UML for BOINC: A Modelling Language Approach for the Development of Distributed Applications based on the Berkeley Open Infrastructure for Network Computing

Christian Benjamin Ries

Director of Studies
Professor Vic Grout

Second Supervisor
Prof. Dr. rer. nat. Christian Schröder

Submitted to the University of Wales
in partial fulfilment of the requirements for the Degree of
Doctor of Philosophy

Department of Computing
Glyndŵr University
11th January 2013

Thesis Declaration

I hereby declare that this work has not been accepted in substance for any degree and is not currently being submitted in candidature for any degree.

Signed..... (Candidate)
Date.....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by references. A bibliography is appended.

Signed..... (Candidate)
Date.....

Statement 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed..... (Candidate)
Date.....

Statement 3

The research was completed under the guidance of
Professor Vic Grout (Main Supervisor), at Glyndŵr University.
In my capacity as main supervisor of the candidate, I certify that the above statements are true to the best of my knowledge.

Signed..... (Main Supervisor)
Date.....

For me — a long journey
has just started!

Abstract

In this thesis an extension of the Unified Modeling Language (UML) 2.4 is elaborated and defined towards the modelling of distributed High Performance Computing (HPC) applications based on an abstraction of the Berkeley Open Infrastructure of Network Computing (BOINC) middleware. The UML profile "UML for BOINC" comprises semi-formal constraints, graphical notations, and code-generation approaches. The general idea behind UML, as part of the Model-driven Engineering (MDE) initiative, is the reduction of the abstraction gap between a software system's specification rules — if possible in a tool-supported automated manner. The discovery of new MDE approaches and methodologies is the moving force behind this research. In this thesis the immense power of UML is used to specify complex systems on a high abstraction level with BOINC for the efficient creation of heterogeneous client-server based high performance computing installations.

In the present study, an effort has been made to find modelling approaches by use of UML for defining an abstraction layer for BOINC's functionalities. Based on the UML profile, a workflow model is provided that allows the specification of an executable application with the help of specialized UML diagram types. In order to test the concepts realized in the UML profile suitable code generators are implemented that transform UML specifications into executable BOINC-projects. Furthermore, a distributed scientific application can be generated which supports the handling of a large set of runtime parameters and user specific scientific computation algorithms. The validation and final storing of scientific results can be easily described by use of a state machine and activity diagrams. Based on these results, a case-study proves the general applicability of the discussed approaches.

This study contributes several emerging modelling approaches and technologies relating to how a BOINC-project can be specified, implemented and maintained. A representative case-study proves the applicability of the new presented principles. The key contributions are a world's first UML 2.4 Profile "UML4BOINC" and several code generation approaches, which allow the specification and creation of BOINC-projects by use of a minimalistic IDE prototype, the so-called Visu@lGrid.

Acknowledgments

Many people helped, inspired, or otherwise facilitated my research during the last years. First, a very special thanks to my two thesis advisors Professor Vic Grout and Prof. Dr. Christian Schröder for their support — both their immense knowledge and their intense coaching made it possible to finish this thesis.

I am also honoured that Dr. Muhammad Younas and Professor Peter Excell have agreed to serve as members of my Ph.D. evaluation committee.

Thanks to the people of Glyndŵr University for their support in research regulations and advice. In no particular order: Dr. Stuart Cunningham, John Davies, Nigel Houlden, Paul Comerford, Mathias Kreider, Ian Sturrock, Hayley Dennis, Julia Wainwright, Nikki Johnson, Misha Jepson, and Dr. Rich Picking.

Thanks to the people of the working group *Computational Materials Science & Engineering* (CMSE) at the Bielefeld University of Applied Sciences. I wish to thank the following people who have, in whatever way, supported me professionally or personally during this period of study. In no particular order: Thomas English, Thomas Hilbig, Mikhail Tolstykh, Lisa Teich, Alexandra Marcelina Boggasch, Dr. Marco La Russa, Andrea Sellmeier, Dorothea Glaunsigner, Nadja Kroke, Gabi Fischer, and Hermann Engesser. Last, but by no means least, I wish to thank my family and their support at any time.

In conclusion, I recognize that this research would not have been possible without the financial assistance of the Federal Ministry of Education and Research (BMBF), the Bielefeld University of Applied Sciences and its Department of Engineering and Mathematics, and express my gratitude to those agencies and institutes.

Declaration

Parts of the research from this thesis that have been published or are presently under review are given below.

The up-to-date list of publications can be found on
`http://www.christianbenjaminries.de/publications`.

Refereed conference papers

- **Christian Benjamin Ries** and Vic Grout. *Code Generation Approaches for an Automatic Transformation of the Unified Modeling Language to the Berkeley Open Infrastructure for Network Computing Framework*. In International Conference on Soft Computing and Software Engineering (SCSE13), San Francisco – CA – USA, 2013 *(acceptance rate below 22%)*.

- **Christian Benjamin Ries**, Christian Schröder and Vic Grout. *Model based Generation of Workunits, Computation Sequences, Series and Service Interfaces for BOINC based Projects*. In International Conference on Software Engineering Research and Practice (SERP12), Las Vegas – NV – USA, 2012 *(acceptance rate 28%)*.

- **Christian Benjamin Ries**, Christian Schröder and Vic Grout. *Approach of a UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC)*. In Proceedings of the IEEE Conference on Computer Applications and Industrial Electronics (ICCAIE 2011), Penang – Malaysia, 2011.

- **Christian Benjamin Ries**, Christian Schröder and Vic Grout. *Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC)*. In Paul S. Dowland, Vic Grout, Bernhard G. Humm and Martin H. Knahl, editeurs, Proceedings of the Seventh Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2011), pages 67-76, Furtwangen – Germany, 2011. Plymouth University.

- **Christian Benjamin Ries** and Christian Schröder. *ComsolGrid – A framework for performing large-scale parameter studies using COMSOL Multiphysics and the Berkeley Open Infrastructure for Network Computing (BOINC)*. In COMSOL Multiphysics Conference 2010, Paris – France, 2010.

- **Christian Benjamin Ries**, Thomas Hilbig and Christian Schröder. *A Modeling Language Approach for the Abstraction of the Berkeley Open In frastructure for Network Computing (BOINC) Framework*. In International Mul-

ticonference on Computer Science and Information Technology (IMCSIT), pages 663-670, Wisla – Poland, 2010.

Books

- **Christian Benjamin Ries**. BOINC – Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing. Springer-Verlag, Berlin Heidelberg, 2012.

Professional Journals

- **Christian Benjamin Ries** and Vic Grout. *Code Generation Approaches for an Automatic Transformation of the Unified Modeling Language to the Berkeley Open Infrastructure for Network Computing Framework*. In International Journal of Soft Computing and Software Engineering (JSCSE), ISSN: 2251-7545, Prefix DOI: 10.7321/jscse, vol. 2, 2013

- **Christian Benjamin Ries** and Christian Schröder. *Public Resource Computing mit Boinc*. Linux-Magazin, vol. 3, pages 106-110, 2011.

Master Thesis

- **Christian Benjamin Ries**. *ComsolGrid – Konzeption, Entwicklung und Implementierung eines Frameworks zur Kopplung von COMSOL Multiphysics und BOINC um hoch-skalierbare Parameterstudien zu erstellen*. Master's thesis, Bielefeld University of Applied Sciences, 2010.

List of Figures

List of Tables

Table of Contents

Part I

The Problem

CHAPTER 1

Introduction

It's not that I am so smart.
It's just that I stay with problems longer.

ALBERT EINSTEIN

1.1 Motivation

When aiming to cope with the ever-present problem of growing software complexity, Model-driven Engineering (MDE) researchers need to invent technologies that developers can use to generate domain-specific development environments [46]. An appropriate modelling language is one of the most essential elements for Model-driven Development (MDD) approaches. MDE is a specialisation of the idea of modelling itself and in this thesis mainly used for software development. For obtaining modelling languages that are adequate, different MDD approaches have defined their own Domain-specific Languages (DSL) or Domain-specific Modeling Language (DSML) in order to present their particular modelling needs. Two of the benefits that the use of DSLs/DSMLs provides to MDD approaches are the following [8]:

- a correct and precise representation of the conceptual constructs related to the application domain, and
- a simplification of the implementation of tools based on the improvement of modelling tasks, development and maintenance of generated software solutions.

MDE and the modelling approaches involved offer a promising methodology to lower the barrier for using complex technologies (Section 2.1). As a matter of fact they enable to use models as the major development artefact [210], provide means for creating different platform specific applications by use of a higher model-based abstraction or increase the product quality, i.e. when the model solves the problem the generated code implementation must not be modified or only slightly. MDE is

a large trend within the software industry and carefully layers additional levels of abstraction onto the underlying hardware [20]: High-level languages have replaced assembly language, libraries and frameworks are replacing isolated code segments in reuse and design patterns are replacing project-specific codes. Nowadays, there are many graphical tools in use [62]: the large body of tools in the Unified Modeling Language (UML) toolkit, the Specification and Description Language (SDL), the Business Process Execution Language (BPEL) and model-driven approaches which ties these tools together with DSLs in order to eliminate the need to write a code manually. Thus, the developmental process can be visualised by UML notation elements (Chapter 4). Accordingly, the relevant context can be interpreted faster that the reading of the code implementation's plain-text, i.e. model visualisations are more intuitive, also known as *"A picture is worth a thousand words"*.

The moving power for the research of this thesis is the creation of methodologies to use the Berkeley Open Infrastructure for Network Computing (BOINC) as a framework within a MDE environment. BOINC is a very prominent grid computing framework which enables the creation of high performance computing installations by means of public resource computing (Chapter 5). The client retrieves a project specific application from the server along with so-called workunits, i.e. a number of parameters usually provided in data files of simple ASCII or binary format that are needed optionally by the application and to perform specific tasks. Each BOINC project has its own infrastructure, i.e. few daemons, periodically executed tasks, hosts, and scientific application for different target platforms [55, 73]. BOINC can solve large scale and complex computational problems. It should be usable for non experts due to visual language. Visual languages are often used in different fields of application. Prominent examples are Labview [201], Lego's NXT-G [61], Microsoft's Visual Programming Language (VPL) [182] and Google's Blockly [179]. These tools provide a steep learning curve, and results can be created efficiently. Based on the UML notation and specifications of specialised viewpoints of the same system, an Integrated Development Environment (IDE) prototype — the so-called Visu@lGrid (meaning *visualised grid*) — can show that the development of a BOINC project can be done more professionally. The focus of this thesis is to reduce the initial and further steps in order to set-up a BOINC project and to create expediently distributed computations.

1.2 Scope of this Thesis

This thesis shall cover the specification of the new *UML for BOINC Profile* (UML4BOINC) and its implications. It provides a means for setting up a BOINC-based project, creating or adding a scientific application and packages of work definitions which have to be performed, i.e. settings of how computa-

tions are parameterised. A UML profile is an approach which can be used to supply a domain-specific dialect for the development of BOINC projects. Thus, it mainly enquires how to reduce the barriers of handling BOINC-related maintenance and administration tasks, e.g. the creation of a new BOINC-based project or the handling of how workunits have to be performed by scientific applications. UML4BOINC has to be supported by additional DSLs and DSMLs which raises the abstraction level and enables a practical application of the UML and DSL/DSML approaches. As a matter of fact, a good DSL/DSML renders professional programmers more productive even if it is not embraced by user programmers. An excellent DSL may end up having a professional programmer to write it — but is reviewable by domain experts [45]. The main scope of this thesis deals with the specification of UML specialisations which are depicted by adapted UML visual notations. Additionally, this thesis covers how the development of BOINC-based projects can be done comfortably by the help of an IDE. At the time of writing this thesis, there does not exist any applicable approach for the creation of BOINC projects within an IDE. Thus, this thesis introduces several approaches of how an IDE can be designed and structured and for which minimalistic graphical widgets must be created in order to represent a BOINC project graphically. This comprises the following contributions of the thesis:

1. An analysis of BOINC's functionalities and architecture requirements toward the abstraction through UML.

2. A semi-formal definition of UML4BOINC. All additional constraints to restrict the use of UML elements are performed by informal statements or by the formal Object Constraint Language (OCL) [127].

3. A definition of UML4BOINC's semantic by introducing new diagram types which are used to cover BOINC's functionalities and in order to enable the use of different viewpoints of the same BOINC project.

4. A methodology describing how the introduced UML4BOINC's stereotypes and illustrated examplary models in this thesis can be used to generate necessary parts of a BOINC project.

5. A proof-of-concept implementation of selected introduced specification and conceptual ideas.

The profile described in this thesis covers most of the BOINC functionalities. Some characteristics are left out due to the fact that they are rarely used by BOINC project developers or only necessary for very special cases as described in Section 5.5.

1.3 Novelty of this Thesis

This thesis introduces a new method for the creation and administration of BOINC-based projects. **Up to the best known state of the art, during the writing of this thesis there exists no comparable approach to work on BOINC projects with the help of UML, code-generation facilities or an additional integrated development environment.** As stated by Werner [74], UML uses multiple modelling paradigms and diagram types to model different aspects of a software system by providing multiple viewpoints on the same model. This enables the application of UML in almost all stages of a software development process. The system can be described in terms of abstract and platform independent to very concrete and platform specific models.

In this thesis, the functionalities and methodologies of a BOINC project are abstracted by a new UML-based abstraction layer. It illustrates in detail how a BOINC project can be set-up and maintained by specialisations of UML elements. Different viewpoints are reviewed and specified; they are able to have a compact view of the BOINC project functionalities. Moreover, this thesis introduces a new visual abstraction of BOINC which is based on the UML notation. Hence, a complete BOINC project, consisting of the scientific application implementation, the workunit creation and maintaining, and the creation of the BOINC infrastructure, can be realised.

1.4 Research Process of this Thesis

BOINC's structural features and functionalities are analysed (Section 5.5) in the first place. Several BOINC projects are therefore developed [186, 91, 57, 73] or maintained [165]. In addition, BOINC's wiki [167] and BOINC's sourcecode [167, SourceCode] provide voluminous information about BOINC's behaviour and how the BOINC projects can be maintained. Numerous methodologies for software development proccesses and teams have been introduced in the last decades: Iterative [17], Spiral [34], Waterfall [166] & V-Model [107], (distributed) XP [14], Scrum [92], Test-driven development (TDD) [75], and more. In this thesis, however the research process is a kind of iterative development, as shown in Fig. 1.1. The process is initialised by the arrow in the top-left corner; one of BOINC's structural features or functionalities (in the following referred to as *features*) which has been analysed previously (Section 5.5). In the next two steps (1-2) the UML specification [125] is analysed and UML metaclasses are chosen and verified within an iteration process; it takes a long time as the features cannot be expressed with UML metaclasses and additional new stereotypes, tag-values and constraints (Chapter 6 and 7). The selection is afterwards only based on a theoretical construct and has to

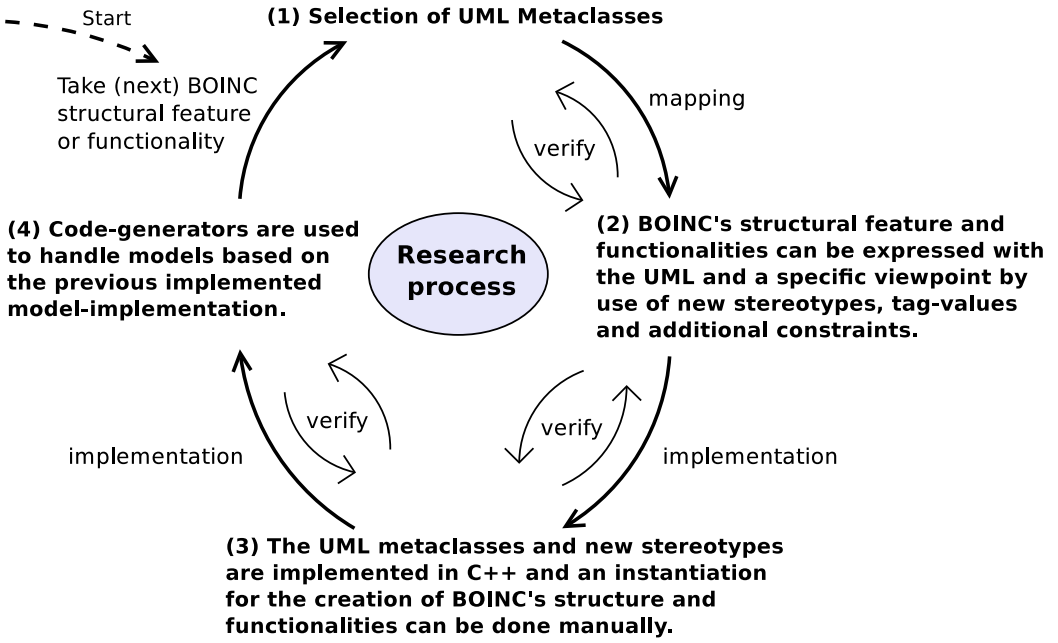

Figure 1.1: The research process in this thesis is based on a circular and iterative process; the modelling of BOINC's entities and how they can be specified with UML with a followed code-generation to create and to maintain a BOINC project automatically.

be transformed in step (3) to an executable implementation which enables use of the model in future developmental phases. Thus, the individual UML metaclasses, the stereotypes, tag-values and constraints are implemented as a C++ class-hierarchy. This hierarchy is verified and modified as long as it does not match the theoretical approach. When the results of the current intermediate step do not fit properly, the constructs of the previous steps have to be modified. Any modification in one of the previous steps needs an additional verification. This iteration process is performed between steps (1-2), (2-3), and (3-4). A code-generator is implemented in step (4), it enables the traversal of the class-hierarchy, to parse the information and to create applications which are used to change the BOINC project structure or the behaviour (Chapter 8).

1.5 Tool technologies

The programming language C++ [118, 141] is mostly used in this thesis, namely; up to the newest C++11 [145] specification. The author is fond of programming with C++ and not only because most of BOINC's implementations are performed in C/C++. As a consequence, C++ is chosen to allow BOINC developers and enthusiasts to adapt the results of this thesis for their personal future work. In addition, the author's daily work takes place within an UNIX-based environment [205]. Most

of the implementations for verifying approaches are done for UNIX, in particular Linux [196] and the Ubuntu distribution [206]. At last, this thesis does not use a specific tool for creating diagrams and UML model visualisations. The following tools are used: Dia[1], VisualParadigm [207] and own implementations with the cross-platform application framework Qt[2] especially in Chapter 7.

1.6 English Differentiation

This thesis exhibits a mix of British English (BE) and American English (AE) as in the case of the words modelling (BE) and modeling (AE). The BE form is used continuously in this thesis, but the application of proper nouns requires the AE form, e.g. Unified Modeling Language (UML), Domain Specific Modeling Language (DSML) or Eclipse Modeling Framework (EMF).

[1] https://live.gnome.org/Dia
[2] http://qt.nokia.com

1.7 Definitions

In this thesis, the first letter of BOINC's provided software components name is always written in upper-case, e.g. Validator or Transitioner. This section introduces some general definitions and word semantics which are used in this thesis.

AST Abstract-syntax Tree. A tree-based structure for programming languages, their language recognition and interpretation.

API Application Programming Interface. A public interface to enable the use software libraries.

BOINC Berkeley Open Infrastructure for Network Computing. This is a grid computing framework by means of public resource computing.

bottom-up Proceeding from the bottom or beginning of a hierarchy or process upwards; non-hierarchical: bottom-up decisions [159, bottom-down].

Clients, Participants, Volunteers The client-side has several names in the context of BOINC. Some research papers state the client-side or users of the client-side as *Volunteers* or *Participants*. For all cases within this thesis, the client-side describes users and hosts which are connected to a BOINC project. In addition, BOINC's Manager and BOINC's Client are on the client-side and the components are part of the *Client Point of View (CPoV)* as shown in Fig. 1.2.

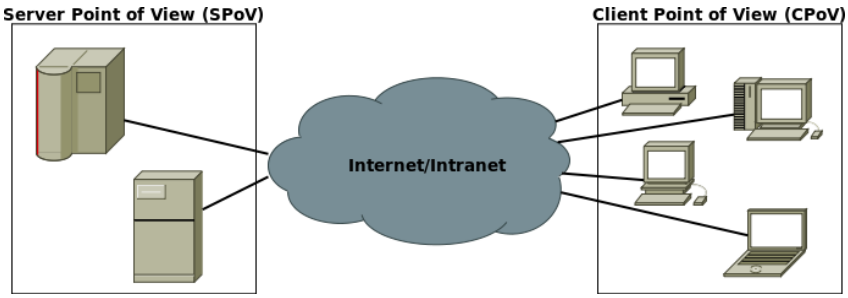

Figure 1.2: The *Server Point of View (SPoV)* is presented on the left-, the *Client Point of View (CPoV)* on the right-hand side. Only applications and architecture elements are defined in the SPoV; they are executed or used on the server-side of a BOINC project. In contrast, the CPoV is used to define applications and architecture elements on the client-side of a BOINC project, i.e. to let users and their hosts participate.

code-centric see *bottom-up*

DSL Domain-specific Language. A language designed to be useful for a specific set of tasks. This is in contrast to a general-purpose modelling language such as the Unified Modeling Language (UML). A DSL is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside of it. DSLs are usually small, more declarative and less expressive than a general-purpose language [96].

DSML Domain-specific Modeling Language. See DSL.

Input A scientific application needs input files to solve computational problems. These input files include arbitrary data values as binary or plain-text tokens.

MDA Model-driven Architecture. This is an approach for IT system specifications that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform [124].

MDD Model-driven Development. This is an approach that aims to separate the act of coding and deploying applications on a specific hardware and programming language/database platform from the practice of gathering and modelling higher level business and organizational processes and requirements [106].

MDE Model-driven Engineering. This aims to raise the level of abstraction in program specification and increase automation in program development. MDE uses models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. An increase of automation in program development is reached by using executable model transformations. Higher-level models are transformed into lower level models until the model can be made executable using either code generation or model interpretation [142].

metacomputer This is an execution environment in which high-speed networks are used to connect supercomputers, databases, scientific instruments and advanced display devices; perhaps located at geographically distributed sites [5].

MOF Meta Object Facility. MOF is an OMG standard for Model Driven Engineering (MDE). MOF is designed as a four level architecture. UML is considered as layer 2 MOF model. Layer 3 (M3) contains the language used by MOF to build meta-models [96]. An OMG standard, closely related to UML, that enables metadata management and modelling language definition [124].

OMG Object Management Group. A consortium that promotes the adoptions of standards [96].

Output A scientific application stores the results of the computation in output files.

PIM Platform Independent Model. A model of a subsystem that contains no information specific to the platform, or the technology that is used to realize it [124].

problem-centric see *top-down*

PSM Platform Specific Model. A model of a subsystem that includes information about the specific technology that is used in the realization of it on a specific platform and hence possibly contains elements that are specific to the platform [124].

RPC Remote Procedure Call. These calls are used in distributed systems to query information of remote functionalities which can be merged dynamically during the runtime.

Scientific Application BOINC is based on a server-client architecture. Server-side services are executed to manage the lifetime of workunits and to handle client requests. The main purpose of the client-side is to execute a *scientific application*, which is used to solve computational problems, e.g. the statistical computation of π [91, Chapter 11] or handling of video sequences (Section 9).

Services, Server All BOINC daemons, databases or webserver instances are executed on the server-side of a BOINC project. For all cases within this thesis, the server-side describes the components which are not executed on the client-side. These components are part of the *Server Point of View (SPoV)* as shown in Fig. 1.2.

top-down This describes the proceeding from the general to the particular: Computing working from the top or root of a tree-like system towards the branches [159, top-down].

1.8 Structure of this Thesis

This thesis is separated into ten chapters. The current chapter gives a short overview of the thesis, i.e. the motivation, the scope, the novelty, a general description of how the research process is realised and which limitations exist. Chapter 2 illustrates the overall research problem, i.e. the research questions covered in this thesis and their focus. The state of the art is described in Chapter 3, i.e. recent work in the research fields of MDD, UML and OCL. A presentation of available

tools and approaches for handling the mentioned fields and an overview of tools and technologies of other grid computing approaches is given. Next, Chapter 4 explains in detail what MDD is and how it can be used in software development processes. The Berkeley Open Infrastructure for Network Computing (BOINC) framework itself is introduced in Chapter 5. It represents BOINC's general idea and field of application, the framework architecture, the tool-chain of the server- and client-side, and finally demonstrates a breakdown of required and optional features which are used to set-up and start a BOINC project. The following three Chapters 6, 7, and 8 contain the original work of this thesis. After a specification of UML4BOINC, the semantic, and the code-generation to perform the transformation from a UML4BOINC model to a BOINC project are described. Chapter 9 shows a case-study and proves the practicality of UML4BOINC. In the end, Chapter 10 completes this thesis with a summary and by stating the contributions and an overview of possible future work.

CHAPTER 2

Problem Statement

Einstein is quoted as having said that if he had one hour to save the world he would spend fifty-five minutes defining the problem and only five minutes finding the solution.

2.1 Introduction

The Berkeley Open Infrastructure for Network Computing (BOINC) framework is based on a client-server architecture. It provides a procedural Application Programming Interface (API) for the creation of BOINC-based scientific applications [58]. In contrast, the Unified Modeling Language (UML) is an object-oriented programming language and system description language [125] which can be used for MDE and MDD applications. In addition the developers of a BOINC project must be familiar with different IT technologies to survey the complete process of BOINC-based distributed projects. While BOINC is used by many different scientific groups and hobbyists [147, 162], MDE and UML are used by several working groups [102, 105, 97]; there exists no way to cope with the complexity of BOINC projects by use of MDE and UML modelling approaches. This thesis is focused on research concerns as described in the following two sections. Section 2.2 deals with the complexity of a BOINC project and Section 2.3 states relevant open questions of handling the BOINC complexity by modelling approaches. The hypotheses of this research are stated in Section 2.4.

2.2 BOINC

2.2.1 Infrastructure

The BOINC infrastructure consists of a server-side and client-side. The server-side is based on a simplified architecture, i.e. every task is separated in its own BOINC service. A minimum of six services is required in order to run a BOINC project (Section 5.3). These services constitute a specific chain of responsibility and each is used to handle a specific task, e.g. BOINC's Validator is only used to verify the results which return from computation. The configuration of a BOINC project must be maintained with attention; every wrong value implies a malfunction of the BOINC project and can result in a broken system [58]. In practice, wrongly configured BOINC services do not work propertly, i.e. a BOINC Assimilator will not store results for a future use or a BOINC Validator is not able to validate results received by clients. In fact, just a few steps are needed for the creation of a BOINC project: (1) download and install prerequisite software packages, (2) download the BOINC sources, (3) configure and build the BOINC software and (4) call scripts for an automatic BOINC project installation [167, ServerIntro]. Next, every part of the new BOINC project can be modified as presented in the following:

- A selection and location of BOINC services can be executed, i.e. in case multiple hosts are used, the execution environment of BOINC services can be assigned to individual hosts. Additional parameter variations can be used, i.e. the same BOINC service is executed on two hosts with different runtime parameters.

- A specific host can be defined as the main installation (Section 6.5.6), i.e. supposing individual hosts are used for the BOINC project server-side, the main installation has to be distributed to all additional used hosts. As a matter of fact, all BOINC services need the project environment for the correct execution.

- The BOINC project database can be combined with replication databases for load-balancing and speed-up purposes.

More detailed adjustments are possible. As [58] describes, there exist typical errors that can occur due to manual editing of the BOINC server configuration files. As a consequence, a significant effect on the system's integrity and application performance can be expected and this results in the mentioned malfunctions.

However, it can be an uncomplicated proceedure from downloading the BOINC sources to running a BOINC project if only one host is chosen. In this case, all work is performed on one host, i.e. BOINC sources can be accessed directly, BOINC services must be started on this host, database and web-server installations are also

directly available and relevant parameters are set only from this host to BOINC's configuration. In addition, only one configured network interface card is necessary and exclusive ports of the Hypertext-Transfer Protocol [Secure] (HTTP[S]) [187, 110] must be enabled for network traffic. Nevertheless, it can be an enormous challenge to set-up a BOINC project with more than one host and when BOINC services are distributed for the execution to individual hosts.

2.2.2 Special Interest Groups

Different special interest groups can administer a BOINC project, i.e. "administrators" configure and maintain the system installation, and "scientists" can create new computation series or workunits and add new scientific application versions. Scientific applications can be implemented in different ways, e.g. scripts, hand-written code in different programming languages (C/C++ and FORTRAN are supported by default) or through ISV (Independent-software Vendor)-Applications or legacy applications [32, 18] (Section 6.6), e.g. [57, 73] present a wrapper for handling COMSOL Multiphysics [199] computations. An application can be written and developed to a great extent from scratch. This approach is the most difficult of all and BOINC's application programming interface (API) is fortunately not changing frequently with each new version. As a matter of fact, 23 different BOINC functions are necessary to implement a successfully running and research relevant application [165, 58, 70]. When a BOINC project is installed and a scientific application implemented, new problems and questions will arise: (i) who can manage a scientific application, i.e. removing or upgrading it and (ii) how can the access of groups and users be restricted, i.e. to selected functions or system calls. These steps can be executed on different hosts; in addition, it is highly error-prone to install and configure all required software components and interface settings, to keep an always valid configuration and stable system when software needs to be upgraded or a host has to be replaced. Besides, it is too cumbersome to create a fast and user-friendly BOINC project only for tests or quality assurance processes.

2.2.3 Handling of Workunits

When a BOINC project is created and started, a scientific application is implemented and added; finally, it is necessary to add workunits which are performed by clients (Chapter 5). The creation of workunits can be performed in several ways: (i) by calling a script, (ii) by the implementation of a tool to call specific BOINC framework functions, or (iii) by use of BOINC's Remote-procedure Calls (RPC). All these approaches need information about the specifically planned input/output files and the handling parameters. BOINC works on individual workunits only; in the case that workunits have to be performed by the same scientific

application (e.g. a set of workunits by one scientist); it is necessary to specify different names for all workunits in order to distinguish them for different scenarios. These scenarios can have individual requirements for the storing of computational results, i.e. storing of the result within user's home directories. Furthermore, the validation process can vary, when different parameters are used for the computations. Both steps need access to the result files; thus, the implementation has to be hidden from the scientist and domain experts. There are BOINC projects that need to perform workunits one after the other (Section 7.4).

2.2.4 Planning and Creation of the Scientific Application

When someone is in need of a BOINC project, then it is mainly used to distribute a scientific application and to solve engineering, scientific or general computational problems. BOINC can be used by everyone, either a person with a deep knowledge of all related technical aspects, or by novices or students who use BOINC for small studies or for testing purposes. However, a scientific application has to be implemented in both cases. Accordingly, it can be necessary to understand BOINC's application programming interface (API) and all relevant BOINC-related methodologies, i.e. how input/output files have to be specified and opened by the scientific application or how an application has to be initialised, performed and finished. In practice, the background technologies need to be familiar (e.g. compiler use, heterogeneous execution environments); it would be good to know how the basic internet protocols work (e.g. Hypertext Transfer Protocol [110]), and how the handling of computational results can be performed (e.g. file-storing or database-storing). In any case, it is necessary to know how the target environment can be controlled (e.g. storing in a database needs knowledge about the Structured Query Language (SQL) [116]); and how the optional computation can be performed on different processors (e.g. central-processing unit (CPU) or graphics-processing unit (GPU)). Whenever BOINC has to be used with legacy-applications, it is mandatory to configure BOINC's wrapper [167, WrapperApp] or the additional third-party wrappers [18, 32, 57, 73]. Moreover, the developer has to know most of the API to select the suitable calls because different scenarios require different API-calls.

2.3 Modelling Abstraction

The biggest challenge concerns UML and how it can be used for modelling BOINC projects; besides, BOINC has to cope with the scenarios and requirements stated above (Section 2.2). The possibility of having different viewpoints of a system enables the creation of developer teams, whereupon any team member works on its own domain, e.g. system administrators are using the Infrastructure view (Sec-

tion 6.5) and researchers are responsible for the creation of scientific applications (Section 6.6). Modelling questions that emerge are the following:

- How can a scientific application be described in general?
- How can BOINC's API-calls be used with UML? In particular, BOINC provides a small set of general functionalities (e.g. file open methods for different operating systems) and specific BOINC related functionalities (e.g. to report how much work is performed for a computation) which can be used by developers and system designers.
- Are there relevant modelling aspects for different architectures or operating systems? Generally, UML is a modelling language and it is possible to restrict its use for a small amount of different systems or architectures. Is it necessary to consider this possible problem or is a general model description adequate?
- How can an independent-software vendor application or a legacy-application be used?
- How can a new version of a scientific application be deployed and which parts, either on the server- or the client-side, have to be recreated in case of a new version? During a manual administration it is necessary to put new scientific versions in a special directory-hierarchy; in addition, a call of a specific tool has to be made because it will announce the scientific application to the context of a BOINC project. As a matter of fact, every version of a scientific application can possess a different handling for workunits because the implementation of the computation can change and thus different values are expected from the workunits. However, it is necessary that these changes are taken into account, e.g. the deployment of a scientific application is only allowed when no "old" workunits are available and open for processing.

2.4 Hypotheses

Based on the previous Section 2.2 and Section 2.3, the following hypotheses are stated:

1. UML can be used for the creation of infrastructure specifications which are used for the deployment of BOINC projects.
2. UML helps different domain experts to use BOINC without the need to implement any line of code.

3. UML can be used as an abstraction layer for BOINC's programming and execution interface.

4. A code-generation process can be used to generate BOINC parts.

The following chapters present the research results and confirm the hypotheses stated above. Chapter 9 will then compare the results with the hypotheses.

2.5 Topics not covered in this Thesis

The previous sections have presented some open issues which are covered in this thesis. However, some research questions cannot be dealt with during this work:

- Graphical-processing Units (GPUs) are not covered directly. They have become more important in recent years and are used in various areas of scientific computing [140] and general engineering computations [132]. Accordingly, different software frameworks were developed: NVIDIA's CUDA [86] or OpenCL as general programming framework for ATI and NVIDIA GPUs [122].

- The Message-passing Interface (MPI) is not covered when a cluster is available on the client-side, such as would be possible to use by a scientific application.

- Multi-thread applications are not directly covered. They are used indirectly but cannot be modelled by the use of the stereotypes introduced in this thesis.

- Performance prediction is not taken into account, i.e. it is not possible to measure how long a specific scientific application will take to complete on workunits.

2.6 Summary

This chapter introduces the BOINC-based research questions and hypotheses. Furthermore, it describes the research questions underlying the model-driven development. The limitations of the thesis are stated in the end.

CHAPTER 3

Related Work

You can only find truth with logic if you have already found truth without it.

GILBERT KEITH CHESTERTON

3.1 Introduction

This chapter discusses the related work of modelling tools for grid systems and public resource computing systems. Several existing tools define their modelling language approach independently and lack support for a real standardised specification.

3.2 Modelling Language Support

3.2.1 Approaches and Specifications

Fowler [45] states that using Domain-specific Languages (DSLs) or Domain-specific Modeling Languages (DSMLs) gives an opportunity to have a grammar that is easier to manipulate. According to this, concerns about the difficulty of designing DSLs arise — language design is difficult and designing multiple DSLs will be too extensive for most projects. This objection is often based on the general purpose of language rather than DSLs. The fundamental issue is getting a good language abstraction — that is the hard part of the task. The difference between the Application Programming Interface (API) design and the DSL design is rather small; designing DSLs is significantly harder than designing good APIs.

In [8], the authors introduce an approach for automatic generation of Unified Modeling Language (UML) profiles based on a DSML. Nowadays, there are several Model-Driven Development (MDD) approaches that have defined DSMLs oriented on representing their particular semantics. The authors state that a UML profile can be generated of an integrated metamodel which is used to map the

DSML elements into a UML profile. 11 rules are described for mapping the integration metamodel to the associated UML elements, i.e. Classes, Properties and Associations; the related lower bound and upper bound constraints, Enumerations, Generalisations, and Data Types. There are also other approaches for the automatic creation of UML profiles [16] that can be partially automated through the identification of specific design patterns. Wimmer et al. [63] propose a semi-automatic approach that introduces a specific language to define the mapping between the DSML metamodel and the UML metamodel; this approach does not support $M : M$ associations, i.e. from a data transformation point of view the question arises how to combine multiple input values to produce multiple output values. As a consequence, the effective application is not fully supported in real MDD approaches when different point of views are used and have to be merged for generating the target application. Currently, no approaches exist which have full capability for the complete automatic generation of a UML profile.

Gerber et al. [47] state that the Meta Object Facility (MOF) and the Eclipse Modeling Framework (EMF) are conceptually very similar, both are based on the concept of classes with typed attributes, operations with parameters, and exceptions by supporting reuse through multiple inheritance. Both frameworks use packages as a grouping mechanism and support nested packages. The main conceptual difference lies in their treatment of relationships between classes. MOF has the first-class concept of Associations as a binary relationship between two classes, i.e. the association ends, which have a navigability property. In MOF, there is a distinction between the relationships which are fundamental concerning the definition of a class and those that are observations about a class, i.e. references access their related class and are not fundamental to the definition of the class. EMF is a *bottom-up* and code-centric approach, whereas MOF is used in a natural way and observes a system by a bird's-eye view, i.e. *top-down* for more problem-centric approaches.

Petri Nets [139] are graphical tools for the description and analysis of concurrent processes; they arise in systems with many components (distributed systems). The components of these nets are called *states* (for substances) and *transitions* (for reactions) which are connected by arrows that show the flow direction. Accordingly, the activities of transitions are subdivided in *give* and *take*. The tokens (visualised as black marks) are moved by the occurrence of transitions. Petri Nets are graphical tools: they possess a formal semantic and they express the most desired routing constructs. In addition, there exist various techniques for proving their properties, they are vendor-independent [81], and they are a means for modelling workflows. UML activity diagrams (see Chapter 4) have also been used for workflow modelling. They almost share the same features as Petri Nets: they are graphical, use bubbles or rectangles and arrows, are vendor-independent and express the favoured routing constructs. Contrarily, the UML activity semantic is not

formal (nor precise) and not intended for workflow modelling.

Pi-Calculus (or π-Calculus) [88] is a process algebra for systems and composed of processes that run simultaneously and interact through channels. As described by Zhang et al. [27], the elementary entities of Pi-Calculus are names and processes. Names play dual roles as communication channels and variables. The basic aim of Pi-Calculus is the fact that processes communicate with each other by transferring names through shared channels. These names can be used by the receiver in order to communicate with other processes. In fact, this concept enables "mobility" [90]. The channels can be used as a *monadic version* (channels carry a single value at a time) or as a *polyadic version* (many values per channel). In addition, Pi-Calculus can be applied for modelling different kinds of systems: the representation of data structures and data types, the description of object-oriented features, and functional programming features. Pi-Calculus possesses a graphical representation as Petri Nets but the mathematical foundations and semantics of Petri Nets and Pi-Calculus are totally different [27].

3.2.2 Tools

Xtext [208] is a framework for the development of programming languages and DSMLs; it is a bottom-up technology and covers all aspects of a language infrastructure from parsers, over linker, compiler or interpreter for Eclipse's [192] Integration Development Environment (IDE) integration. Xtext is supplied with default settings and can be modified to fulfill the developer's need. The core of Xtext's generated infrastructure is based on EMF, which is otherwise based on the Ecore metamodel [191]. Models conforming to the Ecore metamodel have a direct mapping to the Java language constructs. The main disadvantage of using EMF for this thesis is that there is no established mapping to the C++ programming language which limits the interoperability between the set of tools available around the EMF technology and the MDD's need to leverage libraries and techniques available in C++, e.g. metaprogramming [133]. EMF4CPP [99] tries to fill this gap by generating C++ implementation artefacts from Ecore-based metamodels. It produces the C++ source code which gets packed as a shared library from the Ecore model. The library enables the manipulation of type entities defined in the metamodel; in addition, it provides the reading model instances and the processing within a C++ application. Unfortunately, the project will not be maintained any more[1].

The Graphical Editing Framework (GEF) [194], Graphical Modeling Project (GMP) [195] and Graphical Modeling Framework (GMF) [84] provide a set of generative components and runtime infrastructures for developing graphical editors based on EMF. During the first stages of research for this thesis, it turned

[1] Last release was on the 1^{st} December 2010

out that these tools have difficulties with constantly changed requirements [58]. First of all, changes in the GMP models need an update of the underlying EMF model; secondly, the relations between the GEF model and GMP/GMF tools have to be modified, i.e. the mapping of drawing tools to their specific drawing methods. In addition, model modifications were not solved automatically and thus a faulty model emerged; moreover, error messages were not meaningful during the model transformation. Sometimes it took a few days to repair the model.

In order to reinforce these statements, Wienands et al. [95] performed an experiment for implementing a visual domain-specific language editor with the help of GMF and its underlying technologies. The authors state that a lot of the challenges during the implementation were caused by a steep learning curve of the underlying technologies; especially by GMF and the technologies it is based upon, namely EMF and GEF. Furthermore, the decomposition is not fully supported, i.e. double-clicking on a diagram element does not open a subdiagram of a different diagram type. GMF does not handle all modelling aspects; the generated code has to be customised to support different features on different code levels, e.g. read-only attributes require code modifications in the object model and the initialisation of model graphs needs modifications in the related diagram code. Nevertheless, the experiment has shown that the development of a visual domain-specific language and MDD approach is less time consuming, i.e. approximately two-thirds of the time was saved. They conclude by stating that productivity increases through this approach.

3.3 Unified Modeling Language (UML)

3.3.1 Approaches and Specifications

Steve Cook states [11, p. 5] that UML is on the one hand often used to model the structure of software systems, e.g. due to class or component diagrams. On the other hand, behaviours are rarely described with UML because of a poorly-defined and poorly-integrated notation, an error-prone and lossy code round-tripping and missing tool support. UML is not defined precisely, e.g. a black diamond symbol is meant as “composition” but it is not clear what will happen when one of the elements (on one side) is deleted; UML does not specify how this will be handled.

Steve Mellor declares [11, p. 7] that UML defines the *syntax* of *diagrams* and not the *semantics* of *models*. The problem is that UML tailors a sketching language, one that provides modellers with a way to draw basic or advanced sketches; thus building more sketches on top of this unsound base is only going to make the problem worse.

Joaquin Miller demonstrates [11, p. 7-9] some failings of UML 2.0. Data types

and classes should be served by a single fundamental architecture, which is fragmented in two concepts within UML. Furthermore, UML uses the same notation for inheritance and subtyping of UML elements, i.e. arcs denote the derived class and subtype relation. The class hierarchy of UML 2 is not a type hierarchy, i.e. there are subclasses that are not like their superclasses. Moreover, Miller discusses the ambiguity of UML; UML uses the term "to represent" with no clean concept of representation, e.g. "A lifeline represents an individual participant in the Interaction." [125, Section 14.3.17] or "Attributes of a class are represented by instances of Property [...]" [125, Section 7.3.7]. In the first case, the individual is typed as an abstract ConnectableElement representing a set of instances [125, Section 9.3.5]. The UML specification does not describe how this representation is instantiated. In the second case, the description is not clear, i.e. part of the system *(instances of Property)* represent something in the model *(attributes of a class)*. Section 4.3.1 describes how the UML specification can be read. Besides, the UML specification lacks a consistent vocabulary and theory, e.g. terms like "trace" are used with different focus [125, Section 14.1].

Bran Selic dismisses Cook's, Mellor's and Miller's arguments [11, p. 10-12] and argues that the dynamic semantics of UML are defined generally and cover a wide range of possible specialisations to diverse domains, e.g. the same semantic can be specialised to support either a synchronous or an asynchronous world view. In addition, Selic agrees that UML is incomplete and lacks precision but he asserts that UML has a semantic and where inappropriate, users have the option to use the MOF to define a language independent of UML. Nevertheless, UML is not an "all or nothing" proposition. Users can pick and appropriate only those parts that are feasible in solving the definition problem caused by DSML and UML.

3.3.2 Tools

The VisualParadigm [207] is a UML tool for modelling the database design, defining requirements, modelling business processes and more. In addition, it can be used to draw UML diagrams (Section 4.3). The VisualParadigm lacks the support of code generation of C++, i.e. only class diagrams and state machine diagrams are supported. Moreover, the profiling of all UML elements is not allowed, e.g. UML's metaclass NamedElement cannot be extended by stereotypes.

IBM's Rational Rhapsody [135] provides a collaborative design and development framework for systems engineers and software developers creating real-time or embedded systems and software. Rational Rhapsody supports teams to collaborate in order to understand and elaborate requirements, to abstract complexity using industry standard languages (UML, SysML [128], AUTOSAR [177], DoDAF [153]), to validate functionality early in development and automate the delivery of innovative and high quality products. The mentioned features can-

not be validated because the license of Rational Rhapsody is too expensive, i.e. several thousand Euros for a single workstation license. Rational Rhapsody uses Windows-dependent libraries [101] and the main research machines are UNIX/Linux-based (Section 1.5).

3.4 Object Constraint Language (OCL)

Most of the tools with Object Constraint (OCL) [127] support are based on EMF. SimpleOCL [163] is a proof-of-concept implementation of the OCL 2.2 standard. SimpleOCL lacks the support of the whole OCL standard, i.e. it does not support OCL's *pre*, *post*, and *inv* declarations. It is supposed to be a navigation language embedded in model transformation languages. Thus, it is not usable for this thesis because the UML profile (Chapter 6) defines several stereotypes with OCL's *inv* declarations.

DresdenOCL [149] provides a set of tools to parse and evaluate OCL constraints on various models such as UML and EMF. On the one hand, Dresden OCL provides tools for Java and Structured Query Language (SQL) [116] code generation. These code generation features are not taken into account within this thesis, i.e. the focus is towards the use of C/C++ (Section 1.5). On the other hand, DresdenOCL can be used to verify most of the OCL statements presented in this thesis. The Toolkit is enhanced and maintained mainly by students and scientific staff of the Software Technology Group at the Technische Universität Dresden, where the project is also coordinated.

Other tools lack the support of OCL which does not make them functional for this thesis: AndroMDA [189] is an open source generation framework that follows the Model-driven Architecture (MDA) paradigm. According to the tool information, it takes model(s) from CASE-tool(s) and generates deployable applications fully [77]. Out-of-the-box, this tool can not be used with C/C++ interfaces; otherwise no C/C++ code is generated.

3.5 Grid Technologies

3.5.1 Approaches and Specifications

BOINC is not the only desktop grid middleware framework. XtremeWeb [43, 164] and Condor [148] are two additional frameworks; the first one is not maintained any more. XtremeWeb and BOINC are based on the pull-model, whereas Condor uses the push-model to get new workunits. In contrast to BOINC, XtremeWeb enables the submission of workunits by participants, i.e. this is supported by an interface similar to batch systems.

The design of XtremeWeb [43] provides a global computing framework for resolving different applications, e.g. for projects led by institutions, commercial firms or open source communities. There are two ways of using XtremeWeb: (1) as a *Volunteer* and (2) as a *Collaborator*. In the first case, a single machine or a network of computers collaborates with a global computation; but then, the second case uses the computing side as an independent XtremeWeb project which can spread its own application and work. When no own work is described, this machine set can be used by other XtremeWeb projects which exchange their work between the XtremeWeb project servers, best known as "peer-to-peer computing".

PlanetLab [10] is a global overlay network for developing and accessing broad-coverage network services. It enables multiple services to run at the same time; each service in its own so-called slice and these slices are the centrepiece of the PlanetLab architecture. The slices are collections of virtual machines, each runs on a physical node. Accordingly, every single node has to provide a virtual machine abstraction. The parent machine's resources (e.g. CPU, local disk storage, network bandwidth, etc.) must be shared equally so that the slices do not interrupt each other. In addition, virtual machines must be restricted, particularly with regard to the volume and the nature of network traffic they can generate. Unlike many testbeds, PlanetLab is implemented over the "real" Internet and experimental PlanetLab services must coexist peacefully with the Internet traffic. In short, "PlanetLab is an overlay testbed designed in order to enable researchers to experiment with network applications and services that benefit from the distribution within a wide geographical area" [143].

The Globus Toolkit [5, 193] is a technology for building grids that provide distributed computing power, storage resources, scientific instruments and other tools to be shared securely across corporate, institutional and geographic boundaries. Globus' goal is to combine several distributed and cluster computing technologies and to provide a basic infrastructure of different services, e.g. Parallel Virtual Machine (PVM) [161] or Message Passing Interface (MPI) [121] as machine-independent communication layers. Almost every large-scale High Performance Computing (HPC) production grid, including Enabling Grids for E-sciencE (EGEE) [151], LCG [169], and NorduGrid [158] in Europe, uses Globus Toolkit components as a basis.

Legion [23, 50] was the first integrated grid middleware architecture to address the complexity of grid environments from first principles. It was designed to provide a powerful virtual machine interface layered over the distributed, heterogeneous, autonomous and fault-prone physical and logical resources that constitute a grid. The Legion architecture and implementations follow these principles: provide a single-system view, supply transparency as a means of hiding details, enable flexible semantics, reduce user effort, decrease "activation energy", do no harm and do not change the host operating systems. Legion's application layer supports

the execution of its own implemented applications, wrapped legacy applications and applications with MPI use. In addition, Legion can be used as data storage in which case Legion manages the data, deciding where to place it, how many copies to generate for higher availability and where to place them. A web-based interface supports the management of users or hosts and enables the merging of two or more Legion grids.

Cunsolo et al. [39, 38] discuss an approach to combine Grid Computing (GC) and Public Resource Computing (PRC) technologies with the Cloud Computing (CC) paradigm; they call it *Cloud@Home*. The authors want to actively involve users into a new form of computing, allowing them to create their own interoperable Clouds. Thus, both the commercial and the volunteer viewpoints exist in Cloud@Home: the end-user orientation of the Cloud is extended to a collaborative two-way Cloud in the former case; users can buy and/or sell their computer resources. Cloud@Home can be also considered as a generalisation and a maturation of the @home philosophy: a context in which users share their resources voluntarily and without any compatibility problem; the term *resources* is interpreted in the more general Cloud sense of *services*. Nevertheless, the following issues, challenges and problems exist: managing of resources, the front-end to users must be as easy as possible, the security system must support authentication, data protection, data confidentiality and integrity; redundancy is necessary, clouds should interoperate, and a business model management must be supported. Cloud@Home is based on a three layer architecture: *front-end layer*, *virtual layer*, and *physical layer*. The front-end layer is responsible for the resource management from the global Cloud system's perspective, e.g. the discovery of vanished or new resources. The virtual layer virtualises the physical resources and offers a homogeneous view of Cloud's resources to the end-users. The physical layer provides physical resources for implementing execution, storage services, mechanisms and tools for managing such resources locally. To the best knowledge of the author there is no implementation at the time of writing the thesis[2].

Costa et al. [37] present a BOINC prototype that can run MapReduce [40] jobs, using a pull-model in which communication is always initiated by the client, instead of the traditional push-model used in cluster, and especially by BOINC. MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and to reduce a function that merges all intermediate values associated with the same intermediate key. After handling the tasks, the results are reduced; it is necessary that all outputs of the previous map step are presented within the same key range. The authors break the general BOINC restriction and establish a communication between

[2] 9^{th} November 2012

clients; thus, the reduction of overhead network to central BOINC server is used to decrease several computation results.

3.5.2 Tools

Pllana et al. [41, 53, 52] present the graphical editor Teuta which enables the creation of grid workflows. Teuta is used for workflow composition, submission, controlling and monitoring. In addition, Teuta employs a DSML for grid workflows based on UML2. In Teuta, the workflow is composed by graphically combining predefined elements of the DSML metamodel.

The goal of ASKALON [42, 3] is to simplify the development and the execution of workflow applications on the Grid. It contains a number of high level services including a workflow scheduler and execution engine, a resource manager for computers, application components and a performance prediction service [68]. The ASKALON workflow composition tool [144] is a graphical editor for the UML-based modelling of Abstract Grid Workflow Applications (AGWLs) [42] and distributed parallel programs. This tool is a customised version of Teuta.

3.6 Summary

This chapter introduced related work of different scientific and industrial fields. The sections are divided into a presentation of approaches, specifications and a list of tools available for different related work. In particular, the work of the modelling field, the Unified Modeling Language (UML), the Object Constraint Language (OCL) and the computing grids is outlined.

CHAPTER 4

Model-driven Development

The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.

DON ROBERTS

4.1 Introduction

This chapter will give a general overview about the Model-driven Development (MDD) methodology, its efforts to create approaches for handling different modelling aspects and how MDD can be applied in the software development process. As stated by France and Rumpe [46], the growing complexity of software is the motivation behind the work on industrialising software development. In particular, current research in the area of Model-driven Engineering (MDE) is primarily concerned with reducing the gap between problem and software implementation domains through the use of technologies that support systematic transformation of problem-level abstractions to software implementations. The complexity of bridging the gap is tackled through the use of models that describe complex systems at multiple levels of abstraction, from a variety of perspectives and through automated support for transforming and analysing models. In the MDE vision, models are the primary artefacts of development and developers rely on computer-based technologies in order to transform models into running systems. Accordingly, the main goal of MDE research is to produce technologies that shield software developers from the complexities of the underlying implementation platform. An implementation platform may consist of networks of computers, middleware and libraries of utility functions, e.g. libraries of persistence, mathematical routines or the BOINC framework. The realisation of the MDE vision requires a handling of extensive interrelated technical problems that have been the focus of software engineering

research over recent years. It may seem that work on MDE centres on the development and use of models based on the Unified Modeling Language (UML). The UML standardisation effort has played a vital role in bringing together a community that focuses on the problem of raising the level of abstraction at which software is developed.

UML is not precise enough to support executable models or fully automated code generation directly. This can be resolved through the use of profiles, which can provide constraints and specialisations of the manifold semantic variation points in the general standard [11].

4.2 Overview of Model-driven Development

The Object Management Group (OMG) was formed as a standard organisation in order to reduce the complexity, lower costs, and accelerate the introduction of new software applications. One of the major initiatives through which the OMG is accomplishing this goal is by the promotion of the Model-driven Architecture (MDA) as an architectural framework for software development. In 2001 the OMG adopted the MDA as an approach for using models in software development. Its three primary goals are *portability*, *interoperability* and *reusability* through architecture and maintenance [25]. MDA is an approach where software systems are defined by using models. Based on these models, their construction is to some extent automatic [87]. A system can be modelled at different levels of abstraction or out of different perspectives. The syntax of every model is defined by a metamodel. It defines the syntax and notation of the modelling languages, a model plays the role of the source code, the generator performs the transformation from models to executable applications and replaces the compiler. While using MDA, it is possible to generate large amounts of source code and other artefacts automatically, e.g. deployment descriptors and make-up of files, based on relatively concise models. This approach enables systems to be modelled by the use of Platform Independent Models (PIM) and generates Platform Specific Models (PSM) through transformation rules or direct mapping relations [124].

4.2.1 Model-driven Architecture

One fundamental aspect of Model-driven Architecture (MDA) is its ability to address the complete development life-cycle, covering analysis and design, programming, testing and component assembly as well as deployment and maintenance. MDA itself is not a new OMG specification but rather an approach of software development which is set up by existing OMG specifications such as the UML and the Meta Object Facility (MOF) [25]. The main idea is to have unique mod-

els which are executable on any target platform, i.e. the developers have no need to implement any line of code or at least a minimum set of lines of code. In recent decades the abstraction of technologies increased continuously. At the beginning, only machine code was implemented, then the first high-level language Fortran [138] was established. The OMG defines some general tokens which are used for MDA [103, 124]:

Computation Independent Model (CIM) This is a view of a system from the computation independent viewpoint and does not show details of the system's structure. CIM contains all of the business rules defined in the core business model [25].

Platform Independent Model (PIM) This is a view of a system from the platform independent viewpoint and exhibits a specified degree of platform independence so as to be suitable for a number of different platforms with similar types. In short, this model contains only the data elements defined in the conceptual model, e.g. the structure of a system without information about software versions [25].

Platform Specific Model (PSM) A view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how this system uses a particular type of platform and software versions.

The MDA guide [103] states some possibilities of how these different models can be used, in particular, which transformations are established between models. Beyond the mentioned notions of CIM, PIM, and PSM, the two key concepts of MDA are *models* and *transformations* [25]. In particular, the MDA guide does not restrict the method of transformations: there are many ways in which such a transformation may be performed [124], e.g. PIM to PSM or PIM directly to a code implementation and therefore to an executable application. Fig. 4.1 is intended to give a diagrammatic overview. A model is prepared by using a PIM specified from a metamodel. Thus, a particular platform is chosen and its transformation is available or prepared. This transformation specification is in this regards a mapping between metamodels. The mapping guides the transformation of the PIM in order to produce the PSM. There are contexts in which a PIM can provide all the information needed for implementation; there is no need to add specialised mappings for transforming the PIM into a specific PSM or use data from additional profiles in order to be able to generate a code. This is the case when mature software components are used, where middleware provides a full set of services and where these components are reused, i.e. necessary architectural decisions are made once for a number of projects. These decisions are implemented in tools, development

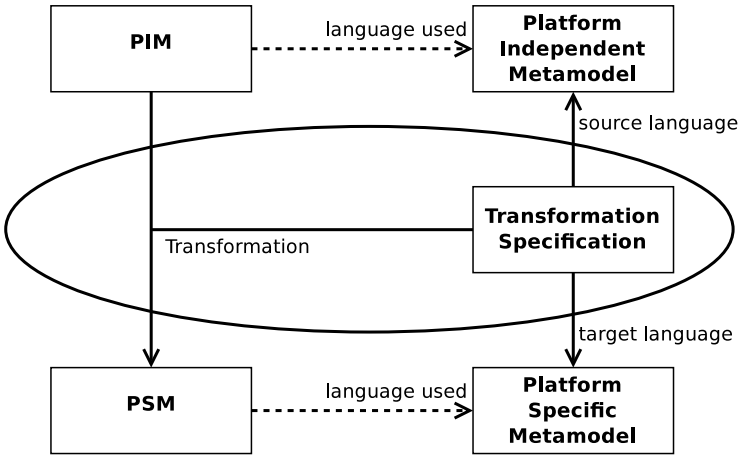

Figure 4.1: Model-driven Architecture approach of metamodel transformation between PIM and PSM [124].

processes, templates, program libraries and code generators. This makes the PIM *computationally complete*; that is, the PIM contains all the information necessary to produce a computer program code. As a consequence, a tool can interpret the model directly or transform the model immediately into a program code [124]. In essence, the foundations of MDA consist of three complementary ideas [2]:

Direct representation Shifts the focus of software development from the technology domain towards the ideas and concepts of the problem domain. The reduction of the semantic distance between the problem domain and the representation allows a more direct coupling of solutions to problems, leading to more accurate designs and increased productivity.

Automation Uses computer-based tools to mechanise those facets of software development that do not depend on human skills. One of the primary purposes of automation in MDA is to bridge the semantic gap between domain concepts and implementation technology by explicitly modelling both domain and technology choices in frameworks and then exploiting the knowledge which is then built into a particular application framework.

Open standards Standards have been one of the most effective boosters of progress throughout the history of technology. Industry standards do not only help to eliminate gratuitous diversity but they also encourage an ecosystem of vendors to produce tools for general purposes as well as all kinds of specialised niches, greatly increasing the attractiveness of the whole endeavour to users. Open source development ensures that standards are implemented consistently and therefore encourages the adoption of standards by vendors.

MDA tackles problems caused by system size through the increase of abstraction level from code-level to model-level [69]. The MDA Guide states that *the magic of software automation from models is truly just another level of compilation* [103, p.1-3]. MDA presents a developmental process based on automatic transformation of models into a code. Fig. 4.2 shows how MDA relates to other ways of using models (M) and code (C). The three first boxes from the left are variants of single-language programming. The code is the central part. If models are used then they are either disconnected or non-executable design models or are visualisations which are produced by reverse engineering techniques. Neither of these approaches use models as first-class artefacts. The three approaches on the right, *round-trip*, *Model-driven Architecture*, and *Model Interpretation*, all fit under the MDE umbrella. These approaches use models as first-class artefacts and they differ only in the priority in which they assign a code [69].

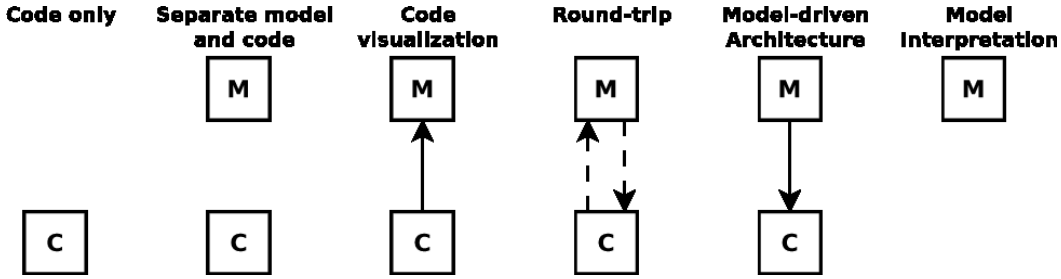

Figure 4.2: This diagram shows six different ways of using models (M) and code (C) in the context of MDE [69, 13].

4.2.2 Domain-specific (Modeling) Language

Domain-specific Modeling Languages (DSMLs) have been around for a long time. Graphical User Interface (GUI) builder systems allow developers to build interfaces by selecting and manipulating buttons and other interface widgets directly, rather than writing pages of repetitive code using a graphics library [2]. A DSL/DSML (hereafter only named "DSL") may either have a textual (i.e. DSL) or a graphical representation (i.e. DSML), i.e. both are concrete syntaxes. These concrete syntaxes are mapped to a language model, i.e. abstract syntax, which describes the modelling elements with which models in the respective domain can be created. In combination with an application library, any general-purpose language can act as a DSL. The library's API constitutes a domain-specific vocabulary of class, method, and function name that becomes available for any general-purpose language program using the library by using object creation and method invocation [19]. The establishment of a model-driven DSL usually means performing three steps [44, 64]:

1. A *language model* is defined to specify the *abstract syntax* of the DSL in a formal way. If a UML-like metamodel is used, a UML profile with stereotypes, tagged values, and constraints is used to model the extensions to the UML needed for defining the DSL formally.

2. For making the language usable, a suitable *concrete syntax* has to be defined. There can be a graphical representation that exposes the abstract model element for selected language elements. The concrete syntax represents the concepts of the abstract syntax. Usually there is a correspondence between concrete and abstract syntax elements. In the case of textual languages the syntax can be described by a grammar which describes both concrete and abstract syntax by specifying terminals, non-terminals and production rules. Finally some kind of development environment needs to be provided.

3. A generator translates the DSL into an executable representation, e.g. C/C++ code which can be compiled in a subsequent step. Therefore, the generator must map the concrete syntax to instances of the abstract syntax. These generate the code in the target programming language. In practice the generator must be able to define the *semantics* of the DSL. An informal description of the language may be given in a natural language by describing the domain itself. But the actual definition of these semantics is performed by implementing the generator back-end. Thus, the semantics of the DSL are defined by giving a translation *(translational semantics)* into some target language which already has some behaviour definition for its elements *(operational semantics)*.

In [19] the authors list some decision patters which can be consolidated when the use of DSLs is an open decision. For these instances, some established and available DSLs are introduced and it is established in which context they are supported. DSLs are used to abstract a complex architecture or hierarchy and narrow the scope of use in a specific direction. This provides a domain-specific notation for developers right from the start. Their importance should not be underestimated as they are directly related to the productivity improvement associated with the use of DSLs. In addition, it is not necessary that DSLs are full Turing machines [85] and they need not be executable. For instance, Microsoft Excel's macro language [198] is a DSL for spreadsheet application which adds programmability to Excel's fundamental interactive mode. It depends on the abstraction of how a DSL is defined and created. A DSL can be created as a textual or graphical representation, therefore three steps are adequate for the creation [64]:

1. A UML based language model is defined to represent the abstract syntax, which specifies how the various language elements of the workflow DSL

interact. Based on this model, UML stereotypes and constraints are defined to model those aspects of the workflow DSL that cannot be expressed in a native UML.

2. For the concrete syntax, in case the DSL is a graphical representation, visual symbols are defined for each of the language elements of the DSL. The abstract syntax model can be mapped into visual or textual elements.

3. The code generation for the target programming language is defined. Each concrete syntax element is mapped to an instance of the respective abstract syntax class, and it is checked that no constraint is violated.

4.3 The Unified Modeling Language Version 2

The Unified Modeling Language (UML) is a standard modelling language for visualising, specifying, and documenting software systems [124]. The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modelling business and similar processes. One of the primary goals of UML is to advance the state of the industry by enabling object visual modelling tool interoperability. However, in order to set up a meaningful exchange of model information between tools, agreement on semantics and notation is required [125]. UML is a visual language for specifying, constructing and documenting the artefacts of systems. It is a general purpose modelling language that can be used with all major object and component methods and can be applied to all application domains, e.g. health, finance, telecommunications [74], aerospace [7].

There are situations in which a language that is so general and of such a broad scope may not be appropriate for modelling applications of some specific domains [7]. This is the case, for instance, when the syntax or semantics of the UML elements cannot express specific concepts of particular systems or when we want to restrict or customise some of the UML elements which are usually too abundant and too general. OMG defines two possible approaches for defining DSLs. The first one is based on the definition of a new language using MOF which aims to provide a language description for metamodels. In this way, the syntax and semantics of the elements of the new language are defined in order to match the specific characteristics of the domain. The second alternative is based on the UML specialisation in which some of the language's elements are specialised; for imposing new restrictions on them while respecting the UML metamodel and leaving the original semantics of the UML elements unchanged, i.e. properties of the UML classes, associations, attributes etc. will remain the same; new constraints will simply be

added to their original definitions and relationships. In order to support the second alternative, UML provides a set of extension mechanisms for specialising its elements, allowing customised extensions of UML for particular application domains, i.e. stereotypes, tagged values, and constraints. These customisations are sets of UML extensions grouped into UML profiles. UML profiles not only can extend themselves but also any other MOF-defined language or even different UML profiles.

As stated in [2] some scientists have complained that UML is too complex to learn or use. It is complex — it is intended to provide a capability that is needed by all kinds of known and unknown frameworks that may be built on it in order to handle real-world problems. But every powerful tool is complex, such as English and Electrical Engineering. Thus, complex tools are needed to deal with the variability and complexity of the real world. However it is not necessary to know all about the English language, electrical engineering or UML in order to use them effectively. UML is structured into separate modules with minimal interaction among themselves. In addition, it includes state machines for describing discrete event-driven behaviour, activity graphs for describing computational flows, and constructs such as components, ports and connectors for assembling large systems from modular pieces.

4.3.1 Structure of UML Version 2

UML 2 is separated into three specifications:

- **Infrastructure Specification** The infrastructure specification provides the core functionalities of the language [126], i.e. Classes, Attributes, Associations, primitive data types and so forth. This part specifies the flexible metamodel library that is reused by the superstructure specification. It contains a single language unit that provides modelling for the kinds of class-based structures encountered in most popular object-oriented programming languages. As such, it ensures an entry-level modelling capability. More importantly, it represents a low-cost common denominator that can serve as a basis for interoperability between different categories of modelling tools [125].

- **Superstructure Specification** The superstructure specification takes account of UML's user level [125]. In particular, this specification defines the static and structural constructs (e.g. classes, components, nodes and artefacts) used in various structural diagrams, such as class diagrams, and deployment diagrams. Another part considers the behaviour constructs which describe the dynamic of systems (e.g. activities, interactions, state machines) used in various behavioural diagrams, such as activity diagrams, sequence diagrams, and state machine diagrams. Finally, this specification provides

information on how UML models can be customised by the use of UML profiles.

- **Object Constraint Language (OCL)** The OCL is a formal language used to describe expressions on UML models [127]. These expressions specify typical invariant conditions that must hold for the system being modelled or queries over objects which are described in a model. When the OCL expressions are evaluated, they do not have side effects, i.e. their evaluation cannot alter the state of the corresponding executing system. OCL expressions can be used to specify operations / actions that, when executed, do alter the state of the system. UML modellers can use OCL to specify application-specific constraints in their models. UML modellers can also use OCL to specify queries on the UML model which are completely programming language independent.

UML employs specialised keywords for describing how associations between metaclasses are used within the metamodel hierarchy [96]:

subsets Two kinds of subsets can be distinguished:

- **derived** If a property is a derived subset, then its value or values can be computed from the value of one or more other properties. That is to say that the values exist in some form somewhere else in the metamodel and that those values can be computed. Since the values for derived subsets are calculated, users cannot add such collections directly.
- **non-derived** Non-derived subsets also apply to properties but such properties contain values that cannot be calculated directly from existing features.

derived unions Derived unions indicate that a feature is the union of one or more collections or scalar.

redefines Redefinition is a way to narrow the scope of a property or to constrain it. It can be used to narrow the type of a property by referring to a more specific type. The redefinition of features which are lists of items replace the entire list. That is, any items contributed via inheritance will be disregarded and the redefined list will be recalculated. The name and visibility of a property are not required in order to match those of any property which it redefines.

Fig. 4.3 shows an excerpt of the UML specification [125, Fig. 7.9, 7.12]; attributes of the classes are not shown. Association specialisations and redefinitions are indicated by appropriate constraints situated in the proximity of the association ends to which they apply (highlighted on the right-hand side). Association ends can have

an optional specification, by use of a *role*, *visibility*, and *multiplicity* as shown on the top left-hand side. When one association end has no multiplicity, then the default is used, i.e. the minimum and maximum is one. The visibility can describe four visibility kinds: public *(+)*, private *(-)*, protected *(#)*, and package *(~)*. A role name can have a slash at the beginning, then it describes a derived association.

The structure shows how the UML Class is modelled. A Class can have superclasses, which are associated by the role *superClass*. In braces it is specified that this association redefines the role *general* of *Classifier*. *Classifier::general* specifies the general Classifiers for this Classifier, *Class::superClass* gives the superclasses of a class. Thus, the specialised association roles should be used. UML

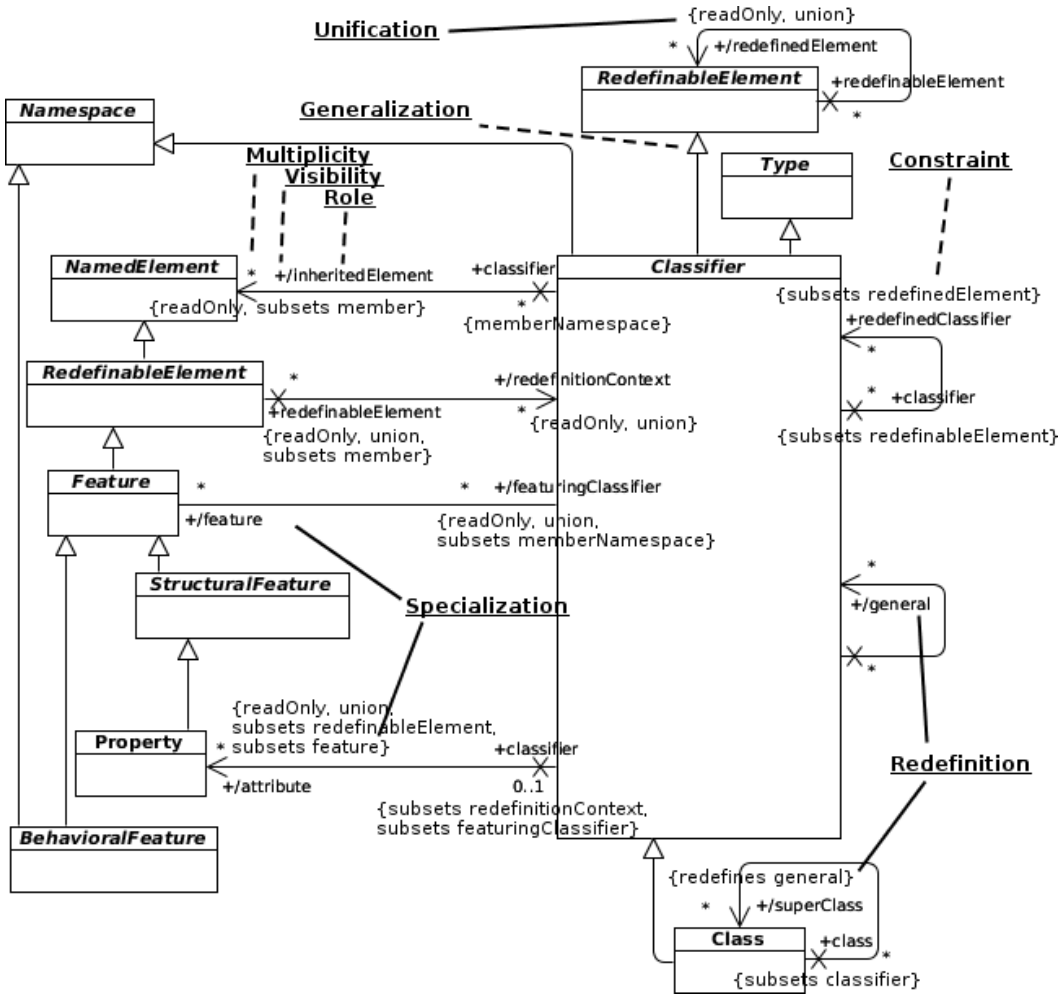

Figure 4.3: UML associations with different keywords: *subsets*, *unions*, and *redefines*.

is based on a class hierarchy. As a consequence, the types of metamodel element instances which are related to the level within the hierarchy can vary, i.e. better known as polymorphism [4]. In particular, *Classifier::feature* is an abstract class,

which can be used to add behaviour to a classifier. The abstract class *Feature* is the base class of a UML Connector to create composite structures. Accordingly, specialised Classifier instances can have an internal structure, due to the fact that UML Property instances can be UML Port instances [125, Fig. 9.4]. This is possible because the UML Kernel package elements are merged for composite structures. In a final step, the top area of Fig. 4.3 shows a union association. All specialised classes of *RedefinableElement* are unified, i.e. the highest specialisation is described by lower layered UML elements.

There are cases, where it is not possible to add elements to collections manually. The collections are filled automatically, this depends on the definition of the *subsets*. For instance if the collection *inheritedMember* of *Classifier (from Kernel, Dependencies, PowerTypes, Interfaces)* is readonly, members must be added to the *member* collection of *Namespace (from Kernel)*.

4.3.2 UML Diagrams

UML 2 has changed significantly, e.g. previous versions have no support to incorporate accurate model descriptions. In addition, the current UML version has more features and types of diagrams. The 2.4 version contains 14 definitions of diagrams; seven are used as structure and the others as behaviour diagrams [125].

UML Viewpoints

UML provides support for separating concerns by letting users model systems from four viewpoints [6]:

1. Models produced from the *static structural* viewpoints describe the system's structural aspects.

2. Developers use the *interaction* viewpoint to produce sequence and communication models that describe the interactions among a set of collaborating instances.

3. The *activity* viewpoint is used to create models that describe the flow of activities within a system.

4. The *state* viewpoint is used to create state machines that describe behaviour in terms of transitions among states.

These viewpoints can be used as a mix; it is not defined if one or more has to be used. It depends on the application whether viewpoints are appropriate for a situation or not.

Structure Diagrams

Class Diagram This diagram deals with the basic modelling concepts of UML, their particular classes and relationships [125, Chapter 7]

Component Diagram This diagram is used to define software systems with different grades of complexity. In particular, this diagram specifies a component as a modular unit with well-defined interfaces that are replaceable within their environment. One aspect of component-based development is the reuse of components according to prior construction. A component is considered as an autonomous unit within a system or subsystem. Ports and interfaces are used to hide the complexity of components and to connect components among each other [125, Chapter 8].

Composite Structure Diagram In this diagram components are interconnected, representing run-time instances which are collaborating over communications links to achieve some common objectives. Thus, communication links are created between ports of components and used to define a component which is independent of its environment. Elements of a system cooperate with each other and produce a specific behaviour within their environment. Collaboration will eventually be exhibited by a set of cooperating instances that communicate with each other by sending signals or invoking operations. This diagram is only used to model the relevant aspects of cooperation instances [125, Chapter 9].

Deployment Diagram This diagram can be used to define the execution architecture of systems that represent the assignment of software artefacts to nodes. They are connected through communication paths to create network systems of arbitrary complexity. In general, artefacts are physical entities and nodes represent hardware devices or software execution environments [125, Chapter 10].

Object Diagram The object diagram can visualise instances of classes which are called *objects*. It shows the current values of one or more selected objects. In particular, this diagram can be used to visualise a snapshot of a specific moment during runtime.

Package Diagram The package diagram defines the basic constructs related to packages and their contents [126, Section 10.4].

Profile Diagram The profile diagram is used to extend the metaclasses from existing metamodels for different purposes [125, Chapter 18]

Behaviour Diagrams

Activity Diagram Activity modelling emphasises the sequence and conditions for coordinating lower-level behaviours. These are commonly called control flow and object flow models. The actions which are coordinated by activity models can be initiated because either other actions finish executing or object and data become available or events occur external to the flow [125, Chapter 12]. Activities can contain different nodes which are used to describe different actions, e.g. the execution of system calls, in order to fork the flow to orthogonal flows or to decide which transition has to be used next. Transitions are created between activities to control the object or control flow. Both flows can be controlled by constraints, the so-called guards which allow the usage of selected transitions to change to the next activity or action.

Use Case Diagram The key concepts associated with use cases are actors and the subject. Use cases are a means for specifying required functions of a system. A subject is a system which considers the application of use cases. The users and any other systems that may interact with the subject are represented as actors. They model entities that are outside the system. The subject's behaviour is specified by one or more use cases, which are defined according to the needs of the actors [125, Chapter 16].

State Machine Diagram The state machine diagram defines a set of concepts that can be used for modelling discrete behaviour or in order to express the usage of protocols which are part of a system through finite state-transition systems. These two kinds of state machines are here referred to as *behavioural state machines* and *protocol state machines*, the second type is not used within this thesis. Behavioural state machines specify behaviour of various model elements, e.g. that can be used to model the behaviour of individual entities [125, Chapter 15].

Sequence Diagram A sequence diagram is a kind of interaction diagram. Within a sequence diagram, lifelines of several objects can be added and messages exchanged. The message exchange can be seen as a sequence and is considered as important for the understanding of the system situation. The data that the messages convey and the lifelines store may also be very important, but this diagram does not focus on the manipulation of data [125, Chapter 14]

Interaction Overview Diagram An interaction overview diagram defines communication and is a variant of the activity diagram which emphasises the high-level control flow. It illustrates an overview of flow control and states which node may contain interaction diagram [134].

Communication Diagram A communication diagram shows the interaction between the objects or roles that are associated with a lifeline and demonstrates the passing of messages. In earlier versions of UML, this diagram was called a collaboration diagram and had a different notation [134].

Timing Diagram A timing diagram conveys the change of a lifeline state or other condition and represents a classifier instance or classifier role over a specific time [134].

4.4 UML and Metamodeling

As a general rule a model describes the elements and types of elements that might exist in a system [7]. For example, in the definition of a "Person" as a class in a model, the use of instances of that class is possible, e.g. such as "John" or "Mary". Following that principle, granted that the system which has to be modelled is a UML system, then the elements which are involved are "Class", "Association", "Package", etc. The OMG defines a four-layered architecture that separates the different conceptual levels by making up a model: the instances, the model of a system, the modelling language, and the metamodel of that language. In OMG terminology these layers are called *M0*, *M1*, *M2*, and *M3*. Fig. 4.4 shows the four layers of the UML specification.

Layer M0: Run-time instances. The M0 layer models the running system and its elements are the actual instances that exist in the system. These instances are, for example, the person "John".

Layer M1: The model of the system. The elements of the M1 layer are *models*. An example would be a UML model of a software system. The M1 layer defines the *class* "Person" and additional *attributes*, e.g. a name for the persons. There is a relationship between M0 and M1 layers. The elements of the M1 layer are *classifications* of elements of the M0 layer. Likewise, each element at the M0 layer is always an *instance* of an element at the M1 layer.

Layer M2: The model of the model (the *metamodel*). The elements of the layer M2 are the modelling languages. Layer M2 defines the concepts that are used to model an element of layer M1. In the case of UML, layer M2 defines "Class", "Attribute", "Association", etc. Just as there is a close relationship between layers M0 and M1 so there is a close relationship between M1 and M2 layers. Every element at M1 is an *instance* of a M2 element, and every element at M2 *categorises* M1 elements. The model that resides at the M2 layer is called a metamodel.

Layer M3: The model of M2 (the *meta-metamodel*). Finally, layer M3 defines the concepts that can be used to define modelling languages. Thus, the concept of the UML Class (that belongs to M2) can be considered as an instance of the corresponding element of M3, which precisely defines what a Class is and its relationships with the rest of the UML concepts, e.g. a UML Class is a classifier, and therefore has an associated behaviour, and a set of attributes and methods, etc.

The modelling language defined for describing the M3 elements is called MOF (Meta-Object Facility, see Section 4.2.1). OMG's specification describes MOF as the foundation of OMG's industry-standard environment where models can be exported from one application, imported into another, transported across a network, stored in a repository and then retrieved, rendered into different formats (including XMI, OMG's XML-based standard format for model transmission and storage), transformed, and used to generate application code [129]. UML is described by its metamodel. If a new object-based visual language other than UML has to be defined, MOF can be used to describe its metamodel.

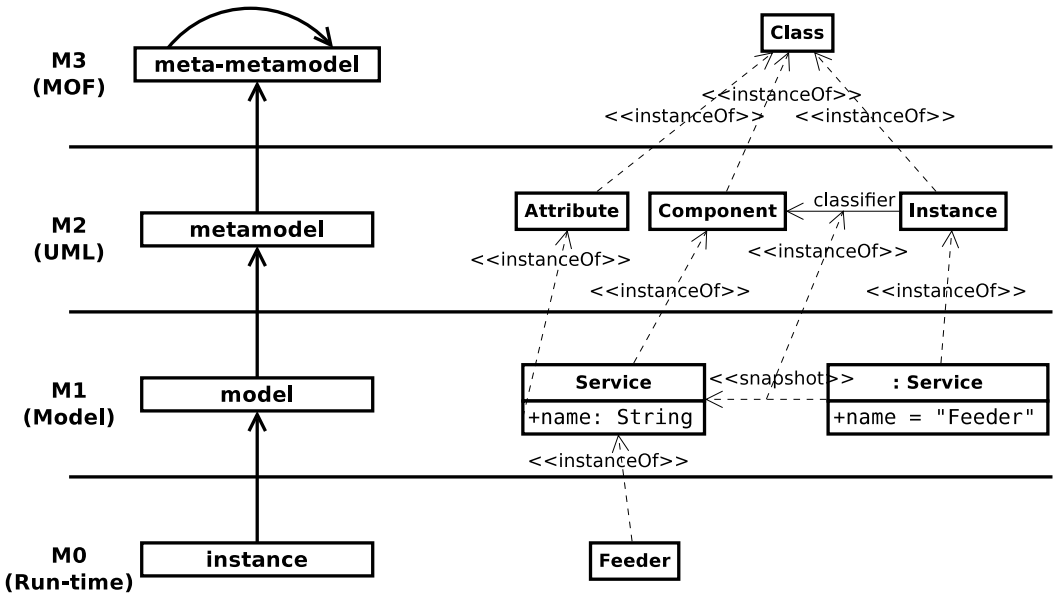

Figure 4.4: Example of UML's four-layer metamodel hierarchy [125].

4.5 UML Profile Mechanism

A UML profile can be created on different levels of complexity. The type of customisation that is required depends often on the nature of the domain and the fact of

how the extended model is going to be used. If one wants to make simple customisations by adding new properties to existing UML metaclasses, then a lightweight extension is the best way. However, if one wants to extend the behaviour of UML, add restrictions on certain collections or take advantages of the more-complex features of UML such as redefinition, then a heavyweight extension works best. Four different extension mechanisms can be used [96, 6]; each one has its pros and cons:

featherweight This mechanism adds keywords, UML itself defines a set of predefined keywords [125, Appendix B].

lightweight This mechanism should be used for a first modelling try, which involves the creation of a new UML profile. A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform domain.

middleweight The middleweight extension increases UML through specialisation of UML metamodels.

heavyweight Heavyweight extensions involve the reuse of copy and merge instead of specialising the types from a reference metamodel as with middleweight extensions.

Bruck and Hussey [96] created a detailed list of comparisons for the different extension types, e.g. ability to evolve or restrict multiplicities. UML can be customised by using a set of extension mechanisms that UML itself provided [7]. More precisely, the profiles package included in UML 2 defines a set of UML artefacts that allows the specification of a MOF model in order to deal with the specific concepts and notation required in particular application domains, e.g. real-time or business process modelling. A UML profile can extend either a metamodel or another profile and is defined in terms of three basic mechanisms: *stereotype*, *constraints*, and *tagged values*. Any metaclass of UML can be stereotyped. In [76] the authors define stereotypes as:

> A stereotype in a modelling language is a well-formed mechanism for expressing user-definable extensions, refinements or redefinitions of elements of the language (directly) modifying the metamodel of the language.

Fig. 4.5 shows how the stereotype mechanism can be classified in four categories [76]:

Decorative Stereotypes This classification allows modifying the *concrete syntax* of a language element. It does not introduce any essential additional information or new concepts into the base language. The represented model and the essence of the language that expresses the model remain unchanged.

Descriptive Stereotypes This classification allows modifying the *abstract syntax* of a language element and defines the pragmatics of the newly introduced element.

Restrictive Stereotypes This classification allows modifying the *semantic* of the newly introduced element. A restrictive stereotype does not change the semantic of the base language — it only extends it. These kinds of stereotypes are used to add missing features to some elements of a language.

Redefining Stereotypes This kind of stereotype *redefines* a language element, changing its original semantics. Using this stereotype, deep and radical changes can be imposed to a language. New language concepts can be introduced.

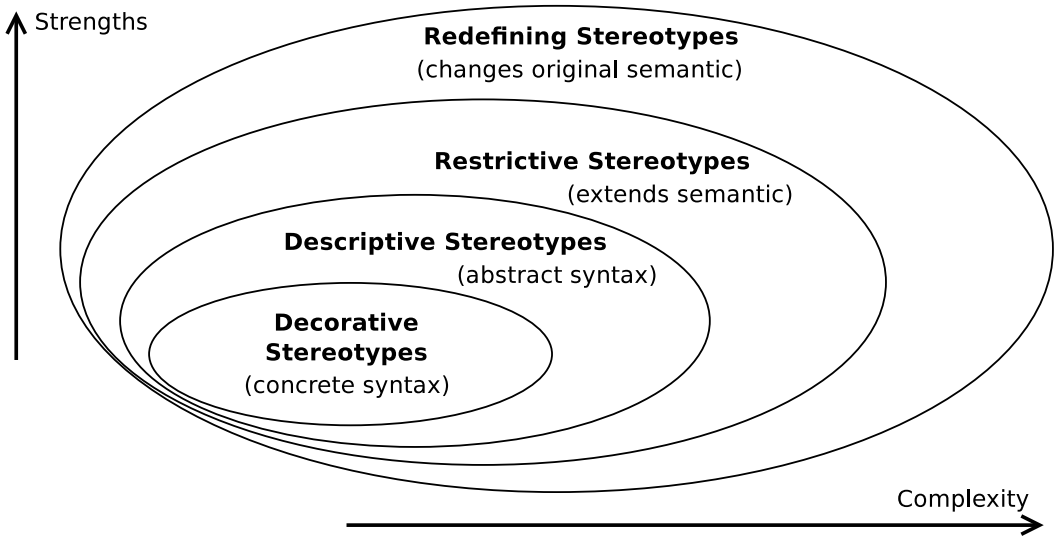

Figure 4.5: Classification of stereotypes according to their expressive strength and level of complexity [76].

The notation for an extension is an arrow pointing from a stereotype to the extended metaclass where the arrowhead is shown as a solid triangle. Fig. 4.6 shows how this can be performed visually. In this example three stereotypes are defined; one extends the metaclass Package and two extend the metaclass Component. *Service* is written in italics and visualises an abstract stereotype which has to be specialised; this is performed by the stereotype *Database*. A stereotype can have additional tagged values, i.e. Host has a tagged value *hostname* of type *String* and *Service* has a tagged value *serviceType* of the (not shown) type *ServiceType*. Tagged values are additional meta-attributes that are attached to a metaclass of the metamodel extended by a UML profile. Tagged values are specified as attributes of the class that defines the stereotype [7]. Furthermore, constraints can be associated to stereo-

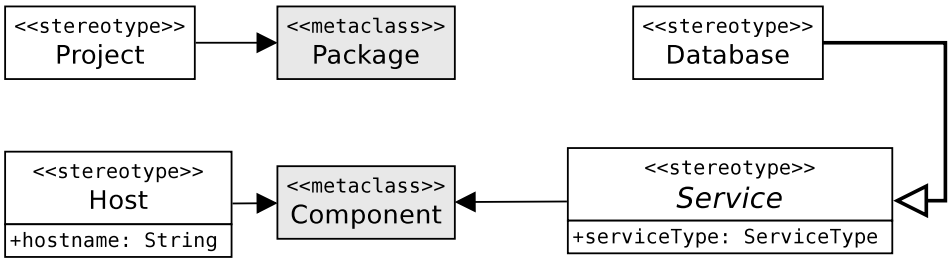

Figure 4.6: Example how UML metaclasses are extended by UML extensions.

types, imposing restrictions on the corresponding metamodel elements. In this way a designer can define the properties of a "well-formed" model. For instance, the *serviceType* can be restricted for the specialisation, i.e. the type of the service must be *Database* or something with a similar meaning. Constraints can be expressed in any language, including natural stated constraints or the formal Object Constraint Language (OCL) as described in Section 4.6 [7].

4.5.1 Definition of a UML Profile

Some guidelines exist [7, 60] to support a UML modeller defining UML profiles. The authors propose the following steps and necessary key elements:

1. A modeller has to define the set of *fundamental language constructs* that will comprise the platform or system, and the *relationships* between them, which can be expressed in terms of a metamodel. If such a metamodel is not available, it can be defined by UML, i.e. classes, hierarchy relationships, associations, etc.

2. Once a domain metamodel is defined, it can be used to define a UML profile. Accordingly, stereotypes are created for each relevant element of the metamodel profile. In order to clarify the connections between the metamodel and the profile, the stereotypes will be named after the corresponding elements of the metamodel. In fact, any element that we might need to define the metamodel can be tagged later with a stereotype.

3. Only the UML metaclasses which are extended by a stereotype are represented within a new UML profile; other metaclasses can still be used. Attributes that appear in the metamodel should be defined as tagged values and should include the corresponding types and initial values.

4. A set of *constraints* that govern how the metamodel elements can be combined to produce valid models can be created. The combination of connections and constraints constitutes the *well-formed* rules of a metamodel. Together with the set of language constructs they represent the *abstract syntax*

of the metamodel. Generally, the abstract syntax is expressed by using the OMG's MOF (Meta Object Facility) [124, 129] language. Here, the MOF-based UML is meant.

5. The *concrete syntax* or notation of the UML profile has to be defined. This is the textual and/or graphical expression of the abstract syntax. Note that it does not have to be derived from or reminiscent of the UML notation. In addition, the *semantic* or meaning of the UML profile has to be specified. This is a specification of the mapping between the elements of the abstract syntax and the corresponding domain entities that the syntactical elements represent.

Once a UML profile is defined, it can be applied for a specific application, i.e. the UML dependency relationship extended by «apply» is used. Furthermore, there is some research on the automatic generation of a UML profile by only comparing the definition of the domain model and the UML elements [8].

4.6 The Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a language adopted by the OMG for expressing constraints and properties of model elements. Examples of constraints include pre- and post-conditions of operations, invariants, derivation rules for attributes and associations, the body of query operations, etc. The word "Constraint" in the name of the language comes from its first version, where only constraints could be expressed. The contexts of constraints are the UML elements that are customised for a particular model. Since OCL 2.0, OCL has evolved to a more expressive query language whose expressiveness is similar to that of Structured Query Language (SQL) [116, 7]. The OCL can be distinguished in three basic complexity levels [77, 26]:

Intra-object Integrity Constraints Constraints restricting the value of the attributes of a single object.

Inter-object Integrity Constraints Constraints restricting the relationships between an object and other objects, instances of different classes. Within this category, it is worthwhile to distinguish the subcategory of Integrity Constraints containing aggregated operations, e.g. *sum*, *count*, *size*, etc.

Class-level Integrity Constraints Constraints restricting a set of objects of the same class, i.e. this is supported by OCL's *allInstances* operator.

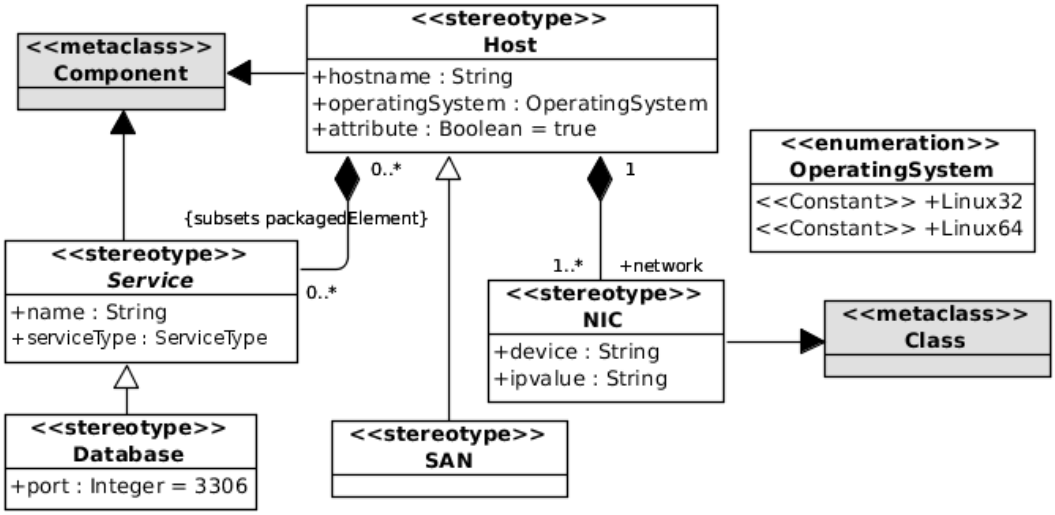

Figure 4.7: Part of the UML4BOINC profile to explain OCL statements.

OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side effects. When an OCL expression is evaluated, it simply returns a value and it cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change, e.g. in a post-condition. OCL is not a programming language; hence, it is not possible to implement program logic or flow control in OCL. OCL is a typed language so that each OCL expression has a type [127]. The OCL can be added to any UML element. As a consequence, OCL statements are related to a specific context. The context can be formed in different ways, i.e. as an invariant or pre- or post conditions. OCL does not only describe conditions but also operations, initialisation of values and derived values.

Fig. 4.7 shows an excerpt of the UML4BOINC profile which is presented in Chapter 6. This model is used to describe the OCL types and operations of the next section driven by an example. The UML4BOINC profile is used for specifying a Host with additional Services which are again installed on these Hosts; they can have one or more Network Interface Cards (NICs) to enable communication between Hosts. The UML metaclass Class is extended by «NIC». The UML metaclass Component is extended by «Service» and «Host». «Host» associates «NIC» and «Service»; «Service» is specialised by «Database» and «Host» is specialised by «SAN». Through this visual representation, one can create models based on this metamodel but the instance of this metamodel cannot be used to create an implementation, i.e. the metamodel does not define which operating system has to be used or how the hostname has to be set. An OCL expression can be an invariant of the restricted type and the expression must be true for all instances of that type

at any time. The OCL expression can be part of a precondition or postcondition. Additional constructs are possible, e.g. the body of operations, initial or derived expressions. An example of some possible constructs is shown in Listing 4.1. The UML metaclass X has an invariant *(inv)* which states that only one instance is allowed. *X* has an operation named *operationName* which checks if the attribute is zero *(precondition)*, sets the counter to one *(body)* and validates if the counter is one *(postcondition)*; thus *body* returns one.

```
context X inv: allInstances()->size() = 1
context X::operationName() : Integer
pre:  self.counter = 0
post: self.counter > 0
body: self.counter = 1
```

Listing 4.1: An example of invariants, preconditions, postconditions and body of operations.

4.6.1 OCL Types

Fig. 4.8 shows the type hierarchy of OCL which has *Classifier (from Kernel, Dependencies, PowerTypes, Interfaces)* [127]. The OCL specification defines a number of types which are similar to other programming languages; among these are primitive types: *Integer*, *Real*, *String*, and *Boolean*. Secondly, OCL defines different *CollectionType*s including a list of particular type elements associated with *elementType*. In addition, the *elementType* covers a collection of different types:

BagType An unordered collection, duplicates are allowed.

OrderedSetType An ordered collection, duplicates are not allowed.

SequenceType An ordered collection, duplicates are allowed.

SetType An unordered collection, duplicates are not allowed.

The types stated above are related to the models which are allowed to be applied by users, i.e. *CollectionType* is the type of *Collection*. Collection is an abstract type with the concrete collection types as subtypes. OCL distinguishes between three different collection types: *Set*, *Sequence*, and *Bag*. A Set is the mathematical set; it does not contain duplicate elements. A Bag is like a set which may contain duplicates (a *multiset* in some notations), i.e. the same element may be twice or more in a bag. A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no defined order. OCL is a 1-based index language whereas other languages like C/C++ are zero-based, i.e. the element of an OCL collection can be accessed by 1 as first index. *TupleType* combines different types into a single aggregate type, i.e. informally known as record type or `struct`. Listing 4.2 shows examples of all available collection types.

```
-- unordered, duplicates allowed
Bag          {1, 2, 3, 2, 1}
-- ordered, no duplicates
OrderedSet {1, 2, 3, 5, 7, 11, 13, 17 }
-- ordered, duplicates allowed
Sequence    {1, 2, 3, 5, 7, 11, 13, 17 }
-- unordered, no duplicates
Set          {2, 4, 1, 5, 7, 13, 11, 17 }
-- Tuple of a two-dimensional coordinate, i.e. x=5 and y=0
Tuple        {x: Integer = 0, y: Integer = 0}.x = 5
```

Listing 4.2: Examples of OCL Collections and a OCL Tuple.

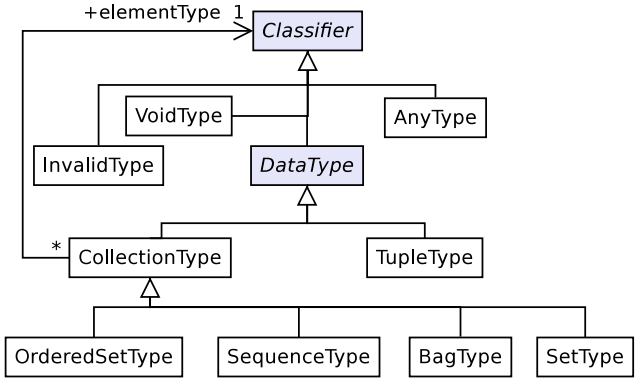

Figure 4.8: Excerpt of the OCL datatypes [127].

4.6.2 OCL Operations for Primitives

Attributes and operations of classes can be accessed directly with OCL; these kinds of constraints are categorised as **intra-object integrity constraints**. The keyword *self* is an optional access operator which is used to access the context available attributes and operations. Listing 4.3 shows how the tag-value *port* is checked for a valid value, i.e. a port must be in the range between 1 and $2^{16} - 1$; *self* can be removed in both compositions. The results can be set in two Boolean values and OCL provides the standard operators >, <, >= and <= for this.

```
-- <<Database>>
context Component inv: self.port >= 1 and self.port <= 65535
```

Listing 4.3: OCL validation of the port range.

In addition, OCL specifies standard operations, e.g. `min` to query the minimum value of two values. Some operations depend upon the type that is used, i.e. an integer cannot be rounded and a real value can contain a floating-point value and therefore can be rounded. Finally, if a «Host» instance has no operating system, *#Linux32* is used as default as shown in Listing 4.4.

```
-- <<Host>>
context Component post:
if operatingSystem.oclIsUndefined() then
  operatingSystem = #Linux32
endif
```

Listing 4.4: Postcondition for the operating system of «Host».

4.6.3 OCL Operations for Collections

This section describes OCL statements which are categorised as **inter-object integrity constraints**. The single navigation of an association results in a Set, combined navigations in a Bag and navigation over associations adorned with *{ordered}* results in an OrderedSet [127]. Listing 4.5 shows how the use of «Database» can be restricted, i.e. only one instance is allowed to be installed on a host. Furthermore, it conveys how associations can be applied when an association role is not named and when the class name with a lowercase first character is used. In Fig. 4.7, the association from «Host» to «NIC» can be used with two equivalent calls: *self.nIC* or *self.network*.

```
-- <<Database>>
context Component inv: host->size() = 1
```

Listing 4.5: OCL invariant that «Database» must have a «Host».

Listing 4.6 shows how named association roles are used. In this example one «Host» can only have «NIC» instances which are unique, i.e. the name of a device and all devices for a internet-protocol value are unique.

```
-- <<Host>>
context Component inv:
  self.network->forAll(n1, n2 | n1.device  <> n2.device implies
                                n1.ipvalue <> n2.ipvalue)
```

Listing 4.6: OCL invariant that a «NIC» must vary on a «Host».

The Listing 4.7 specifies that a «SAN» cannot have a «Service». Thus, the service association is checked for emptiness.

```
-- <<SAN>>
context Component inv: service->empty()
```

Listing 4.7: OCL invariant that «SAN» has no services.

OCL specifies several operations for the different collections. It is possible to cast between different collection types. For this, each collection type has four operations: *asBag()*, *asSequence()*, *asSet()*, and *asOrderedSet()*. Each of them returns the named collection, i.e. *asBag()* returns a Bag.

Finally, OCL statements can be used to restrict the use of instances itself. Listing 4.8 defines that all created hosts must have a unique hostname. This kind of OCL statement is categorized as **class-level integrity constraints**.

```
— <<Host>>
context Component inv: allInstances()->unique(hostname)
```

Listing 4.8: OCL invariant that all hostnames must be unique.

4.7 Summary

This chapter presented the Model-driven Engineering (MDE) in software development. In particular, it has been demonstrated how the Model-driven Development (MDD) is supported by different approaches. OMG's Model-driven Architecture (MDA) allows the creation of different metamodels which are able to be transformed into metamodels themselves. Thus, the Unified Modeling Language (UML) can be used as a Domain-specific Language (DSL) to allow developers to create specific viewpoints of the same system environment. Furthermore, it has been explained how UML and its semantics work and how the formal Object Constraint Language (OCL) allows the restriction of the use of UML. The chapter ends with an illustration of UML layers and its metamodel concepts and a final description of extension mechanisms which are based on stereotypes, tag values and constraints.

CHAPTER 5

Berkeley Open Infrastructure for Network Computing

BOINC (Berkeley Open Infrastructure for Network Computing) is a software system that makes it easy for scientists to create and operate public-resource computing projects. It supports diverse applications, including those with large storage or communication requirements. PC owners can participate in multiple BOINC projects, and can specify how their resources are allocated among these projects.

DAVID P. ANDERSON

5.1 Introduction

Berkeley Open Infrastructure for Network Computing (BOINC) is a Grid Computing (GC) middleware framework which allows the creation of High Performance Computing (HPC) installations by means of Public Resource Computing (PRC). PRC, also known as "Global Computing" or "Peer-to-peer computing" (P2P), uses these resources to do scientific supercomputing. Despite the fact that BOINC supports the creation of PRC systems, it works slightly differently from GC and P2P systems [28] (Section 5.1.1). BOINC enables the solution of large scale and complex computational problems. It supports diverse applications, including those with large storage or communication requirements [31]. The target platforms or devices are not limited. BOINC-based applications can be executed on various target devices with different software environments, e.g. Windows, Linux, Mac, and mobile devices based on Android or iOS [33, 190, 157]. Since 1999, SETI@home [15, 1], a basis of BOINC has attracted millions of participants worldwide. In addition it encourages public awareness of current scientific research, catalyses global communities which are centered on scientific interests and gives the public a measure of control over the directions of scientific progress.

5.1.1 BOINC in Constrast to Grid Computing

BOINC and GC share the goal of better utilising existing computing resources. However, there are profound differences between these two paradigms [31]:

- GC involves organisationally-owned resources: supercomputers, clusters, and PCs owned by universities, research labs, and companies. These resources are managed centrally, mostly powered and connected full-time with high-bandwidth network links. There is a symmetric relationship between organisations: each one can either provide or use resources.

- In contrast, BOINC involves an asymmetric relationship between projects and participants [29]. BOINC projects are typically small academic research groups with limited computer expertise and manpower. Most participants are individuals who own Windows, Macintosh, or Linux PCs that are connected to the Internet by telephone or cable modems or DSL (Digital subscriber line) and often behind network-address translators (NATs) or firewalls. The computers are turned off or disconnected from the Internet frequently. The participants are no computer experts, only participate in a project when interested and thus receive "incentives" such as credit and screensaver graphics. BOINC projects have no control over participants and cannot prevent malicious behaviour.

Accordingly, there are different requirements for middleware for BOINC than for GC, e.g. BOINC's features such as redundancy computing, cheat-resistant accounting and support for user-configurable application graphics are not necessary in a Grid-system. Conversely, GC has many requirements that BOINC does not. GC architectures must accommodate many existing commercial and research-oriented academic systems and must provide a general mechanism for resource discovery and access. In fact, it must address all the issues of dynamic heterogeneous distributed systems [31]. Fig. 5.1 shows a comparison of BOINC and GC by Anderson [28]. On the left-hand side, the figure illustrates the approach of BOINC, whereas the right-hand side shows the general GC. Consequently, BOINC clients are *loosely coupled* and GC systems are *coherent*.

5.1.2 Potential of BOINC

In [29] a study on BOINC and SETI@home [1, 15] is presented. From the participating hosts, 25% had 2 or more CPUs and the average memory was 819MB RAM (Random-access Memory) and 2.03GB swap. The average network throughput was 289Kps. The BOINC client measures the amount of total free space as 12 Petabytes. When the article was written, there were 1 million participants involved, i.e. a few hundred thousand for each project. The average host lifetime

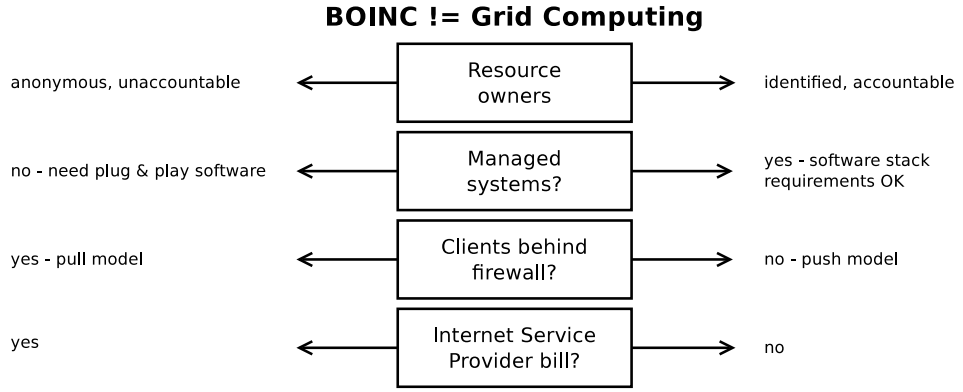

Figure 5.1: Comparison of BOINC in contrast to Grid Computing [28].

is 91 days. The average active-fraction (fraction of the time that BOINC is running when BOINC is allowed to compute and communicate) was 0.84. SETI@home had a potential processing rate of 149.8 TeraFLOPS. The host pool provides processing at a sustained rate of 95.9 TFLOPS and it also has the potential to provide 7.74 Petabytes of storage, with an access rate of 5.27 Terabytes per second [67].

5.1.3 Goals

The goal of BOINC is to collect as many spare cycles as possible. Currently the largest number of computers available is those desktop machines (PCs) which are mostly owned and used by individual persons at home. Thus, the number one target of BOINC is the large set of home computers [12]. BOINC's general goals are [31]:

Reduce Barriers Anderson states that a BOINC project can be created within a week of initial work and a hour of maintenance per week when the developer or research scientist has moderate computer skills. In [70] it has been stated that the initial work can take more than half a year with time variation in maintenance which depends on the current state of a BOINC project, i.e. if some hardware has to be replaced or software dependencies have to be solved.

Share Resources & Autonomy BOINC-based projects are autonomous and are not authorised or registered centrally. Each project operates independently with its own servers. Participants of a single BOINC project can participate in multiple projects and are able to assign a "resource share" individually. Thus, they determine how to divide limited resources among projects because computing power and disc space are limited.

Diverse Applications BOINC accommodates a wide range of different applications; it provides a flexible and scalable mechanism for distributing data

and its scheduling algorithms match requirements intelligently with resources. BOINC provides Application Programming Interfaces (APIs) for C/C++ [118] and Fortran [138]. An application can consist of several files, e.g. multiple programs such as a scientific application, a screensaver and SQLite[1] databases. New versions of applications can be deployed without participants being involved.

Reward Participants BOINC projects provide an award system in order to attract and retain participants; it consists of credits: a numeric measure of how much computation the particpants have contributed.

5.1.4 Features

BOINC software includes server-side components such as a Scheduler, programs which manage the distribution of task and data collections [30] and also a web-based interface for clients and project administrators [71]. In addition, BOINC provides the following features which make it a widely used PRC middleware framework [31, 152]:

Redundant Computing BOINC provides support for redundant computing, a mechanism for identifying and rejecting erroneous results. A project can specify that N results should be created for each workunit. Once $M <= N$ have been distributed and completed, an application-specific Validator is called to compare the results and possibly select a canonical result. If no consensus is found, or if results fail, BOINC creates new results for the workunit and continues this process until either a maximum result count or a timeout limit is reached. Section 5.3.9 describes this in more detail.

Homogeneous Redundancy BOINC provides a feature called homogeneous redundancy for scientific applications. When this feature is enabled, the BOINC Scheduler sends results for a given workunit only to hosts with the same operation system name and CPU vendor. BOINC can distinguish four operating systems (Linux, Windows, Darwin, FreeBSD) and 80 processor types (e.g. Intel Xeon, Celeron, Pentiums, and others) [91, p.50]. In this case, strict equality can be used to compare results. Finally, BOINC is compatible with other schemes for ensuring result correctness [48].

Security The BOINC middleware framework protects against several types of attacks, i.e. it uses digital signatures based on public-key encryption to protect against the distribution of viruses.

[1] http://www.sqlite.org [accessed 11^{th} April 2012]

Multiple Servers and Fault Tolerance BOINC projects can have separate scheduling and data servers with multiple servers of each type (Section 5.3). Clients try alternate servers automatically; if all servers are down, clients implement exponential back-off [123] to avoid flooding the servers when they come back up.

System Monitoring Tools BOINC includes a web-based system for displaying time-varying measurements, e.g. CPU load, network traffic, or database table sizes.

Support for Diverse Applications The BOINC client is available for most common platforms, e.g. Mac OS X, Windows, Linux and other Unix systems. It can use multi-core CPUs and GPUs.

Tunes Work Distribution The BOINC client fetches enough work to keep its host busy for a participant-specifiable amount of time. This can be used to decrease the frequency of connections or to allow the participant's computer to keep working during project downtime.

Locality Scheduling The BOINC core client implements a local scheduling policy, i.e. maximise resource usage, satisfy result deadlines, respect the participant's resource share allocation among project preferences, and maintain a minimal "variety" among projects. It is used to decide when work has to be fetched from projects and what tasks have to be executed at a given point.

Open and Extensible Architecture Existing applications in common languages (e.g. C/C++, Fortran) can run as BOINC applications with little or no modification. An application can consist of several files, e.g. multiple programs and a coordinating script. New versions of applications can be deployed without participant involvement, i.e. when the client fetches new workunits, it also retrieves a new application version.

Asynchronous-messages BOINC offers a so-called trickle-message mechanism, providing bidirectional, asynchronous, reliable, ordered messages, piggybacked onto the regular server-client RPC (Remote Procedure Call) traffic. This can be used to convey credit or to report a summary of a computational state [67]. Users like to see an accrual of their credits on a daily basis which can actually cause problems for projects with long workunits. The trickle-message mechanism was put into the climateprediction.net (CPD) software [93]. Basically, this is a ping from the client to the central BOINC project server in order to inform the project which users are active, how much time they have spent running CPD and where they are in the processing of

the computation. The trickle-messages return some diagnostic data at certain intervals, i.e. through these data scientists are able to see which runs are unstable or will be finished soon [35].

5.2 Architecture

BOINC uses a server-client architecture (so-called master-slave model), i.e. the server controls the execution of a BOINC project. In addition, it organises the workunit handling, validates and assimilates results and provides the download/upload facilities which are necessary for a client's queries. Finally, the server is in charge of the following aspects [49]:

- *Hosting the scientific experiment:* A project is composed of a scientific application and some input files. The executable is classified according to the target platform (e.g. Microsoft Windows, GNU/Linux, Mac OS X) and architecture (e.g. x86 32–64 bits and SPARC [79]).

- *Workunits creation and distribution:* A workunit describes how the experiment must be run by the clients, i.e. the name of the scientific application, the input/output files and optional command line arguments. Input/output files contain a number of parameters in ASCII or binary format that are optionally needed by the scientific application to perform a specific task.

The BOINC client executes the application, i.e. performs the calculations and sends its results back to the server which assembles these into a "global" result or stores them at specific places, e.g. in a specific directory or database [58]. The client initiates the communication by sending a request over HTTP to the server. The request is a XML (Extensible Markup Language) [117] dialect which describes the information about the hardware and the availability of the machine on which it is running. After establishing the communication, the server sends a reply consisting of a XML script that describes the new tasks and files related to the application. Then the client can download all the necessary binaries and files for running the application and — after completing — uploads the results [71]. A distinguishing feature among GC systems is the task distribution mechanism. Basically, there are two concepts:

pull-model: The clients ask for work from the server.

push-model: The server pushes tasks to the clients.

BOINC is a clear example of the pull-model. Whenever a client is free, it asks for workunits from the server [12].

The set-up of a BOINC project requires the installation of a GNU/Linux server with additional software components, e.g. Apache webserver [187], MySQL [200] and PHP [188]. Secondly, a scientific application has to be implemented in C/C++ [141] or Fortran [138]. Thus, three possible scenarios can be employed to support BOINC:

Scenario 1: *Porting.* This is the most common method. A researcher has to adapt his application source code in order to support BOINC. The current code-base can be modified although the researcher has to rewrite the whole code to fit the other use-cases.

Scenario 2: *Wrapping.* The BOINC team provides a tool called *wrapper* which enables running legacy-applications or applications where the code-base is not accessible; there are also other wrapping approaches [18, 57, 73]. Wrapping can only be used when the application is linked statically, if not, all external libraries have to be provided for the execution.

Scenario 3: *Virtualization.* VirtualBox [202] or VMware [209] can be used to make uniform the client's operating system and architecture by employing a virtualisation layer. The virtual machine is provided by a BOINC project and can be controlled by a wrapper.

Section 5.4.2 describes these three scenarios in more detail.

5.3 Server Components

The server-side of the BOINC project uses a number of separate daemons which share a common MySQL[2] database. Fig. 5.2 shows daemons which are basically used on the BOINC's server-side. A minimum of five daemons has to be installed and started to enable BOINC's workflow, i.e. workunit creation, distribution, performing, validation and assimilation. Each of these daemons can be distributed among several machines; as long as they share the same BOINC project installation as in Fig. 5.3 they can be located in different cities as has been organised in the IberCivis [154] project [12]. Fig. 5.3 shows how the Network Filesystem (NFS) [108] can be used to share a BOINC project installation. Host "01" is defined as a *main* host, i.e. this host performs the main installation of a BOINC project with all relevant configurations and applications. The main host must share the BOINC project installation directory. In this example, three hosts are attached to each other and they are importing the shared directory. Only one configuration file has to be modified and each daemon and task can be executed on several hosts.

[2]http://www.mysql.de [accessed 11^{st} April 2012]

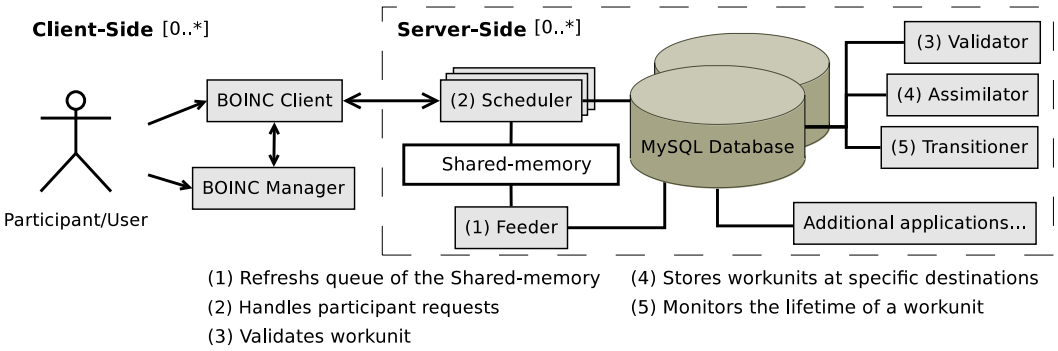

Figure 5.2: BOINC's server-client architecture [31, 30]: the left shows the client-side, the right shows the server-side; One BOINC project can have zero or unlimited attached participants and each participant can be attached to zero or unlimited BOINC projects.

BOINC's management script parses the configuration on each host and decides whether a BOINC daemon or task has to be started on a specific host or not. This allows the setting up of load-balancing for all BOINC daemons and tasks.

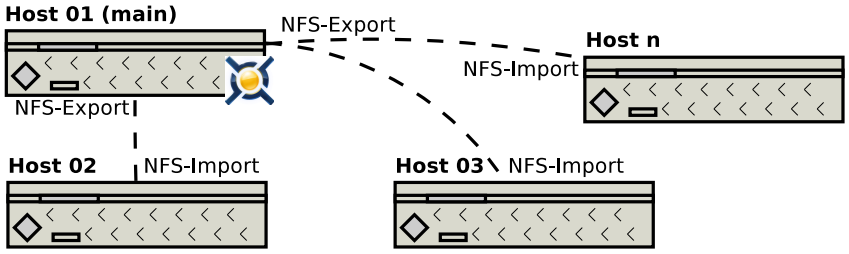

Figure 5.3: The 'main' host exports the BOINC project installation directory to external hosts through the use of a Network File System (NFS) [108], e.g. to enable load-balancing.

5.3.1 Database

The data back-end of a BOINC project is a MySQL database. In this database all the information about users, scientific applications and their different versions, workunits and the current state of processing is stored. Within this thesis we call this database the **master database**. Additional databases can be configured and added, e.g. to use them for storing the results of computations. This kind of databases will be called **science databases**. Both — master and science — can have an optional replication, this will be called a **replica database**.

master database: BOINC's main database used for the back-end.

science database: Optional database to store computation results.

replica database: Optional database to replicate above database kinds.

5.3.2 Scheduler

The Scheduler handles requests from BOINC clients. Each request includes a description of the host, a list of completed instances and a request for additional work which is expressed in terms of the time the work should take to complete. The reply includes a list of instances and their corresponding workunits. The handling of a request involves a number of database operations: reading and updating records for the user account, the host and the various jobs and instances [30].

Furthermore, the Scheduler estimates the computing power of the client hosts and allocates as many workunits to every client as can be processed within a day. On the other hand BOINC avoids frequent work request operations from clients who enable the service of a very large number of clients per day. BOINC does not trust clients because it can allocate workunits redundantly, i.e. the same workunit is sent to several clients. When a certain number of redundant workunits is consistent, then the result is accepted and will be flagged by the Transitioner as 'ready for validation' (Section 5.3.3) [12].

5.3.3 Transitioner

The Transitioner examines workunits and analyses which state change has occurred, e.g. a workunit has been returned from a participant and is ready for reporting. Depending on the situation, it may generate new instances; flag the workunit as having a permanent error or triggers validation or assimilation of the workunit [30]. It keeps track of the results in progress and makes sure that they move properly down the pipeline [71]. The Transitioner is a black-box element. The runtime behaviour can be configured with parameters during the call for execution. Four parameters can be modified in total (Appendix A.1).

5.3.4 Feeder

The feeder streamlines the Scheduler's database access. It maintains a shared-memory segment containing workunit relevant information which increases the performance on the workunit accesses:

- static database tables such as scientific applications, platforms, and application versions
- a fixed-size cache of unsent instance/job pairs. The Scheduler finds instances that can be sent to a particular client by scanning this memory segment [30].

The feeder is a black-box element. The runtime behaviour can be configured with parameters during the call for execution. Ten parameters can be modified in total (Appendix A.2).

5.3.5 Validator

The Validator performs two functions. At first, it checks whether a sufficient number (a 'quorum') of successful results has been returned and then compares and tests them for a 'consensus'. The method of comparing results (which may consider the platform-varying floating point arithmetic) and the policy of determining consensus (e.g. best two out of three) are supplied by the application. When a consensus is reached, a particular result is designated as the *canonical result*. This also includes the fact that, if a result arrives after a already reached consensus, it will be compared with the canonical result; this determines whether the user gets credit. The Validator compares the instances of a job and selects a canonical instance representing the correct output. It determines the credit granted to users and hosts, returns the correct output and updates those database records [30, 71, 131]. BOINC provides two standard Validators:

Trivial Validation This Validator checks if a specific amount of CPU cycles is not exceeded.

Bitwise Validation This Validator checks if two files are binarily equal through the comparison of MD5 (Message-Digest) [112] checksums and output files.

In addition to these two Validators, the BOINC project owner can implement their own validation process. The BOINC Application Programming Interface (API) provides functions which can be used for this purpose [91]. The Validator has to be provided for each scientific application. This means that it is not enough to create a scientific application code for a certain application but that the Validator code should also provide a more complex programming of BOINC [12]. The default Validators are black-box elements. The runtime behaviour can be configured by parameters during the call for execution. Ten parameters can be modified in total (Appendix A.3).

5.3.6 Assimilator

The Assimilator is the mechanism by which the project notifies the workunit completion, i.e. if the mechanism is performed successfully or unsuccessfully and for workunits that have a canonical instance or for which a permanent error has occurred. It is performed exactly once per workunit. Handling a successfully completed workunit might involve writing outputs to a scientific database or archiving the output files on hard-disk drives. When the workunit has failed, the function might write an entry in a log or send an e-mail [30, 131]. BOINC provides sample Assimilators which can be used out of the box:

Dummy Assimilation This Assimilator handles workunits but does not "really" assimilate them; only a log message is created.

Archive Assimilation This Assimilator stores successful and failed workunits into two specific directory destinations in which just the canonical results are handled; other results are handled by an error message in the root directory of the destination.

Single Job Assimilation This Assimilator performs only one workunit and exit. The canonical result is stored in a specific directory.

In addition, the BOINC project owner can implement their own assimilation processes. The BOINC API provides functions which can be used for this purpose [91]. The default Assimilators are black-box elements. The runtime behaviour can be configured by parameters during the call for execution. Eight parameters can be modified in total (Appendix A.4).

5.3.7 Structure of Workunits

A workunit is a package of one or more input files. These files can be plain-text or binary files. For workunit creation and distribution, two templates files are required: (1) the input template file and (2) the output template file. Each input or output file must be defined within these template files otherwise they are not recognised by BOINC. In cases in which output files are optional they can be tagged as *optional* within the output template file. BOINC provides some approaches to create workunits: (1) submitting of jobs on the command line [167, JobSubmission], (2) implementation of an own work-creation tool by use of BOINC's API [167, WorkGeneration] [57], (3) remote submitting of workunits [71, 9] or (4) workunit creation via web-interfaces [59].

A scientific application does not use the physical filenames. Therefore virtual filenames are specified within the template files. This allows the distribution of an unchanged scientific application with hard-coded filenames. The BOINC client and BOINC API are responsible for resolving the physical filename during the opening call of a virtual filename. Fig. 5.4 shows how scientific applications are executed, i.e. each execution is within a specific slot. Exemplarily, two virtual files are shown: (1) *data.xml* and (2) *result.xml*. In this case they are virtual files that act as symbolic-links, i.e. the files contain the XML tag "link" which contains the relative path to the physical file. BOINC provides functions to resolve the filename of the physical file, i.e. `boinc_fopen()`. This case shows one used slot, the project Spinhenge@home contains two physical files and it can be assumed that the second file is used by an additional slot.

It is not required to use BOINC's file-handling functionalities. When the sources of the scientific application are not available, another approach has to be used, e.g. a legacy-application. Accordingly the listed input and output files within

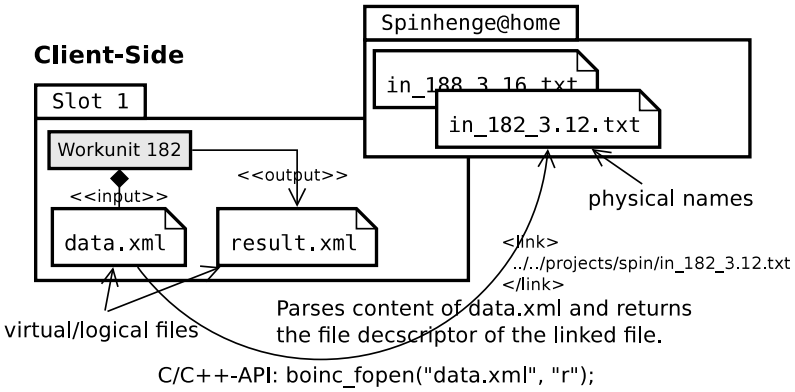

Figure 5.4: BOINC's virtual and physical name and slot mechanism [167, AppIntro].

the templates can be marked as "copy them to the slot" and opened directly. Under these circumstances the BOINC client is responsible for the copy processes.

5.3.8 Template Files

BOINC's workunit template files are fundamental for workunit creation. The creation will fail when they are not filled in correctly, e.g. the number of future workunit results by one client must be under or equal to the maximum number of future workunit results (Section 6.8.9). Furthermore, the template files describe how the input and output files of a computation are handled [167, JobSubmission], e.g. where they can be downloaded, how much floating-point operations (FLOPS) are allowed to finish the computation, when the results have to be finished (so-called *deadline*), or how much results are necessary to trigger the validation process. The computation's output files or results can be restricted by a maximum file size, i.e. if these files exceed the specified file size then the computation is marked as failed. The result template can be used to specify the following preferences for the result files: the virtual filenames, whether they are optional or the validation is not required, whether the files have to be copied to a BOINC's client project directory and reported without delay when they are uploaded.

5.3.9 Workunit processing

A workunit is performed simultaneously on several clients. The BOINC framework provides a form of *redundant computing* in which each computation is performed on multiple clients. Thus, the results are compared and accepted only when a 'consensus' is reached. Otherwise new results must be created and sent. Fig. 5.5 to Fig. 5.7 show how workunits are performed by the use of following configuration values [167, JobIn]:

min_quorum = 2: The minimum size of a 'quorum'. The Validator runs when there are two — in this case — successful results.

target_nresults = 3: Defines how many results have to be created initially. This must be at least *min_quorum*. It may be more to reflect the ratio of result loss, or to get a quorum more quickly.

max_total_results = 10: If the total number of results for this workunit would exceed ten, it is declared as error. This prevents the collection of workunits that are never reported, e.g. because they crash the core client. Without a maximum borderline BOINC's Transitioner will never flag a related workunit as "finished" and as a consequence the current computation series will run indefinitely.

These configuration values have to be set during the workunit creation, i.e. they can be defined within the input template file or can be defined in a BOINC C-programming structure named `WORKUNIT`. This structure has to be passed to BOINC's function `create_work()` as a first parameter and additional other parameters, e.g. the content of the input template file. Therefore, the workunit will be created and stored in the master database. Fig. 5.5 shows the creation of one workunit *(Time 0)*; after one time-step three workunit results are created. They are sent and performed differently on clients. The validation can be triggered when two results are returned *(Time 8)*. After successful validation they are assimilated. In the meantime *(Time 9)* still one workunit result is performed; the deadline is not reached yet and when the result is returned it will be validated and the user will be granted credits. What happens when the deadline is reached is shown in Fig. 5.6. Result 2 is lost but two results are already returned. As a consequence of `min_quorum` being 2, the workunit is validated, assimilated and finished. The maximum number of allowed workunit results is ten. Thereupon an additional workunit result *(Result 4)* is created when a previous result had an error *(Result 2)*. Only when no second result is returned, are additional workunit results created and sent to clients.

The reason for the creation of multiple similar workunit results is that BOINC cannot guarantee that one client is available until the deadline of a workunit is reached or that the performance finished without any other failures. As a consequence, a workunit is distributed to more than one client. Hence, the BOINC administrator can decide how many of several workunit results have to be performed before it is marked as failed or how many have to be returned before a validation check can be performed successfully (or not). A BOINC 'result' abstracts an instance of a computation. Typically, BOINC's server-side sends 'results' to clients who are performing the computation and returns the result. During this process, many things can happen to a result [131]:

- The client computes the result correctly and returns it.
- The client computes the result incorrectly and returns it.
- The client fails to download or upload files.
- The scientific application crashes on the client.
- The client does not return anything because the running of the BOINC client breaks or stops.
- The Scheduler is not able to send the result because it requires more resources than any client processes.

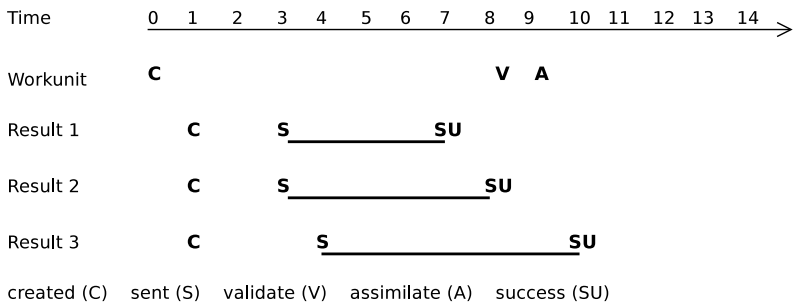

Figure 5.5: BOINC creates three results (`target_nresults=3`) automatically and they are sent at different moments. At Time 8, two successful results have returned. Consequently, the Validator is invoked; it finds a consensus and the workunit is assimilated. At Time 10, Result 3 arrives; validation is performed again, henceforth to check whether Result 3 gets credit [131]

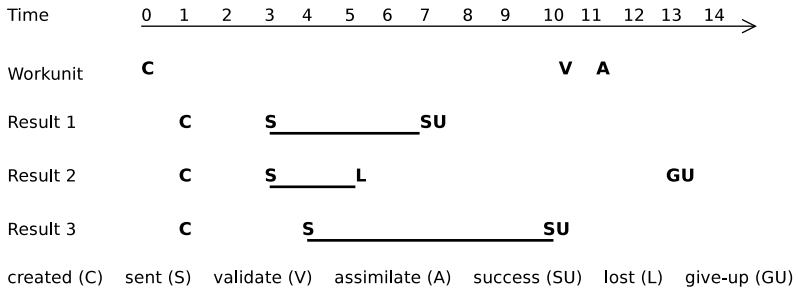

Figure 5.6: Result 2 is lost, i.e. there is no reply to the BOINC Scheduler. When Result 3 arrives a consensus is found and the workunit is assimilated. At Time 13 the Scheduler 'gives up' on Result 2, i.e. this enables the deletion of the canonical result's output files which are needed to validate late-arriving results [131]

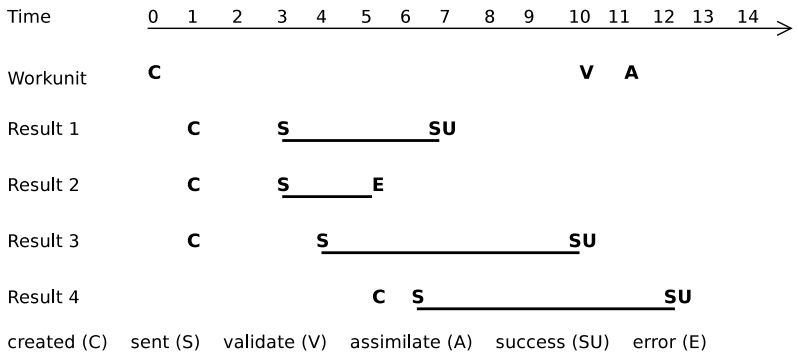

Figure 5.7: Result 2 returns an error at Time 5. This reduces the number of outstanding results to 2; because `target_nresults` is 3, BOINC creates another result (Result 4). A consensus is reached at Time 9, before Result 4 is returned [131]

5.4 Client Components

In [71] the client components are described in detail. Users who wish to contribute to the BOINC project, download and install the BOINC client software on their machines. The client software is developed for Windows, Mac and Linux operating systems. In order to attach the client to a project, the user must obtain an unique identification (ID) number by registering on the project Web site. Once the user enters the user ID and the project URL, he or she becomes attached to that particular project. A single user can attach to as many projects as desired. The user's computer cycle would then split between these different projects. The client software performs CPU scheduling of jobs between the projects and maximises concurrency by using multiple CPUs when possible and overlaps both communication and computation. The user has a significant amount of control over the client. While using the preferences in the client software, a user can change the amount of work their computer has to do for the project. Once the computation is completed, the client software notifies that the results are ready for transfer. The tasks are scheduled on a FIFO basis on the client but, if there are multiple projects, then a local scheduler algorithm will distribute the work cycle equally amongst the jobs. In addition, the software reports the amount of time spent on each project. The users are given credits for volunteering their computer. Security is one of the most important issues that arise while using the client software. The entire project related data and programs are hashed and signed by a private key. This ensures that the code the users are getting is from the right project and that is has not been tampered with by a malicious code. Besides, the client software provides solutions to a lot of project security-related issues where the user can try to return wrong results or get more credits. This is achieved by comparing or distributing the same work to at least five people and then checking the results for only allocating credits to users who have the right results. There are other features, such as limiting the size of the output file to the server and equally distributing the credits between users in order to avoid other project related security problems.

The client-side of BOINC users can have two basic applications: (i) BOINC client and (ii) BOINC manager (Section 5.4.1). The BOINC manager is an optional application and presented in Section 5.4.1. The BOINC client is necessary for the communication between BOINC users and a BOINC project. When a computation is performed, the scientific applications are stored and executed in a safe place, the so-called BOINC slot. Fig. 5.8 shows on the left-hand side the client-side of BOINC users. As an example, three slots are created and each slot works on its own workunit, i.e. Slot 1 performs workunit 1, Slot 2 performs workunit 65, and Slot 3 performs workunit 182. The execution is orthogonal. At any specific point in time, it is not specified which Slot is activated. The BOINC-Client manages the run-time behaviour through the use of the client-scheduling preferences, i.e. after

how many minutes a project has changed, if the processing on CPUs or GPUs is preferred.

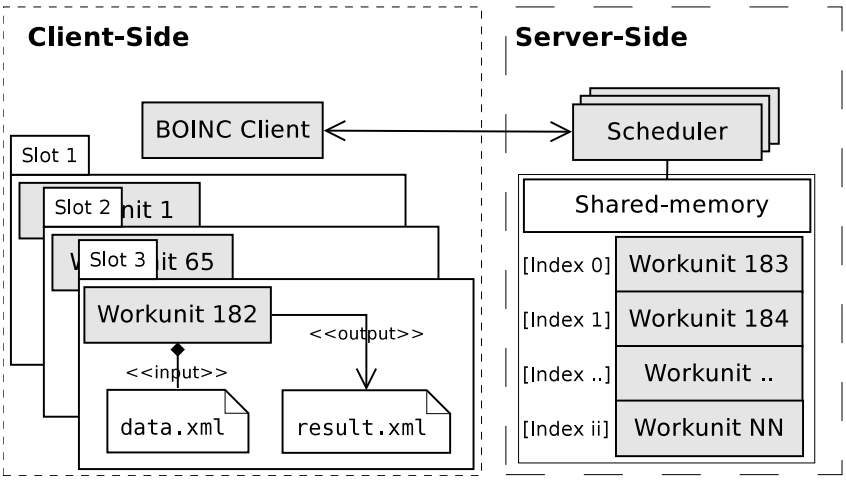

Figure 5.8: BOINC's slot mechanism to handle individual workunits.

In the meantime, the BOINC manager and BOINC client Remote Procedure Calls (RPCs) are used to control the execution of a scientific application in their slot. Four RPCs are available:

Suspend: The execution of the scientific application is suspended. The user can decide whether the application will be kept in memory or if it is removed from memory; this can be configured in the configuration dialogues of BOINC's manager.

Resume: A suspended scientific application is resumed.

Quit, Abort: These two RPCs are similar with the exception that *Quit* is used to stop the execution by the BOINC project server-side and *Abort* is a RPC used by BOINC users to stop the execution manually. The BOINC project will be informed about the manual aborting.

5.4.1 Manager and Command-Line

The BOINC manager is a graphical front-end for the BOINC client. The manager can connect to several clients simultaneously. The BOINC client can be configured and maintained with the manager. It allows seeing all current registered BOINC projects and demonstrates statistics of the granted credits. In addition, information about the current process is presented: how the project proceeds, how long it will take to finish the computation of a specific workunit and more. This controlling can be performed graphically or by a command-line tool. The call for this tool can

be modified with several parameters, e.g. to retrieve information of the processing state, or to trigger the benchmark to check the computation power. The communication between the BOINC client and these two tools is network-based by TCP/IP or UDP/IP network protocols. This communication is a kind of Remote Procedure Call (RPC) [114]. Within this thesis, the BOINC manager and command-line tool are not focused upon.

5.4.2 Scientific Application

A scientific application can be implemented by a scientific researcher but this can be a challenging task as mentioned in Section 5.1.3. In contrast, legacy-applications can be deployed on a BOINC infrastructure [57, 12]. A scientific application can be a single executable application or a package of several files, i.e. more than one executable application, signature files and a descriptive file. The descriptive file contains information about the files in the package [167, AppVersionNew], i.e. the physical file name, the logical file name, URLs of where the files can be downloaded, optionally whether the files have to be copied to the slots or compressed. The descriptive file is similar to workunit template files.

Scenario 1: Porting

An accessible code-base can be ported to BOINC. This can be done to some extent without BOINC related functionality, with support of basic BOINC functionalities or with advanced BOINC support. For the execution of scientific applications it is not necessary to support any BOINC functionality, nor essential to distribute any input files with the workunits. The scientific application can be executed without any modification. It must at least support the target platforms, but this is handled by BOINC's server-side scheduling and the configured platforms.

For all three scenarios, the structure of the scientific application is basically the same [24], i.e.

- initialisation of the BOINC middleware internals with optional flags to enable diagnostic functionalities,
- opening of input files,
- checking for a checkpoint file, if one exists then open and read the content,
- (optionally) reserve the target processor, e.g. the Graphics-processing Unit (GPU),
- do the core computation of the scientific application, e.g. perform a statistics computation [165], do some filtering for film sequences [186, 56], create ping tests for websites [150],

- create checkpoints frequently during core computation, in addition send the current 'fraction done' value to BOINC's client and handle asynchronous messages,
- finally, the scientific application is completed and will exit.

The core computation can be implemented in different ways: (1) as a sequentially processed application or (2) as a multi-threaded CPU or/and GPU application. Both approaches need a different initialisation of the BOINC middleware framework. Furthermore, the handling of the second approach poses new challenges, i.e. deadlock or livelock, and race-conditions. Multi-thread applications can be implemented with several different technologies:

Threads: A modern operating system executes applications in threads to support process scheduling [94]. As a result, developers can implement their own thread-based applications which are executed within the same memory-space. A prominent thread standard is specified by the IEEE POSIX 1003.1c specification [120].

Shared-memory: OpenMP [115] is a candidate which has an API to support multi-platform shared-memory parallel programming in C/C++ and Fortran. As a result, it is feasible to increase the amount of usable computation power.

Message-Passing: This technique allows combining of distributed resources. Message Passing Interface (MPI) [121] is a prominent middleware to merge these computational resources. BOINC's current methodology does not forbid the use of MPI but it is not fully supported, i.e. BOINC does not provide any API calls to identify if there is a cluster on one client-side.

Scenario 2: Wrapping

When a legacy-application has to be distributed, BOINC provides a wrapper to use such an application [167, WrapperApp]. This wrapper only works properly when all the following constraints are fulfilled:

- Each task has a fixed (and known) number of input and output files,
- tasks are executed sequentially in a fixed order,
- no task requires any special preparations before execution or post-processing function after execution, and
- the tasks do not start other processes, e.g. do not act themselves as a wrapper for another legacy application.

GenWrapper [18] aims to provide a generic solution for wrapping and executing an arbitrary set of legacy applications by utilising a POSIX like shell scripting environment. It should describe how the application is going to run and how the workunit should be processed. GenWrapper provides the following enhancements over the BOINC Wrapper [12]:

- Generic POSIX shell scripting language for describing tasks and for performing any operation before and after execution,
- tasks may be executed in any order,
- ability to execute tasks that require special preparations,
- ability to run tasks with variable numbers of input and output files,
- support for batching of variable numbers of smaller tasks together into a single workunit,
- ability to monitor and control any child task of the legacy tasks.

Scenario 3: Virtualization

This scenario uses a specialisation of BOINC's wrapper to execute a virtual machine (VM) [167, VboxApps] [146]. Thus, the VM is used for unifying the application environment, i.e. within the VM only one operating system is installed and will be executed. The developer has to be sure that the distribution of such an operating system installation within a VM is allowed by the relevant policies, e.g. Linux can be distributed and Microsoft's Windows must be licensed for any individual PC. Users register on a BOINC project, thereupon they download the scientific application (i.e. the wrapper), retrieve workunits, and additional workunit input files. The wrapper is executed within BOINC's client slots. It creates a slot-subdirectory, copies the input files to this slot-subdirectory, and starts the VM. Then the VM tries to mount the slot-subdirectory to its local file-hierarchy, executes the VM included scientific application, performs the calculation and works like a normal scientific application as described by the first scenario. After completion, the VM is deleted. The wrapper exits normally and returns the computational results in the mounted directory by the VM. Thus, they are available within BOINC's client slot.

5.5 BOINC's Requirements of a Working Project

Previous sections have given an overview of which structural features and functionalities are provided by the BOINC framework; in the following named as *features*.

Name	Description
BOINC's default services	Some BOINC services have to be configured and started during runtime, i.e. Feeder, Transitioner, Validator, Assimilator, and Scheduler.
Scientific Application	This entity includes the core of a computation and handles workunits.
Workunits	Any computation needs information for performance.
Database	Provides information about the BOINC project, e.g. workunits, users or hosts.
Assimilation's target	Returned and validated workunit results have to be stored somewhere, either within a specific directory or in a specific database.

Table 5.1: BOINC project required features.

Not all of these features are required to set-up and run a BOINC-based distributed computational project, e.g. a website presentation is not necessary, it is used to present results and to give general project information. Nevertheless is still necessary to have a web server on one or more hosts which provides an interface for downloading workunits and to upload computational results. Besides, the BOINC framework enables consumers to distribute server-side components and services to individual hosts. They can be simplified by only using one host with an all-in-one installation. In general, BOINC's features can be categorised as **required** and **optional**. Table 5.1 lists all required features and Table 5.2 lists all optional features.

5.6 Summary

This chapter introduced the fundamentals of BOINC's architecture, its general idea, a technical overview and the fields of application. In the first sections, the differences between BOINC — based on the Public Resource Computing (PRC) principles — and Grid Computing (GC) are given. These sections are followed by a clarification of BOINC's potential, its goals and features. Based on the fact of how a BOINC created project is working, necessary server-side tools and applications for processing the project are presented and described briefly. Next, relevant components and implementation details for the creation of a future application are outlined. Finally, the required and optional BOINC parts for starting a BOINC project are summarised.

Name	Description
Website	A website is used to provide information about the BOINC project. In particular, it provides detailed server status notification, can be depicted with news about the project, present a forum for user-communication and possesses a login form for administration tasks.
Asynchronous Messages	This kind of communication is only used for long-running computations and is optional. Long-running computations can be handled like general computations. Asynchronous messages are partly covered in this thesis (Section 6.11).
GPU-Computing, Thread-Computing	The scientific application can be executed on different targets (e.g. Graphical-processing units (GPU)) or different execution approaches (e.g. single-thread or multi-thread processing). Both ways require the use of third-party libraries, e.g. CUDA [86], OpenCL [122] or OpenMP [115]. This thesis does not focus on these approaches.
Credit System	For scientific or engineering computations, credits are unnecessary; as described in Section 5.1.4 they are used in order to attract and retain participants [30]. Thus, BOINC's default credit system is used; it is not covered in this thesis.
Forum	The mentioned forum for *Websites* is used to provide a way to communicate with participants. This feature is not necessary for the creation of a BOINC project and therefore not within the scope of this thesis.

Table 5.2: BOINC project optional features.

Part II

The Solution

CHAPTER 6

UML for BOINC Profile Definition

A lot of things which come with a high profile will always be criticised one way or another.

EVELYN GLENNIE

6.1 Introduction

This chapter introduces all UML stereotypes for UML4BOINC which allows specifying the structure and behaviour of individual BOINC projects. This chapter lists all new stereotypes and describes how they are extending the UML specification. The semantic of the several stereotypes is out of this chapter's scope. As a consequence, Chapter 7 is used to describe the semantic of UML4BOINC's stereotypes.

For clarity, there are two auxiliary OCL operations defined which are re-used in the informal constraints and Object Constraint Language (OCL) rules of the stereotypes [74, pp.103ff]. The OCL operation *isStereotypedBy* results in a *true* value if the given element is extended by the specified stereotype classifier. Otherwise, it results in a *false* value.

```
UML4BOINC::isStereotypedBy(e: Element, s: Stereotype) : Boolean;
post: result=e.extension->exists(e | e.type=s)
```

All elements within UML4BOINC must have an unique name. Therefore the UML NamedElement is stereotyped directly to support this specification.

```
context UML4BOINC::NamedElement inv:
  name->notEmpty() implies
    allInstances->forAll(n1, n2 | n1.name <> n2.name)
```

6.2 Reading the Profile Definition

A UML profile restricts the use of the general UML by use of stereotypes. Stereotypes are closed with so-called Guillemets, e.g. «Name of a Stereotype».[1] Any

[1] Guillemets are French quotation marks.

stereotype can specify tag-values, associations and additional informal or rule-based constraints to restrict the use of the UML metaclass and to restrict the meaning of the tag-values and associations. Tag-values are similar to attributes to declare additional information in the context of a UML metaclass. Associations are used to add relations between UML metaclasses and other stereotypes. Constraints can be defined as informal rules and with the help of the Object Constraint Language (OCL) in a formal way.

In this chapter, there is a specific notation used for all UML metaclass extensions [74]. Names of tag-values, associations and additional operations are shown *italic*. References to UML metaclasses are capitalized, e.g. Event, Transition or Trigger. Attributes, associations and operations of the metaclasses are written in lower-case letters and are underlined. For any tag-value or association where no multiplicity is given, the default is zero-or-one — in UML described as [0..1] — , i.e. the value for the tag-value or association is optional. Enumeration literals are always written with the prefix # when no scope is defined, e.g. *#public* or *VisibilityKind::public*. For a comfortable handling of all stereotypes, all belonging tag-values are specified as public, i.e. in the figures a + is used as the prefix.

Stereotype specifications are all introduced in the same format. The individual specifications are split into five parts: (1^{st} part) a general description, i.e. in which context the stereotype is used and why it is specified, (2^{nd} part) the extended UML metaclass, (3^{rd} part)–(4^{th} part) the tag-values and associations of the stereotype, and (5^{th} part) describes informal and OCL constraints. Informal constraints are only used when the OCL cannot impress the constraints directly. All OCL constraints are in the context of the current context, i.e. the OCL context preamble is not shown (Section 4.6). All stereotypes in the following sections are arranged alphabetically; they are not prioritised. Finally, this chapter only specifies all stereotypes, the semantic is introduced in Chapter 7 and how they can be applied is shown in Chapters 8 and 9.

6.3 UML Profile for BOINC

The UML profile UML4BOINC is used to specialise UML's specification for a specific point of view (POV). The use of UML4BOINC must enable one to use UML elements for modelling of different focused BOINC projects, e.g. the same set of UML elements must allow the creation of a BOINC project for statistical computation or movie sequence processing. UML itself specifies three applicable diagram types: (1) structure, (2) behaviour, and (3) interaction. UML4BOINC adapts these diagram types and uses them for the following scenarios:

Structure A BOINC project infrastructure is described within this diagram type. Relations between relevant components can be created, e.g. hosts and shares.

In addition the structure of workunits, input/output files and checkpoints can be described.

Behaviour & Interaction The behaviour of the components of a BOINC project are defined, e.g. in case a specific Validator is necessary or when specific tasks have to be executed. Moreover, the behaviour of the distributed scientific application can be described. These diagram types are also used for the dynamic creation of workunits and for describing the workunit's lifetime, e.g. when a workunit is processed and successfully stored. In particular, as a minimum set, the following scenarios can be described:

- How input and output files can be opened by a scientific application, Validator or Assimilator.
- Handling of workunits or series when they are planned to be created, or when they are finished or failed.
- Handling of asynchronous-messages between the server-side and client-side of all communicating participants of a BOINC project.

6.4 Types of Diagrams

UML4BOINC is subdivided into packages. Each package has its own specific domain. Fig. 6.1 shows how the several packages are related to each other. In total eight packages are defined; six of them define special diagram types and they are labelled with the prefix `(D)`. These six diagrams are used for the mentioned *Structure* and *Behaviour & Interaction* diagram types. In UML4BOINC these diagrams are defined as packages and can be used by different domain experts. Fig. 6.1 visualises the dependencies between these six diagrams and additional two packages: (i) *Trickle* and (ii) *Project*. *Project* is the main package and imports all diagram types [24]. It is recommended that tool vendors support *Project* and not just selected packages. The six diagrams are used to visualise the context of a BOINC project. In fact, visualisation is performed for two purposes: (1) for communicating an idea, and (2) for discovering the idea itself. While graphics have been utilised for both thinking and communication for thousands of years, the evolution of the computer has dramatically changed the way these graphics are (i) designed, (ii) rendered, and (iii) interacted with [78, 66]. In general, the same underlying model can be rendered and maintained by the help of different views; known as the Model-view controller (MVC) concept [83]. Here, the controller decides which information is shown by the different views and how data change requests are handled. With the help of the mentioned six diagrams, the context of a BOINC project and its related elements can be focused by the use of specific views. When they are merged a complete and full-operational BOINC project can be generated.

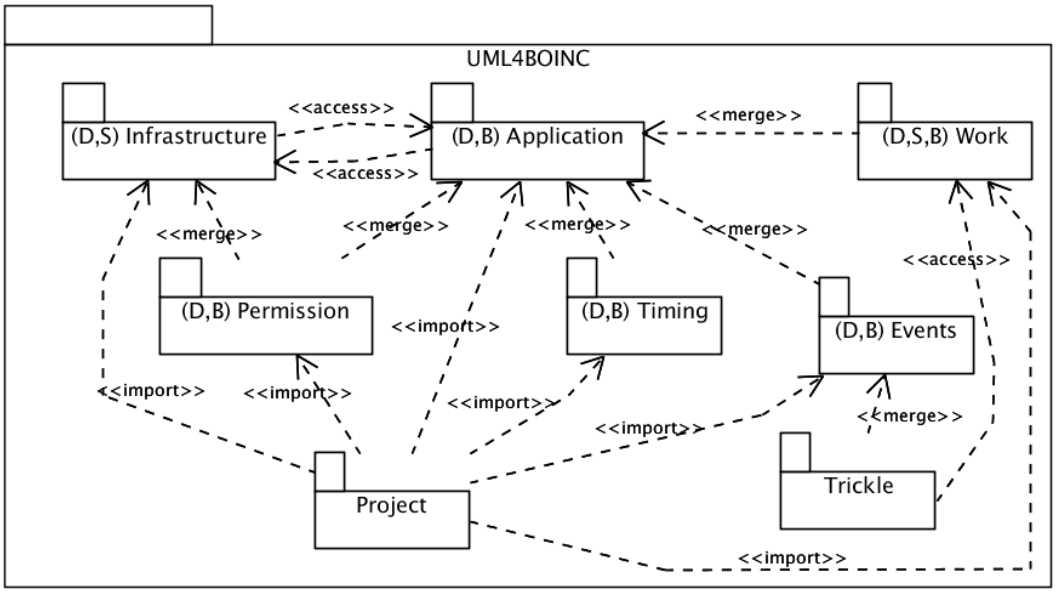

Figure 6.1: Diagram packages of UML4BOINC; The prefixes of the package names describe how the several packages are used: `(D,S)` includes UML elements for a structure diagram, `(D,B)` includes UML elements for a behaviour diagram, and `(D,S,B)` includes both, structure and behaviour elements.

The two packages *Infrastructure* and *Application* are the most autonomous ones and do not merge any other UML4BOINC packages. Certainly the Infrastructure package has access to the Application package and vice versa, i.e. to allow the deployment of a BOINC project with the information as to which scientific application should be provided and used by BOINC's services (which is necessary for the selection of computational results when an individual Validator or Assimilator is used). Fig. 6.1 does not show the hierarchy related to UML's specification and metaclasses, e.g. the *Infrastructure* package merges elements of the UML PackagingComponents package [125, Fig. 8.1] to enable the creation of hosts and embedded provided BOINC services. Section 6.5 to 6.12 state the relation to the UML specification and which UML packages the metaclasses contain which are extended by the stereotypes in this chapter. Furthermore, it can be seen in Fig. 6.1 that the package *Application* is automatically available when one of the following packages are used: *Permission*, *Timing*, *Work*, or *Events*. This is reasonable, because of the fact that these packages are used to set-up restrictions on different implementations, e.g. the timing package is used to describe when [unit of time] tasks or other applications are executed. Appendix B contains six sections which give an overview of the latter stereotypes described in this chapter and states in which UML4BOINC package they are included. Furthermore, these sections can be used as a short reference for all introduced stereotypes in this thesis.

6.4.1 UML4BOINC's Diagrams

1. **Infrastructure Diagram:** This diagram is used to specify the infrastructure of one BOINC project and to add third-party libraries which are necessary during runtime. It is used to add different hosts, Storage Area Networks (SAN), BOINC services and periodically executed tasks; all mentioned elements are shown as placeholders and have no implementation during their initialisation; in this diagram only the infrastructure of a BOINC project is created. This diagram shows the associations between hosts, e.g. where computational results should be stored. Chapter 9 contains a use-case of this diagram. The infrastructure diagram is a kind of UML Composite Structure and UML Deployment diagram [55]. Section 6.5 contains the description of allowed elements and their specific stereotypes.

2. **Application Diagram:** Within this diagram, behaviours are specified, e.g. a scientific application or tasks. Furthermore, it is possible to add wrappers which would make it feasible to use de-facto industry standard software components. This type of diagram is a mix of a UML StateMachine diagram and a UML Activity diagram. On the one hand, if a legacy-application like COMSOL Multiphysics were used [57, 73, 199], it is not necessary to define a whole state machine or activity, because the underlying wrapper-implementation handles the execution [58]. On the other hand, if an individual implementation is required, the structure of a scientific application can be modelled by a set of UML Actions. Section 6.6 contains the description of allowed elements and their specific stereotypes.

3. **Permission Diagram:** Role-based Access Control (RBAC) is specified in this diagram. Within this diagram, any user, role and permission can be added. Associations are used to associate a set of permissions which are associated to resources, i.e. users are not associated directly to resources (e.g. hosts), they are associated to roles, these roles are associated to a set of permissions and finally to resources [55, 36]. Resources are cross-referenced to specific UML4BOINC elements, because these elements are part of the Infrastructure diagram and not visualised within the Permission diagram. Section 6.7 contains the description of allowed elements and their specific stereotypes.

4. **Work Diagram:** This diagram is used to specify the structure of workunits. One can specify which file aliases must be used within the *Application Diagram* and the structure of input files and output files. Additional information about computation series can be specified, i.e. computations are planned to be performed and for starting previous results are required. These depen-

dencies can be created by the use of associations between two or more computation series. This diagram type is a kind of a UML Class diagram [54]. UML4BOINC's Work package itself contains more stereotypes to specify how computations are handled. These stereotypes extend the UML4BOINC Application diagram with the ability to monitor the lifetime of workunits and can be used to specify the behaviour of workunit's handling different computation states. Sections 6.8 to 6.10 contain the description of allowed elements and their specific stereotypes.

5. **Events Diagram:** Unlimited events can be added and each is a start of a sequence to handle individual events, e.g. if a host sends a message it can be forwarded to other hosts; like chat messages. BOINC can handle asynchronous messages between the server-side and client-side, so-called trickle-messages as described in Section 5.1.4. In this diagram the message type can be specified. In addition, an event-handler's implementation can be referenced to elements within the mentioned *Application Diagram*. This type of diagram is a kind of a UML Sequence diagram. Furthermore, when a server-side host becomes unreachable, an administrator can be informed automatically. Section 6.11 contains the description of allowed elements and their specific stereotypes.

6. **Timing Diagram:** As mentioned for the *Infrastructure Diagram*, a BOINC project can have periodically executed tasks. They can be further specified in this *Timing Diagram* where time-ranges for the execution of tasks can be defined. This diagram is a kind of a UML Activity diagram. The relevant stereotypes of this diagram are presented in Section 6.12.

6.5 Infrastructure

Fig. 6.2 shows an overview of all specified stereotypes to model the infrastructure of a BOINC project. In addition, it shows which UML metaclasses are extended. Constraints are not included; they are described in the individual stereotype Section 6.5.1 until Section 6.5.23.

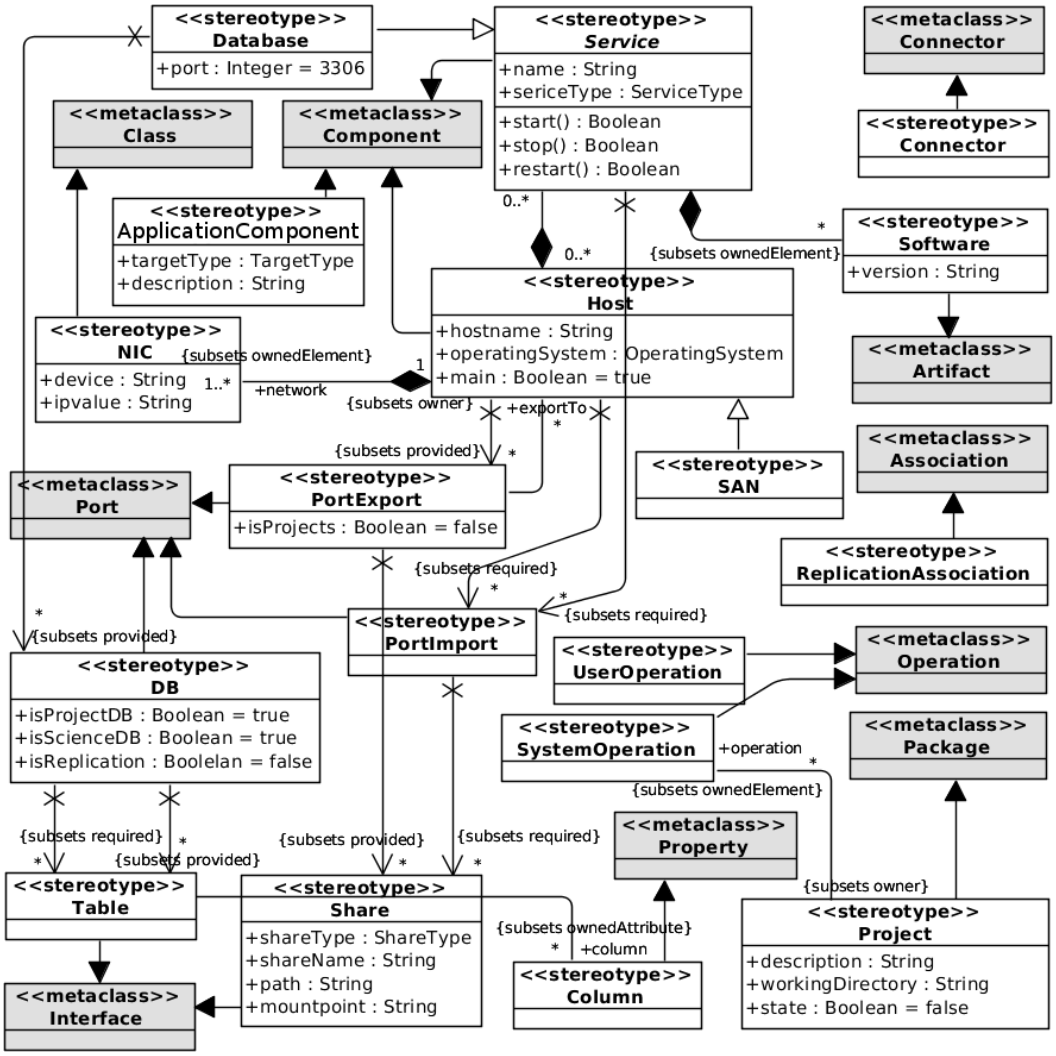

Figure 6.2: UML4BOINC stereotypes to restrict the use of UML metaclasses for the infrastructure modelling of a BOINC project; numerous stereotypes are included in the Infrastructure package which extend ten different UML metaclasses.

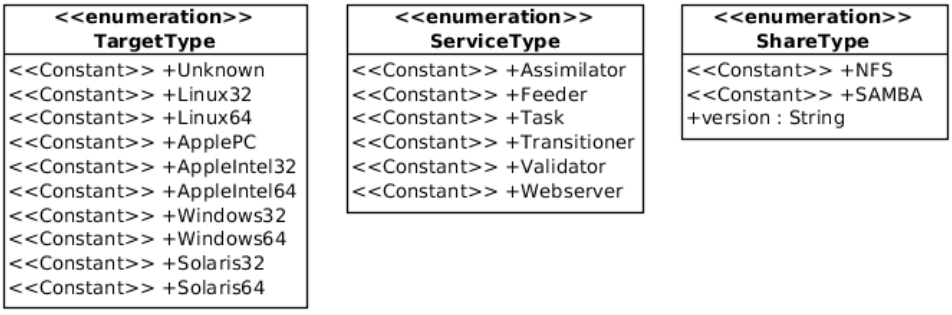

Figure 6.3: Enumerations used to specify different kinds of scientific applications, services, and network exchange protocols, i.e. Network-file system (NFS) [109] or SAMBA [204].

6.5.1 ApplicationComponent

«ApplicationComponent» extends the UML metaclass `Component (from BasicComponents)` and describes a scientific application, which is used for computations. This Component can be substituted by another only if the two types conform [125, p.150]. Here, the Component is only used to inform which application is provided by a BOINC project. This Component must only have required and provided interfaces to access input files and to create output files. name is used as the name for the physical instance of the scientific application.

UML MetaClass
Component (from BasicComponents, PackagingComponents) [125, Chapter 8.3.1]

Tag-Values
targetType : TargetType [*] – Specifies the target types, e.g. Linux32 or Windows64.
description : String [1] – A short descriptive text of this application, e.g. "Spinhenge@home – Monte Carlo experiments of nanosystems" [165] (see Section 9.2).

Associations
No additional associations.

Informal and OCL Constraints
[1] At least one target type must be set.

```
self.targetType->size() > 0
```

6.5.2 Column

«Column» extends the UML metaclass `Property (from Kernel, AssociationClasses, Interfaces)` which describes individual database table

columns.

UML MetaClass
Property (from Kernel, AssociationClasses, Interfaces) [125, Chapter 7.3.45]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] The owner is stereotyped by «Table».

```
isStereotypedBy(interface, Table)
```

[2] «Column» is never read-only.

```
isReadOnly = false
```

[3] «Column» never redefines or subsets a property.

```
redefinedProperty->empty() and subsettedProperty->empty()
```

[4] «Column» has no settings of the UML AssociationClasses package.

```
associationEnd->empty() and qualifier->empty()
```

6.5.3 Connector

«Connector» extends the UML metaclass `Connector (from BasicComponents)` and connects Ports extended by «PortExport» or «PortImport» with a Port in the same context which is extended by the same stereotype, i.e. the connection ends must have the same stereotype. Fig. 6.4 shows this principle, where the BOINC service webserver on host "lmboinc-web" owns one exporting Port (P1). The host itself has a Port (P2) to export data. P2 is connected with the internal port P1 and this approach hides the internal structure of hosts and offers an easier replacement of a host, i.e. only the external Ports must fit. In case BOINC services are not visible or specific Ports are not visible for external components, it is required to create an external Port to interact with external BOINC services; this is known as delegation. It does not matter, for the external environment, what the internal structure looks like; the same can be done for «PortImport». For «DB» the connector can be used with the same principles.

UML MetaClass
Connector (from BasicComponents) [125, Chapter 8.3.4]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] A Connection has no behaviour.

```
contract->empty()
```

[2] A *delegation* can only connect Ports extended by «PortExport» or «PortImport».

```
kind = ConnectorKind::delegation implies
  end->forAll(e | isStereotypedBy(e, PortExport) or
                  isStereotypedBy(e, PortImport))
```

[3] An *assembly* Connector can only connect Interfaces extended by «Share» or «Table».

```
kind = ConnectorKind::assembly implies
  end->forAll(e | isStereotypedBy(e, Share) or
                  isStereotypedBy(e, Table))
```

6.5.4 Database

«Database» extends the UML metaclass `Component (from BasicComponents)` and is used to add database instances to hosts. «Database» must have Ports extended by «DB» to provide individual database contexts, e.g. BOINC's master/science or replica databases.

UML MetaClass
Component (from BasicComponents, PackagingComponents) [125, Chapter 8.3.1]

Tag-Values
port : Integer = 3306 [1] – Used as port number for IP communications.

Associations
No additional associations.

Informal and OCL Constraints
[1] All Ports must be extended by «DB» and only provided Ports are allowed. No directly used provided Interfaces are allowed.

```
provided->forAll( i | isStereotypedBy(i, DB)) and required->isEmpty()
```

[2] The IP port for network communications must be set.

```
port->notEmpty() implies port > 0 and port <= 65535
```

6.5.5 DB

«DB» extends the UML metaclass `Port (from Ports)` and is used to provide access to databases. «DB» describes BOINC's master/science database and optional replica databases. If a master/science database should be used with a replica

database instance, the replica database can be installed on individual hosts within a BOINC project. As a consequence, the database instances must be associated by an Association extended by «ReplicationAssociation» (Section 6.5.11). The kind of a database is specified by *isProjectDB* and *isScienceDB*. As default, an instance of «DB» is used as master and science database.

UML MetaClass
Port (from Ports) [125, Chapter 9.3.12]

Tag-Values
isProjectDB : Boolean = true [1] – Specifies that this «DB» is used as a data backend for a BOINC project, i.e. all information about users, hosts, supported scientific applications and versions, etc. are stored in this database.
isScienceDB : Boolean = true [1] – Specifies that this «DB» is used to store computational results or other user-defined sets of data.
isReplication : Boolean = false [1] – Specifies that this «DB» is used as a replication for an another «DB» context.

Associations
No additional associations.

Informal and OCL Constraints
[1] Only Interfaces extended by «Table» can be used for provided Interfaces.

```
provided->forAll(p | isStereotypedBy(p, Table))
```

[2] name is the name of the used database, e.g. "spin" of the BOINC project Spinhenge@home.[2]
[3] Minimum of one «DB» must exist, i.e. a BOINC project needs a database as data back-end; this one must be BOINC's project database.

```
allInstances()->select(isProjectDB=true)->size() = 1
```

[4] When this «DB» is specified as project's database, its name is set to «Project»'s name.
[5] If this «DB» is a replica, then it cannot be a science or project database context.

```
isReplication = true implies
  isProjectDB = false AND isScienceDB = false
```

[6] Only read-only queries are allowed when this instance is specified as a replica database.

6.5.6 Host

«Host» extends the UML metaclass `Component (from BasicComponents)` and is used to specify a host on the SPoV. The number of hosts is not

[2] http://spin.fh-bielefeld.de

limited. A host can nest any BOINC service, e.g. Feeder or Transitioner. In addition, these BOINC services can be deployed on different hosts. A BOINC project needs a minimum of one host which provides the BOINC installation with all BOINC tools and configuration files as a «Share» for other hosts — if available —, i.e. one host must have *main = true*. This main host must have a Port extended by «PortExport» to provide the mentioned «Share» to other hosts. As consequence the other hosts must have a minimum of one Port extended by «PortImport» and an associated «Share» to import the exported shares. Any host can have a different operating system (OS). Despite the fact that BOINC only targets Unix platforms [167, ServerIntro], UML4BOINC allows the specification of other platforms. *operatingSystem* is used to distinguish the different OS environments. UML4BOINC does not specify how the model is used for the configuration of hosts. OSs can differ and the tool which supports UML4BOINC must handle different OS types.

UML MetaClass
Component (from BasicComponent, PackagingComponents) [125, Chapter 8.3.1]

Tag-Values
hostname : String [1] – This is a network-wide addressable name of a host. Network addresses by all network descriptions must be mapped to this *hostname*, i.e. hostname "vg-grid" must be addressable by an IP address within a network.
main : Boolean = true [1] – If this tag-value is *true*, then this host owns the main BOINC project installation and must provide it to external hosts. If it is *false*, then the BOINC installation must be imported to this host.
operatingSystem : OperatingSystem [1] – Specifies which OS is used for this host.

Associations
network : NIC [1..]* – A host must have a network definition to interconnect with other hosts. Without a network specification it is not possible to have a valid configuration.

Informal and OCL Constraints
[1] *hostname* must be set and unique.

```
hostname->notEmpty() and allInstances()->unique(hostname)
```

[2] *operatingSystem* must have a value.

```
operatingSystem->notEmpty()
```

[3] Only BOINC relevant elements can be added to this component.

```
packagedElement->size() >= 1 implies
  packagedElement->forAll( e : PackageableElement |
                isStereotypedBy(e, Service) or
                isStereotypedBy(e, Software) or
                isStereotypedBy(e, Database))
```

[4] Only one host has *main* with a value of *true*.

```
allInstances()->select(main = true)->count() = 1
```

[5] If this host has *main* = *true*, at least one Port extended by «PortExport» must be added to this Component and an Interface extended by «Share» to provide the BOINC project installation must be connected. In other circumstances, when this host has *main* = *false*, at least one Port extended by «PortImport» must be added to this Component and connected with the previous shared interface.

```
if main = true then
  provided->exists( p : Port |
    isStereotypedBy(p, PortExport) and
    p->exists( i : Interface | isStereotypedBy(i, Share)))
else
  required->exists( p : Port |
    isStereotypedBy(p, PortImport)
endif
```

[6] Specified OSs must be valid for a host platform and architecture, i.e. a 64-bit based OS is not executable on a 32-bit architecture.

6.5.7 NIC

«NIC» extends the UML metaclass `Class (from Kernel)` and specifies network settings for a Network Interface Card (NIC) on an individual «Host». *ipvalue* can contain any IP description, e.g. *192.168.0.10/24* for an IP version 4 value [113] or *fe80:0:0:0:0:0:c0a8:a* for an IP version 6 value [111] (same addresses).

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
device : String [1] – Name of the used device, e.g. *eth0* or *eth1*.
ipvalue : String [0..1] – Contains an IP network address.

Associations
No additional associations.

Informal and OCL Constraints
[1] The owner of a «NIC» can only be a Component extended by «Host».

```
isStereotypedBy(self.class, Host)
```

[2] If *ipvalue* has no value, then for all modelling purposes *device* must be used.

```
if ipvalue->empty() then
  ipvalue = device
else false endif
```

6.5.8 OperatingSystem

«OperatingSystem» extends the UML metaclass `Enumeration (from Kernel)` and contains enumeration literals for each planned or supported operating system, e.g.:

- *#Linux32* for a Linux system with 32-bit architecture support,
- *#Linux64* for a Linux system with 64-bit architecture support or
- *#Windows32* for a Windows system with 32-bit architecture support.

A minimum of one enumeration literal must be defined by tool developers.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

6.5.9 PortExport and PortImport

«PortExport» extends the UML metaclass `Port (from Ports)` and specifies directories which are shared between hosts. «PortImport» extends the UML metaclass `Port (from Ports)` and is used to import these directories. An export or import definition can be owned by a «Host» or a «Service» stereotype. In case a Port is owned by a «Host», this Port can be connected to internal Components extended by the same stereotype as shown in Fig. 6.4. When a Port is owned by a BOINC service (internal Port) and is not visible to external Components, but must be used by them, this Port must have an external Port to provide access. These external Ports are not allowed to be private, i.e. if a BOINC service owns a Port and it is not public, it must be connected with a Connector extended by «Connector» to a Port which is added to a Component extended by «Host». Fig. 6.4 shows how this looks, where a Port of a BOINC service is connected to a Port by a «Host». Ports should be used for one specific purpose, e.g. to export computational results of different scientific projects. The minimum number of Ports on a «Host» is one, because BOINC sources need to be exported to all hosts or imported by hosts (Section 6.5.6).

UML MetaClass
Port (from Ports) [125, Chapter 9.3.12]

Tag-Values for «PortExport»

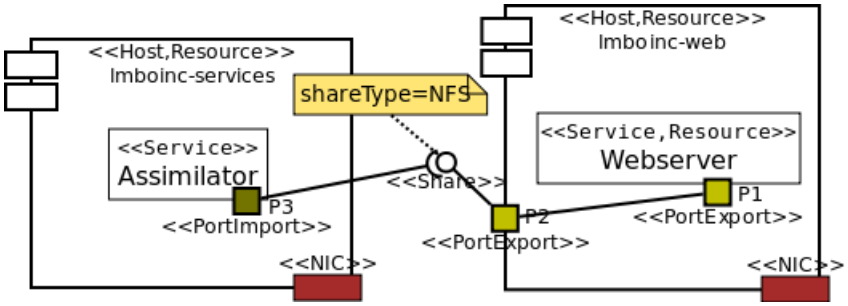

Figure 6.4: Example how to use «PortExport» and «PortImport» connected with an Interface extended by «Share».

isProjects : Boolean = false [0..1] – If the owner of this Port is BOINC's main host, then *isProjects* is *true*.

Tag-Values for «PortImport»
No additional tag-values.

Associations for «PortExport»
exportTo : Host [*] – Association to hosts which are allowed to use this Port and all provided Interfaces. This Association is not necessary to visualise. It is based on the connections between Ports and Interfaces, e.g. Fig. 6.4 has a «Share» between the host "lmboinc-web" and "lmboinc-services" and *exportTo* has "lmboinc-services" as an associated element.

Associations for «PortImport»
No additional associations.

Informal and OCL Constraints
[1] The owner must be a «Host» or a «Service» extended Component.

```
isStereotypedBy(self.class, Host) or
isStereotypedBy(self.class, Service)
```

[2] A Port which is owned by a Component extended by «Host» is public and a service.

```
visibility = VisibilityKind::public and isService = true
  implies isStereotypedBy(self.class, Host)
```

[3] «PortExport» does not require Interfaces, it is used to provide Interfaces. «PortImport» does not provide Interfaces, it requires Interfaces.

```
-- <<PortExport>>
   self.required->isEmpty() and self.provided->notEmpty()
     implies self.provided->forAll(i | isStereotypedBy(i, Share))
-- <<PortImport>>
   self.required->notEmpty() and self.provided->isEmpty()
     implies self.required->forAll(i | isStereotypedBy(i, Share))
```

[4] *exportTo* is filled with Components extended by «Host» which are connected indirect via Ports and Interfaces.

```
   if self.exportTo->size() > 0 then self.exportTo->forAll(
     required->forAll(isStereotypedBy(port.class, Host)) or
     required->forAll(isStereotypedBy(port.class, SAN))
   ) else false endif
```

[5] If `isProjects = true`, then the owner of this Port is a «Host».

```
   isProjects = true implies isStereotypedBy(self.class, Host)
```

[6] All Interfaces must be unique in the context of a Port, i.e. it is not allowed to connect a Port twice to the same Interface.

6.5.10 Project

«Project» extends the UML metaclass `Package (from Kernel)` and is used to describe an individual BOINC project. This package contains all information to set-up a complete BOINC project. name is used as the name of this BOINC project, e.g. *spin* of "Spinhenge@home" (see Section 9.2) or *lmboinc* for "Linux-Magazin BOINC" (see Section 9.3). Fig. 6.1 shows how this Package is related to the other packages of UML4BOINC. «Project» imports all six diagram packages to allow one to set-up a complete BOINC project.

UML MetaClass
Package (from Kernel) [125, Chapter 7.3.38]

Tag-Values
projectName: String [1] – The name of the project, e.g. "spin" or "Kreiszahl@home". *description : String [1]* – A more detailed informal text, e.g. "Spinhenge@home" or "Kreiszahl@home – Monte-Carlo-Simulation".
workingDirectory : String [0..1] – Directory where a BOINC project is installed and executed on any used «Host»; this directory is used on any server-side host.
state : Boolean = false [1] – Describes the state of a BOINC project; if it is activated *(true)* or deactivated *(false)*, i.e. execution is started or stopped.

Associations
operations : SystemOperation [*] – This association is used to add behaviours, e.g. "bin/start" or "bin/stop" of a BOINC project.

Informal and OCL Constraints

[1] «Project» is public.

```
visibility = VisibilityKind::public
```

[2] *operations* must have a minimum of three Operations, which are used to start, stop and restart a BOINC project.

```
operations->size() >= 3
```

[3] All operations must be extended by «SystemOperation».

```
operations->forAll(o | isStereotypedBy(o, SystemOperation))
```

[4] If *workingDirectory* has no value then a default directory is used; it can be defined individually by tool vendors.

6.5.11 ReplicationAssociation

«ReplicationAssociation» extends the UML metaclass `Association (from Kernel)` and is used to set-up replications of databases. Fig. 6.5 shows how «ReplicationAssociation» is used. This Association can be used between Ports extended by «Port», or to create an Association between NICs of two or more Components extended by «Host». In cases where a full database replication is intended on a single host, a «ReplicationAssociation» instance can be established directly between the database services. Fig. 6.6 shows a second scenario where replications are modelled on different hosts. Any host has an NIC and can be associated to BOINC's master/science databases (Section 6.5.4). It is not permitted to associate two replication databases.

UML MetaClass

Association (from Kernel) [125, Chapter 7.3.3]

Tag-Values & Associations

No additional tag-values and associations.

Informal and OCL Constraints

[1] An Association can be used as a link between «DB» and «NIC». A link between two «DB» instances is only allowed when the «DB» instances are in the same context.

```
memberEnd->size() = 2 implies
  if memberEnd->forAll(e1, e2 | e1.namespace = e2.namespace) then
    memberEnd->forAll(type | isStereotypedBy(type, DB)) or
    memberEnd->forAll(type | isStereotypedBy(type, NIC))
  else
    (isStereotypedBy(memberEnd->at(1), DB) AND
     isStereotypedBy(memberEnd->at(2), NIC)) or
    (isStereotypedBy(memberEnd->at(2), DB) AND
     isStereotypedBy(memberEnd->at(1), NIC))
  endif
```

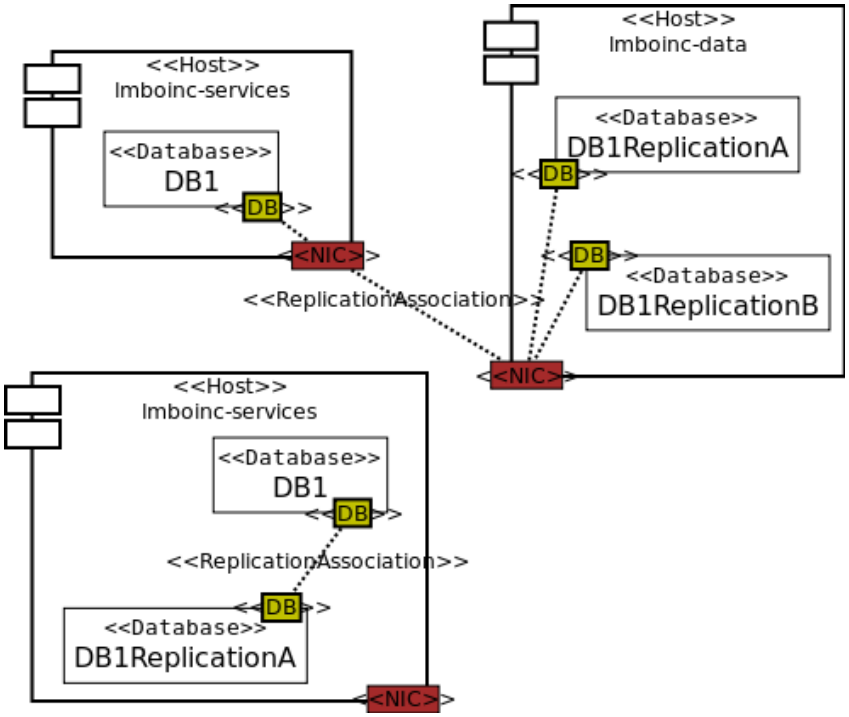

Figure 6.5: Two cases how «ReplicationAssociation» can be used: (i) to establish an association between NICs of two individual Components extended by «Host» and (ii) to associate two Ports extended by «DB» on a specific «Host».

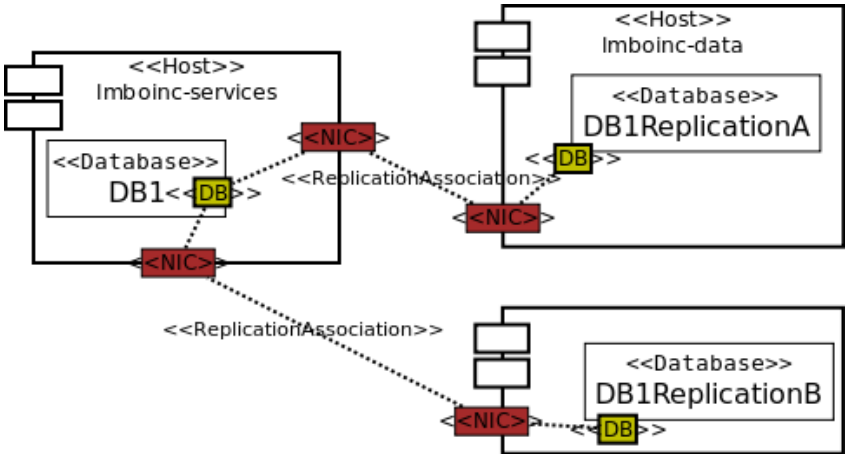

Figure 6.6: More than one database replication can be used for a database; they can be distributed to individual Components extended by «Host». This case needs associations between NICs.

6.5.12 SAN

«SAN» specialises «Host» and describes an external host with capabilities to store data, e.g. workunits or computational results; also known as a Storage Area Network (SAN).

UML MetaClass
Component (from BasicComponent, PackagingComponents) ; Specialises «Host»

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] *operatingSystem* is empty.

```
operatingSystem->isEmpty()
```

[2] BOINC services are not allowed, only exporting Ports are allowed.

```
packagedElement->size() = 0 and required->empty()
```

[3] «SAN» does not own BOINC sources.

```
main = false
```

6.5.13 Service

«Service» extends the UML metaclass `Component (from BasicComponents)` and is used to add BOINC services to a host, e.g. Feeder or Transitioner. It depends on the service type specified by *serviceType* which functionality a «Service» instance offers. A service instance can be manifested for different cases. It is necessary to specialise «Service» for the different possible BOINC services, i.e. if a Feeder is used then this component can be specialised as *Feeder*. As a consequence «Service» is abstract. «Service» provides the specification for different specialisations to allow models for different manifestations.

UML MetaClass
Component (from BasicComponents, PackagingComponents) [125, Chapter 8.3.1]

Tag-Values
state : Boolean [1] – Specifies that this BOINC service is started or has to be started.
serviceType : ServiceType [1] – Specifies the occurrence of the service.
command : Operation [1] – Specifies the behaviour of this service.
outputFile : String [0..1] – Filename of the file where output messages are stored during runtime.

pidFile : String [0..1] – Filename of the file where the service PID (process identifier) is stored, when the service is started.
debugMode : Integer = 2 [0..1] – Level of detail of debug messages; following values are allowed: *1 = minimal*, *2 = normal*, *3 = debug* or *4 = high*; default is *2*.

Associations
software : Software [*] – Associates software artefacts [125, Section 10.3.1] which are necessary for the execution of services. This association is used to add required software libraries or software packages which are used during the runtime of services, e.g. if computational results are packed in ZIP packages then ZIP must be available on the host where the relevant service is executed.

Operations
Service::start() : Boolean – Starts the execution of the service.
Service::stop() : Boolean – Stops the execution of the service.
Service::restart() : Boolean – Restarts the current service.

Informal and OCL Constraints
[1] «Service» is owned by a host.

```
isStereotypedBy(self.class, Host)
```

[2] This is an abstract component.

```
isAbstract = true
```

[3] *debugMode* must be between 1 and 4.

```
debugMode->notEmpty() implies debugMode >= 1 and debugMode <= 4
```

[4] required contains only «PortImport» when «Service» specifies an Assimilator.

```
serviceType = ServiceType::Assimilator implies
  required->forAll(p | isStereotypedBy(p, PortImport))
```

6.5.14 Service (specialised with *serviceType = ServiceType::Assimilator*)

This specialisation is used to describe Assimilators and implies *serviceType = ServiceType::Assimilator*. BOINC's default Assimilators can be used (Section 5.3.6). In addition the BOINC project can have individual assimilation behaviour. One host can own several assimilation services; each of them can have the same configuration. It is allowed to add similar assimilation services to different hosts.

Tag-Values
mod : Integer [2] – Defined by BOINC itself to select workunit results. The first value is substituted as N and the second value is substituted as R. These two substitutions are used within the formula $Workunit :: id \,\%\, N == R$ to process specific

workunit results (Section 6.8.9). This allows BOINC's assimilation processes to run for different sets of workunit results.
sleep : Integer = 5 [1] – Seconds between periodic execution; default is 5 seconds. If one call is still running, it will not execute again and the service will wait for another period.
onepass : Boolean = false – Exits after execution; sleep is neglected.
appname : String [1] – Assimilate workunits of this specific scientific application.

Informal and OCL Constraints
[1] This «Service» is an Assimilator.

```
serviceType = ServiceType::Assimilator
```

6.5.15 Service (specialised with *serviceType = ServiceType::Feeder*)

This specialisation is used to describe BOINC's Feeder and implies *serviceType = ServiceType::Feeder*. Feeder cannot be modelled by the BOINC project owner. BOINC's default Feeder is used and can be used for several scientific applications. The Feeder creates a shared-memory segment used to pass data to CGI scheduler processes (Section 5.3.2). This data includes applications, application versions, and workunit items (an unsent result and its corresponding workunit); the tag-values *apps*, *randomOrder*, *priorityOrder*, *priorityOrderCreateTime*, and *purgeState* are based on BOINC's wiki description [167, BackendPrograms].

Tag-Values
mod : Integer [2] – See Section 6.5.14.
sleep : Integer = 5 [1] – See Section 6.5.14.
onepass : Boolean = false – See Section 6.5.14.
allapps : Boolean = false – In case a BOINC project provides more than one scientific application, this BOINC service can be used for all of them (*allapps = true*) or only selected ones (*allapps = false*) specified by *apps*.
apps : String [0..]* – Get work only for the given named scientific applications.
randomOrder : Boolean = false [1] – Enumerate work items in order of increasing *result.random*; table field of the BOINC database table *result*.
priorityOrder : Boolean = false [1] – Enumerate work items in order of decreasing *result.priority*; table field of the BOINC database table *result*.
priorityOrderCreateTime : Boolean = false [1] – Enumerate work items in order of decreasing *result.priority*, then increasing *workunit.id*; both are table fields of BOINC database tables *result* and *workunit*.
wmod : Integer [0..]* – Similar to *mod*; in contrast *wmod* is used to select specific

workunits (not only the several workunit results).
purgeState : Integer = 60 [1] – Removes workunits from the shared-memory segment that have been there for longer than 60 minutes but have not been assigned to any client.

Informal and OCL Constraints
[1] This «Service» is a Feeder.

```
serviceType = ServiceType::Feeder
```

[2] Only *allapps* or *apps* is used.

```
(apps->notEmpty() implies allapps = false) or
(apps->empty()    implies allapps =  true)
```

6.5.16 Service (specialised with *serviceType = ServiceType::Validator*)

This specialisation is used to describe Validators and implies *serviceType = ServiceType::Validator*. BOINC's default Validators can be used (Section 5.3.5). In addition the BOINC project can have individual validation behaviour. The tag-values *maxClaimedCredit*, *maxGrantedCredit*, *grantClaimedCredit*, and *updateCreditJob* are based on BOINC's wiki description [167, ValidationIntro].

Tag-Values
mod : Integer [2] – See Section 6.5.14.
sleep : Integer = 5 [1] – See Section 6.5.14.
onepass : Boolean = false – See Section 6.5.14.
appname : String [1] – See Section 6.5.14.
maxGrantedCredit : Float – Grant no more than this amount of credit to a result.
updateCreditJob : Boolean – Updates the state of validated workunit items to BOINC's record database.
creditFromWU : Boolean – Credit is specified by the workunit definition (Section 6.8.9).
grantClaimedCredit : Float – Grant the claimed credit, regardless of what other results for this workunit are claimed.
maxClaimedCredit : Float – If a result claims more credit than this value, mark the workunit as invalid.

Informal and OCL Constraints
[1] This «Service» is a BOINC validator.

```
serviceType = ServiceType::Validator
```

6.5.17 Service (specialised with *serviceType = ServiceType::Webserver*)

If one or more webservers are modelled, they can be responsible for different duties, e.g. one host provides download handling of workunits or one host provides BOINC's Scheduler for user requests. In addition this «Service» allows specifying on which hosts web features are installed, e.g. message forums for user communication. *instances* allows specifying how many instances of a webserver are started during the initialisation. The higher the number of instances, the more precreated threads for handling of user requests are initialised [100]. *port* specifies the HTTP(S) connection port. HTTP-request's default port is 80; HTTPS-request's default port is 443 (*useSecure=true*). By default, this «Service» provides all web features, i.e. one instance provides interfaces for download and upload of workunits, handling of general client requests, a website for the individual BOINC project and a user forum.

Tag-Values

instances : Integer = 5 [1] – Number of parallel executed service threads.
port : Integer = 80 [1] – Specifies the IP port to listen for incoming connections.
useSecure : Boolean – Specifies if Secure-Socket Layer (SSL) connections are used.
isMaster : Boolean = true – Specifies if this webserver component is the main host of a BOINC project, in this case this webserver is used for the BOINC master URL where the main communication elements of a BOINC project are installed, e.g. website of the project (Section 5.3.2).
isScheduler : Boolean = true – Specifies if the host provides the BOINC Scheduler.
isDownload : Boolean = true – Specifies if the host provides downloads of workunits.
isUpload : Boolean = true – Specifies if the host allows uploading of workunit results.
isForum : Boolean = true – Specifies if the host provides a user forum.

Informal and OCL Constraints

[1] This service is a webserver.

```
serviceType = ServiceType::Webserver
```

[2] *port* for network communications must be set.

```
serviceType = ServiceType::Webserver
  implies self.port > 0 and self.port <= 65535
```

[3] Only one instance can be the master of a BOINC project.

```
allInstances()->select(isMaster=true)->size() = 1
```

6.5.18 ServiceType

«ServiceType» extends the UML metaclass `Enumeration (from Kernel)` and is used to specify the type of a BOINC service, e.g. Feeder or Transitioner. There is a dependency between the service type and the attributes and functionalities of a BOINC service (Section 6.5.13 – 6.5.17). These stereotypes have a minimum set of six enumeration literals for specifying BOINC's default services.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

Informal and OCL Constraints
[1] All enumeration literals must be unique.

```
ownedLiteral->forAll(el1, el2 | el1->isUnique(el2))
```

[2] «ServiceType» has a minimum set of enumeration literals.

```
ownedLiteral->contains(Set{
  Assimilator, Feeder, Task, Transitioner, Validator, Webserver
})
```

6.5.19 Share

«Share» extends the UML metaclass `Interface (from Interfaces)` and specifies where exported directories of individual hosts are imported on individual hosts. Exported directories are specified as provided Interfaces. Otherwise they are specified as required Interfaces. In case the parent Port of a «Share» is a «Host» and configured as the main host of a BOINC project, the values of the connected Interfaces are set automatically by the current «Share», i.e. the *mountpoint* on all hosts is similar and the same share protocol specified by *shareType* is used.

UML MetaClass
Interface (from Interfaces) [125, Chapter 7.3.24]

Tag-Values
shareType : ShareType [1] – Specifies which technology is used for sharing.
shareName : String [0..1] – A specific name for a «Share».
path : String [1] – The directory which is exported.
mountpoint : String [1] – Mountpoint of the exported directory.

Associations
No additional associations.

Informal and OCL Constraints
[1] Instances must be visible within Packages extended by «Project».

```
visibility = VisibilityKind::protected
```

[2] *path* and *mountpoint* must have valid formats which are used by the operating systems on the target hosts, e.g. `/media/files` on Linux systems and `c:\media\files` on Windows systems.

6.5.20 ShareType

«ShareType» extends the UML metaclass `Enumeration (from Kernel)` and specifies the share protocol between Components extended by «Host», e.g. Network File System (NFS) or SAMBA.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

Tag-Values
version : String [0..1] – Specifies which version is used for the different technologies, e.g. it can be necessary for NFS to specify a special version, because NFSv4 requires TCP but when only UDP is necessary, NFSv2 or NFSv3 are adequate.

Associations
No additional associations.

Informal and OCL Constraints
[1] If version is not valued, then the highest available version on a «Host» is used.

```
context Enumeration
pre: version->isEmpty()
def: getVersion() : String
body:
  -- Retrieve highest version of current
  -- EnumerationLiteral, e.g. NFSv4 if literal is NFS.
```

6.5.21 Software

«Software» extends the UML metaclass `Artifact (from Artifacts, Nodes)`.

UML MetaClass
Artifact (from Artifacts, Nodes) [125, Chapter 10.3.1]

Tag-Values
version : String [1] – Specifies which version this «Software» is.

Associations
No additional associations.

Informal and OCL Constraints
No additional constraints.

6.5.22 SystemOperation

«SystemOperation» extends the UML metaclass `Operation (from Kernel, Interfaces)` and specifies system or BOINC service routines, e.g. routines for Validator or Assimilator.

UML MetaClass
Operation (from Kernel, Interfaces) [125, Chapter 7.3.37]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] Only available and executable binaries and scripts can be extended by «SystemOperation».

6.5.23 Table

«Table» extends the UML metaclass `Interface (from Interfaces)` and allows a BOINC service to access database tables.

UML MetaClass
Interface (from Interfaces) [125, Chapter 7.3.24]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] name is used as the name for a table in a database.
[2] Properties are extended by «Column».

```
ownedAttribute->forAll(a | isStereotypedBy(s, Column))
```

6.5.24 UserOperation

«UserOperation» extends the UML metaclass `Operation (from Kernel, Interfaces)` and specifies operations which users can execute.

UML MetaClass
Operation (from Kernel, Interfaces) [125, Chapter 7.3.37]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] Only existing users can execute these operations.

6.6 Application

The Application package has the focus to cover BOINC related functionalities by a specific set of UML elements. In addition, this package does not give a strict boundary of allowed UML elements, i.e. this package introduces stereotypes for a handful of specialised elements which are applicable for handling of BOINC's framework functions or for performing inter-process communication. UML4BOINC's Application package can be used to model the following aspects of a BOINC project:

1. A scientific application can be created by the use of a set of UML Activities and UML Actions. In contrast, a legacy-application can be added to the BOINC project. In both cases, any application is described as a platform independent version. The model transformation to an executable code recognises the information of the Infrastructure package, i.e. the Infrastructure package is used to specify which target architectures and platforms are supported.

2. The scientific application can be assembled with periodically executed functionalities.

3. Atomic functions can be created, i.e. they cannot be interrupted and must be finished before any events are handled by subsequent actions.

4. Asynchronous messages can be transmitted from BOINC's client- and server-side to the opposite side.

5. The current execution mode can be checked, i.e. if the scientific application is executed within a BOINC-client context.

6. It can be checked if a network connection is available.

7. Input and output files can be associated within UML Activities, i.e. any workunit related file and checkpoint file can be modelled as sets of parameters and are usable via UML ObjectFlows.

8. The state of the current process can be described by the use of a UML Statemachine. Therefore six stereotypes in the Section 6.6.14 to 6.6.19 are specified. Fig. 6.8 shows the relation between these stereotypes.

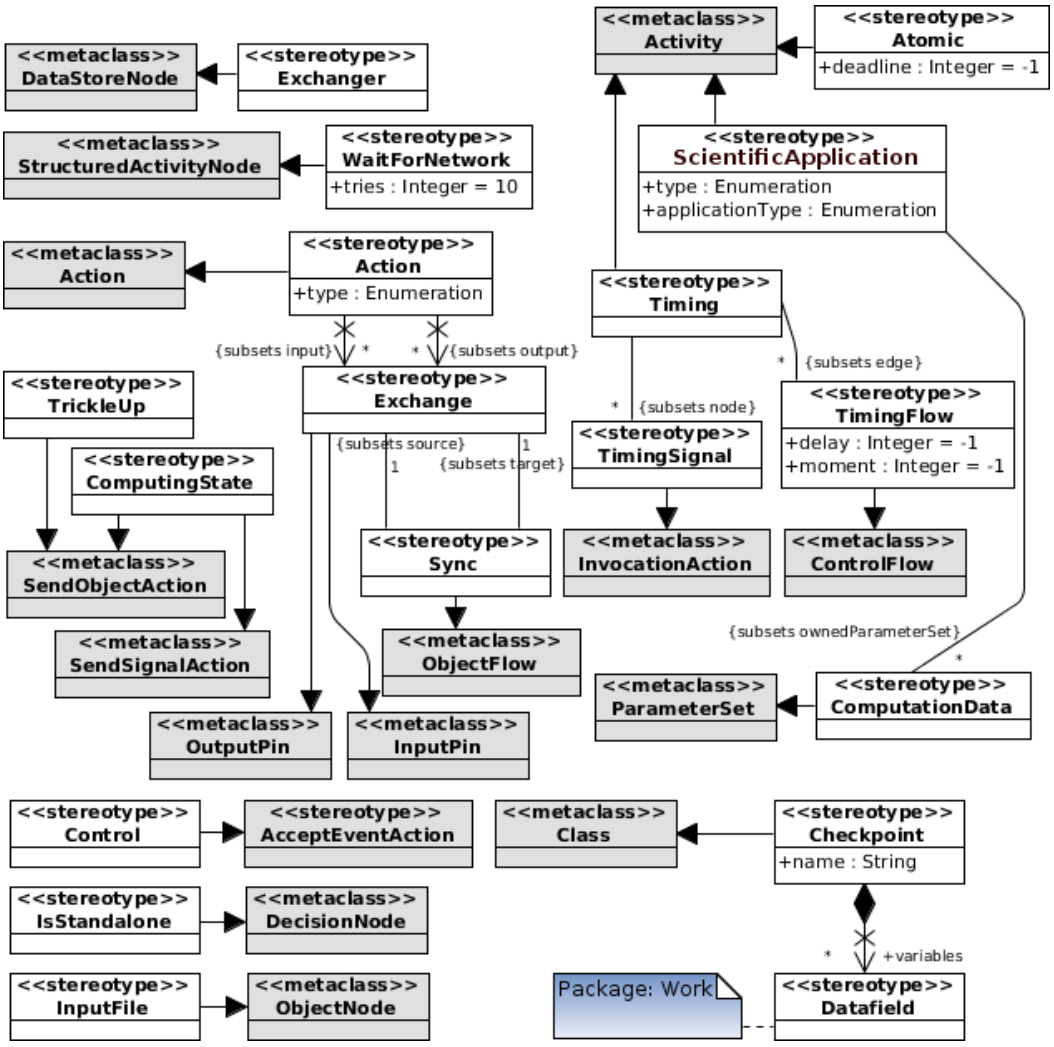

Figure 6.7: Stereotypes for the creation of scientific applications, BOINC services, and tasks.

6.6.1 Action

«Action» extends the UML metaclass `Action (from CompleteActivities, FundamentalActivities, StructuredActivities, CompleteStructuredActivities)` and is used as an abstraction for BOINC's framework API. «Action» allows the abstraction of different functionalities by use of just this single stereotype. The execution behaviour can be modified by *type*. As a consequence, *type* specifies how many InputPins and OutputPins an Action has. It is assumed that «Action» is performed without any failures, but under any circumstances an ExceptionHandler can be associated to any instance of «Action». Specialisations of «Action» are allowed, e.g. to add more recognitions of different file types when *type* is *FileInfo*. Fig. 6.9 shows an overview of the base

Enumerations and Classes which are required for the different «Action» types. Section 7.3 shows how these different specifications can be used.

UML MetaClass
Action (from CompleteActivities, FundamentalActivities, StructuredActivities, CompleteStructuredActivities) [125, Chapter 12.3.2]

Tag-Values
type : Kernel::Enumeration – Specifies «Action»'s behaviour; five types are pre-specified by UML4BOINC:

- *#Locking* – Is used to lock/unlock one file to allow/not allow access. This Action has one InputPin which is used as the file path and one output pin which provides the current lock state.

- *#FileInfo* – Queries file information, e.g. the file size. This Action has one

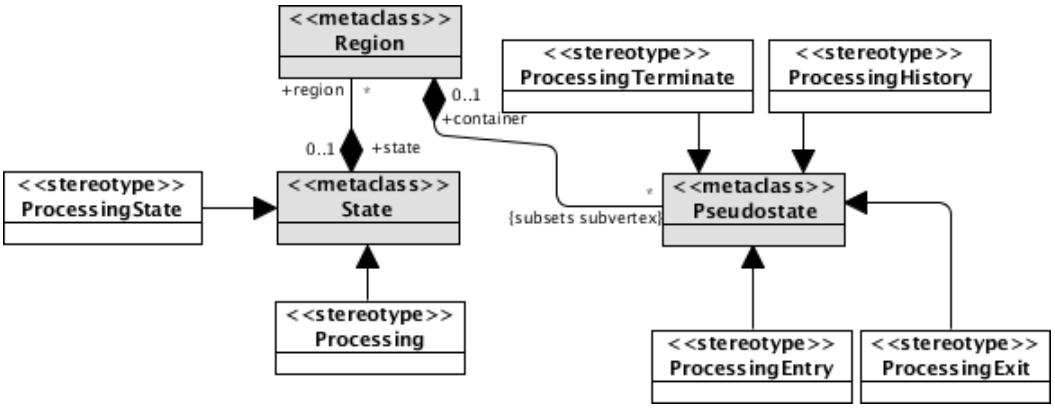

Figure 6.8: Stereotypes for the creation of state machines which are used to handle different states during the execution of a scientific application.

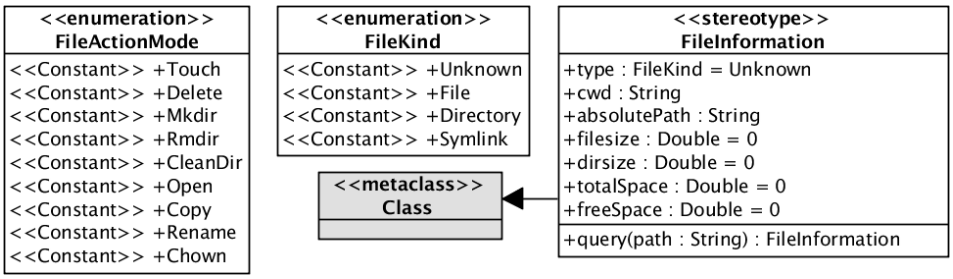

Figure 6.9: The Enumeration of the left-hand side is used to specify the execution mode of «Action»; Section 6.6.9 describes the enumeration literals in more details. The Enumeration in the middle is used to describe the type of a specific file when file information is queried by «Action» and *type* is *#FileInfo*; «FileInformation» keeps the queried file information.

InputPin which is used as a file path and one OutputPin to deliver the file information to the next action.

- *#FractionDone* – Forwards the current fraction done value of the computation to the BOINC client. This Action has one InputPin which is used to deliver the fraction done value.

- *#TrickleDown* – Handles asynchronous messages. This Action has one OutputPin which contains the message token.

- *#FileHandling* – This Action has three InputPins: (1) to set the mode of file handling, (2) to set the file path, and (3) to add parameters related to the used mode.

- *#Timing* – This Action is used within «Timing» for periodically execution and has no InputPins or OutputPins.

It depends on *type* as to how many InputPins and OutputPins this Action has.

Associations
No additional associations.

Informal and OCL Constraints
[1] *type* specifies how many InputPins and OutputPins this action has, how the pins are typed, and named.

```
if type = #Locking then
  input->size() = 1 and output->size() = 1 implies
   input->at(1).type = PrimitiveType::String
    and input->at(1).name = "path"
    and output->at(1).type = PrimitiveType::Boolean
    and output->at(1).name = "state"
else
  if type = #FileInfo then
    input->size() = 1 and output->size() = 1 implies
     input->at(1).type = PrimitiveType::String
      and input->at(1).name = "path"
      and isStereotypedBy(output->at(1).type, FileInformation)
      and output->at(1).name = "finfo"
  else
    if type = #FractionDone then
      input->size() = 1 and output->size() = 0 implies
       input->at(1).type = PrimitiveType::Real
    else
      if type = #TrickleDown then
        input->size() = 0 and output->size() = 1 implies
         isStereotypedBy(output->at(1).type, TrickleMessage)
      else
        if type = #FileHandling then
          -- Separate constraints!
          input->size() >= 2 and output->size() = 1 implies
            input->at(1).name = "mode"
             and oclIsKindOf(input->at(1), ValuePin)
             and oclIsKindOf(input->at(1).value.type, FileActionMode)
```

```
            and input->at(2).name = "path"
            and input->at(2).type = PrimititveType::String
        else
          if type = #Timing then
            input->size() = 0 and output->size() = 0
          endif
        endif
      endif
    endif
  endif
endif
```

[2] When *type* is *#FileHandling* the number of input pins and their names are related to mode.

```
if type = #FileHandling then
        v : input->at(1).value;
  states2p : Set{Touch, Delete, Mkdir, Rmdir, CleanDir};
  states3p : Set{Rename, Copy, Open, Chown};

  if states2p->contains(v.value) then
   -- nothing special
  else
    if v.value = #Copy or v.value = #Rename then
      input->at(3).name = "newpath" and
      input->at(3).type = PrimititveType::String
    else
      if v.value = #Chown then
        input->at(3).name = "owner" and
        input->at(3).type = PrimititveType::Integer
      else
        if v.value = #Open then
          input->at(3).name = "filemode" and
          input->at(3).type = PrimititveType::String
        else
          -- Something is wrong ;-(
        endif
      endif
    endif
  endif

endif
```

[3] When *type* is *#Timing* the incoming flow must have ForkNode or InitialNode as source.

```
  type = #Timing and incoming->size() = 1 implies
    incoming->at(1).source.oclIsKindOf(ForkNode) or
    incoming->at(1).source.oclIsKindOf(InitialNode)
```

6.6.2 Atomic

«Atomic» extends the UML metaclass `Activity (from from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)` and is used as a safeguard for non-interruptible Action executions. An atomic Activity cannot be interrupted by any external event,

when it is entered it has to be finished. When an atomic activity is entered the Activity has to be performed in total before any new received events are handled. In case the atomic Activity is performed as an opaque thread, any received event can be performed by other threads. In addition they will no longer be available for later Actions which follow until the atomic Activity. It is necessary that all actions have an outgoing edge for subsequent actions. Otherwise it is possible that an external action is invoked and «Atomic» is not freed, i.e. the operation is still in process and cannot be executed again.

UML MetaClass
Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities) [125, Chapter 12.3.4]

Tag-Values
deadline : Integer = -1 – Any Activity extended by «Atomic» has a deadline, which is used to specify when «Atomic» has to be finished at the latest. The deadline is endless when the default value is used.

Associations
No additional associations.

Informal and OCL Constraints
[1] All included InvocationsActions must have an outgoing edge.
[2] When an exception is raised «Atomic» will process forward until the FinalNode is reached. It is not guaranteed that the exception is still valid until «Atomic» has completed.

6.6.3 Checkpoint

«Checkpoint» extends the UML metaclass `Class (from Kernel)` and is used to specify how checkpoint files are structured. Fig. 6.10 shows the simplified structure of checkpoint specifications. *Datafield* is borrowed from the workunit UML4BOINC profile specification and is used with the same restrictions.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
name : String [1] – The file name of the checkpoint file.

Associations
variables : Datafield [*] – A checkpoint file can have several data fields and each can have several sub-elements. *type* is used to specify the type of the data field. In case *type* is valued with *FileType::File* a physical file is used and the related association cannot have sub-elements (Section 6.8.1).

Informal and OCL Constraints
No additional constraints.

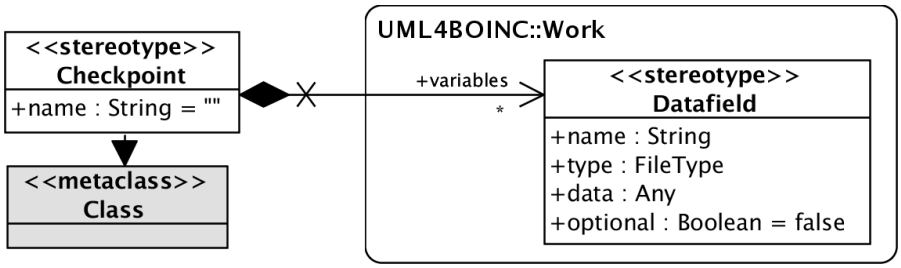

Figure 6.10: «Checkpoint» uses the same specification elements for the structure as used for workunit specification. Different types of the data fields are possible. If the type is *FileType::File* then a physical file is used and the opening process is responsible for the current opening, otherwise «Datafield» offers the current opening functionality (Section 6.8.1).

6.6.4 ComputationData

«ComputationData» extends the UML metaclass `ParameterSet (from CompleteActivities)` and is used for input and output data objects and tokens for «ScientificApplication» instances typed as *#Application*, *#Wrapper* or *#Task*. The Parameters of ParameterSets can be used as input and output parameters. The associated Parameters are used to offer an interface between «InputFile» and «ScientificApplication»; the same principles are used for the result files of a computation. The use of external ObjectNodes is not required, the Parameters themselves can be typed with «Input» or «Checkpoint» extended data types. The only reason for the use of a ParameterSet is to make the visualisation within the Application diagram more compact. It would be possible to use several Parameters, but ParameterSet makes it possible to combine all necessary data values and tokens. The number of Parameters within «ComputationData» is related to the number of modelled «Input» files of a workunit and if a checkpoint is available. Also the number of Parameters for computational results is related to the number of modelled «Result» files of a workunit.

UML MetaClass
ParameterSet (from CompleteActivities) [125, Chapter 12.3.43]

Tag-Values and Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] For input data the number of Parameters is the number of input files of a workunit plus one (checkpoint) and for output data the number of Parameters is the number of result files of a workunit.

[2] All Parameters in this set can have an incoming object flow which is connected to an ObjectNode extended by «InputFile» or the owned Parameters are typed with «Input» or «Checkpoint» data types.

6.6.5 ComputingState

«ComputingState» extends the UML metaclasses `SendObjectAction (from IntermediateActions)` and `SendSignalAction (from BasicActions)` and is used to announce that the scientific application is finished. It is used by «ScientificApplication» typed as: *#Application* or *#Wrapper*. A screensaver or tasks do not need any invocation actions, because it has no relevance in the context of the computation handling. «ScientificApplication» can only send two different state descriptions:

- for signalling that the computation is finished, or
- to signal that the computation failed.

#Suspended could be a third state but this state cannot be controlled by the scientific application itself, only external control events can be used to suspend any «ScientificApplication» instance.

UML MetaClass
SendObjectAction (from IntermediateActions) [125, Chapter 11.3.44]
SendSignalAction (from BasicActions) [125, Chapter 11.3.45]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «ComputingState» can only have incoming edges from Activities extended by «ScientificApplication».

```
incoming->forAll(i | isStereotypedBy(i, ScientificApplication))
```

[2] No outgoing edges are allowed.

```
outgoing->size() = 0
```

[3] Only two instances are allowed.

```
allInstances()->size() <= 2
```

6.6.6 Control

«Control» extends the UML metaclass `AcceptEventAction (from CompleteActions)` and is used to control the execution of scientific applications

extended by «ScientificApplication» and typed as *#Application* or *#Wrapper*. The events themselves do not provide any restrictions on how the events must be handled by the scientific application. An application's implementation itself is responsible for this. It is not necessary to add any «Control» events, in case an event is sent but no receivers are available the events are mapped as followed:

- event(*Suspend*): The scientific application will start at the initial point of the activity diagram.
- event(*Resume*): The scientific application will resume at the initial point of the activity diagram.
- event(*Quit*): The related scientific application activity is exited and no computation result is created.
- event(*Abort*): The related scientific application activity is exited and no computation result is created.

UML MetaClass
AcceptEventAction (from CompleteActions) [125, Chapter 11.3.2]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «Control» can be used to handle four different events: *Suspend*, *Resume*, *Quit* and *Abort* as discussed in Section 5.4.

```
self.name = "Suspend" or self.name = "Resume" or
self.name = "Quit"    or self.name = "Abort"
```

[2] «Control» must be owned by an Activity which is extended by «ScientificApplication» and which is not typed as a *Screensaver*.

```
isStereotypedBy(context, ScientificApplication) implies
  context.type <> Screensaver
```

6.6.7 Exchange

«Exchange» extends the UML metaclasses `InputPin (from CompleteStructuredActivities)` and `OutputPin (from CompleteStructuredActivities, StructuredActivities)`. These UML metaclasses are only allowed for data object and token exchange between an instance of «ScientificApplication» and an instance of «Exchanger». During runtime the pins always have the up-to-date data objects or tokens which are usable within

the implementation of «ScientificApplication» or can be read by other instances of «ScientificApplication». At any time the data objects and tokens are up-to-date to all associated instances of «ScientificApplication». OutputPins extended by «Exchange» are never used as an output of the control flow. UML defines Actions as "[...] when completed, an action execution offers any object tokens that have been placed on its output pins and control tokens on all its outgoing control flows (implicit fork), [...]" [125, p.321]. Under any circumstances the output pins extended by «Exchange» are never used as transition for the control flow, an available output pin (not extended by «Exchange») or an outgoing control flow is used always. It is not explicitly specified by UML4BOINC that an Action must have an outgoing control flow or OutputPin to guarantee a fully operable Activity.

UML MetaClass
InputPin (from CompleteStructuredActivities) [125, Chapter 12.3.32]
OutputPin (from CompleteStructuredActivities, StructuredActivities) [125, Chapter 12.3.40]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] Instances of «Exchange» can only be used by Actions when they are owned by instances of «ScientificApplication».

```
isStereotypedBy(self.owner, ScientificApplication)
```

[2] PortOutput can only be used by Actions when the owning Activity has *type* valued as *#Application* or *#Wrapper*.

```
self.owner->applicationType = #Wrapper or
self.owner->applicationType = #Application
```

[3] OutputPins extended by «Exchanger» are never used as control flow exit or object flow exit.

6.6.8 Exchanger

«Exchanger» extends the UML metaclass `DataStoreNode (from IntermediateActivities)` and is used to share data between different «ScientificApplication» instances within a BOINC slot. Only applications within a BOINC slot are allowed to access «Exchanger». Section 7.3 describes how this stereotype has to be used.

UML MetaClass
DataStoreNode (from IntermediateActivities) [125, Chapter 12.3.21]

Tag-Values & Associations

No additional tag-values and associations.

Informal and OCL Constraints
[1] The objects are ordered and the access is possible via an *index* or a *name* (index range is defined in the OCL [127, Section 11.6.3]).

```
self.ordering = ordered
```

[2] This object node can be used in three states: (1) *#Computing*, (2) *#Reporting* and (3) *#Checkpointing*.

```
self.inState->size() = 3 implies
  self.inState->including(Set{
    "Computing", "Reporting", "Checkpointing"
  })
```

[3] «Exchanger» is only visible for instances within the state *OnClient*.

6.6.9 FileActionMode

«FileActionMode» is an enumeration of the following values as shown in Fig. 6.9:

- *#Touch* – A file is created or the timestamp is updated at the point of time of the call (when the file exists).
- *#Delete* – Removes a file.
- *#Mkdir* – Creates a new directory.
- *#Rmdir* – Removes an available directory.
- *#CleanDir* – Removes the content of a directory.
- *#Open* – Opens a file for writing and reading.
- *#Rename* – Changes the name of a file or directory.
- *#Chown* – Changes the file owner and assigned group.

BOINC itself supports a generalized interface for the use on Windows, Linux and Mac OS X systems. «FileActionMode» can be mapped directly between the UML model and BOINC's functionalities.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

Informal and OCL Constraints
[1] «FileActionMode» cannot be extended by enumeration literals.

6.6.10 FileInformation

«FileInformation» extends the metaclass `Class (from Kernel)` and is used for storing file information, e.g. the byte size a specific physical file has. This Class is filled by «Action» when the tag-value *type* is set to *#FileInfo*.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
type : FileKind = #Unknown [1] – Type of file, e.g. directory or symbolic-link.
cwd : String– The current working directory; not related to the file.
filesize : Integer = 0 – The file size in bytes.
dirsize : Integer = 0 – The directory size in bytes.
absolutePath – The absolute path of the file, i.e. `/home/cr/file1` on UNIX systems or `c:\home\cr\file1` on Windows systems.
freeSpace : Integer = 0 – The free and open for use disc space in bytes of the device which provides the file.
totalSpace : Integer = 0 –The total available disc space in bytes of the device which provides the file.

Operations
FileInformation::query(path : String) : FileInformation – Used to query file information which are stored in FileInformation's tag-values.

Informal and OCL Constraints
No additional constraints.

6.6.11 FileKind

«FileKind» is an enumeration of the following values as shown in Fig. 6.9:

- *#Unknown* – The type of the file is unknown.
- *#File* – It is a general file.
- *#Directory* – The file is a directory.
- *#Symlink* – The file is a symbolic-link.

It depends on the underlying operating system how many file types are possible, i.e. UNIX systems support file based FIFOs (First-in-First-out) or MMAPs (memory-mapped file input/output). Here, only three specific file types are recognised, others are typed as *#Unknown*. Additional type checks can be added in specialisation of «FileKind» and «Action». Furthermore, symbolic-links are not supported on Windows systems, as a consequence *#Symlink* can be set to *#Unknown*. «FileKind»

is only used once by «FileInformation» in Section 6.6.10.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

Informal and OCL Constraints
[1] This Enumeration can be extended by enumeration literals to add more file types.
[2] *#Symlink* is not available on Windows systems.

6.6.12 InputFile

«InputFile» extends the UML metaclass `ObjectNode (from BasicActivities, CompleteActivities)` and is used for the specification of input files of an «ScientificApplication» instance. How input files are processed is decided by the *Datafield::FileType* values of «Datafield» which are associated by «Input». For primitive datatypes UML4BOINC supports them out of the box [126], for additional specialised datatypes the operation *Datafield::open()* is defined and can be used. This ObjectNode has no selection behaviour. All data objects and tokens are available within the scientific application context. This stereotype must have an outgoing object flow which can be charged with selection behaviour. At any time «InputFile» allows only one data token, i.e. this is the physical file (input or checkpoint) itself which has its own structure specification as presented in Section 6.8.1.

UML MetaClass
ObjectNode (from BasicActivities, CompleteActivities) [125, 12.3.38]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This ObjectNode can only be stereotyped by «Input» or «Checkpoint».

```
isStereotypedBy(self.type, Input) or
isStereotypedBy(self.type, Checkpoint)
```

[2] This ObjectNode is read-only, i.e. no incoming object flows are allowed.

```
self.incoming->isEmpty() and
self.outgoing->notEmpty()
```

[3] Only one data token is allowed (the physical input file).

```
self.upperBound = 1
```

[4] This ObjectNode has no selection behaviour. The selection is controlled by outgoing object flows.

```
self.selection->isEmpty()
```

[5] This ObjectNode can only be used in the states *OnClient* (SPoV) and *Computing* (CPoV).

```
inState->including(Set{OnClient, Computing})
```

6.6.13 IsStandalone

«IsStandalone» extends the UML metaclass `DecisionNode (from IntermediateActivities)` and is used for checks if the current execution is in a namespace of the BOINC client. It is possible that a scientific application is executed without any BOINC control mechanism, i.e. to enable one to check the runtime behaviour during the development of a scientific application. «IsStandalone» allows the creation of computational flows with different control flows; one can be used with necessary BOINC functionalities, and in case the scientific application is not in the namespace of a BOINC project the flow can be created with dummy functions for BOINC methods, e.g. with default values. UML4BOINC does not restrict the use to where the outgoing edges are connected with the same following Actions.

UML MetaClass
DecisionNode (from IntermediateActivities) [125, Chapter 12.3.22]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «IsStandalone» must have two outgoing edges. One of these edges must have a *guard* valued by *true* and the other must be valued with *false*.

```
self.outgoing->size() = 2 implies
  self.outgoing->select(o | o.guard = true)->size() = 1 and
  self.outgoing->select(o | o.guard = false)->size() = 1
```

[2] For the decision no input data is necessary.

```
self.decisionInputFlow->empty()
```

6.6.14 Processing

«Processing» extends the UML metaclass `State (from BehaviorStateMachines, ProtocolStateMachines)` and is used to model the behaviour of how asynchronous messages, checkpointing and computational processing are handled. Fig. 6.8 gives an overview of the relation between the stereotypes in Sections 6.6.14 to 6.6.19.

UML MetaClass
State (from BehaviorStateMachines, ProtocolStateMachines) [125, Chapter 15.3.11]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This state has a maximum of three and a minimum of two regions.

```
region->size() >= 2 and region->size() <= 3
  implies isOrthogonal = true
```

[2] The individual regions do not own a final state.

```
region->forAll(subvertex->select(oclIsKindOf(FinalState))->size() = 0)
```

[3] Only one instance of «Processing» is allowed for a scientific application.

```
allInstances()->size() = 1
```

[4] Outgoing transitions must use an exit point extended by «ProcessingExit».
[5] «Processing» is a composite and orthogonal state. It must have an entry or an exit Pseudostate.

```
isComposite = true and isOrthogonal = true and
isSubmachineState = false and isSimple = false implies
  connectionPoint->notEmpty()
```

[6] This state does not have any behaviour when it is entered, exited or in use. The behaviour is described by its owned regions and states.

```
doActivity->empty() and entry->empty() and exit->empty()
```

6.6.15 ProcessingEntry

«ProcessingEntry» extends the UML metaclass `Pseudostate (from BehaviorStateMachines)` and is used as an entry point for the computational region of «Processing».

UML MetaClass
Pseudostate (from BehaviorStateMachines) [125, Chapter 15.3.8]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This Pseudostate describes an entry point.

```
kind = PseudostateKind::entryPoint
```

[2] The owner of this entry Pseudostate must be extended by «Processing».

```
isStereotypedBy(self.state, Processing)
```

6.6.16 ProcessingExit

«ProcessingExit» extends the UML metaclass `Pseudostate (from BehaviorStateMachines)` and is used as an exit point for the computational region of «Processing».

UML MetaClass
Pseudostate (from BehaviorStateMachines) [125, Chapter 15.3.8]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This Pseudostate describes an exit point.

```
kind = PseudostateKind::exitPoint
```

[2] The owner of this entry pseudostate must be extended by «Processing».

```
isStereotypedBy(self.state, Processing)
```

6.6.17 ProcessingHistory

«ProcessingHistory» extends the UML metaclass `Pseudostate (from BehaviorStateMachines)` and is used as a shallow or deep history point for the computational region of «Processing». It describes the initial point when a checkpoint is created and the scientific application is resumed.

UML MetaClass
Pseudostate (from BehaviorStateMachines) [125, Chapter 15.3.8]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This Pseudostate describes a history point.

```
kind = PseudostateKind::shallowHistory or
kind = PseudostateKind::deepHistory
```

[2] The owner of this entry pseudostate must be extended by «Processing».

```
isStereotypedBy(self.state, Processing)
```

[3] Only one history state can exist.

```
allInstances->size() <= 1
```

6.6.18 ProcessingState

«ProcessingState» extends the UML metaclass `State (from Behavior-StateMachines, ProtocolStateMachines)`. A scientific application can have several states. It depends on the type of the scientific application which states are necessary to model. Section 7.3 describes some possible scenarios and how «ProcessingState» can be used.

UML MetaClass
State (from BehaviorStateMachines, ProtocolStateMachines) [125, Chapter 15.3.11]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «ProcessingState» has no regions.

```
region->size() = 0
```

[2] «ProcessingState» is not composite nor a submachine state. As a consequence «ProcessingState» cannot have any entry or exit points and describes a simple state.

```
isComposite = false and isOrthogonal = false and
isSubmachineState = false
  implies
    isSimple = true      and
    connection->empty() and
    connectionPoint->empty()
```

[3] This state has a behaviour.

```
doActivity->notEmpty()
```

[4] The owner is stereotyped by «OnClient» or «Processing» when the state is part of a region.

```
if self.region->notEmpty() then
  region->forAll(isStereotypedBy(stateMachine, Processing))
else
  isStereotypedBy(submachine, OnClient)
endif
```

[5] A minimum of two exit pseudostates must exist to allow the handling of finished or suspended computations. Section 7.3 shows an example of how this constraint is used by a scientific application.

```
allInstances()->size() >= 2
```

6.6.19 ProcessingTerminate

«ProcessingTerminate» extends the UML metaclass `Pseudostate (from BehaviorStateMachines)` and is used to terminate the execution of a sci-

entific application. When this pseudostate is used to terminate the execution, no BOINC checkpoint has to be created, and no clean-up handling is called. This termination is comparable with the kill call on UNIX systems [94].

UML MetaClass
Pseudostate (from BehaviorStateMachines) [125, Chapter 15.3.8]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] Only one termination state is allowed on the client-side.

```
allInstances()->size() <= 1
```

[2] The termination state is owned by the Activity which is extended by «OnClient».

6.6.20 ScientificApplication

«ScientificApplication» extends the UML metaclass `Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)` and is used to specify a scientific application. During runtime of a BOINC project, only one instance of «ScientificApplication» is necessary, but this one can be available in different versions for different platforms, e.g. version 1.0 for Windows systems or version 1.3 only for Linux systems. Within this Activity the use of ActionNodes is not restricted, all structures are allowed. The call of this «ScientificApplication» is restricted, it can only be executed once and additional invocations will not create and execute additional «ScientificApplication» instances. Furthermore, «ScientificApplication» can only be invoked on CPoV. «ScientificApplication» does not specify how the specified *applicationType* is used, it is assumed that the vendor of code generation systems specifies any relevant enumeration literals which are supported by individual systems. *applicationType* is used to specify which independent-software vendor (ISV) application has to be wrapped.

UML MetaClass
Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities) [125, Chapter 12.3.4]

Tag-Values
type : Kernel::Enumeration – This tag-value is used to specify which kind of «ScientificApplication» is modelled. Default values are: *#Application*, *#Wrapper*, *#Task*, and *#Screensaver*.
applicationType : Kernel::Enumeration – This tag-value is recognised when *type* is *#Wrapper*. It is used to specify which ISV application is wrapped.

Associations
No additional associations.

Informal and OCL Constraints
[1] Minimum enumeration literals for *type* are: *#Application*, *#Wrapper*, *#Task*, and *#Screensaver*.

```
self.type->including(Set{Application, Wrapper, Task, Screensaver})
```

[2] *applicationType* is recognised only when type is *#Wrapper*.
[3] «ScientificApplication» can only own one ParameterSet and this must be stereotyped as «ComputationData».

```
self.ownedParameterSet->size() = 1 implies
  isStereotypedBy(self.ownedParameterSet->at(1), ComputationData)
```

[4] The Activity can only be invoked one time and it cannot be called again when it is still running.

```
self.isSingleExecution = true and self.isReentrant = false
```

[5] Only one instance of «ScientificApplication» is allowed.

```
self.allInstances()->size() = 1
```

[6] «ScientificApplication» allows manipulating non-local variables when a DataStoreNode extended by «Exchanger» is available for data exchange between applications, i.e. a scientific application and a screensaver.

```
self.edge->select(e |
  isStereotypedBy(e.target, Exchanger))->size() = 1
    implies self.isReadOnly = false
```

[7] Each «ScientificApplication» instance with a different *type* exists only once.

```
allInstances()->forAll(t | allInstances->select(type = t)->size() <= 1)
```

[8] Only two instances of «ScientificApplication» are allowed and one of them must be used for a screensaver.

```
allInstances()->size() = 2 implies
  allInstances()->select(
    a | a.type = Screensaver
  )->size() = 1
```

[9] This Activity and any specialisation cannot be called directly by other Activities or Actions.

6.6.21 Sync

«Sync» extends the UML metaclass `ObjectFlow (from BasicActivities, CompleteActivities)` and is used to exchange data objects and tokens between scientific applications in BOINC's slots on the CPoV. In Activity diagrams the use of two outgoing flows (ControlFlow and ObjectFlow) are allowed.

ObjectFlows are only used to store data objects or tokens in the DataStoreNode and the execution of the scientific application will step forward. The used transition must be a ObjectFlow or a ControlFlow and should not be extended by «Sync». The selection of the next flow is determined by the guards of the flows. Storing of data objects and tokens is only allowed by instances of «ScientificApplication» when they are typed as *#Wrapper* or *#Application*. Restoring of data objects and tokens is allowed by all instances of «ScientificApplication». *#Screensaver*s can only restore data objects and tokens and never have storing access to a DataStoreNode. «Sync» can be used between UML Activities extended by «ScientificApplication» or OutputPins extended by «Exchange» and InputPins extended by «Exchange» of UML Actions.

UML MetaClass
ObjectFlow (from BasicActivities, CompleteActivities) [125, Chapter 12.3.37]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] It is only permitted to send or receive data objects or tokens on a single flow, i.e. only one sender and receiver.

```
self.isMulticast = false and self.isMultireceive = false
```

[2] If the source of «Sync» is an OutputPin extended by «Exchange», then the target must be a DataStoreNode extended by «Exchanger». This constraint is used to store data objects and tokens in a DataStoreNode.

```
isStereotypedBy(source, Exchange)
  implies isStereotypedBy(target, Exchanger)
```

[3] If the source is a DataStoreNode extended by «Exchanger», then the target must be an Activity extended by «ScientificApplication» or an InputPin extended by «Exchange».

```
isStereotypedBy(source, Exchanger) implies
  isStereotypedBy(target, ScientificApplication) or
  isStereotypedBy(target, Exchange)
```

[4] If the source is a DataStoreNode extended by «Exchanger», then the selection behaviour returns the object with a specific *index* or *name*.

```
ObjectFlow::selection(index : Integer) : Any
  body { source.ownedElement->(index) }

ObjectFlow::selection(n : String) : Any
  body { source.ownedElement->select(name = n) }
```

6.6.22 Timing

«Timing» extends the UML metaclass `Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)` and is used to specify periodically executed operations. «Timing» is processed in a thread and can have several different Actions. UML4BOINC does not specify how the data is synchronised between Actions. «Timing» can have several input parameters which are changeable by all embedded Actions. «Timing» has only one initial node which can be connected to a ForkNode or directly to ActionNodes. Fig. 7.4 shows on the top right-hand side a small example of how «Timing» is used: There is one InitialNode connected to a ForkNode, which has three outgoing edges. The first edge has an InvocationAction as target and the other two are used for Actions. The InvocationAction symbolises the end of the flow; in the other two cases the flow is directly finished by a FinalNode. There, the ForkNode creates three threads in which the Actions are performed. Within BOINC's implementation the «Timing» is entered every second, «TimingFlow» is used to specify different execution behaviours. «Timing» has to be run in an extra thread but this is organised by BOINC's API and the modeller has only to fill the function by its own implementation.

UML MetaClass
Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities) [125, Chapter 12.3.4]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
No additional constraints.

6.6.23 TimingFlow

«TimingFlow» extends the UML metaclass `ControlFlow (from BasicActivities)` and is used to connect Actions within «Timing». *delay* can be used to set-up different offsets when Actions have to be executed for the first time until the scientific application has started, e.g. one Action is executed 10 seconds from the first initialisation of the scientific application.

UML MetaClass
ControlFlow (from BasicActivities) [125, Chapter 12.3.19]

Tag-Values
delay : Integer = 1 [1] – A value in seconds when an Action is executed after the first entering of «Timing» or last execution of the related Actions.
moment : Integer = -1 [1] – Instant of time when the «TimingFlow» has to be used

to execute the target action.

Associations
No additional associations.

Informal and OCL Constraints
[1] When a ForkNode is available in «Timing», «TimingFlow» can only be used between a ForkNode and Actions, otherwise «TimingFlow» has to be used between the InitialNode and an Action.

```
self.activity.node->select(oclIsKindOf(ForkNode))->size() = 1 implies
    source.oclIsKindOf(ForkNode) and (
      target.isStereotypedBy(TimingSignal) or
      target.isStereotypedBy(Action))
```

[2] *delay* specifies a value in seconds. When the connected Actions or InvocationAction is entered, *moment* is set to the future timestamp which describes the period when the action has to be executed again, i.e. $moment = CURRENTTIMESTAMP + delay$.

6.6.24 TimingSignal

«TimingSignal» extends the UML metaclass `InvocationAction (from BasicActions)` and is used to send signals to client's statemachine (Section 6.6). It can be used for periodic execution of checkpoint creation. With the help of «TimingFlow» the checkpoint period can be handled by the developer of the scientific application.

UML MetaClass
InvocationAction (from BasicActions) [125, Chapter 11.3.20]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «TimingSignal» is not allowed to have outgoing control flows. The incoming control flow must have ForkNode as source and only one incoming flow is allowed.

```
self.outgoing->isEmpty() and
self.incoming->size() = 1 implies
  self.incoming->at(1).oclIsKindOf(ForkNode)
```

[2] «TimingSignal» can only be added to «Timing».

```
isStereotypedBy(self.activity, Timing)
```

6.6.25 TrickleUp

«TrickleUp» extends the UML metaclass `SendObjectAction (from IntermediateActions)` and is used to signal to a client's statemachine that an

asynchronous message is available which has to be sent to the BOINC project server. When the signal is invoked, the execution of the current flow is continued immediately. The asynchronous message is removed by «TrickleUp» after handling and an attached outgoing control flow is used.

UML MetaClass
SendObjectAction (from IntermediateActions) [125, Chapter 11.3.44]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] The outgoing flows are control flows.

```
self.outgoing->forAll(o | oclIsKindOf(ControlFlow))
```

[2] «TrickleUp» has two InputPins.

```
-- [1] variety
-- [2] : TrickleMessage
input->size() = 2
```

6.6.26 WaitForNetwork

«WaitForNetwork» extends the UML metaclass `StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)` and is used for waiting for an available network interconnection, e.g. when a BOINC project needs to communicate to external internet services. «WaitForNetwork» does not need any parameter values. *tries* is used as an execution constraint which describes how often a network check is performed. «WaitForNetwork» checks periodically every *tries* seconds, i.e. one check each second. Two outgoing flows are specified to describe the result of the waiting process:

- one for an available network connection and
- one when no network connection is available; it is used for *tries* seconds.

UML MetaClass
StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities) [125, Chapter 12.3.48]

Tag-Values
tries : Integer = 10 [1] – Specifies how often this Activity will check for an available network connection. This value describes seconds, default is 10 seconds.

Associations

No additional associations.

Informal and OCL Constraints

[1] «WaitForNetwork» has two outgoing control flows: (1) is guarded by *true* and (2) is guarded by *false*.

```
self.outgoing->size() = 2 implies
  self.outgoing->at(1).guard = true and
  self.outgoing->at(2).guard = false
```

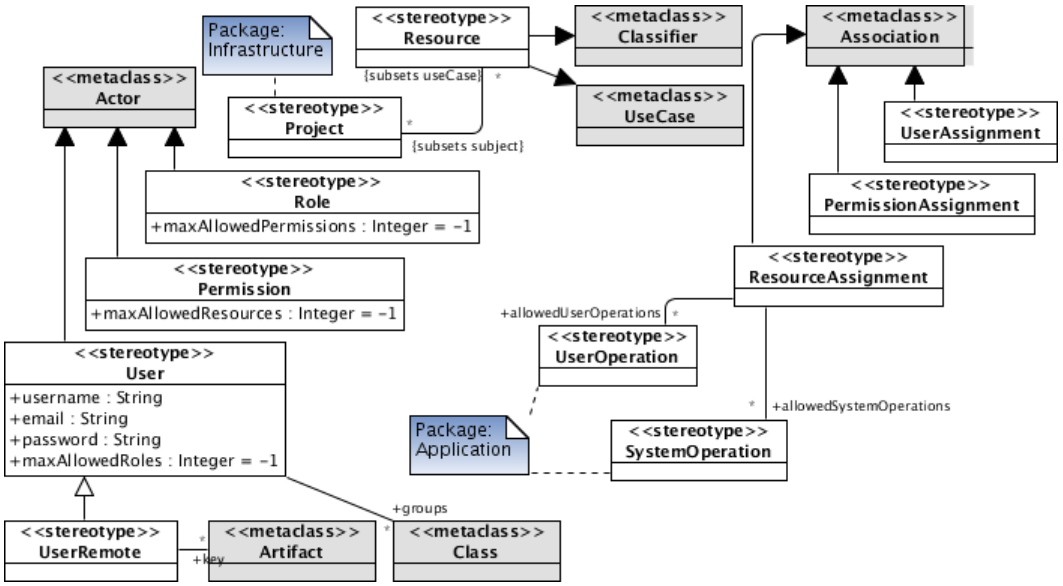

Figure 6.11: Stereotypes for the creation of Role-based Access Control (RBAC) models.

6.7 Permissions

In [36] the authors give an idea of an approach for specifying user control mechanisms where each user can be associated to roles, sets of permissions and individual resources; this approach is extended in this thesis. This thesis introduces restrictions that «Permission» descriptions can only be associated to elements which are extended by «Resource» and are used as UML UseCases. Individual users can be added as Actors extended by «User» or «UserRemote», roles can be specified as Actors extended by «Role», and a set of permissions can be specified with Actors extended by «Permissions». Based on this, there is a strict coherence in how associations must be used, i.e. «User» instances must be associated to «Roles» and these to «Permissions». Finally, «Permissions» can be associated with elements extended by «Resource».

Fig. 6.12 shows the general approach of the Permission diagram and how the introduced stereotypes are applicable. «Share» and «PortExport» are extended to allow user authenticated access to previously provided data shares (Section 6.5). For this case each «Share» can have an individual set of permissions or the whole Port has one set of permissions which is used for all associated shares.

6.7.1 Permission

«Permission» extends the UML metaclass `Actor (from UseCases)` and is used to specify a set of allowed permissions which are associated by roles. «Permission» can be associated to several «Resource» elements, the maximum number

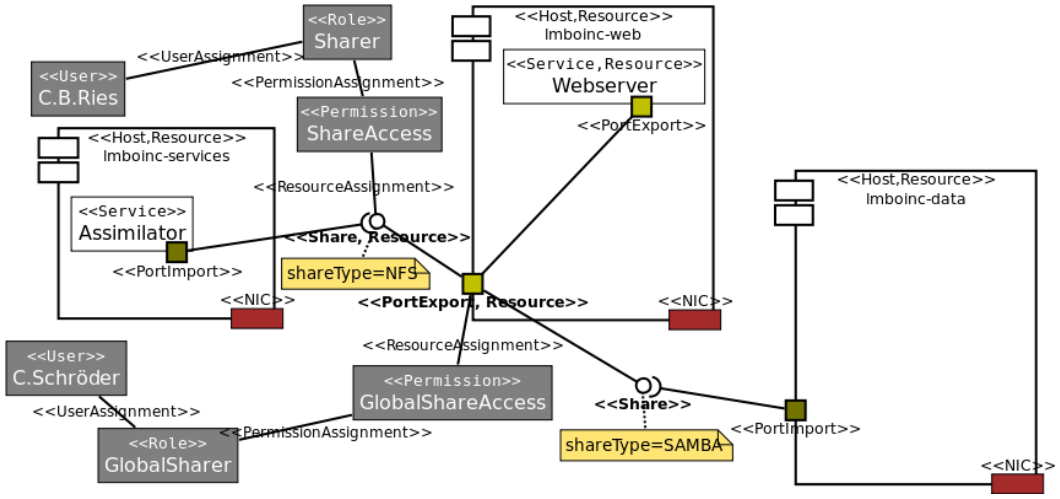

Figure 6.12: Example of «PortExport» and «PortImport» with additional stereotype «Resource» to allow «Permission» assignments.

of associations can be restricted with *maxAllowedResources*.

UML MetaClass
Actor (from UseCases) [125, Chapter 16.3.1]

Tag-Values
maxAllowedResources : Integer = -1 [1] – Maximum number of «Resource» elements permitted for «Permission».

Associations
No additional associations.

Informal and OCL Constraints
[1] If *maxAllowedResources* is -1 then the value is not limited.

6.7.2 PermissionAssignment

«PermissionAssignment» extends the UML metaclass `Association (from Kernel)` and assigns «Roles» to «Permissions».

UML MetaClass
Association (from Kernel) [125, Chapter 7.3.3]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] One side of this association is connected to «Permission» and the other side is connected to «Resource».

```
memberEnd->select(p | isStereotypedBy(p, Permission))->size() = 1 and
memberEnd->select(p | isStereotypedBy(p, Resource))->size() >= 1
```

6.7.3 Resource

«Resource» extends the UML metaclasses `Classifier (from UseCases)` and `UseCase (from UseCases)` and is used to specify BOINC components as resources which are usable by external actors, e.g. real persons or system tools.

UML MetaClass
Classifier (from UseCases) [125, Chapter 16.3.2]
UseCase (from UseCases) [125, Chapter 16.3.6]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] Only elements extended by stereotypes of the Infrastructure package can be extended by «Resource», in particular: «Service», «Host», «Database», «PortExport», «Share», «DB», and «SAN».
[2] «Permission» can only be assigned to elements extended by «Resource».

6.7.4 ResourceAssignment

«ResourceAssignment» extends the metaclass `Association (from Kernel)` and assigns «Permissions» to instances of a BOINC project which are extended by «Resource», e.g. provided interfaces extended by «Share» to exchange files between hosts.

UML MetaClass
Association (from Kernel) [125, Chapter 7.3.3]

Tag-Values
No additional tag-values.

Associations
allowedUserOperations : UserOperation [*] – List of operations which users can execute. These operations are modelled by administrators or developers, i.e. scripts to create specific statistics or to call an individual script within a BOINC project.
allowedSystemOperations : SystemOperation [*] – List of operations which administrators or system tools can execute. These operations are available on hosts of the server-side. The difference between this association and *resourceActions*

is, that this association describes only operations which are modelled by the developer.
resourceActions : ResourceAction [*] – These are actions of operations by a specific BOINC resource, i.e. start and stop scripts of a BOINC project. Any resource can have optional constraints for how these actions have to be called and which parameter should be used, e.g. copy calls need information **what** should be copied and **where** it should be stored; in this case two parameters are needed.

Informal and OCL Constraints
[1] One end of this association is connected to «Permission» and the other end is connected to «Resource».

```
self.memberEnd->select(p | isStereotypedBy(p, Permission))->size() = 1 and
self.memberEnd->select(p | isStereotypedBy(p, Resource))->size()  >= 1
```

6.7.5 Role

«Role» extends the UML metaclass `Actor (from UseCases)` and specifies «Roles» for «Users». Users can not be associated directly to system resources. They must have a role which is associated by a set of permissions. Roles can be used by several users and therefore the administration of user memberships to resources can be administrated effortlessly, i.e. an association of an user to a role can be removed and this can involve the removing of several associated permissions, otherwise the user assignment to all resources assigned by the related role has to be removed in single steps. The number of assigned permissions is restricted by the value of *maxAllowedPermissions*.

UML MetaClass
Actor (from UseCases) [125, Chapter 16.3.1]

Tag-Values
maxAllowedPermissions : Integer = -1 [1] – Maximum number of «Permissions» permitted for a «Role».

Associations
No additional associations.

Informal and OCL Constraints
[1] If *maxAllowedPermissions* is -1 then the value is not limited.

6.7.6 User

«User» extends the UML metaclass `Actor (from UseCases)` and describes a physical user which has different permissions within a BOINC project. In particular, the user's access control and administration permissions is described by

associations to «Roles» and «Permissions».

UML MetaClass
Actor (from UseCases) [125, Chapter 16.3.1]

Tag-Values
username : String [1] – Name of this user.
email : String [1] – E-mail address of this user, necessary for log-in.
password : String [1] – Password for this user, necessary to log-in.
maxAllowedRoles : Integer = -1 [1] – Maximum number of allowed roles for this user.

Associations
groups : Class [*] – Groups in which this user can be a member.

Informal and OCL Constraints
[1] If *maxAllowedRoles* is -1 then the value is not limited.

6.7.7 UserAssignment

«UserAssignment» extends the UML metaclass `Association (from Kernel)` and assigns «Users» to «Roles».

UML MetaClass
Association (from Kernel) [125, Chapter 7.3.3]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] One side must be connected to «User» and the other side must be connected to «Role».

```
self.memberEnd->select(p | isStereotypedBy(p, User))->size()  = 1 and
self.memberEnd->select(p | isStereotypedBy(p, Role))->size() >= 1
```

6.7.8 UserRemote

«UserRemote» specialises «User» and adds remote functionalities to allow the login of a user by use of the secure-shell (SSH).

UML MetaClass
Actor (from UseCases) [125, Chapter 16.3.1]

Tag-Values
No additional tag-valus.

Associations

key : Artifact [*] – These are Secure Shell (SSH) keys for an automatic log-in on one server-side host. More than one is allowed because the user can enable automatic logins from several client-side hosts.

Informal and OCL Constraints

[1] Key must be set.

```
self.key->notEmpty()
```

[2] Only this user can login to hosts on the server-side, i.e. for manual administration tasks.

6.8 Workunit Structure

For computations with BOINC it is necessary to have one or more workunits which contain descriptive information on how to handle these workunits by computations. Workunits could contain several files, e.g. additional configurations, data sets, definitions of algorithms and arbitrary extra files. Fig. 6.13 shows the part of UML4BOINC's definition for workunit structures. The three stereotypes «Workunit», «Input», and «Output» are directly based on BOINC's workunit system. All tag-values of these three stereotypes are mapped to attributes of BOINC's workunit template files. The only exception is the tag-value *unique*. If *unique* is *true*, then all input files are renamed to be unique within a BOINC project. The other presented stereotypes are extensions to fulfil UML4BOINC's workunit handling and processing approach as described in more detail in Section 7.4. Some tag-values of the stereotypes in this section have a specific datatype: *Any*. Any is a special naming and means that every dataset is allowed, i.e. during runtime the correct datatype is automatically selected and used.

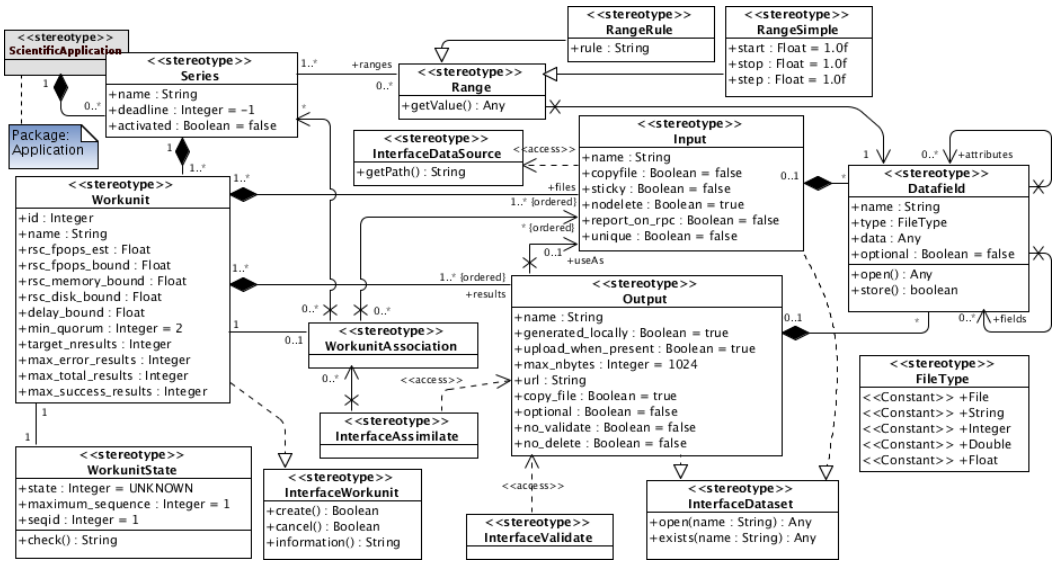

Figure 6.13: This abstract model describes the functionality of dynamic series creation. Workunit's result files can be specified as input files for subsequent workunits. Furthermore, the structure allows validation and assimilation processes to be specified.

6.8.1 Datafield

«Datafield» extends the UML metaclass `Class (from Kernel)`. All instances of «Datafield» have two associations which can be used to create tree structures with several pieces of information for a «Workunit» embedded in an

«Input» instance. With this methodology different structures are possible as shown in Section 7.4. When *type* is *FileType::File*, *open()* and *store()* must be specified by the user. UML4BOINC does not specify how the files can be expressed with the help of UML. The user is responsible for correct file handling, e.g. if the file is a ZIP-archive a corresponding correct implementation of ZIP-functionalities must be used when datasets are accessed or stored.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
name : String – Name of the «Datafield» itself.
type : FileType – Specifies the type for an instance of «Datafield».
data : Any – Contains the set of data. *Any* is a place holder and specifies that any type of data can be added.
optional : Boolean = true – The data field is optional and must not be specified during runtime.

Associations
fields : Datafield [0..]* – The data field can have several leaf data fields. This association makes it possible to create hierarchies of data.
attributes : Datafield [0..]* – A data field can have attributes, to specify a data values with more details, e.g. a currency value has a specific currency unit.
input : Input [0..1] – Owner of an instance of «Datafield».
output : Output [0..1] – Owner of an instance of «Datafield».

Operations
open() : Any – This operation is used to add a special implementation to open a «Datafield», i.e. each datafield can be specified by an arbitrary datatype, in case user-defined datatypes are used an additional *open()* implementation can support this arbitrary datatype.
store() : Boolean – The opposite of opening is storing and this operation is used to implement a special implementation for additional arbitrary datatypes.

Informal and OCL Constraints
[1] If a «Datafield» contains a physical file, *type* is set to *FileType::File* and no associated data fields or attributes are allowed.

```
self.type = FileType::File implies
  self.attributes->isEmpty() and
  self.fields->isEmpty()
```

[2] «Datafield» can only be owned by «Input» or «Output».

```
input->notEmpty() or output->notEmpty()
```

[3] UML's primitive datatypes must be supported without any additional *open()* and *store()* operations [126].

[4] For selecting and data access purposes the name must be set.

```
name->notEmpty()
```

6.8.2 FileType

«FileType» extends the UML metaclass `Enumeration (from Kernel)` and is used to specify user-defined datatypes which are used for input and output file attributes. As a default the UML PrimitiveTypes without *UnlimitedNatural* have to be supported by all implementations [126, p.213]. A developer can add their own datatype specification. The UML PrimitiveType *Real* is supported by *Double* and *Float*.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] Most of the UML infrastructure specification PrimitiveTypes must be supported [126], e.g. *Boolean*, *Integer*, or *String*.

```
ownedLiteral->includes(Set{Boolean, Integer, String,
                           Double, Float, File})
```

6.8.3 Input (implements «InterfaceDataset»)

«Input» extends the UML metaclass `Class (from Kernel)` and specifies an input file of a workunit. An «Input» classifier can have unlimited embedded data fields associated by *datafield*. BOINC itself defines a fragile way of workunit creation based on virtual and physical file names. UML4BOINC works with an arbitrary name for input and output files and physical files are associated by «Datafield». The tag-values *copyfile*, *sticky*, *nodelete*, and *report_on_rpc* are described in BOINC's wiki [167]

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
name : String – The name of the input file which is used within the implementations, e.g. scientific applications or workunit creation processes. This name can be seen as a virtual file name, similar to BOINC's system. It depends on the data fields for how the name is used in the context of UML4BOINC. If the *type* of a data field is *FileType::File* then the name of the input file is mapped to the physical

file. Under other circumstances a temporary file is created and the input filename is mapped to this physically created file.
copyfile : Boolean = true – The file is copied into job's slot directory.
sticky : Boolean = false – The file remains on the client after the computation is finished.
nodelete : Boolean = false – The file is not deleted from the server after the computation is completed.
report_on_rpc : Boolean = true – If present report file in each Scheduler request (sticky files).
unique : Boolean = true – If *unique* is *true* all input files are renamed to be unique within a BOINC project.

Associations
workunit : Workunit [1..]* – Owner of input files.
datafield : Datafield [*] – Data fields of an input file.

Informal and OCL Constraints
[1] Each «Input» classifier must have an owner associated by *workunit*.

```
self.workunit->notEmpty()
```

[2] The name of an input file must be unique in the namespace of a specific BOINC project.

```
allInstances->forAll(c1, c2 | c1.name <> c2.name)
```

6.8.4 Output (implements «InterfaceDataset»)

«Output» extends the UML metaclass `Class (from Kernel)` and describes how output files are structured. Results of workunit processing can create several output files. In case a result file is optional the tag-value *optional* is set to *true*. The tag-values *name*, *generated_locally*, *upload_when_Present*, *max_nbytes*, *url*, *no_validate*, *no_delete* and *optional* are described in BOINC's wiki [167].

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
name : String – Similar to the «Input» stereotype (Section 6.8.3).
generated_locally : Boolean = true – Always include this for output files.
upload_when_present : Boolean = true – If present, clients will report this computation immediately after the output files are uploaded. Otherwise they may wait up to a day.
max_nbytes : Integer = 1024 – Maximum file size. If the actual size exceeds this, the file will not be uploaded, and the job will be marked as an error.

url : String – The URL of the file upload handler. It can be set explicitly, or `<UPLOAD_URL/>` can be used to fill *url* automatically by the BOINC framework.
copy_file : Boolean = true – If present, the file will be generated in the slot directory, and copied to the project directory after the job has finished. Use this for ISV applications.
optional : Boolean = false – If output files are absent, the scientific application must create the file, otherwise the computation will be marked as an error.
no_validate : Boolean = false – If true, this file is not included in the result validation process; relevant only if BOINC's standard sample bitwise validator is used.
no_delete : Boolean = false – If present, the file will not be deleted on the server even after the computation is finished.

Associations
useAs : Input [0..1] – In case a workunit is part of a sequence to be performed «Series» this file is used as an input file for workunits to be performed later.
workunit : Workunit [1..]* – Owner of this result.
datafield : Datafield [*] – Describes the structure of the result file.

Informal and OCL Constraints
No additional constraints.

6.8.5 Range

«Range» extends the UML metaclass `Class (from Kernel)` and can be used for the specification for how attributes of workunit input files are valued in a dynamically executed context. Two specializations are provided by UML4BOINC's «RangeRule» and «RangeSimple».

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
No additional tag-values.

Associations
series : Series [1..]* – One «Range» can be owned by several «Series» specifiers.
datafield : Datafield [1] – One specific data field of a workunit input file is valued by «Range». In case a workunit has one input file with several data fields, each field can be set automatically by «Range».

Operations
getValue() : Any – This operation returns a value which is used to set attributes of «Datafield». In implementation it must be a virtual operation and must be offered by specializations of «Range».

Informal and OCL Constraints
[1] «Range» is abstract.

```
isAbstract = true
```

6.8.6 RangeRule

«RangeRule» extends the UML metaclass `Class (from Kernel)` and specializes «Range» (Section 6.8.5). This stereotype is used to specify rules for range creation. The tag-value *rule* can be used to add a user specific script or implementation which is used during runtime to create user-defined ranges.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
rule : String – This tag-value can contain a user-specific rule how the tag-value *data* of «Datafield» has to be set. The interpretation of this rule is provided by the development environment.

Associations
No additional associations.

Informal and OCL Constraints
No additional constraints.

6.8.7 RangeSimple

«RangeSimple» extends the UML metaclass `Class (from Kernel)` and specializes «Range» (Section 6.8.5). This stereotype can be used as a default set for different ranges, where a specific value is set for variables during runtime, e.g. *start* is set to 1.0f, *stop* to 10.0f and *step* is 0.1f then the creation of ninety-one workunits will use 1.0f, 1.1f, . . . , 10.0f as values and all workunits differ. The same rule can be created manually by the user when «RangeRule» is used.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
start : Float = 1.0f – First used value for the tag-value *data* of «Datafield».
stop : Float = 1.0f – Last used value for the tag-value *data* of «Datafield».
step : Float = 1.0f – Steps between last and new used values for the tag-value *data* of «Datafield».

Associations
No additional associations.

Informal and OCL Constraints
No additional constraints.

6.8.8 Series

«Series» extends the UML metaclass `Class (from Kernel)` and is used to set-up workunits which are performed sequentially, i.e. one workunit needs input data or results of previous workunits and not until they are finished can new workunits after current position in the sequence be created. UML4BOINC supports three «Series» approaches as described in Section 7.4.3.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
name : String [1] – Any «Series» must have a specific name. This name must be unique in the BOINC project environment. There is no direct use for this name within UML4BOINC, but for later implementation in an Integrated Development Environment (IDE) this name can be used as an associative association.
deadline : Integer = -1 – This tag-value is used as a UNIX timestamp based deadline. A «Series» must be finished before this deadline is reached. In cases where this is not possible, the deadline can be set to -1; then the deadline is not set and the workunit's creation part of a BOINC project will wait indefinitely until all workunits of a «Series» are finished or the project administrator cancels the execution manually.
activated : Boolean = false – This tag-value describes if a «Series» is activated and is enabled for processing. If *false*, no workunits of this «Series» will be created. As a consequence, in the case that three «Series» are defined, and the first and third «Series» are activated and the second is not activated, the third «Series» will never be processed. The process will get stuck at the end of the first «Series».

Associations
ranges : Range [0..]* – An association to a rule-set for how attributes of workunits are valued when workunits are created.
workunit : Workunit [1..]* – «Series» owns several workunits which have to be performed.

Informal and OCL Constraints
No additional contraints.

6.8.9 Workunit

«Workunit» extends the UML metaclass `Class (from Kernel)` and is the root of a workunit specification within a BOINC project. In UML4BOINC the root

is changed to «Series» which has an association to several «Workunit» instances. Furthermore, «Workunit» has associations to «Input» instances to describe which files are necessary to start the computation, secondly associations to «Output» instances are used to create the correct number of output files. Within BOINC a workunit is a fundamental part of a computation beside the scientific application, validation and assimilation parts. When no workunit is defined a BOINC project could be started but no computations are performed. It is possible to create only one workunit which is used by any client, therefore *max_total_results* has to be set to a value of how many clients are planned.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7] (implements «InterfaceWorkunit»)

Tag-Values
id : Integer – An unique identifier for a workunit in the context of a BOINC project.
name : String – An unique name for a workunit in the context of a BOINC project.
rsc_fpops_est : Float – The estimated number of floating-point operations on the client-side.
rsc_fpops_bound : Float – Number of floating-point operations which can be used for the computation of a workunit on the client-side.
rsc_memory_bound : Float – Number of bytes which a workunit can use on the client-side.
delay_bound : Float – Deadline of a workunit. The workunit has to be performed until this point in time.
min_quorum : Integer = 2 – Defines how many times a workunit has to be performed before it is passed to the next step within BOINC's processing chain, i.e. to mark a workunit as ready for validation.
target_nresults : Integer – This value is used to initialize the number of workunits when a workunit is flagged as ready for computation. It defines how many parallel computations can be performed of the same workunit. This value must be positive and not zero and may not exceed *max_success_results*.
max_error_results : Integer – Number of failed workunit results, when it is exceeded the workunit is marked as failed.
max_total_results : Integer – Maximum number of workunit computation tries (Section 5.3.9).
max_success_results : Integer – Maximum number of successfully processed parts of a workunit.

Associations
series : Series [1] – Any «Workunit» is owned by a «Series» instance.
files : Input [1..] {ordered}* – Input files which are used to add data for computations.
results : Output [1..] {ordered}* – Output file specification and description for how

the result has to be stored until the computation has finished.
workunitAssociation : WorkunitAssociation [0..1] – This association is used in the context of a dynamic «Series». The association is used by BOINC's assimilation process to check if a workunit is part of a sequence and if a following workunit has to be created.
workunitState : WorkunitState [1] – Information of the processing state of this «Workunit» instance.

Informal and OCL Constraints
[1] A workunit has several attributes which are related to each other. BOINC provides an interface for workunit creation and following constraints are based on BOINC's implementation.[3]

```
rsc_fpops_est          > 0  and
rsc_fpops_bound        > 0  and
rsc_disk_bound         > 0  and
target_nresults        > 0  and
max_error_results      > 0  and
max_total_results      > 0  and
max_sucess_results     > 0  and
max_sucess_results  <= max_total_results and
max_error_results   <= max_total_results and
target_nresults     <= max_sucess_results
-- At this position a check for assigned workunits is possible.
-- In addition, if an identifier is set the workunit
-- is updated and not created.
```

6.8.10 WorkunitAssociation

«WorkunitAssociation» extends the UML metaclass `Association (from Kernel)` and enables one to define a «Series» with sequential computations. «InterfaceAssimilate» delegates these computations and can have an association to «WorkunitAssociation». As described in Section 6.10 the "Next" exit pseudostate in Fig. 7.6 is used when one workunit is assimilated and has additional workunits to be performed. The following pseudocode demonstrates how the assimilation process can decide if one workunit is in a sequence and if a workunit follows:

```
Let ws As workunitAssociation.workunit.workunitState
If ws.seqid < ws.maximum_sequence Then
 For ro In Output
  Set workunitAssociation.input = ro
    Where
      workunitAssociation.input.name = ro.useAs.name
 EndFor
EndIf
ws.seqid = ws.seqid + 1
```

A new workunit is filled with «Datafield» values of one «Output» when they are associated by *useAs*. As a consequence, *useAs* must only be set when «Output»

[3] Revision 22328

is used for one «Input» configuration. The fact is, when no *useAs* is available then it makes no sense to check for a sequence. In the case when all results of a workunit are required, BOINC's Assimilator can create additional workunits with a duplicated configuration, e.g. a workunit is missed to complete a sequence and is cancelled by BOINC clients too often or workunit's *delay_bound*[4] is reached. For this occurrence the workunit can be copied and added to a «Series». When a workunit is part of a sequence, the workunit's name has a special format to distinguish workunits. A similar approach for rBOINC is used [9]. rBOINC defines a specialized workunit name and this format is modified to **"NNN-SEQ-XX-YY"**:

- **NNN** is the name of the WU, and
- **SEQ** is a start pattern for a sequence description.

The embedded string **"-XX-YY-"** is defined as follows: **XX** is the current sequence identifier and **YY** is used for the maximum number of sequences. This workunit name format is used to select sequenced workunits (see Operation `querySequencedWorkunits()`).

UML MetaClass
Association (from Kernel) [125, Chapter 7.3.3]

Tag-Values
No additional tag-values.

Associations
workunit : Workunit [1] – Owner of a «WorkunitAssociation» instance.
series : Series [*] – This association can be used to connect workunits with any «Series».
input : Input [*] {ordered} – This association can be used to fill input files for workunits to be performed later.

Operations
A workunit can be cancelled at any time. This is achieved by *InterfaceWorkunit::cancel()* and it is necessary to cancel workunits which are related to a cancelled workunit, i.e. when the current WU is cancelled and has associated workunits, these must be also cancelled because they can never be processed with missing «Input» values. «WorkunitAssociation» has an additional OCL operation to query which workunits are in the current sequence:

```
querySequencedWorkunits(seqid : Integer, name : String) : Set(Input);
querySequencedWorkunits = self.series->select(
   s | s.workunit->select(w | w.workunitState.seqid > seqid AND
   -- NNN-SEQ-XX-YY
   w.name.substring(1, w.name.strpos("-SEQ") = name)
)
```

[4] Deadline of a workunit.

With this OCL statement, workunits can be selected which are later defined within a sequence and as a consequence they can be cancelled. Cancelled workunits can have other associated workunits and they must be cancelled as well. The call of *Workunit::cancel()* is used to cancel one workunit. «Series» can be cancelled with this concept, it is sufficient to call *cancel()* during the iteration of all associated workunits.

Informal and OCL Constraints
No additional constraints.

6.8.11 WorkunitState

«WorkunitState» extends the UML metaclass `Class (from Kernel)` and adds information on the processing state of a specific «Workunit» instance. «WorkunitState» is associated by «Workunit» and from the beginning of its existence the state of a workunit can be queried at any time. The tag-value *state* holds the current state and can be valued with the following variables:

- **Created** Workunit is created.
- **Failed** Workunit has failed and cannot be finished.
- **Validation** Workunit has to be validated.
- **Assimilation** Workunit has to be assimilated.
- **Computation** Workunit is in progress and clients work on it.
- **Done** Enough clients have worked on one workunit and it can be moved to the validation and assimilation process.
- **Finished** Workunit is finished and ready for later use, e.g. to create a new «Series» or to use their computational results.
- **Canceled** Workunit is cancelled by an administrator or by other processes, e.g. when sequenced workunits have failed or are cancelled.

For UML4BOINC it is important what the state of a workunit is; as a matter of fact, the state value is responsible for deciding which actions are performed during the workunit processing, i.e. when a workunit fails the assimilation process has to decide if it should be performed again. The accessory method *check()* is used to query the current state of a workunit and returns a descriptive text value, i.e. the string contains the current state with additional, more precise, information such as the *timestamp* of the last check or how long a workunit is currently processing. The

other two tag-values *maximum_sequence* and *seqid* are used for the «WorkunitAssociation» in the next section.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
state : Enumeration – Specifies the current workunit processing state.
maximum_sequence : Integer = 1 – Maximum identifier for workunits.
seqiq : Integer = 1 – Identifier of the current workunit within a sequence.

Associations
workunit : Workunit [1] – Associated workunit for this state information.

Operations
check() : String – Queries the current state of a workunit and returns a descriptive text value.

Informal and OCL Constraints
[1] Allowed processing states of a workunit.

```
state = #Created     or state = #Failed        or
state = #Computation or state = #Done          or
state = #Validation  or state = #Assimilation or
state = #Canceled    or state = #Finished
```

6.9 Workunit Handling

BOINC has several components which need to access «Input», «Output» and their embedded «Datafield» fields. Fig. 6.13 shows three interfaces to access them: (1) «InterfaceAssimilate», (2) «InterfaceDataset» and (3) «InterfaceValidate».

6.9.1 InterfaceAssimilate

«InterfaceAssimilate» extends the UML metaclass `Interface (from Interfaces)` and is used to access workunit output files and to create new workunits.

UML MetaClass
Interface (from Interfaces) [125, Chapter 7.3.24]

Tag-Values
No additional tag-values.

Associations
workunitAssociation : WorkunitAssociation [0..]* – This association is used to create additional workunits when «Series» to be performed later are defined.

Informal and OCL Constraints
No additional constraints.

6.9.2 InterfaceDataset

«InterfaceDataset» extends the UML metaclass `Interface (from Interfaces)`. This interface is used to allow external applications to open input and output files on a standard way and always with the same procedure, e.g. Fig. 6.13 shows two interfaces («InterfaceValidate» and «InterfaceAssimilate») which use «InterfaceDataset» to access input and output files and their embedded data fields.

UML MetaClass
Interface (from Interfaces) [125, Chapter 7.3.24]

Tag-Values & Associations
No additional tag-values and associations.

Operations
open(name : String) : Any – This operation is used to open a data field of an input or output file. The *name* is mapped to the corresponding data field. It returns the data of the data field which is returned by *Datafield::open()*.
exists(name : String) : Boolean – This operation is used to check if specific data

fields of an input or output file exist. The *name* is mapped to the corresponding data field. This operation returns *true* if the data field exists, otherwise *false* is returned.

Informal and OCL Constraints
[1] The additional operations are included in ownedOperation.

```
self.ownedOperation->includes(Set{open, exists})
```

6.9.3 InterfaceDataSource

«InterfaceDataSource» extends the UML metaclass `Interface (from Interfaces)`. «InterfaceDataSource» (IDS) is an interface which is implemented by a «Service» component for workunit creation [55]. This component can have several connections to data sources. When these data sources signal new available data packages, IDS provides, with the help of *getPath()*, a file path which is usable for «Input» and a corresponding «Datafield» has *#File* as *type*.

UML MetaClass
Interface (from Interfaces) [125, Chapter 7.3.24]

Tag-Values & Associations
No additional tag-values and associations.

Operations
getPath() : String – Returns the file path of a physical file.

Informal and OCL Constraints
[1] The additional operation is included in ownedOperation.

```
self.ownedOperation->includes(Set{getPath})
```

6.9.4 InterfaceValidate

«InterfaceValidate» extends the UML metaclass `Interface (from Interfaces)`. All implementations of this interface have access to the output files of a workunit. Only then it is possible to validate completed workunits.

UML MetaClass
Interface (from Interfaces) [125, Chapter 7.3.24]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
No additional constraints.

6.10 Workunit Creation

This section presents an idea of new stereotypes which support developers with the ability to define the chain of workunit creation, processing and finishing.

6.10.1 Assimilation

«Assimilation» extends the UML metaclass `StateMachine (from BehaviorStateMachines)`. This statemachine is used to model the assimilation process for a specific BOINC project. Fig. 6.14 shows how BOINC's assimilation process can be abstracted with the help of UML.

UML MetaClass
StateMachine (from BehaviorStateMachines) [125, Chapter 15.3.12]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «Assimilation» can only be part of SPoV.

```
isStereotypedBy(region, WorkunitProcessing)
```

[2] This stereotype can only own exit points extended by «WorkunitExit» and which are typed as *Retry* or *Next*.

```
connectionPoint->forAll(p |
  isStereotypedBy(p, WorkunitExit) and
  (p.type = #Retry or p.type = #Next))
```

6.10.2 MonitorFinished

«MonitorFinished» extends the UML metaclass `FinalState (from BehaviorStateMachines)` and is used as an end of state machines within «WorkunitMonitor». Any instance of a final state within the same context has the same behaviour and exits the execution of a workunit monitoring (*mode* is *#Workunit*) or a whole workunit series (*mode* is *#Series*).

UML MetaClass
FinalState (from BehaviorStateMachines) [125, Chapter 15.3.2]

Tag-Values
mode : Enumeration [1] – The finished «WorkunitMonitor» extended Region.

Associations
No additional associations.

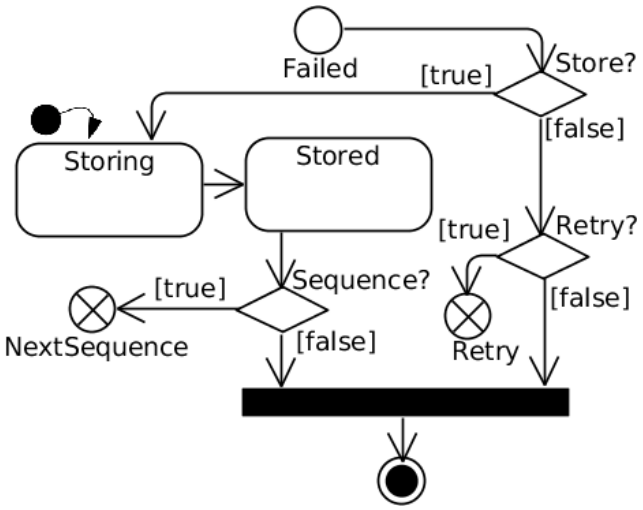

Figure 6.14: BOINC's assimilation process can use BOINC's offered standard Assimilator which will only store correct and failed result files in specific places or the BOINC project developer/administrator can implement their own assimilation process.

Informal and OCL Constraints

[1] «MonitorFinished» is used to finish a workunit series or the monitoring process of any workunit.

```
mode.ownedLiterals->including(Set{Series, Workunit})
```

[2] The mode specifies which region of «WorkunitMonitor» owns an instance of «MonitorFinished».

```
if mode = #Series then
  isStereotypedBy(self.region, SeriesRegion)
else
  if mode = #Workunit then
    isStereotypedBy(self.region, WorkunitRegion)
  endif
endif
```

6.10.3 OnClient

«OnClient» extends the UML metaclass `StateMachine (from Behavior-StateMachines)` and is modelled as a statemachine for the execution of a scientific application on the CPoV. Any owned structure or behaviour is executed on the client-side. Workunits in this state can only be controlled by asynchronous messages (Section 6.11).

UML MetaClass

StateMachine (from BehaviorStateMachines) [125, Chapter 15.3.12]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «OnClient» has no behaviour on the server-side.
[2] «OnClient» has only one region and only one incoming edge.
[3] Only one instance of this stereotype is allowed.

```
allInstances()->size() = 1
```

6.10.4 SeriesRegion

«SeriesRegion» extends the UML metaclass `Region (from Behavior-StateMachines)`. This region is only used for the monitoring of a whole «Series» and only three states need to be described:

Processing: A «Series» is in process. This is the case when one workunit is still open for processing, which is neither performed by a client nor cancelled by a BOINC administrator.

Finishing: In this state no workunits are open for processing. All workunits are finished and returned back to the BOINC project, validated and assimilated.

Cancellation: In this state the «Series» is no longer performed, all open (i.e. not already performed) workunits will be cancelled. In addition all subsequent workunits or series are cancelled.

When a series is cancelled the monitoring process is terminated and all related data can be removed and the environment can be cleaned.

UML MetaClass
Region (from BehaviorStateMachines) [125, Chapter 15.3.10]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This region must have at least one UML FinalNode extended by «SeriesFinished».

```
self.subvertex->contain(v | isStereotypedBy(v, SeriesFinished))
```

6.10.5 SeriesState

«SeriesState» extends the UML metaclass `State (from BehaviorState-Machines)`. This state is used to describe the current state of a series lifetime. A series can be in three states:

Processing: A series has workunits to be performed.

Cancellation: BOINC project's administrator has requested to cancel a series. All related workunits have to be cancelled. In addition all by the current series followed series have to be cancelled. Workunits can be still performed on the client-side, the current state machine is on the server-side and is terminated after all workunits and series are cancelled. Already performed workunits can be cancelled by asynchronous messages. If they are not cancelled by this technique, they will be finished on the client-side and returned after the computation is finished. They are not performed by any validation or assimilation process on the server-side.

Finishing: There are no additional workunits to be performed. In this state the series is checked for subsequent series definitions. If subsequent series are defined the workunits will be created and then the state machine of the current series is exited. The new workunits are created within a new series and as a consequence a new state machine is created.

«SeriesState» can only be part of a region extended by «SeriesRegion» (Section 6.10.4).

UML MetaClass
State (from BehaviorStateMachines) [125, Chapter 15.3.11]

Tag-Values
type : Enumeration [1] – Specifies which kind of state is specified.

Associations
No additional associations.

Informal and OCL Constraints
[1] «SeriesState» is used to model three different processing states.

```
type = #Processing or type = #Cancellation or type = #Finishing
```

[2] «SeriesState» can only owned by a UML Region extended by «SeriesRegion».

```
isStereotypedBy(owner, SeriesRegion)
```

6.10.6 SeriesTerminated

«SeriesTerminated» extends the UML metaclass `PseudoState (from BehaviorStateMachines)`. This state terminates the monitoring process and cancels all currently performed workunits. When this state is entered the current series and subsequent series with all related workunits are cancelled.

UML MetaClass
PseudoState (from BehaviorStateMachines) [125, Chapter 15.3.9]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This PseudoState is used as a terminate node.

```
self.kind = #terminate
```

6.10.7 Validation

«Validation» extends the UML metaclass `StateMachine (from BehaviorStateMachines)` and is used to show that a workunit is currently in validation. The validation process can have two results: (i) the workunit's result is valid or (ii) the result is not valid. In both cases the workunit has to be performed by BOINC's Assimilator, but the assimilation process differs. If a workunit is not valid the exit point typed *#NotValidated* is used and passes over to the entry point typed *#Failed* of the UML Activity extended by «Assimilation». In the other case the UML Activity is exited normally and the assimilation process is entered. BOINC offers standard validators which are applicable directly when no individual validation procedures are necessary. Fig. 6.15 describes how the validation process is achieved on SPoV. As described in Section 5.2, two standard validators are provided by BOINC:

Bitwise Validator: Checks if the MD5 checksums of two result files are equal.

Trivial Validator: Checks if the amount of CPU time for the computation is not exceeded.

In addition, the developer of a BOINC project can implement their own validation process specification.

UML MetaClass
StateMachine (from BehaviorStateMachines) [125, Chapter 15.3.12]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] If the validation failed the exit point typed *#NotValidated* has to be used.

6.10.8 WorkunitEntry

«WorkunitEntry» extends the UML metaclass `PseudoState (from BehaviorStateMachines)`. This stereotype is used to start the assimilation process

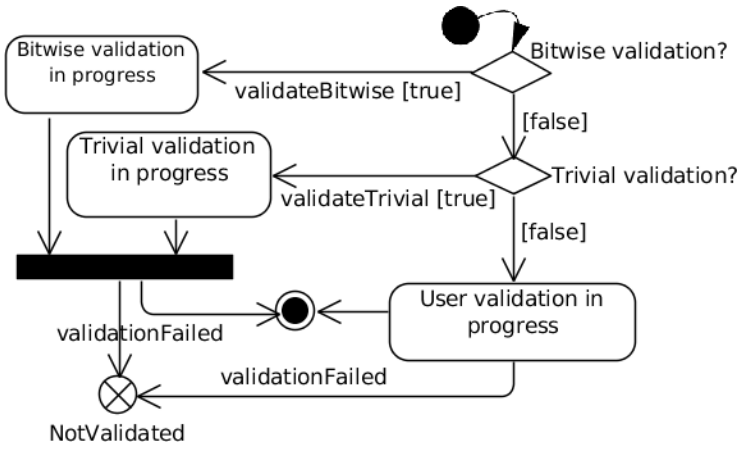

Figure 6.15: BOINC's validation process can use one of two standard Validators or the developer can implement their own validation process.

for failed workunits. With the state machine «Assimilation» it can be made distinct where failed or successfully performed workunits are stored, additionally, whether a workunit is stored in a database or within the file system can be defined.

UML MetaClass
PseudoState (from BehaviorStateMachines) [125, Chapter 15.3.8]

Tag-Values
type : Enumeration [1] – Defines the behaviour of this exit point.

Associations
No additional associations.

Informal and OCL Constraints
[1] «WorkunitEntry» is used as an entry point.

```
self.kind = #entryPoint
```

[2] The entry point can be used to model a specific behaviour and therefore it can be owned by only one specific UML StateMachine extended by «Assimilation».

```
type = #Failed implies isStereotypedBy(owner, Assimilation)
```

6.10.9 WorkunitExit

«WorkunitExit» extends the UML metaclass `PseudoState (from BehaviorStateMachines)` and is used as an exit point. This exit point can have three behaviours selected by the user:

NotValidated: The result is not validated.

Retry: The workunit of a result has to be performed again.

Next: The workunit of the result is part of a sequence and the following workunit has to be created and performed.

It depends on the selected behaviour which UML element is allowed to own «WorkunitExit». This exit point is used by validation and assimilation processes and describes the results of both activities.

UML MetaClass
PseudoState (from BehaviorStateMachines) [125, Chapter 15.3.8]

Tag-Values
type : Enumeration [1] – Specifies which behaviour is described by this exit point.

Associations
No additional associations.

Informal and OCL Constraints
[1] «WorkunitExit» is used as an exit point.

```
self.kind = #exitPoint
```

[2] The exit point can be used to model three behaviours.

```
type.ownedLiterals->including(Set{NotValidated, Retry, Next})
```

[3] It depends on the value of *type* which UML Element can own this instance.

```
if type = #NotValidated then
  isStereotypedBy(owner, Validation)
else
  if type = #Retry or type = #Next then
    isStereotypedBy(owner, Assimilation)
  endif
endif
```

6.10.10 WorkunitMonitor

«WorkunitMonitor» extends the UML metaclass `State (from Behavior-StateMachines)`. «WorkunitMonitor» extended States have two regions. The first one is used to monitor a series specification and the second one is used to monitor the lifetime of all workunits of a specific «Series» instance. In any case, if a sequence of workunits is defined or not, there is always an instance of a «Series» monitoring region.

UML MetaClass
State (from BehaviorStateMachines) [125, Chapter 15.3.11]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This state must have two regions: (i) to monitor the state of one workunit, (ii) to monitor the state of a whole series.

```
region->size() = 2 implies
  region->select(r|isStereotypedBy(r,WorkunitRegion))->size() = 1 and
  region->select(r|isStereotypedBy(r,  SeriesRegion))->size() = 1
```

6.10.11 WorkunitProcessing

«WorkunitProcessing» extends the UML metaclass `StateMachine (from BehaviorStateMachines)`. Fig. 7.6 shows the embedded structure of this state machine. Any workunit has a dedicated instance of this state machine for processing them. It is possible to define individual validation and assimilation implementations for each workunit. Workunits can be distinct based on their *id*. With the help of «WorkunitProcessing» the BOINC project administrator can query where a specific workunit is currently performed.

UML MetaClass
StateMachine (from BehaviorStateMachines) [125, Chapter 15.3.12]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
No additional constraints.

6.10.12 WorkunitRegion

«WorkunitRegion» extends the UML metaclass `Region (from Behavior-StateMachines)` and is used to monitor all workunits of a specific series. Only two different state kinds are defined to be added for this region (Section 6.8.11).

UML MetaClass
Region (from BehaviorStateMachines) [125, Chapter 15.3.10]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] This Region has at least one UML FinalNode extended by «WorkunitFinished».

```
self.subvertex->contain(v | isStereotypedBy(v, WorkunitFinished))
```

[2] All UML States have to be extended by «WorkunitState».

```
self.subvertex->select(oclIsKindOf(State))
              ->forAll(s | isStereotypedBy(s, WorkunitState)))
```

6.10.13 WorkunitState

«WorkunitState» extends the UML metaclass `State (from Behavior-StateMachines)`. It depends on the state kind as to which elements are allowed to own a specific state. *#Idle* and *#Process* are allowed to be defined within a UML Region extended by «WorkunitRegion». *#Created*, *#Finished* and *#Canceled* are owned by a UML StateMachine extended by «WorkunitProcessing». The kind of states has following meaning:

#Idle No workunits are created nor available for computations.

#Process Workunits are distributed or ready for distribution to clients.

#Created A state machine extended by «WorkunitProcessing» is created and the workunit is ready to be performed by clients.

#Finished A workunit is already performed by a client and the workunit is ready for validation.

#Canceled The processing of a workunit failed on the client-side or the workunit is cancelled by a BOINC administrator, i.e. cancelled by an asynchronous message (Section 6.11).

All kinds are required to be defined for a BOINC project. Fig. 7.5 and 7.6 show how the different states have to be connected.

UML MetaClass
State (from BehaviorStateMachines) [125, Chapter 15.3.11]

Tag-Values
type : Enumeration [1] – Specifies which kind of state is specified.

Associations
No additional associations.

Informal and OCL Constraints
[1] «WorkunitState» is used to model five different processing states.

```
type = #Created   or type = #Finished or
type = #Canceled  or type = #Idle     or
type = #Process
```

6.10.14 WorkunitTransition

«WorkunitTransition» extends the UML metaclass `Transition (from Be-haviorStateMachines)`.

UML MetaClass
Transition (from BehaviorStateMachines) [125, Chapter 15.3.14]

Tag-Values
mode : Enumeration [1] – Specifies how this Transition is used.

Associations
No additional associations.

Informal and OCL Constraints
[1] «WorkunitTransition» can be used in two modes: *Detach* and *Scatter*.

```
mode->ownedLiterals->including(Set{Detach, Scatter})
```

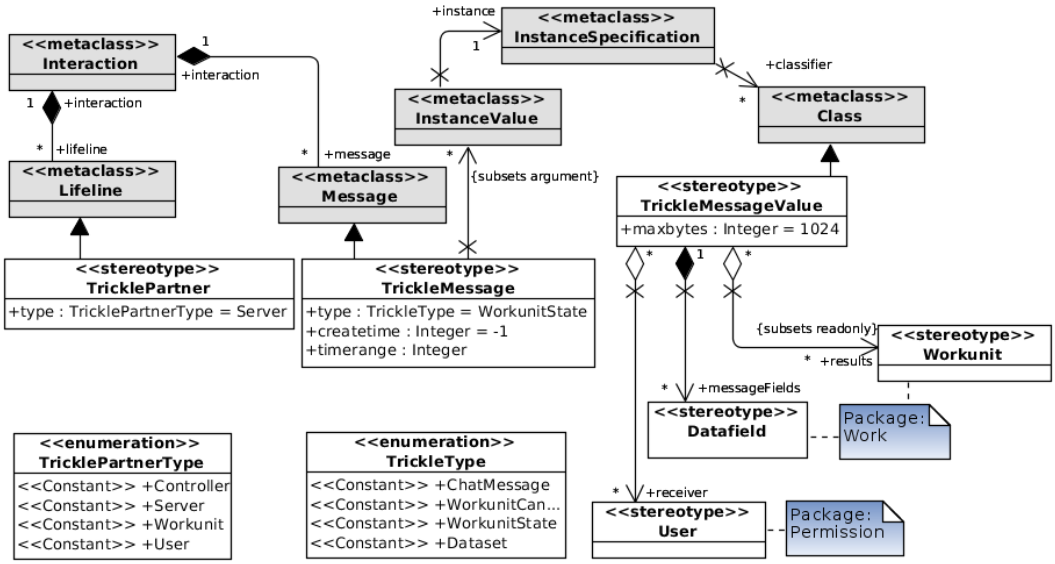

Figure 6.16: Metaclasses and stereotypes for the creation of asynchronous-message calls for BOINC's trickle messages.

6.11 Asynchronous-messages

BOINC's trickle message facilities work through the use of asynchronous-messages. Both sides — the BOINC project services on the server-side and participants on the client-side — can start a communication. This communication has no specific state frame. The server and clients send messages without any information if the messages will be received in the future. If message handling and monitoring of the message state is required, further implementation of the communication infrastructure is necessary. The framework BOINC provides does not support this out of the box. BOINC's communication framework for trickle-messages can be modelled by a small range of stereotypes as shown in Fig. 6.16. «TricklePartner» is used to specify communication ends, e.g. an individual user specified by *TricklePartnerType::User* or a BOINC service specified by *TricklePartnerType::Server*. Between them, different messages can be transferred. Therefore it is the responsibility of the modeller how the messages are structured. «TrickleMessageValue» can contain arbitrary datasets.

6.11.1 TrickleMessage

«TrickleMessage» extends the UML metaclass `Message (from BasicInteractions)` and is used to set-up the communication between a BOINC project and user/participants of a BOINC project. Users are specified with two types:

- Users can be administrators, or actors with specific roles, e.g. scientist or developer (Section 6.7.5).
- Users are participants of a BOINC project and spend their computational resources, they can communicate with other participants but the main purpose for «TrickleMessage» is to keep a BOINC project up-to-date on the computation process.

The use of this stereotype is specified by the Lifeline type (Section 6.11.3).

UML MetaClass
Message (from BasicInteractions) [125, Chapter 14.3.18]

Tag-Values
type : TrickleType = TrickleType::WorkunitState – *type* is used to specify the purpose of an asynchronous-message. Predefined values are specified by *TrickleType* in Section 6.11.5.
createtime : Integer = -1 – Point of time of the message's creation.
timerange : Integer [1] – Definition of a deadline when a message has to be delivered.

Associations
No additional associations.

Informal and OCL Constraints
[1] «TrickleMessage» can only have «TrickleMessageValue» extended arguments.

```
self.argument->forAll(a | isStereotypedBy(a, TrickleMessageValue))
```

[2] «TrickleMessage» is an asynchronous-call.

```
messageSort = MessageSort::asynchCall
```

[3] Messages from Lifelines typed as *Workunit* and *User* can only be sent to Lifelines typed as *Server*.

Notation
It is not necessary to show *createtime* in the graphical view. *createtime* must be auto filled by model interpreters during the creation of new trickle messages.

6.11.2 TrickleMessageValue

«TrickleMessage» extends the UML metaclass `Class (from Kernel)` and is used as a container for arbitrary structured asynchronous-messages. These messages are used as communication tokens between a BOINC server and corresponding BOINC clients.

UML MetaClass
Class (from Kernel) [125, Chapter 7.3.7]

Tag-Values
maxbytes : Integer = 1024 – Size in bytes of the maximum allowed message size.

Associations
receiver : User [*] – The message will be sent to all hosts of these users.
results : Workunit [*] {readonly} – A message is sent to individual users associated by *receiver*. Each user can have several processing workunits. One message will be multicast to any workunit of the user. The fact is the BOINC server does not know which workunit is processed when the message is transmitted and therefore the BOINC server does not know if the user can handle messages immediately.
messageFields : Datafield [*] – Developers can set-up individual asynchronous-messages. Section 6.8.1 describes *Datafield* in more detail.

Informal and OCL Constraints
[1] The whole size of an asynchronous-message does not exceed *maxbytes*.
[2] «Datafield» can be used to describe physical files. When one or more are specified, the IDE should embed this physical file within the message.

Notation
It is not necessary to show *maxbytes* in the graphical view, but it can be shown in curly brackets after or below the name of a trickle message.

6.11.3 TricklePartner

«TricklePartner» extends the UML metaclass `Lifeline (from BasicInteractions, Fragments)` and is used to specify communication partners for BOINC's server-side and client-side.

UML MetaClass
Lifeline (from BasicInteractions, Fragments) [125, Chapter 14.3.17]

Tag-Values
type : TricklePartnerType = Server [1] – «TricklePartner» represents the *Server*, *Client* or *User* life line.

Associations
No additional associations.

Informal and OCL Constraints
[1] Lifeline must represent the BOINC server components (Section 5.3) or the client-side components specified by one of two specific modes: (i) Workunit or

(ii) User (Section 5.4) or a BOINC project administrator on the server-side lifeline.

```
represents->notEmpty() implies
  isStereotypedBy(represents->at(1).type, Application) or
  isStereotypedBy(represents->at(1).type, Service    ) or
  isStereotypedBy(represents->at(1).type, User       )
```

[2] On the client-side the user's (typed as *TricklePartnerType::Users*) lifeline is multivalued, on the server-side the user's (typed as *TricklePartnerType::Controller*) Lifeline represents a single selected user, e.g. an administrator or scientist.

```
type = Controller implies selector->notEmpty()
```

Notation
The graphical view of the sequence diagram must not show all relevant data for the success of asynchronous-messages. In reality some tag-values must be filled with auto values, e.g. the point of creation must not be modelled by an administrator or a user.

6.11.4 TricklePartnerType

«TrickleParterType» is an enumeration of the following values:

- *Controller* – This enumeration literal describes a user with specific administration rights, i.e. a user can cancel one or more workunits or can transfer a new set of data to specific hosts (in case they are computing a long-running computation). Represented users of this Lifeline can be users which are locally logged-in on BOINC's project server or can be remotely logged-in users.
- *Server* – The Lifeline is used to describe a BOINC project service, in particular an asynchronous-message handler.
- *Workunit* – Message is sent to a specific workunit to all hosts which compute this workunit.
- *User* – Messages are sending to specific users.

The enumeration literals can be extending by user-defined literals.

Informal and OCL Constraints
[1] «TrickleParterType» has *Controller*, *Server*, *Client* or *User* as minimalistic enumeration literals.

```
self.literals->including(Set{Controller, Server, Workunit, User})
```

6.11.5 TrickleType

«TrickleType» is an enumeration of the following values:

- *ChatMessage* – A new chat message is received and should be forwarded to other BOINC users (server-side) or should be displayed in the BOINC manager (client-side).
- *WorkunitCancel* – A specific workunit should be cancelled, the workunit has to be specified in more detail within the *messageFields* of a trickle message. This message is sent by a BOINC project and the client must not reply on receiving.
- *WorkunitStatus* – A BOINC project can query the status of specific workunit on the client-side. The workunit has to be specified in more detail within the *messageFields* of a trickle message. Both, the server- and client-side can initiate the communication for status requests or to inform the BOINC server in advance.
- *Dataset* – This type is used to exchange data sets or tokens between the server- and client-side. Section 6.11.1 describes this in more detail because of the fact that the data has to be fitted into the trickle message and can not exceed the maximum size in bytes of a trickle message.

UML4BOINC does not give any restrictions how the implementation of a scientific application has to interact on different used types.

UML MetaClass
Enumeration (from Kernel) [125, Chapter 7.3.16]

Informal and OCL Constraints
[1] This Enumeration can be extended by arbitrary enumeration literals.

6.12 Tasks and Timing

UML makes it possible to set-up observers for the execution of sequenced actions, when it is necessary to start and finish the execution in a specific time window. This is not necessary for BOINC and, as a consequence, all relevant UML classifiers are simplified. This section introduces stereotypes to specify periodically executed tasks and applications. Fig. 6.17 gives an overview of the relations and dependencies between all these stereotypes. BOINC is a framework without any real-time restriction or guarantee of time of delivery, i.e. the shortest period between two task executions is one second. In addition, BOINC itself provides no system to execute any tasks or application automatically. This is performed by UNIX's cron daemon.

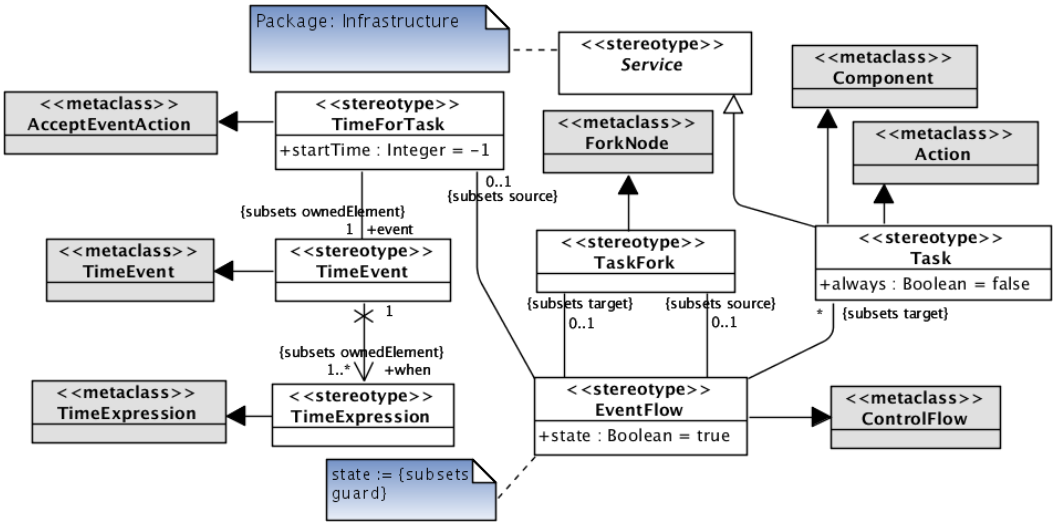

Figure 6.17: Overview of the Timing classes and their relations and dependencies.

6.12.1 EventFlow

«EventFlow» extends the UML metaclass `ControlFlow (from BasicActivities)`. *guard* of ActivityEdge (Package IntermediateActivities) can be used; but for easier understanding and handling an extra tag-value is specified. *state* is always *true*, when source is «TimeForTask» and target is «TaskFork». EventFlow can be used to connect one «TimeForTask» specification with a «Task» or «TaskFork». *state* can be used as ControlFlow's guard [125, Fig. 12.11]. UML4BOINC specifies its own guard to enable the flow on «EventFlow» and let the interpretation open if it should be handled by guard or *state*.

UML MetaClass
ControlFlow (from BasicActivities) [125, Chapter 12.3.19]

Tag-Values
state: Boolean = true [1] – Enables/Disables transitions on this flow.

Associations
No additional associations.

Informal and OCL Constraints
[1] If an «EventFlow» is used between «TimeForTask» and «TaskFork» the tag-value *state* is always *true*.

```
isStereotypedBy(target, TaskFork) implies
  isStereotypedBy(source, TimeForTask) and state = true
```

[2] «EventFlow» cannot be redefined.

```
redefinedEdge->empty()
```

[3] If more than one «Task» is planned to be executed a «TaskFork» must be used.

6.12.2 Task

«Task» extends several UML metaclasses and merges their attributes and conventions. The specialized Action has ActivityNode as one of its base classes [125, Sec.12.3.2]. A task must specialise a Component; in this case the «Task» can be deployed on different hosts and necessary configuration values can be retrieved [160], i.e. the hostname where hosts have to be executed. Furthermore, a Component offers the implementation. The period of a task is specified by «TimeForTask» and its related time expressions. The specification for whether a «Task» is enabled/disabled is specified by «EventFlow» and its tag-value *state* (Section 6.12.1). «Task» is only responsible for calls of Activities, therefore no input or output pins are necessary. The called Activities are responsible for the handling of input and output values.

UML MetaClass
Component (from BasicComponents, PackagingComponents) [125, Chapter 8.3.1]
Action (from CompleteActivities, FundamentalActivities, StructuredActivities, CompleteStructuredActivities) [125, Chapter 12.3.2]

Tag-Values
always : Boolean = false [1] – This is a BOINC related tag-value [160]. It runs this task regardless of whether or not the project is enabled (for example, a script that logs the current CPU load of the host machine).
output : String [1] – This is a BOINC related tag-value [160]. Specifies the output file to output; by default it is `COMMAND_BASE_NAME.out`. Output files are stored in the directory `log_X/` of the BOINC project installation, where `X` is the host.

Associations
No additional associations.

Informal and OCL Constraints
[1] Unnecessary tag-values and associations of the metaclasses.

```
-- Action (from BasicComponents, PackagingComponents)
localPrecondition->empty() AND
localPostcondition->empty() AND
input->empty() AND
output->empty()
```

[2] «Task» has only one incoming and one outgoing event flow.

```
incoming->size() = 1 AND outgoing->size() = 1 implies
  incoming->forAll(e | isStereotypedBy(e, EventFlow)) AND
  outgoing->forAll(e | isStereotypedBy(e, EventFlow))
```

[3] «Task» is a specification of «Service» and has its own service type.

```
serviceType = ServiceType::Task
```

[4] Restriction for the base-class «Service».

```
-- The task itself is always enable,
-- the control is done by <<EventFlow>>.
inv: state = true
-- A task must have an implementation or physical available tool.
inv: not command->empty()
```

[5] «Task» can only be started. Tasks are not allowed to act as daemons, they should start and, when finished they should stop and exit. `Service::stop()` and `Service::restart()` have no implementation.

6.12.3 TaskFork

«TaskFork» extends the UML metaclass `ForkNode (from Intermediate-Activities)` and must be used when more than one task is executed when a time event is triggered.

UML MetaClass
ForkNode (from IntermediateActivities) [125, Chapter 12.3.30]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] «TaskFork» can have only one incoming edge but several outgoing edges. An incoming edge must have «TimeForTask» as source, outgoing targets must be «Task» elements.

6.12.4 TimeEvent

«TimeEvent» extends the UML metaclass `TimeEvent (from SimpleTime)`. This event is an expired deadline which is used periodically, i.e. when the described time period is reached the same values are used for the next time.

UML MetaClass
TimeEvent (from SimpleTime) [125, Chapter 13.3.27]

Tag-Values
No additional tag-values.

Associations
when : TimeExpression [1..]* – The UML specifies 'when' as: association *when : TimeExpression [1]*. This requires specifying duplicates of «TimeEvent» when different call periods are necessary for the same task. For this reason UML4BOINC overwrites *when* with «TimeExpression» to allow multiple 'when' definitions.

Informal and OCL Constraints
[1] The time event is always relative.

```
isRelative = true
```

6.12.5 TimeExpression

«TimeExpression» extends the UML metaclass `TimeExpression (from SimpleTime)` and is used to specify time values, i.e. '2 hours' or '10 minutes'. This value must be constrained by valid values, which are based on BOINC's specification. Only the following descriptions are allowed: *n* [*days* | *hours* | *minutes* | *seconds*]. As a consequence, only valid related values are possible, e.g. for 'hours' only values between 1 and 24 are possible. In UML, expr can be specified as 'relative' or 'absolute', in the case of BOINC, only relative expressions are used. These expressions are named "periods". The UML specification constrains the use of expr, the default value of expr has the form [125, p.468]: 'after' followed by given time values, e.g. 'after n days'.

UML MetaClass
TimeExpression (from SimpleTime) [125, Chapter 13.3.28]

Tag-Values & Associations
No additional tag-values and associations.

Informal and OCL Constraints
[1] observation is used for "[...] refers to the time and duration observations that are involved in *expr*." [125, Section 13.3.28]; here, referencing is not necessary for BOINC's case. UML4BOINC specifies fixed time values for execution.

```
observation->isEmpty()
```

6.12.6 TimeForTask

«TimeForTask» extends the UML metaclass `AcceptEventAction (as specialized)`. BOINC's time configuration is not real-time based, and this makes it possible to use an approach for timing specifications. «TimeForTask» is used to specify periodic tasks. Within AcceptEventAction it is specified which interval tasks are executed. «TimeForTask» is the root for task definitions which are executed in different periods. The base class AcceptEventAction allows triggering this event manually by a user.

UML MetaClass
AcceptEventAction (as specialized) [125, Chapter 12.3.1]

Tag-Values
startTime : Integer = -1 [1] – This tag-value is used to specify the start time for a task execution. The specification of *startTime* can be useful when more than one task is specified on a host. In this case *startTime* is used as an offset and several tasks can be configured to be executed as a sequence, i.e. during the start-up of a BOINC project, the load of a host can be reduced.

Associations
event : TimeEvent [1] – This association specifies when this AcceptEventAction is triggered.

Informal and OCL Constraints
No additional constraints.

6.13 Summary

In this chapter, the UML Profile for the Berkeley Open Infrastructure for Network Computing (UML4BOINC) has been specified. The profile is described by a set of semi-formal stereotypes. Additional tag-values and associations between UML4BOINC's stereotypes and UML metaclasses have been introduced. Most of the stereotypes have been constrained by the use of a formal language, the Object Constraint Language (OCL). Not all cases are possible to describe formally and, in such cases, informal constraints are given. An informal overview of the semantic and contextual relationship between the stereotypes and the BOINC-system is given. Chapter 7 describes the semantic in more detail, depicted by short examples.

This chapter has given an overview of six new, introduced diagram types,

which are used to create different viewpoints of a BOINC-based project. After this, the different stereotypes for the context of these diagrams are presented. The infrastructure stereotypes can be used for the creation of the infrastructure on BOINC's server-side and to specify which scientific applications for which target platforms and architectures are used for distribution. This section is followed by the introduction of stereotypes to allow developers modelling scientific applications by the use of specific UML metaclasses. This is followed by the presentation of stereotype specifications for the creation of Role-based access control (RBAC) descriptions. After this, stereotypes for the specification of workunits are proposed. This proposal allows specifying of the structure of workunits, and how series of several autonomous or dependent series are handled by a BOINC-project. The last two sections present stereotypes for the specification of periodically executable tasks and event-handling aspects.

CHAPTER 7

Semantic of the UML4BOINC Profile

All our work, our whole life is a matter of semantics, because words are the tools with which we work, the material out of which laws are made, out of which the Constitution was written. Everything depends on our understanding of them.

FELIX FRANKFURTER

7.1 Introduction

This chapter explains UML4BOINC's semantic of the stereotypes introduced in Chapter 6. Six sections are used, one for every UML4BOINC package containing a diagram. Therefore the order of the sections in this chapter is as follows: (1) Infrastructure in Section 7.2, (2) Application in Section 7.3, (3) Work in Section 7.4, (4) Permission in Section 7.5, (5) Timing in Section 7.6, and (6) Event in Section 7.7. The semantic is illustrated by using UML examples. Some are created within Visu@lGrid [184] and others with VisualParadigm [207].

7.2 Semantic of the Infrastructure Modelling

The Infrastructure diagram is used for the creation of a viewpoint which enables the recognition of how a BOINC project server-side is configured. «Host» is a component which can include several «Service» or «Database» components. Every service must be added and specified as a BOINC service, i.e. as BOINC's Feeder or Transitioner. Furthermore, the component can be used as a Task. The enumeration «ServiceType» provides six different types for the specification of these services. When a BOINC service needs advanced software or libraries for starting execution

or during runtime, the software and libraries can be added in different versions through instances of «Software».

«Database» is a service which can be added to any «Host». A database itself is a specialisation of a BOINC service which can be started, stopped or restarted by the operations being part of «Service». Three different database types can be specified (Section 5.3.1):

- a BOINC project database as shown in Fig. 6.3,
- a science database to store computational results or other datasets and
- database replications for the previous databases.

«Database» itself is only used as a place-holder for additional database contexts. This concept is related to MySQL, i.e. MySQL is the software which provides the option to add a high number of individual databases. UML4BOINC works similarly and «Database» services need so-called contexts to specify usable database storages which are, in turn, specified by «DB». «Database» can have an unlimited number of «DB» contexts; including a unique name and several individual tables specified by «Table». The two tag-values *isProjectDB* and *isScienceDB* of «DB» specify how the context is used within a BOINC project. When «DB» is defined as the project database, UML4BOINC specifies that it cannot be used as a scientific database at the same time, i.e. a second database context has to be added. Thus, only one context can be specified as a project database; it has to be used by all services when access to project information and datasets is necessary. The structure of the database context is predetermined by BOINC's framework developers and it is not necessary that it can be changed by UML4BOINC. In other cases the structure can be specified by the use of «Tables» and «Column». When a database is used as a replication, the database is represented by an optionally connected «ReplicationAssociation». Any database context can be specified as a replication instance. Therefore neither *isProjectDB* nor *isScienceDB* is recognised and it is used only when associated to a second database, namely a project or science database. In addition, it is possible to associate two distributed database contexts which are located on different hosts. The association has to be made via two network-interface cards; each must be part of the relevant host. This modelling approach allows restriction of the communication pathway. Besides, two database contexts can be associated directly on the same host.

Any «Host» or «SAN» needs a network specification, otherwise communication is not granted and clients cannot connect to a BOINC project in order to retrieve workunits. In case the «Host» or «SAN» have more than one network-interface card, they can be used to restrict the communication pathways, e.g. database connections are only allowed on specific network-interface cards which

are used as in-house network devices and not linked to internet-connected networks.

Fig. 7.1 shows an example of how the Infrastructure diagram can be used. The diagram contains two hosts: (1) *lmboinc-main* and (2) *lmboinc-tasks*, and one storage area network: *lmboinc-images*. *lmboinc-main* is specified as the BOINC project's main host. It has the BOINC project installation and provides the installation containing the directory to the second host *lmboinc-tasks*. The two hosts have individual export and import ports named "BOINC" and are connected via an interface extended by «Share». This configuration provides the BOINC project installation to external hosts and is required for the execution of self-implemented Tasks and services on the hosts. All hosts within a BOINC project can share data among each other; accordingly they get ports extended by «PortExport» and «PortImport» which must be associated to each other by «Share» instances. A «PortExport» with a provided «Share» can be used by more than one «PortImport», i.e. a «PortExport» is a global definition within a BOINC project. When the user wants authorised access, both Ports must be extended by «Resource» and associated to a set of «Permissions» (Section 6.7). The same methodology is specified for databases and tables, i.e. «DB» is like the two ports described and a «Table» is like a «Share». These principles are based on the UML specification for creating components which are fully replaceable when all ports of another component are equal [125, Section 8.3.1].

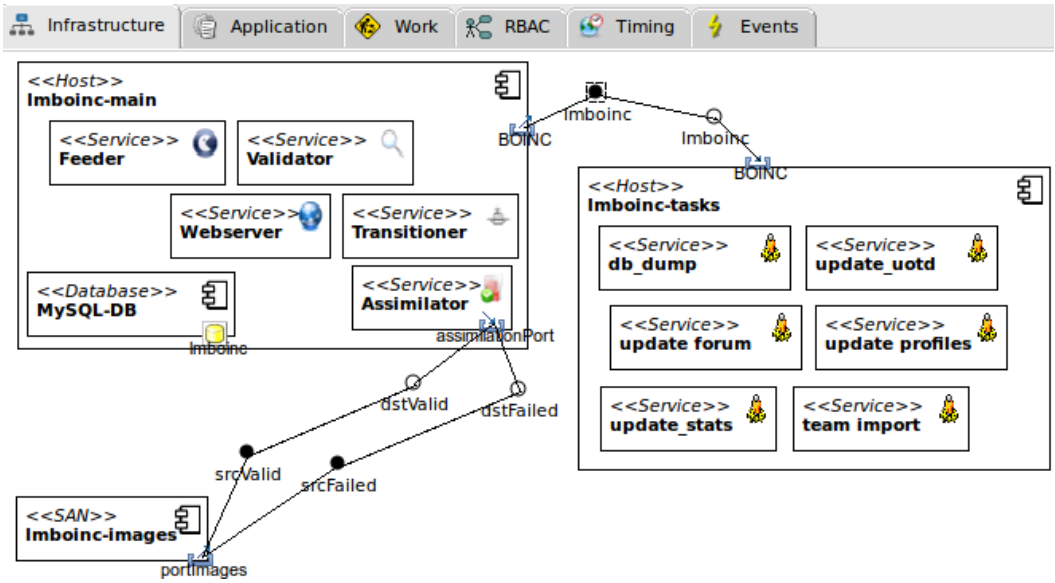

Figure 7.1: Example for the use of an Infrastructure diagram.

lmboinc-main possesses on the one hand an Assimilator as a service which obtains on the other hand a «PortImport» to store computational results in a specific

target. In this case, the import uses two exported «Share» interfaces of the storage area network *lmboinc-images*. The Assimilator can use any target directory or «Table» and «Column» descriptor of «DB» contexts (described later). The model interpreter or code-generation facility is responsible for handling different cases: (i) filesystem-based storing or (ii) database-based storing of computational results. The Assimilator can have unlimited import ports.

7.3 Semantic of the Application Modelling

A science application is specified by «ScientificApplication» and all supported target platforms must be added to *targetType*, e.g. *#Linux32* or *#Linux64*. Software packages and libraries can be specified by additional «Software» associations. The lifetime of a scientific application can be expressed by a UML StateMachine. Fig. 7.2 shows the investigated state machine which is used in order to express the lifetime of BOINC-based scientific applications. Section 6.6 has introduced and specified several stereotypes which are applied in this state machine. The execution of this example state machine starts at “Initialization” on the bottom left-hand side and follows the transitions. In “Ready“, a general set-up can be performed, e.g. executed, and repetitive methods can be added (Section 6.6.22) or handling of asynchronous-messages can be activated (Section 8.10). When the state “Processing” with its three regions is reached, the process will split into three orthogonally executed processes. Regions can only be part of UML States. In addition, every UML Region must offer its own UML States and UML Transitions [125, Chapter 15]. The bottom UML Region can have a history state which is used for BOINC’s checkpoint mechanism. When a computation is resumed and a checkpoint available, the history state is used to enter the computation. After restoring, the computation starts in the state “Computing” again.

In Fig. 7.2 the top and middle UML Regions are optional. The top is used for asynchronous-messages, the middle for checkpoint creation or updates when one checkpoint is already present. The structure of these two regions is minimalistic: they are entered and the process will idle until the transitions from “IdleCheckpointing” to “Checkpointing” or from “IdleReports” to “Reporting” are triggered. These two regions work in an iterative manner until the states “Checkpointing” or “Reporting” are finished or the idle states are entered again. The transitions from the left to the right can be activated by various triggers: (i) time triggers, e.g. after 60-seconds, or (ii) external triggers, e.g. from other states. The BOINC Manager can arrange time trigger defaults on any BOINC user or host.

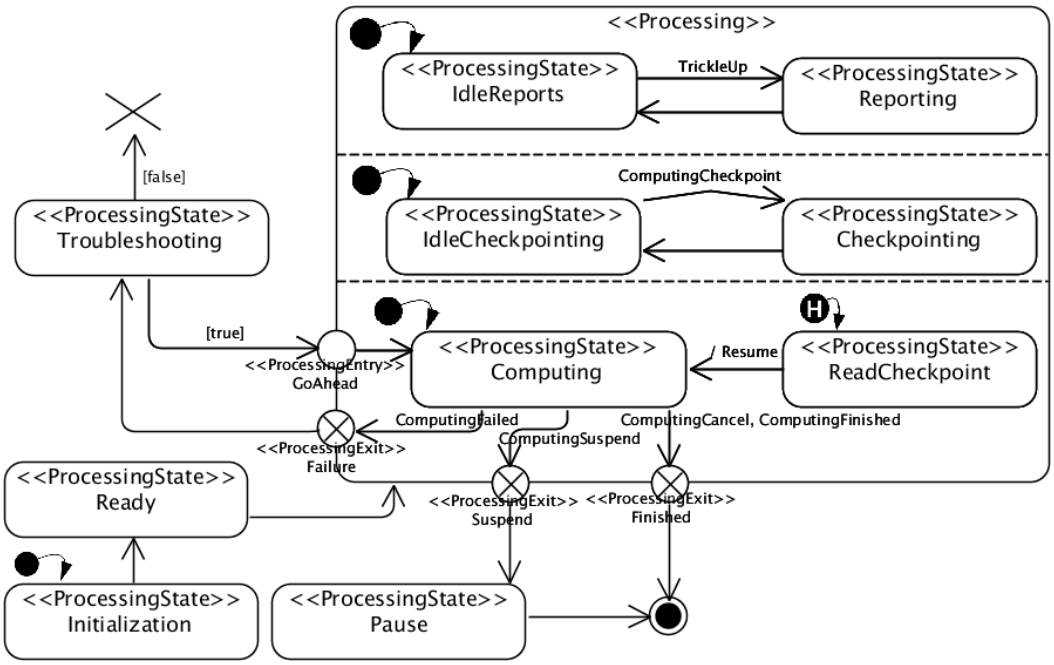

Figure 7.2: UML4BOINC's fundamental UML Statemachine for the lifetime of a BOINC scientific application.

7.3.1 Client's States

Several states are considered for scientific applications. It is not necessary that all states are used but in this thesis they are the pathfinder for the abstraction of runtime behaviour. Ten states will be introduced and analysed in total. Four of them, namely — *IdleCheckpointing*, *Checkpointing*, *IdleReports*, and *Reporting* — are, as mentioned in the previous section, optional.

Actions and Activities

The following actions or activities are recommended for the several state nodes:

Initialization: BOINC's client runtime is initialised at this stage, e.g. diagnostic flags are set, the target device (e.g. GPU) is opened and reserved, start parameters are parsed, the existence of all input files is checked, asynchronous-message handling is activated and the inter-process communication starts.

Ready: The initialisation is performed and the computation can start. In addition, pre-conditioning of Markov chain Monte Carlo (MCMC) methods can begin [80].

Computing: Computation's core is processed, i.e. the actions for resolving the engineering or scientific problem are executed.

Pause: Clean-up and creation of scientific application and specifically formatted checkpoint files.

ReadCheckpoint: An available checkpoint is opened and all datasets are read-in. The core computation can resume at the last saved position.

Troubleshooting: During computation progress failures can occur and thus handled at this state. If it is impossible to deal with the failure, the scientific application will be terminated or the computation resumed.

IdleCheckpointing: This state idles until a checkpoint has to be created or updated.

Checkpointing: A checkpoint will be created.

IdleReports: This state idles until one or more asynchronous-messages are received, or have to be transmitted. In addition, exchanges of datasets for the inter-process communication can be performed.

Reporting: Asynchronous-messages are handled and the inter-process communication is performed.

Transitions Events

The following Events are recommended concerning several transitions:

ComputingFailed: This event occurs when exceptions or general failures have been signalled during the computation.

ComputingCancel: This event is accomplished by the BOINC-client.

ComputingFinished: The computation has finished, so this event must be sent from the scientific application implementation.

ComputingSuspend: This event is established by the BOINC-client.

ComputingCheckpoint: This event can be sent at any time from every external action. In general, it will be forwarded in a timing interval which can be initialised in the state “Initialization”.

TrickleUp: A trickle-message is ready for transmitting.

Transitions Actions

The following action is recommended concerning several transitions:

Resume: The computation has to be started at a specific point which enables the restoration of the checkpoint datasets.

7.3.2 Computing on the Client Side

It is important to note that the execution is encapsulated in a specific namespace, i.e. the execution is within a slot on the client-side (Section 5.4). For all UML elements within an application where the execution's behaviour is affected by the developer or user, this behaviour influences only this slot, e.g. a UML Activity has the attribute isSingleExecution; when it is set to *true*, all invocations of the UML Activity are handled by the same execution instance [125, p.326]. The implementation of scientific applications can be performed by UML Activities and UML Actions [51]. For this purpose all UML ControlNodes are allowed to be used:

- decision/merge node
- fork/join node
- initial node
- activity/flow final nodes

Fig. 7.3 shows three different «ScientificApplication» instances. Each of them has a different *type* definition:

- the top one is typed as *#Application*,
- the middle one is typed as *#Screensaver*, and
- the bottom one is typed as *#Wrapper*.

The «ScientificApplication» in the middle has a final node; the other two have outgoing transitions with target nodes for sending signals. A screensaver is an external application and is only used in order to visualise the computation process for users (Section 5.4.2). These three different applications are connected to a UML DataStoreNode [125, Chapter 12.3.21] and extended by «Exchanger». UML's DataStoreNode is used for data exchange between applications on BOINC's client-side. Whereas all actions within activities that are extended by «ScientificApplication» are allowed to synchronise data values with this UML DataStoreNode, the «ScientificApplication» types as *#Screensaver* have solely read-only access. For this purpose, a UML ObjectFlow which is extended by «Sync» (Section 6.6.21) and a UML OutputPin/InputPin which is extended by «Exchange» (Section 6.6.7) are specified.

UML4BOINC specifies that final nodes for «ScientificApplication» are optional. It is allowed to send signals in order to inform other parts of the application when the execution has finished or when a failure occured. The represented state machine in Fig. 7.2 catches these signals and acts on them. When the computation

is finished successfully, the signal “ComputingFinished” is sent, if not, “ComputingFailed” will be forwarded.

The state machine decides thereupon which exit node is used. “ComputingFinished” uses the exit node “Finished” and the state machine ends; “ComputingFailed” uses “Failure” as the exit node. An error handling can be used in this case. In addition, the state “Troubleshooting” is activated in order to handle failures. If the failure can be handled and the computation resumed, the entry node “GoAhead” will resume the computation; otherwise the state machine is terminated.

The start-up of «ScientificApplication» needs only a small set of file information: (i) checkpoint, (ii) input files and (iii) output files descriptions. Section 6.6.3 introduces the first one. Section 6.8 introduces stereotypes for the other two descriptions. Fig. 7.3 shows an application with three input files at the top: (i) **f1**, (2) **f2**, and (3) **c**. The first two are input files which are defined by the workunit input and the output files descriptions; a «Checkpoint» file can be used optionally. The input files are provided as UML ObjectNodes; thus, the embedded data can be selected for UML’s ObjectFlow of nodes and Parameters which respectively consist of a UML ParameterSet attached to «ScientificApplication». «ScientificApplication» typed as *#Wrapper* can be seen as an opaque application with a specific interface to the environment. During the execution of «ScientificApplication», the process can only be controlled by external signals and the «ScientificApplication» element can be checked by four Remote Procedure Calls (RPCs) as presented in Section 5.4: *Suspend*, *Resume*, *Quit* and *Abort*. Fig. 7.3 shows three of them.

7.3.3 Actions and Activities

BOINC offers a handful API functionalities. They enable the execution of methods and operations within an atomic context or in time periods, to query file information, to check the computation state and so-called fraction done values, to send and receive asynchronous-messages between BOINC participants or to manipulate physical files by their file name, e.g. to rename or to erase them. Fig. 7.4 shows an overview of all UML4BOINC’s specified stereotypes which support the development of BOINC-related scientific applications.

7.4 Semantic of the Workunit Modelling

BOINC projects use a fine-grained file-based system to set-up workunits. For computations with BOINC it is necessary to have one or more workunits which contain descriptive information about how computations can handle these workunits. Workunits can contain several files, e.g. additional configurations, datasets, defini-

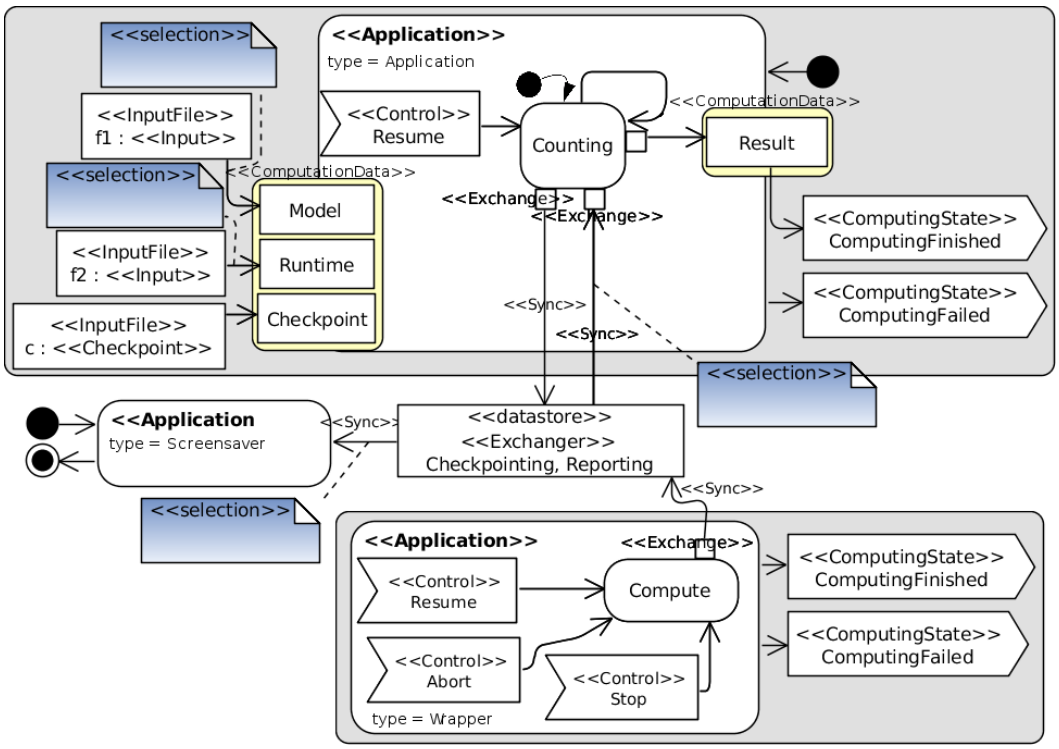

Figure 7.3: Three «ScientificApplication» instances with individual purposes: a manually modelled scientific application is situated at the top, a wrapper which is used for an Independent Software Vendor (ISV) application is at the bottom and a screensaver implementation in the middle. All three applications can exchange datasets through a DataStoreNode which is extended by «Exchanger».

tions of algorithms and arbitrary extra files. Section 6.8 introduces stereotypes to cope with these modelling aspects. UML4BOINC allows one to define a workunit's input, output files and multiple data fields for these files and how they are related within a computation process, e.g. if results are used for subsequent computations. With UML4BOINC, workunits can be specified by the Work package (Section 6.8) which includes all required modelling aspects to create workunit packages with all relevant information to perform specific computational tasks [31]. Before a workunit can be added to a BOINC project, it is necessary to create several input files for a computation and datasets. As a consequence, two additional template files are required for describing the structure of a workunit package:

1. Listing 7.1 demonstrates an input template example and describes which files are used as input and which flags are set, and

2. Listing 7.2 shows a result template example to describe how output files must be named by the scientific application or how many bytes they can allocate during the storing.

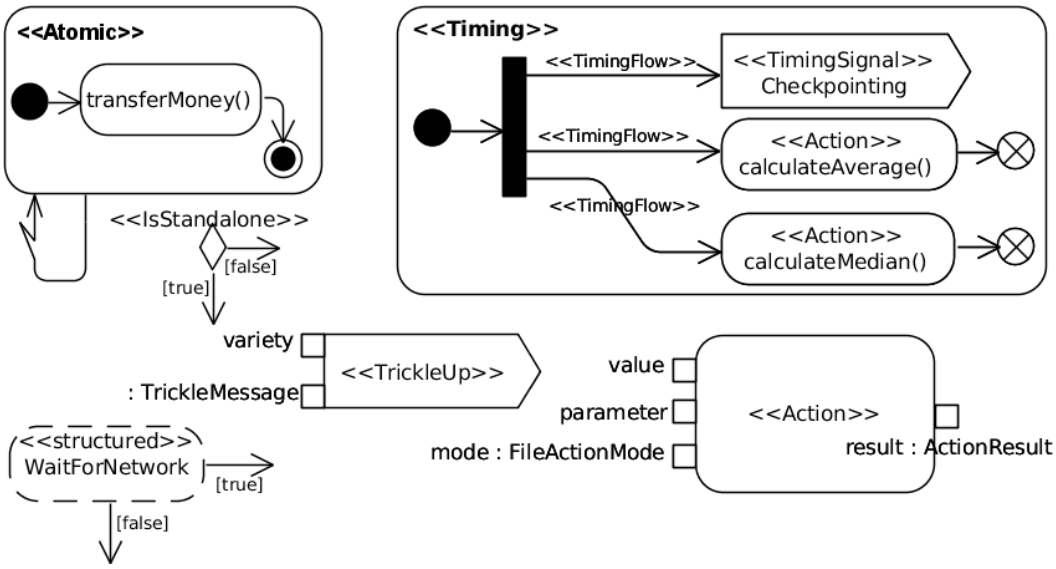

Figure 7.4: Eight stereotypes are defined for the abstraction of BOINC's general functionalities, e.g. to rename files or to query a network connection. «Action», situated on the bottom right-hand side, covers 13 functions which have to be activated by the tag-value *type* of «Action» and the input parameter *mode* (Section 6.6.1). «Atomic» is used to have atomic execution areas where the execution cannot be interrupted and has to be finished before subsequent actions are performed. The decision node «IsStandalone» is used to check if the execution of a scientific application is performed in a BOINC context or not (useful for executions with a testing purpose). «TrickleUp» is used to send asynchronous messages to a BOINC project. «WaitForNetwork» informs the BOINC user that a network connection is necessary to perform computational steps and that «Timing» and its nodes are executed periodically when a scientific application is performed.

```
<file_info>
  <number> 0 </number>
</file_info>
<file_info>
  <number> 1 </number>
</file_info>
<workunit>
 <file_ref>
  <file_number>                    0 </file_number>
    <open_name>          lmboinc.xml </open_name>
 </file_ref>
 <file_ref>
  <file_number>                    1 </file_number>
    <open_name>       archive_in.zip </open_name>
 </file_ref>
         <rsc_fpops_est>               1e+10 </rsc_fpops_est>
       <rsc_fpops_bound>               1e+11 </rsc_fpops_bound>
      <rsc_memory_bound>               1e+08 </rsc_memory_bound>
        <rsc_disk_bound>               1e+08 </rsc_disk_bound>
           <delay_bound>              604800 </delay_bound>
            <min_quorum>                   2 </min_quorum>
       <target_nresults>                   3 </target_nresults>
     <max_error_results>                   1 </max_error_results>
     <max_total_results>                  10 </max_total_results>
   <max_success_results>                  10 </max_success_results>
</workunit>
```

Listing 7.1: An example of a BOINC template input file; it describes two input files (`lmboinc.xml` and `archive_in.zip`), how many floating-point operations per seconds (flops) are estimated for the complete computation, how much memory in bytes can be allocated or how long the computation can take. Additional information is used for the server-side scheduling of workunits: either the minimal amount of valid workunit results required in order to validate the individual workunit or the number of distribution or it will be flagged as failed (Section 5.3.9).

```
<file_info>
                  <name>          <OUTFILE_0/> </name>
            <max_nbytes>            1000000000 </max_nbytes>
                  <url>         <UPLOAD_URL/> </url>
                       <generated_locally/>
                     <upload_when_present/>
</file_info>
<result>
 <file_ref>
  <file_name> <OUTFILE_0/> </file_name>
  <open_name> archive_out.zip </open_name>
  <copy_file/>
 </file_ref>
</result>
```

Listing 7.2: An example of a BOINC template output file; it specifies the name of the result file (`archive_out.zip`) or how many bytes the result file can allocate on the hard-disk drive.

Fig. 6.13 shows the introduced stereotypes which allow specifying the structure of workunits and how they can be opened and stored. «Input» is used to describe input files and «Output» describes result files. A workunit can own several file instances and each of them can have a distinct «Datafield». «Datafield» is used to describe data values for input files. The data format is not restricted and, for this reason, two methods are defined:

1. *open()* is used to access datasets and

2. *store()* is used to store datasets.

The reason for these additional methods is that the embedded data can have different formats, i.e. values have to be encrypted during saving or specific embedded function-calls must be used during data access in case a file is packed as a ZIP-archive [168, Zip_(file_format)]. These functions can be set by a developer to supply special opening and storing methods for currently unknown data types. There is no reason to allow a «Datafield» to be used by «Input» and «Output» at the same time. As a consequence, only one owner of the root «Datafield» is allowed. This root and all other instances of «Datafield» have two associations which can be used to create tree structures with several pieces of information for a workunit which is embedded in an «Input» file. This methodology enables different structures, e.g. a XML-structure can be created as shown in Listing 7.3. The use of these associations is restricted. If one «Datafield» is associated by *attributes*, then it cannot have additional associations. Each «Datafield» has the tag-values *name*, *type*, *data*, and *optional*. At any rate, *name* must be user-defined when a «Datafield» is used, whereas the other tag-values are optional and their use depends on the function. Listing 7.3 shows an example of the first three tag-values [54]:

name: *person*, *interests*, and *topics* are names,

type: *"C.B.Ries"* and *"Research, Sport"* are of the enumeration type *FileType::String*, and

data: mentioned string values are the real embedded information.

```
<person name="C.B.Ries">
  <interests topics="Research, Sport"/>
</person>
```

Listing 7.3: Example of «Datafield» usage to define a XML structure.

7.4.1 Data values of Workunits

«Range» is used for the specification of how attributes of workunit input files are valued in a dynamically executed context. During workunit creation, data fields of input files or the input files themselves can be specified by «Range». Thus values can be generated by «Range» specializations:

(a) «RangeRule» (Section 6.8.6) and

(b) «RangeSimple» (Section 6.8.7).

«Range» is an abstract classifier and must be specialised by «RangeRule» or «RangeSimple» or an individual specialisation. «RangeRule» can specify a rule-set for value creation. Accordingly, the tag-value *rule* can be filled with a user-defined rule, e.g. each workunit within a specific «Series» can have a corresponding mode for algorithms. «Range» specifies an operation *getValue()* which is used to query the related «Datafield» value. As shown in Fig. 6.13, each rule can modify only one «Datafield». «RangeSimple» is used to have a range-loop for one specific «Datafield»; three tag-values are specified for this scope: (1) *start*, (2) *stop*, and (3) *step*. In combination with different additional rules each call of *getValue()* can increment the lower-bound value *start* by *step* to the upper-bound *stop*. One «Range» can be owned by several «Series». As a consequence it must be possible to retrieve which «Series» is calling *getValue()*. For this reason tag-value *activated* is defined. When this tag-value is valued by *true*, the association between «Range» and «Series» can be used to query the currently used «Series». This allows «Range» to access all information of a «Series» with associated «Workunits».

7.4.2 Lifetime of Workunits

Fig. 7.5 shows the first state machine diagram for UML4BOINC's workunit modelling approach. Depending on the computation scenario covered in the BOINC project, it is possible to define how workunits are created. The state machine will always start at the initial point in the top-left corner and the process is directly subdivided into two parts with two transitions stereotyped by «WorkunitTransition». BOINC's workunits are working on the client side and independently from other processes. In addition, they are structured elements and in this regard not conceived to have behaviour. As a consequence, other components which may possess behaviour definitions have to deal with them. As all workunits are public within BOINC's domain, every component can have access and modify them. «WorkunitTransition» in mode *Detach* creates two orthogonal regions for the lifetime monitoring of a «Series» and all associated workunits. The top region is responsible for workunit monitoring and the bottom region controls the current «Series». The

transition between “Idle” and “Process” is triggered by *dataAvailable* and *nextSequence*. *dataAvailable* is called by the «InterfaceDataSource» and thereupon the file path is used to define a new workunit. Fig. 7.6 shows the state machine of a single workunit. In that state machine “Assimilation” has a “Next” named exit pseudostate and *nextSequence* is triggered when this exit is entered. As a result, a new workunit is created. It is defined that this exit pseudostate can only be used when a workunit is part of a sequence as described in the next section. All available and new workunits are scattered and monitored at the initial point of the state machine. The top region turns to the left when no more workunits are in process. The bottom region monitors the lifetime of a «Series» and the “Processing” is only left under two circumstances: (1) the «Series» has to be cancelled and (2) the processing is complete and all results can be merged in “Finishing”, e.g. an average of all results of a Monte Carlo method computation can be calculated.

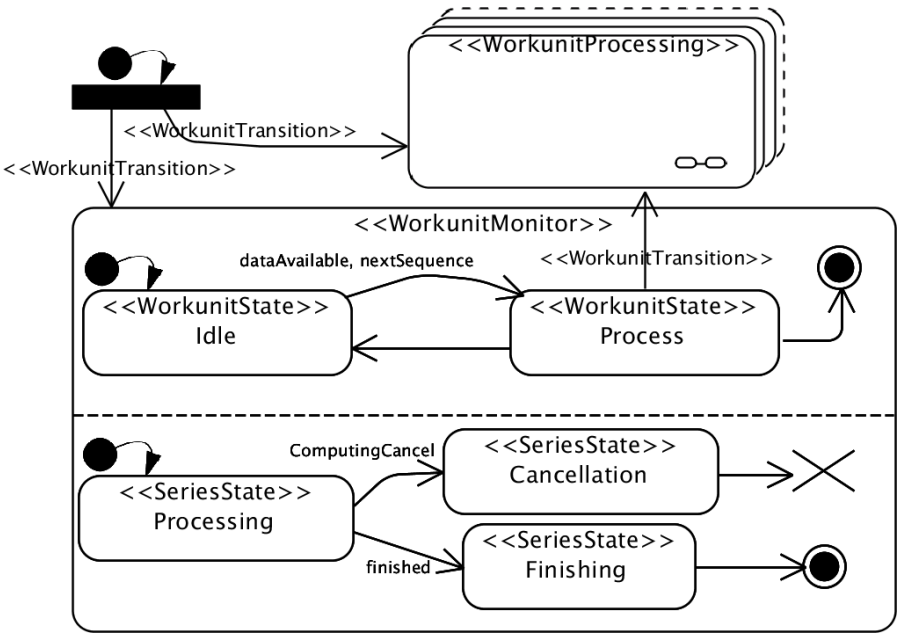

Figure 7.5: First part of UML4BOINC’s state machine for workunit creation and monitoring «Workunit» and «Series» instances. The State’s top region is responsible for the workunit monitoring and creates new workunits when either the data is available or the following workunit of a sequence has to be performed. The bottom region monitors a «Series»; it handles the cancelling events for a «Series» instance and states when it is finished or whether results can be merged.

7.4.3 Series Creation and Handling

«Series» allows the creation of different computational series, whereas the result of one series can be used by subsequent series. It is possible to associate individual

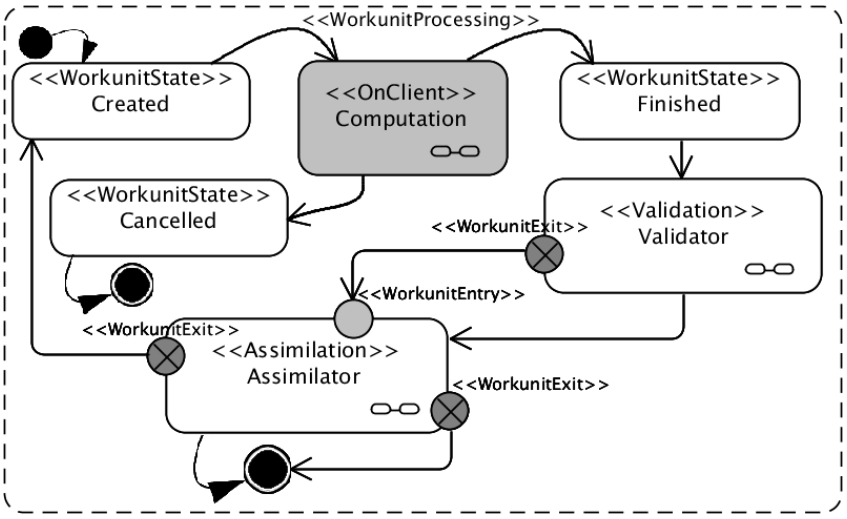

Figure 7.6: Second part of UML4BOINC's workunit state machine diagram. During the computation of a workunit, clients can decide to cancel a current workunit and thus need to change a workunit's state. It will be validated after completion; when the validation fails, the exit pseudostate is used. The following assimilation state repeats this workunit and creates a new one with same «Input» values. "Next" is used when the workunit reaches a sequence, otherwise the state machine is finished.

workunit results and specify them as input files for other workunits. «Series» can be used in three different approaches. Fig. 7.7 shows them:

- **Static Series:** All workunits are created before a «Series» will be created. Only these known workunits are handled by a BOINC project.
- **Continuous Series:** This configuration has no workunits at the beginning of a «Series». Workunits are created on demand, e.g. when new data packages are available or when a time slot is reached.
- **Dynamic Series:** In this configuration the previous two possibilities are merged.

Any «Series» instance has a *deadline* which specifies when it has to be finished.

7.5 Semantic of the RBAC Modelling

Fig. 7.8 shows the use of the Role-based Access Control (RBAC) stereotypes for a BOINC project. In [36], the authors give an idea of an approach to define a user control mechanism by the use of UML; each user can be added to roles and individual resources. This approach is adapted and extended a little. UML4BOINC

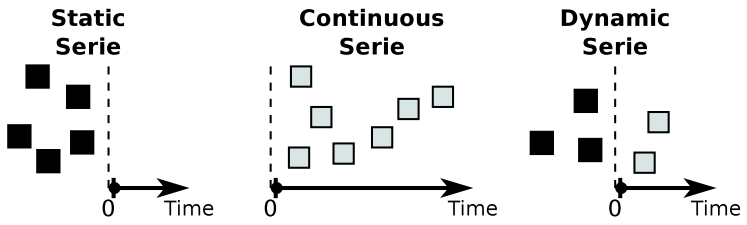

Figure 7.7: UML4BOINC allows «Series» to be defined in three different ways: (1) all workunits must be available before a BOINC project starts, (2) workunits are created and added continuously during runtime and (3) a mix of the first and second; some workunits are available at the beginning, and others are added to a BOINC project during runtime.

defines that permission rules can only be associated with elements which are extended by «Resource». Individual users can be defined with classes extended by «User», roles are defined with classes extended by «Role» and a set of permissions is defined with classes extended by «Permissions». Accordingly, there is a strict coherence of how associations must be used, i.e. «User» instances must be associated to «Roles» and these to «Permissions». Finally, «Permissions» could be associated with elements extended by «Resource». Fig. 6.3 shows a first example of how RBAC can be applied to models based on UML4BOINC. On the right-hand side, the BOINC project "lmboinc" and relevant information are shown. In this case, two resources are illustrated: (i) *Result handling* and (ii) *Workunit handling*. The RBAC describes restrictions of the users "C.B.Ries" (U1) and "C.Schröder" (U2. Both have different roles and possess an associated set of permissions. U1 has the permissions to *open*, *rename* or to *remove* computational results, whereas U2 is allowed to *create* new workunits and series.

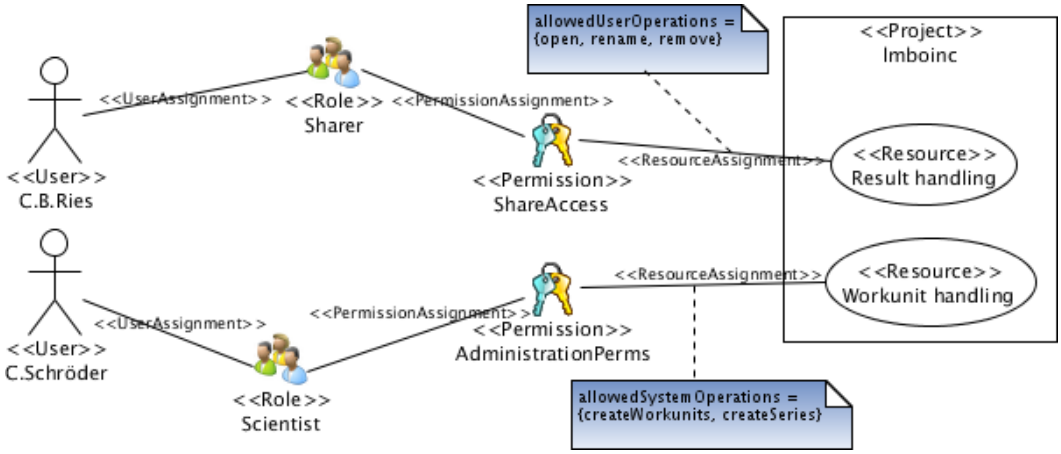

Figure 7.8: Example for a Role-based Access Control (RBAC).

7.6 Semantic of the Timing Modelling

BOINC's timing events are trivial and have no real-time requirements or requirement to be synchronised. The Timing diagram is only used to specify executed activities regularly. These activities can contain arbitrary actions. In particular, it is not of interest what the activities are doing during their execution, but only that they have an initial point for entering them. Fig. 7.9 shows seven tasks which are executed in four different periods: (1) each day, (2) once a week, (3) after 12-hours and (4) after 6-hours. From this specification, it is not clear when the tasks are really executed because this behaviour depends on two values:

- A task or an application will only be started when the previous run is finished. Therefore, the execution can be disabled for several periods due to long running activities.
- BOINC keeps track when the tasks are finally executed. The cron daemon [155, cron] tries to execute all tasks every five minutes as a default configuration. In this case the execution of the tasks can have an offset of five minutes from the first execution.

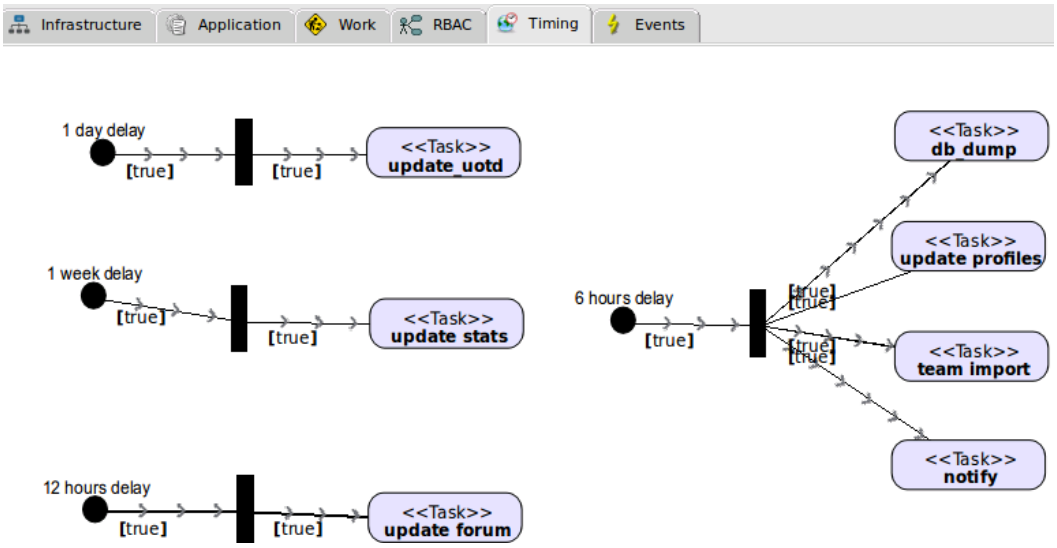

Figure 7.9: Timing diagram for timed Tasks which are executed in a specific period.

In order to handle this problem, it is possible to set a first start time by use of the tag-value *startTime* (Section 6.12.6). According to this, the tool vendor can recognise this value to set-up a specific tool for handling the execution or to start the cron daemon directly when the start time is reached. In addition, BOINC's five-minute default for the cron daemon can be configured automatically by the greatest common time value divisor of all available and enabled tasks. This will reduce

the number of checks for the cron daemon. The «EventFlow» connectors can be enabled or disabled at any time. *state* is used as a guard to decide whether the «EventFlow» is feasible for following the control flow. «EventFlow» connectors can connect «TimeForTask», «TaskFork» and «Task» elements. When the connector between «TimeForTask» and «TaskFork» is disabled, no following «Task» is executed. In the circumstances that the «EventFlow» is enabled but one of the three following connectors is disabled, only the connected «Task» of the disabled connector is not executed; the other tasks will be executed. One time expression can be used for several tasks. Hence, the event acceptor has to connect to a «TaskFork» element which will split the control flow; the individual flows work independently from the others and are used to execute tasks. The flow finishes after the execution of the task. «Task» itself is not implemented in the Timing diagram. It has to be performed within the Application diagram. The visualisation is realized within the Timing Diagram and referred to as a feature of the UML Classifier base class (Section 4.3.1).

7.7 Semantic of the Event Modelling

BOINC supports asynchronous-messages for which a BOINC service has to be added on the server-side; the client-side must support the handling of sending and receiving messages. The communication can be unidirectional or bidirectional. On the one hand, both communication partners can be configured as sender **or** receiver, further, they can support the sending **and** receiving of asynchronous-messages at the same time. The message handling is based on a BOINC service which is a single application on the server-side. This service has to be added to a host within the Infrastructure diagram. In particular, this service is used to handle specific *variety* identifiers (Section 6.6.25). Asynchronous-messages can be visualised by UML's sequence diagram. The BOINC project, registered users or hosts can be modelled by Lifelines. A third Lifeline can be used for user interactions, e.g. when the BOINC project administrator wants to cancel the processing of a specific workunit. UML4BOINC sequences can be added within the events diagram and used for asynchronous-messages or events. On the client side two different types of receivers are possible.

- A workunit or
- the user who is registered on a BOINC project.

In the first case only the workunit is affected, whereas the second case only sends messages to hosts of a specific user. Thus, the message is sent as a multicast to all hosts of a user. A user can be applied for a representation on the server- and/or

client-side. A «TrickleMessage» can consist of different states: *complete*, *lost* and *found* [125, p.508]. A message is lost when the receiver vanishes and the message cannot be delivered within a specific time range specified by the tag-value *TrickleMessage::timerange*. This value can be a specific time in the future or a delta value calculated by the *TrickleMessageValue::createtime*, i.e. *TrickleMessageValue::createtime* plus *TrickleMessage::timerange* is the time when a message has to be delivered. The recognition depends on the implementation of the UML4BOINC supporting tool.

Fig. 7.10 shows how the user "C.B.Ries" cancels the performing of workunits which are named "wu*"[1]. In this case, the message `1: cancelWorkunits` is sent to the asynchronous-message handler on the server-side. The handler relays this message as `1.1: cancelWorkunits` to all users which are registered and which are performing workunits of a specific scientific application. As a consequence, the workunits are cancelled and the execution of the scientific application is terminated. Section 8.10 shows how such a sequence diagram is transformed into executable code.

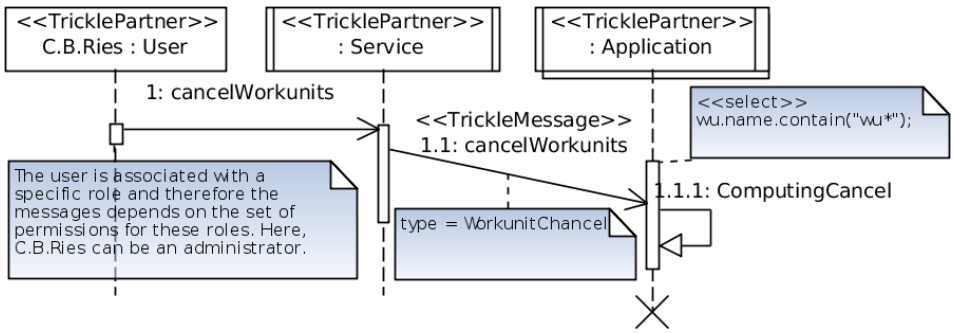

Figure 7.10: Illustration of sequences to cancel the performing of specific workunits; the user "C.B.Ries" cancels the execution of all workunits which are named "wu*".

7.8 Summary

This chapter presents the semantic of UML4BOINC. It has been demonstrated how the infrastructure of a BOINC project can be specified through the use of the Infrastructure package. The next step explains how the stereotype of the Application package is relevant in order to model a scientific application with different numbers of input/output files used. Furthermore, examples for the usage of the state machine and the handling of asynchronous-messages and the checkpointing have been presented thoroughly and the semantics for modelling activities and actions are introduced. The second half of this chapter demonstrates the semantic of work

[1]The star is a wildcard and means: all variants are selected.

handling, how permission control can be modelled and how timed tasks can be executed. The chapter closes with the handling of asynchronous-messages, which allow sending commands or general informal messages. Beside the concepts available in UML, the focus of this chapter is on the new language features described in Chapter 6.

CHAPTER 8

Code Generation Methodologies

In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever.

NIKLAUS WIRTH

8.1 Introduction

This chapter describes the verification process of the UML4BOINC stereotypes (Chapter 6) and the semantic (Chapter 7) of the stereotypes. Several methods enable the verification and this thesis presents three different methods (Section 8.2.1): (i) specifications of Domain-specific Modeling Languages (DSMLs), (ii) the use of C++-Models, and (iii) the use of Visual-Models created with Visu@lGrid [184]. As a consequence, specific code-generators for the transformation of these models are implemented into applicable parts for a BOINC project. As for the understanding of how the transformation is realised, a brief introduction about the language-recognition and how code-generators can be implemented by use of ANTLR (ANother Tool for Language Recognition) [89] is given. This chapter does not cover all transformations because most of them can vary, i.e. they depend on the target language (e.g. C++) and how tool-vendors handle semantic-models. In addition, steps three and four are realised within the research iterations (Section 1.4) of this thesis.

8.2 Code Generation

8.2.1 Approaches

The code-generation generator-backend is often realised by one of the following three approaches [44]:

Patterns: This approach allows a search pattern to be specified on the model graph. A specific output is generated for every match.

Templates: Code-generation is based on templates in most language workbenches or IDEs. As the name suggests, code-files are the basis in the target language. Expressions of a macro language are inserted or replacement patterns are used for specifying the generator instructions. Ordinary programming languages are often used in order to describe the behaviour of the generator and to state which instructions are used for the target code, e.g. Java or C/C++ in ANTLR.

Tree-traversing: This kind of code generation approach specifies a predetermined iteration path over the Abstract-syntax Tree (AST). Generation instructions are defined for every node type or leaf and executed when a node or leaf is passed. This kind of generation approach is mainly used in classical compiler construction and textual languages, e.g. as used by ANTLR in Section 8.2.2.

The three code generation approaches can be combined. Accordingly, this thesis applies a combination of the second and third approach, i.e. when the AST is created, the leaves of the AST are transformed into code-fragments and merged in existing template files. An AST is not always created before the templates are filled, in some cases the DSL-/C++-/Visual-Model (Section 8.1) are directly used to emit the BOINC parts, e.g. when the task configuration is generated for the server-side.

Fig. 8.1 shows the code-generation processing parts which are used in this thesis. In particular, seven parts are used to verify the UML4BOINC approach. Section 8.3 introduces the first part (1) and the result of it allows the generation of C++-code which is usable for the creation of semantic models. Such a semantic model can be created in several ways: (3) with a Domain-specific Modeling Language (DSML) description, (4) as a direct C++ implementation based on the previously created semantic model and (5) as a visual description, such as UML or with the help of UML4BOINC's specific tool Visu@lGrid as shown in Fig. 7.1. These three approaches conclude in (6) and describe a BOINC project. The description is transformed in different parts which are relevant for BOINC, e.g. configurations, implementations of services and scientific applications or the creation of workunits (Chapter 5). For the steps between (1), (2), (6), and the creation of BOINC parts, the software library *libries* (7) [185] is used.

8.2.2 ANother Tool for Language Recognition (ANTLR)

ANTLR (ANother Tool for Language Recognition) [89] is a parser generator used to implement DSL interpreters, compilers and other translators, e.g. for the han-

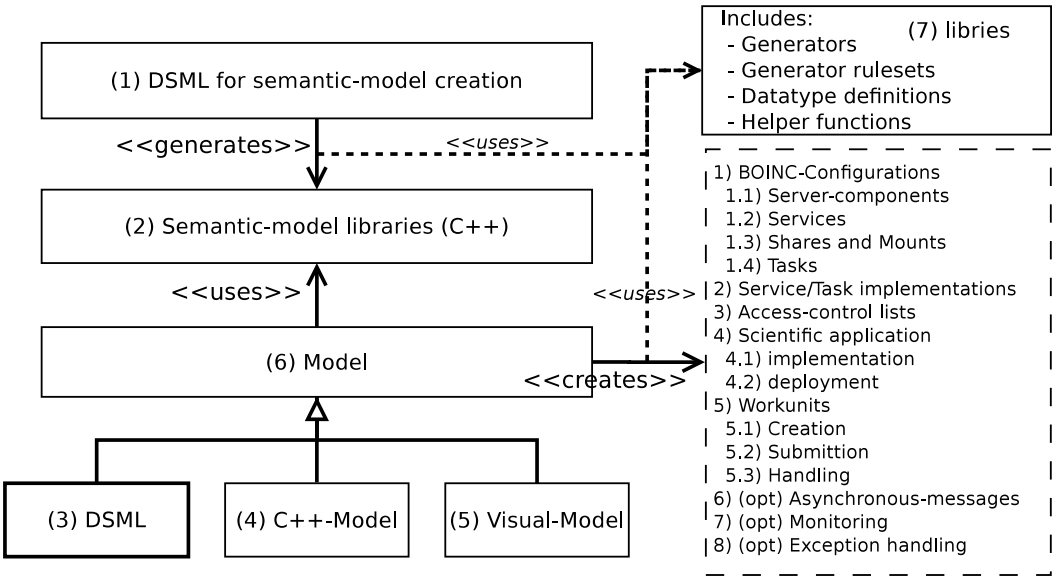

Figure 8.1: Architecture of the code-generation process; different models are used to generate a BOINC project with the components and configurations it uses.

dling of specific configuration file formats, network protocols, text-processing languages, protein patterns and genome sequences. A minimum set of two steps is necessary for language recognition: *lexical analysis* and *parsing* [21]. Fig. 8.2 shows how language recognition works in general. An input stream such as `1 + 21 => res` consists of different bytes. This stream can be filtered by a *lexer*[1] which will split the different independent bytes into constituent tokens. This process is based on a user-defined rule-set, a so-called syntax or grammar definition. The connection between the *lexer* and *parser* and the input stream is shown below. Every number is defined as an `INT` (integer), the plus sign as `ADD` (addition or summation) and => is used to assign (`ASSIGN`) 'something' to `res` (ID, an identifier or an arbitrary name). The underscores are only shown in order to visualise the gaps (so-called whitespace) between the tokens; they are not recognisable by the *parser*. At this stage, the input stream is only filtered and transformed into several tokens. The *parser*[2] is the core component for interpreting the first input stream. The allowed order of tokens is defined by the given syntax. Every token has its own meaning, e.g. `ADD` can be used to add-up or subtract two integers. In addition, the chain of different tokens can have a different meaning. The parser is used to interpret the tokens, i.e. either an optional Abstract-syntax Tree (Section 8.2.3) can be created and analysed or the tokens are interpreted in order to execute specific behaviours directly.

[1] A lexer is a tool for lexical analysis.

[2] The parser takes the list of tokens produced by the lexer and arranges them optionally into a tree-structure (called the syntax tree) that reflects the structure of the program [21].

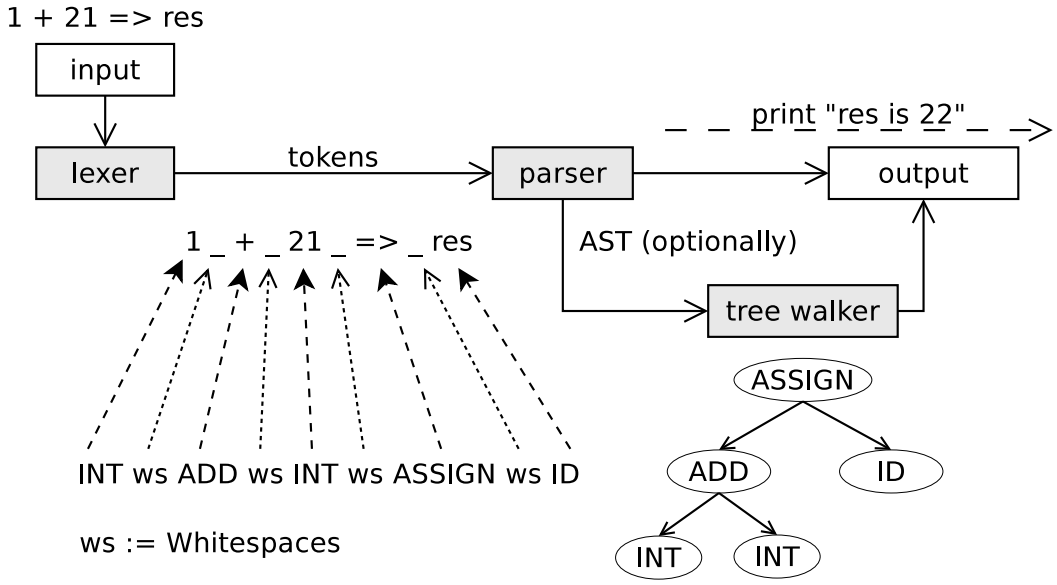

Figure 8.2: General process description of language recognition.

8.2.3 Abstract-syntax Tree (AST)

Generally, an Abstract-syntax Tree (AST) is an abstract representation of any formatted input or token stream. As illustrated in Fig. 8.2, the creation of an AST is optional. It is possible to parse, interpret and act on the input stream directly, i.e. for simple input it can be an easier approach. When the syntax and complexity of the input becomes more extensive, the transformation to an AST can make the interpretation more efficient, i.e. for the computer and more importantly for the developer. Furthermore, the maintaining syntax and relevant AST-structure become more manageable. The AST is created by adding mapping rules directly to the syntax of the new language:

```
multiplication: v1=INT '*' v2=INT -> ^(MUL $v1 $v2) ;
```

The operator -> introduces the tree construction rules. The body of the rule is a list where the first element is the node type (`MUL`) and followed by the child nodes which are the factors for the multiplication in this case [82]. By means of this approach two syntax trees are defined, i.e. an input syntax on the left-hand side and an AST-syntax on the right-hand side.

8.2.4 ANTLR and the Extended Backus-Naur Form (EBNF)

The EBNF (Extended Backus-Naur Form) is a syntactic metalanguage which is a notation for defining the syntax of a language through a set of rules. Each rule names a part of the language (called a non-terminal symbol of a language) and then

defines its possible forms. A terminal symbol of a language is an atom that cannot be split into smaller components of the language [119]. The EBNF has a small set of syntax rules [137] and ANTLR applies most of the EBNFs with modifications as shown in Table 8.1. Fig. 8.2 shows a small summation of two integer values and

	EBNF	ANTLR
‘a’ zero or one time	[a]	a?
‘a’ zero or more	{a}	a*
‘a’ minimum of one	a {a}	a+
‘a’ or ’b’	a | b	a | b
(‘a’ or ’b’) and ‘c’	(a | b) c	(a | b) c

Table 8.1: Excerpt of the comparison of EBNF’s and ANTLR’s syntax [89, 119, 136, 130].

an assignment to the variable *res*. Listing 8.1 states the EBNF-syntax of this small command. The first line defines which tokens are applied and how the addition has to be written by users. Any token is defined in the last five lines, i.e. integers only consist of numbers, the identifiers (IDs) can be written by a chain of characters, numbers and underscores but have to start with a character. The last line defines whitespaces (WSs) and includes an ANTLR specific part. It is possible to define channels which are used within ANTLR and during the lexical analysis and parsing process. In this case WSs are written to *HIDDEN*, i.e. they are removed from the token stream.

```
summation: INT ADD INT ASSIGN ID ;

ADD:      '+' ;
ASSIGN:   '=>' ;
INT:      '0'..'9'+ ;
ID:       ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS:       (' '|'\n'|'\r'|'\t')+ {$channel=HIDDEN;} ;
```

Listing 8.1: EBNF-syntax for the addition of two integers and assignment to a variable.

The main problem of this small example is that only one addition can consist of two integers. Listing 8.2 includes a small modification which allows creating unlimited lines of additions and chains of additions as shown in Listing 8.3.

```
summation: (INT (ADD INT)* ASSIGN ID)+ ;
```

Listing 8.2: Improved EBNF-syntax to allow unlimited additions.

```
10 + 11 => result_addition1
20 + 22 + 222 => result_addition2
```

Listing 8.3: Example of how to use the EBNF of Listing 8.2.

Finally, the ANTLR syntax enables direct AST creation. Listing 8.4 shows how the AST on the bottom right-hand side of Fig. 8.2 can be described. The ANTLR

operator -> is used to map the EBNF-syntax to a valid AST-syntax. The result of this mapping and its dataset is shown in Fig. 8.3. Appendix C.1 includes the complete sources for this example and can be used directly.

```
summation: leaf* -> leaf* ;
leaf: INT ('+' INT)* '=>' ID -> ^(ASSIGN ^(ADD INT*) ID);
```

Listing 8.4: Definition of the Abstract-syntax Tree (AST) of the summation in Fig. 8.2.

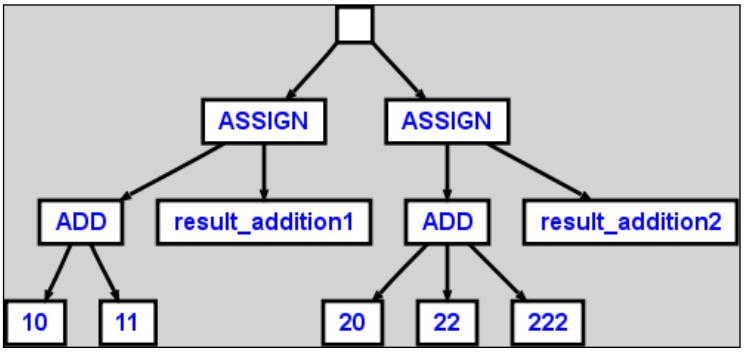

Figure 8.3: Abstract-syntax Tree (AST) for the summation of unlimited integer values.

8.2.5 Result

Fig. 8.2 shows the parser during the language-recognition and with two outgoing transitions: (1) a direct transition to output and (2) a detour of AST creation and subsequently finished by an output generation.

Parser ⇒ Output

This approach is also called "embedded translation" because it populates the semantic model from the embedding code to the parser; it implements the semantic model at appropriate points in the parse [82]. In this case the summation of Listing 8.3 is interpreted when it is read. Listing 8.5 shows a modification of the previous example in order to sum-up an unlimited amount of integers and for printing the result. Two variables are initialised at the start: (i) to store a variable name and (ii) to store the sum of the summation. Whenever a summation begins, the variables are initialised in a new way and previous values are not valid. Lines 6-7 are used to do the summation, in line 8 the name of the variable is cached in the variable `name`. Finally, line 10 prints the result of the summation. During the research process of this thesis, it was not easy to maintain the approach, especially when the syntax and the generation became more complex. The DSLs have been changed often and with the direct interpretation of the input the output generation must be adjusted within the generation process. By use of an AST between the interpreting

of input data and the generation of usable sourcecode the transformation can be unified. Changes of the DSL only influence the description how the AST has to be formed which is more compact in its description format as the generation of output.

```
leaf  // Java-based embedded translation
@init {
  String name = "";
  int value = 0;
} :
    v1=INT {value+=Integer.parseInt($v1.text);} ('+'
    v2=INT {value+=Integer.parseInt($v2.text);})*
    '=>' n=ID {name=$n.text;}
  {
    System.out.println(name + ": " + value);
  } -> ^(ASSIGN ^(ADD INT*) ID) ;
```

Listing 8.5: Example of an embedded translation.

Parser ⇒ AST ⇒ Output

The use of an additional AST can be more explicit and easier to maintain [98]. Listing 8.6 shows the same result as the previous example in Listing 8.5 but with less code and a clearer code view.

```
root  // Java-based AST translation
@init {
  String name = "";
  int value = 0;
} : ^(ASSIGN ^(ADD (v=INT{value+=Integer.parseInt($v.text);})*)
                   n=ID {name=$n.text;})
  {
    System.out.println(name + ": " + value);
  } ;
```

Listing 8.6: Example of an AST code-generation.

8.3 Generation of the Semantic Models

The research for this thesis required the implementation of different approaches, in order to verify and modify models in an iterative way (Section 1.4). All models are based on an object-oriented hierarchy implemented in the C++ programming language. The creation of the hierarchy is supported by a self-defined DSML and allows the specification of all necessary information in a compact manner: (i) datatype specifications, (ii) creation of enumerations, (iii) classes with attributes and methods, (iv) derivations of classes and (v) different kinds of associations between classes and their different multiplicities, i.e. compositions and aggregations. This approach allows any unscheduled requirement changes to be handled during the research process. One major advantage is that the creation of class-hierarchies

is much easier and the produced code has an improved quality. Furthermore, architectural changes to the class-hierarchy can be performed in the code-generator or its code-templates; thus, it is directly used for all parts of the class-hierarchy.

Listing 8.7 shows an example of the created DSML. The complete syntax for this DSML, its AST and parser can be found in Appendix C.1. Line 1 of Listing 8.7 specifies additional header files. Lines 3-4 indicate a mapping of used datatypes within the DSML which are therefore mapped in the target's programming language datatypes. Lines 6-8 specify an enumeration which can be used as a datatype within the DSL. The remaining lines specify three classes: *Project*, *Host* and *NIC*. *Project* has one attribute *name* and one association to one or more *Host* instances. *Host* itself is a composition of one or more *NIC* instances. The code-generation produces setter and getter methods and routines automatically in order to check the multiplicities during the adherence of instances to associations or compositions. Finally, every class gets factory and destroyer functionality [83].

```
1  includes { "General.h" }
2
3  datatype String  "std::string";
4  datatype Integer "int";
5
6  OperatingSystem [
7    Ubuntu = 1, UbuntuServer = 2
8  ] { /* empty */ }
9
10 Project {
11   name : String [1];
12   // composition application : Application [1];
13   association hosts : Host [1..*];
14 }
15
16 Host {
17   composition network : NIC [1..*];
18   // association : NIC [1..*];
19 }
20
21 NIC {
22   device : String [1];
23   ipvalue : String [0..1];
24 }
```

Listing 8.7: DSML description for the creation of a class-hierarchy.

Fig. 8.4 shows the resulting AST generated by Graphviz's tool *dot*, i.e. the alignment of all nodes is performed automatically and appears therefore unstructured. The root is the node *VGSMSTART* in the middle. The enumeration leaf is shown on the left-hand side, the directives and datatype specifications are below and the classes and attributes, compositions and associations are specified at the top-right-hand side.

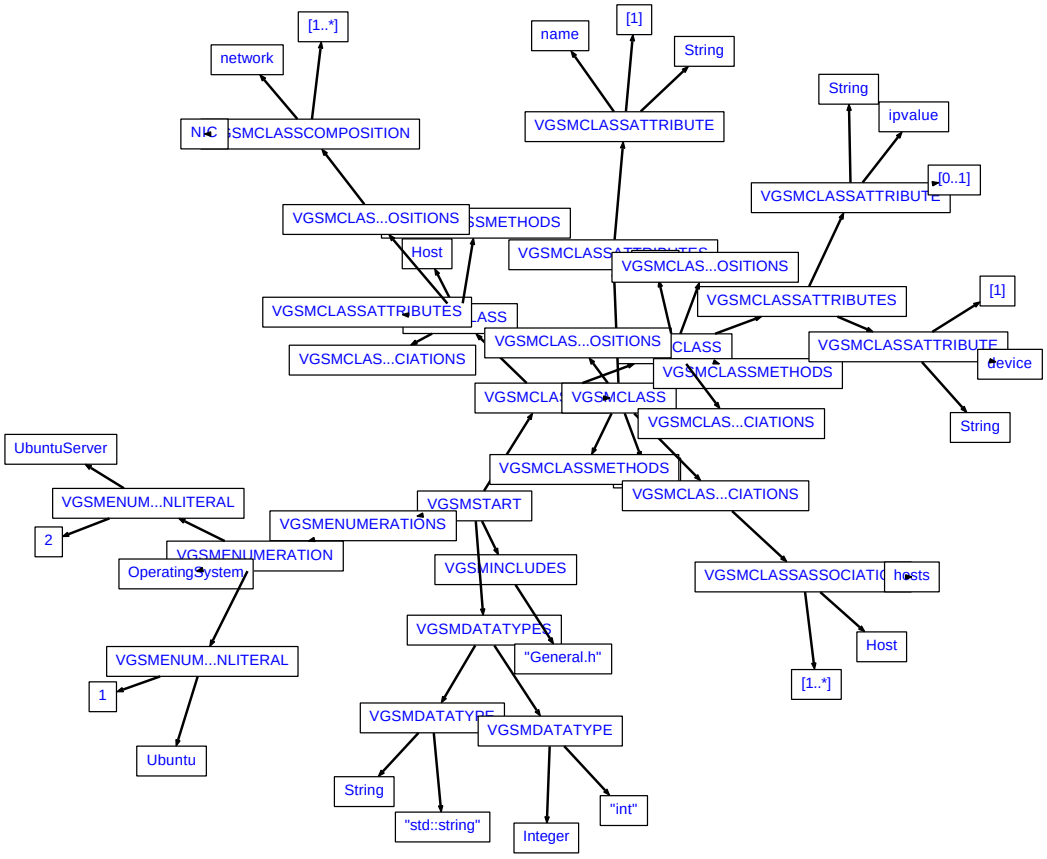

Figure 8.4: AST of the semantic model for the DSML description of Listing 8.7.

8.4 Generation of Statemachines

UML's StateMachine package has been used for the research of this thesis [125, Chapter 15]. There are some state machine implementations for the combination with C++ [178, 180, 72] which are partly based on the UML specification. All of them are restricted in their use (e.g. orthogonal regions are not supported) and do not support all ways of UML's StateMachine modelling (e.g. entry and exit points are not supported). Concerning this, the *UML 2 Statemachine for C++* (UML2STM4CPP) [183] and UML's StateMachine in version 2.4 support have been created and applied for all relevant scenarios. UML2STM4CPP is a developer framework which enables the implementation of applications with state machines as an implementation frame. By means of UML2STM4CPP only one DSL specification is needed to create executable applications for Linux, Mac OS X, and Microsoft Windows. Appendix C.2 includes the DSL-syntax. UML2STM4CPP hides the complexity of software systems through the modelling language. UML2STM4CPP's DSL – named after its author – *Ries Statemachine* (RSM) enables the creation of hierarchy state machine constructs. This DSL is

used and transformed into a C++ code. RSM is especially implemented for this research and can be used for other projects [183].

Fig. 8.5 shows a small state machine which is based on an initialisation state, choice state, two system states, and a final state. The initialisation state has only one outgoing transition and a choice state as target which possess three outgoing transitions: (a) two are guarded and (b) one is used as default. Choice states set up the separation of transitions into multiple outgoing paths. The decision is made by the transition's guards. When more than one guard evaluates to `true`, an arbitrary path is selected; but when there is no evaluation to `true`, the model is considered to be ill-formed. In order to avoid this case, it is recommended to define one outgoing transition as a predefined "else" guard for every choice vertex [125]. In

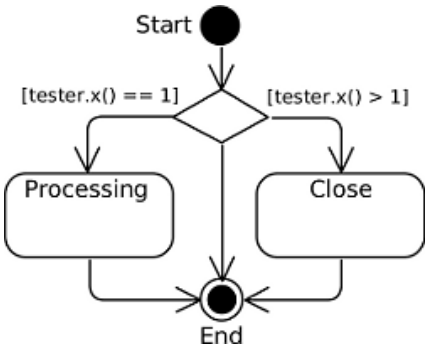

Figure 8.5: Use of UML Pseudostates with guarded transitions and a default transition.

this example *Processing* and *Close* do not include complex behaviours, i.e. they are used to show some information for internal available variables. When *End* is reached, the execution of the state machine finishes. The decision has two outgoing transitions with guards: (a) `tester.x() == 1` and (b) `tester.x() > 1`. `tester` is an instance of a so-called UML2STM4CPP anchor element as seen in lines 1-5. Anchors are used to fill state machine states with functionality. In appendix C.2, Listings C.7 and C.8 contain an example of a `tester`'s class-type.

Listing 8.8 shows the DSL definition for the state machine in Fig. 8.5. The state machine has two areas: (1) `States{...}` defines states and regions, and (2) `Transitions{...}` defines transitions, guards and events between states and regions. The DSL defines all used nodes, i.e. decision node, initial node, action node, and final node. Not all pseudo nodes are supported directly, i.e. they are simulated by the definition of relevant incoming and outgoing transitions, e.g. a decision node is just a node with one incoming transition and one or more guarded outgoing transitions. As a consequence, transitions can be created between a source node and a target node. The outgoing transitions are interpreted as a decision node. When the source node is finished, the guarded transitions are evaluated and the first true evaluated transition or the default transition is used. Finally, the

DSL is equipped with extra code anchors:

<[onEntry CODE -]> This code is executed when a state is entered.

<[- CODE -]> This code is the main part of a state. When this execution is finished, it is followed by an optional `onExit` code.

<[onExit CODE -]> This code is executed when the state is left. The execution is started and finished before the action of the next transition has to be performed.

The combination of these three anchors with the first anchor in lines 1-5 allows the fulfilling of the state machine with arbitrary behaviour.

```
1  Anchors <[--
2    #include <Tester.h>
3    Tester tester; int x;
4  --]>
5
6  Statemachine Pseudostates {
7    States {
8      Initial Start
9      Simple Processing <[onEntry tester.show(); --]>
10                       <[-- tester.processing(); --]>
11                       <[onExit  tester.show(); --]>
12     Simple Close
13     <[onEntry
14       x = tester.x(); tester.show(); tester.setX(345);
15     --]>
16     <[-- tester.close(); --]>
17     <[onExit
18       tester.setX(x); tester.show();
19     --]>
20     Final End
21   }
22
23   Transitions {
24   // Owner   source  + event [guard] / action == target
25   //-------------------------------------------------------------------
26      RegionMain: Start  [tester.x() == 1] / tester.setX(2)  == Processing
27      RegionMain: Start  [tester.x() > 1]  / tester.setX(3)  == Close
28      RegionMain: Start                                      == End
29      RegionMain: Processing                                 == End
30      RegionMain: Close                    / tester.show()   == End
31   }
32 }
```

Listing 8.8: A UML2STM4CPP DSL example of the state machine in Fig. 8.5.

8.5 Scientific Application

BOINC offers an Application-programming Interface (API) for the procedural implementation of scientific applications. UML is mainly an object-orientated specification and modelling approach. The developer is responsible for handling of

asynchronous-messages in the procedural approach, for sending status messages or the current state of computation to the BOINC client, for handling input and output files, for initialising the application and finishing the execution. Generally, some tasks have to be performed whenever a new scientific application has to be implemented. In order to avoid these incidents, an Object-orientated Programming (OOP) abstraction is implemented and set up on the top of BOINC's API [58]. Fig. 8.6 shows the abstraction layers of this approach. The OOP layer provides a top-down approach where most of BOINC's functionalities are merged in encapsulated objects and used as building blocks [185]. The biggest issue for

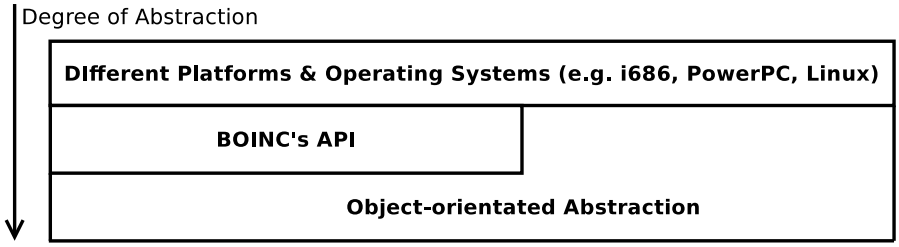

Figure 8.6: Object-orientated Programming layer for BOINC.

a scientific application is the computation's core. The core provides the "intelligence" and creates engineering or scientific results. As a matter of fact, it makes sense that only this core function has to be implemented for a scientific application and all other parts should be normalised or not implemented when not necessary to fulfil the computational task. Fig. 8.7 shows an excerpt of the OOP approach. In this case, two scientific applications are given: (i) *LMBoinc* (see Section 9.3) and (ii) *Spinhenge* (see Section 9.2). In both cases, only the `doWork` method has to be implemented. The computation is called within *Main* and is responsible for the instantiation of the BOINC framework and OOP layers, i.e. this can be a function, object or application implementation. This example specifies and initialises input and output files, variables for storing the checkpoint file and exchanging the interprocess communication. *Main* is instantiated by BOINC's client. The client has access to the `MVC::Handler` which provides information about the computation state; in addition, the handler receives the signals which are used to pause, resume, exit and abort the execution. In this thesis, *Main* is implemented as a state machine which is introduced by the UML stereotypes of Chapter 6 and shown in Fig. 7.2.

8.5.1 State Machine of a Scientific Application

Listing 8.9 contains the state machine which is specified by the RSM DSL from Section 8.4. The same names are used for the state nodes. The fundamental idea of the state machine is based on the fact that only relevant functions are used for creating various state nodes, i.e. the state "Initialization" is used for initializing

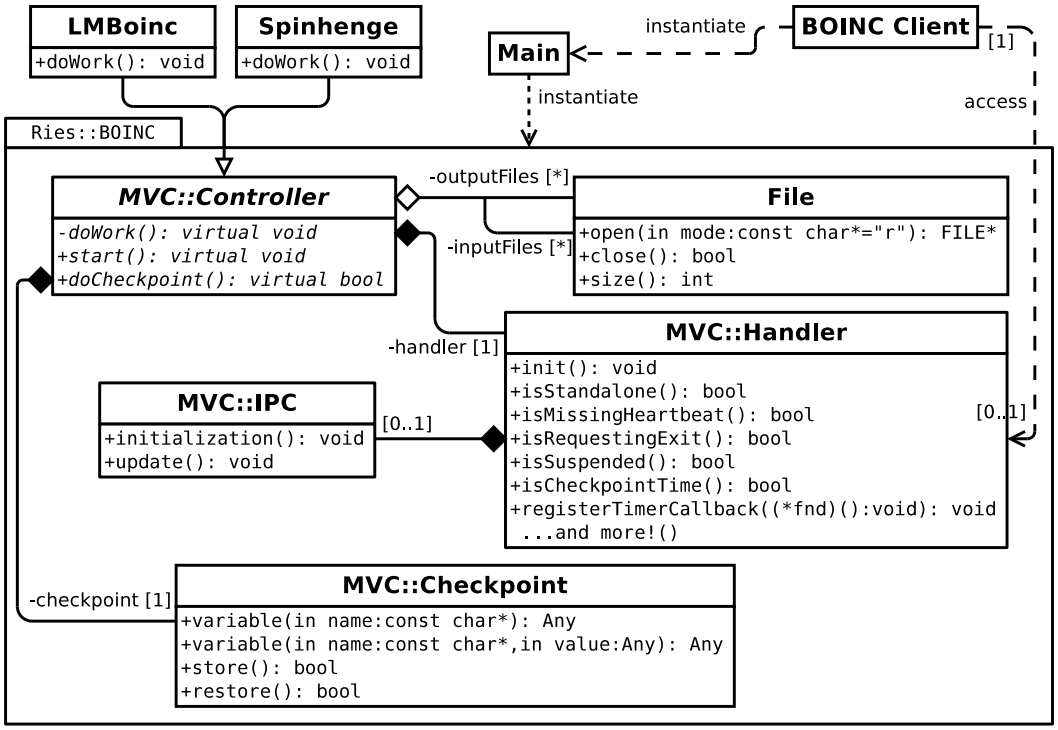

Figure 8.7: Object-orientated Abstraction of BOINC's procedural API.

BOINC's runtime environment or "Computing" is used for processing the core computation; thus, it will be called `doWork()` as in the previous Section 8.5. The state machine possesses three Regions:

R1: This Region is used for the reporting process and handling of asynchronous-messages,

R2: the second Region is used for the checkpointing process and the inter-process communication, and

R3: this Region specifies the computation, checkpointing and incoming and outgoing transitions of the scientific application.

The Transitions of the processing state and the three Regions can be triggered by external events and are able to call actions when they are used. Section 7.3.1 describes which implementation is recommended for these actions.

```
1 Anchors <[-- ... --]>
2
3 Statemachine ThesisStateMachine
4 {
5  States {
6   Initial Start
7   Simple  Initialization
8   Simple  Ready
```

```
9    Regions Processing {
10    R0 { States {
11     Initial Start0
12     Simple  IdleReports
13     Simple  Reporting
14    } }
15    R1 { States {
16     Initial Start1
17     Simple  IdleCheckpointing
18     Simple  Checkpointing
19    } }
20    R2 { States {
21     Initial Start2
22     History H2
23     Simple  ReadCheckpoint
24     Simple  Computing
25    } } :
26     Entry  GoAhead
27     Exit   Failure
28     Exit   Suspend
29     Exit   Finished
30    :
31   }
32   Simple    Pause
33   Simple    Troubleshooting
34   Final     Finish
35   Terminate Abort
36  }
37
38  Transitions {
39   // Owner source + event [guard] / action  == target
40   RegionMain: Star                          == Initialization
41   RegionMain: Initialization                == Ready
42   RegionMain: Ready                         == Processing
43
44   RegionMain: Failure                       == Troubleshooting
45   RegionMain: Troubleshooting [false]       == Abort
46   RegionMain: Troubleshooting [true]        == GoAhead
47   RegionMain: GoAhead                       == Computing
48
49   RegionMain: Suspend                       == Pause
50   RegionMain: Pause                         == Finish
51   RegionMain: Finished                      == Finish
52
53   R0: Start0                                == IdleReports
54   R0: IdleReport + TrickleUp                == Reporting
55   R0: Reporting                             == IdleReports
56
57   R1: Start1                                == IdleCheckpointing
58   R1: IdleCheckpointing + ComputingCheckpoint == Checkpointing
59   R1: Checkpointing                         == IdleCheckpointing
60
61   R2: Start2                                == Computing
62   R2: H2                                    == ReadCheckpoint
63   R2: ReadCheckpoint / Resume()             == Computing
64   R2: Computing + ComputingFailed                 == Failure
65   R2: Computing + ComputingSuspend                == Suspend
66   R2: Computing + ComputingFinished ComputingCancel == Finished
67  }
68 }
```

Listing 8.9: RSM Statemachine of the UML Statemachine of Fig. 7.2.

8.5.2 Mapping of Activities and Actions

Most of UML4BOINC's stereotypes can be mapped directly to BOINC's API-calls or with less code-complexity. Section 6.6 introduces a handful of stereotypes which are used to abstract BOINC's API functions. The current section illustrates how the code-generation from stereotypes to implementation code can be performed.

«Atomic» and its Code-Mapping

Atomic functions cannot be interrupted and have to be finished before subsequent actions or activities are executed. Listing 8.10 shows a small code-snippet. Line 2 opens and line 5 closes the atomic area; thus, any action or activity is called and cannot be interrupted between these calls. Fig. 8.8 shows a UML model consisting of an atomic area. It executes an activity with an initialisation and final node and then executes the "transferMoney()" action. An additional exception transition can be added in order to recognize the visualisation. It starts and ends with the atomic activity. Accordingly, exceptions cannot interrupt the execution.

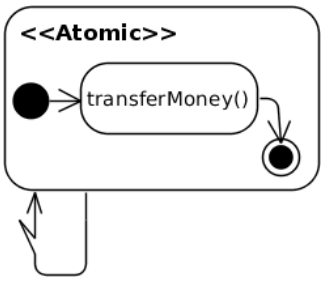

Figure 8.8: Code-mapping for UML4BOINC's «Atomic».

```
1 /// Start: <<Atomic>>
2 boinc_begin_critical_section();
3   transferMoney();
4   /*...*/
5 boinc_end_critical_section();
6 /// End: <<Atomic>>
```

Listing 8.10: Code-implementation example of BOINC's atomic mechanism.

«Action» & *type=#FractionDone* and its Code-Mapping

This type of «Action» is used in order to inform the BOINC client how much work has been accomplished. It is a value between zero and one, i.e. this value can be used as a percentage descriptor. The mapping of «Action» in BOINC's API is a single line: boinc_fraction_done(fdone). *fdone* is the input pin and requires a floating-point value (PrimitiveType::Real [126]) which has to be calculated by the scientific application developer.

«Action» & *type=#FileInfo* and its Code-Mapping

Section 6.6.10 introduces the stereotype «FileInformation» as an extension of UML's metaclass Class and with several tag-values and an operation. It is possible to query general information of specific files in combination with «Action» and *type = #FileInfo*. Listing 8.11 implements «FileInformation» directly as a C++-class, namely tag-values as attributes and the operations as methods. «Action» is mapped to line 29: (a) the input pin parameter *path* is used as `query`'s parameter, and (b) the output pin *finfo* as a new variable which assigns the returned value of `query`.

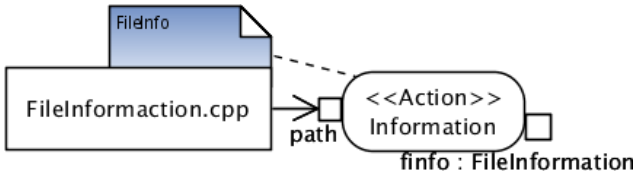

Figure 8.9: «Action» to query file information.

```
1  class FileInformation {
2  public:
3   enum FileKind {Unknown=0, File=1, Directory=2, Symlink=3};
4
5   static FileKind type;
6   static std::string cwd, absolutePath;
7   static double filesize, dirsize;
8   static double totalSpace, freeSpace;
9
10  static FileInformation query(const char *path) {
11   FileInformation fi;
12   if(is_file(path))         type = FileKind::File;
13   else if(is_dir(path))     type = FileKind::Directory;
14   else if(is_symlink(path)) type = FileKind::Symlink;
15   else                      type = Unknown;
16   char buffer[4096] = {'\0'};
17   boinc_getcwd(buffer); cwd = buffer;
18   relative_to_absolute(path, buffer); absolutePath = buffer;
19   file_size(path, filesize);
20   dir_size(path, dirsize);
21   get_filesystem_info(totalSpace, freeSpace);
22   return fi;
23  }
24 };
25
26 /* ...initialise default values! */
27
28 int main(int argc, char **argv) {
29  FileInformation finfo = FileInformation::query("FileInformation.cpp");
30  /*...*/
31 }
```

Listing 8.11: Code-mapping for UML4BOINC's stereotypes «FileInformation» and «Action» for file information queries.

«Action» & *type=#Locking* and its Code-Mapping

BOINC provides the functionality for preventing access to specific files, i.e. the BOINC-structure `FILE_LOCK` is used for locking and unlocking a file. Every file needs its own locking instance. Fig. 8.10 shows «Action» with one input pin to lock/unlock the file "FileInformation.cpp". The state of the lock/unlock-call is stored in the output pin *state* after execution. When something goes wrong, an exception called *ExceptionLocking*, which can be caught and handled individually, will be raised. Listing 8.12 shows an extension of `FILE_LOCK`. It will be extended by an exception handler when the locking or unlocking fails. Every locking mechanism is only usable in its context. When a global lock has to be used for specific files, it is necessary to define its lock instance in a global manner. The locking mechanism is implemented in lines 1-24- It must be provided by tool developers when being used. The instance `lockInstance01` realises the approach within the `main`-function. Lines 29-33 lock access to the file "FileInformation.cpp", whereas lines 37-41 unlock the file. Arbitrary function calls are possible between these parts and additional UML parts can be transformed into a code in order to fill this area.

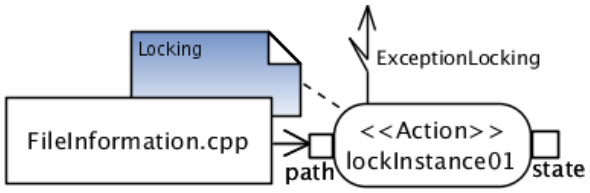

Figure 8.10: «Action» for locking/unlocking files.

```
1  class ExceptionLocking : public std::exception {
2   std::string m_errmsg;
3  public:
4   ExceptionLocking(const char *errmsg)
5    : std::exception(), m_errmsg(errmsg)
6   { /*...*/ }
7   ~ExceptionLocking() throw() { /*...*/ }
8   virtual const char* what() const throw() {
9    return m_errmsg.c_str();
10  }
11 };
12
13 struct Locking : public FILE_LOCK {
14  bool operator()(const char *path) {
15  if(!locked) {
16   if(lock(path) < 0)
17    throw ExceptionLocking("lock() failed");
18   } else {
19   if(unlock(path) < 0)
20    throw ExceptionLocking("unlock() failed");
21   }
22   return locked;
23  }
```

```
24 };
25
26 int main(int argc, char **argv) {
27  Locking lockInstance01;
28
29  try {
30   bool state = lockInstance01("FileInformation.cpp");
31  } catch(ExceptionLocking & ex) {
32   std::cout << "Locking failed: " << ex.what() << std::endl;
33  }
34
35  // ...do something!
36
37  try {
38   bool state = lockInstance01("FileInformation.cpp");
39  } catch(ExceptionLocking & ex) {
40   std::cout << "Locking failed: " << ex.what() << std::endl;
41  }
42 }
```

Listing 8.12: Code-mapping for locking/unlocking files.

«Action» & *type=#FileHandling* and its Code-Mapping

Fig. 8.11 and Fig. 8.12 show how UML modelling can be realised when UML4BOINC's «Action» and *type=#FileHandling* are used. «Action»'s characteristic is specified by the input pin *mode*; it states how many input pins can exist in one action. The notes in Fig. 8.11 and Fig. 8.12 describe the modes with one and two additional input pins. These actions and individual characteristics can be directly transformed into C++-implementations. Table 8.2 shows the related mapping of *mode*'s value to the suitable BOINC API-call. This is a trivial approach and some modes can need more specific implementation, e.g. the mode `Open` can be used for virtual filenames or/and for filenames with immediate access. In case the filename is a virtual name, it is necessary to resolve the physical filename. Listing 8.13 shows a general approach for file opening. Various problems can arise when the file is in a specific file format, e.g. it is a ZIP-archive and the opening differs from these approaches. Opening approaches can be handled directly by the code-generation or implemented in the OOP abstraction layer (Section 8.5). In this regard, code-generation is mostly a tool-specific task and can be handled in different ways.

```
1 std::string inputDataIn;
2 boinc_resolve_filename_s(path, inputDataIn);
3 int res = boinc_fopen(inputDataIn, filemode);
```

Listing 8.13: General code-implementation for file opening.

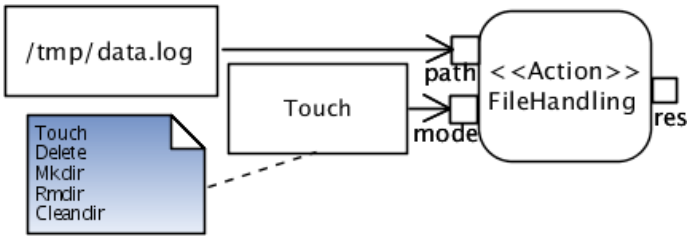

Figure 8.11: «Action» for file handling with two parameters.

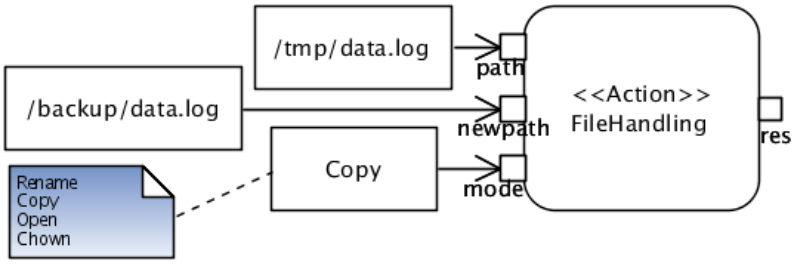

Figure 8.12: «Action» for file handling with three parameters.

Mode	BOINC's Function Call
	one additional input pin
Touch	`int res = boinc_touch_file(path);`
Delete	`int res = boinc_delete_file(path);`
Mkdir	`int res = boinc_mkdir(path);`
Rmdir	`int res = boinc_rmdir(path);`
Cleandir	`int res = clean_out_dir(path);`
	two additional input pins
Copy	`int res = boinc_copy(path, newpath);`
Chown	`int res = boinc_chown(path, owner);`
Rename	`int res = boinc_rename(path, newpath);`
Open	`FILE *res = boinc_fopen(path, filemode);`

Table 8.2: Mapping of UML to BOINC's API.

«Action» & *type=#TrickleDown* and its Code-Mapping

Trickle-messages which are received on the client-side are not handled immediately. The scientific application is responsible for collecting all trickle-messages; for this reason UML's AcceptEventAction [125, Section 12.4] is not used. The collection process can be performed in the first region of the state machine as in Fig. 7.2. The trickle-messages are provided by the output pin of «Action». Listing 8.14 shows the receiving of trickle-messages and a suggestion for handling them.

```
1  char trickleFilename[32] = {'\0'};
2  int retval = boinc_receive_trickle_down(trickleFilename, 32);
3  FILE *trickleFile = boinc_fopen(trickleFilename, "r");
4  if(trickleFile) {
5   /// (1) read in the trickle-message content
6   /// (2) parse the content
7   /// (3) create a specific TrickleMessage instance
8   ///     and fill this instance with content of the message
9  }
```

Listing 8.14: Approach of how the trickle-message handlers can be implemented on the client-side.

«TrickleUp» and its Code-Mapping

Fig. 8.13 shows UML4BOINC's stereotype «TrickleUp» for sending trickle-messages from every host to a BOINC project. Generally, this stereotype can only be used by clients. Listing 8.15 shows an example of how the stereotype can be implemented. A vector container for storing string values is defined in the first line. This container provides the planned content of the trickle-message which is created in lines 5-10. The sending itself is performed in line 11. The stereotype does not provide an output pin for handling the return value of the sending function. Thus, it is checked in order to enable error treatment. However, the reception of trickle-messages and their handling on the server-side require more work (Section 8.10). Thus, the content of the trickle-message must be parsed and handled individually. Section 6.11 describes the trickle-message stereotypes in more detail.

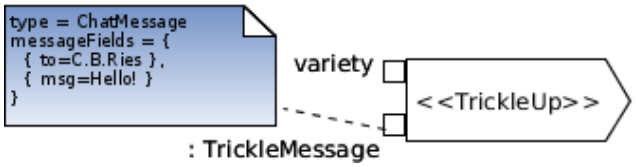

Figure 8.13: Code-mapping for UML4BOINC's «TrickleUp».

```
1  std::vector<std::string> messageFields;
2  /* ...fill messageFields with data! */
3  if(messageFields.size() >= 3) {
4   char msgUp[1024] = {'\0'};
5   snprintf(msgUp, 1024,
6     "<to>%s</to>\n<from>%s</from>\n<msg>%s</msg>\n",
7     messageFields[0],
8     messageFields[1],
9     messageFields[2]
10 }
11 int ret = boinc_send_trickle_up((char*)variety, msgUp);
12 if(ret) {
13   /// Error-handling, throwing of an exception, something else...
14 }
```

Listing 8.15: Code-implementation example for trickle-message sending on the client-side.

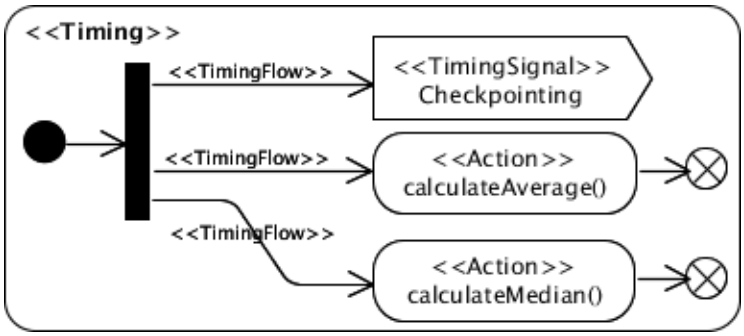

Figure 8.14: Code-mapping for UML4BOINC's «Timing».

«Timing» and its Code-Mapping

Periodically executed function-calls can be implemented in a scientific application, e.g. to calculate an average value or to handle input-/output calls. BOINC provides a mechanism which enables adding of specific functions. Accordingly, a function can be used as callback-function by BOINC's internal. UML4BOINC specifies «Timing» in order to add several different function calls as shown in Fig. 8.14. Only «Timing» has to be added to BOINC's callback mechanism. Listing 8.16 shows what the generated code of Fig. 8.14 can look like. The first three lines are dummy implementations and can be replaced by more complex action flows. The last execution is compared to its individual delay (Section 6.6.23) value (Line 5) by `timingFunction`. When the current time is less than the last `timingMoments` value plus the `timingDelays` value, then the action is executed. This implementation approach does not allow execution of the three functions in an orthogonal way. It depends on the code-generation facilities which are provided by tool-vendors. Finally, the `timingFunction` has to be registered for execution within BOINC's runtime environment and is performed in line 29.

```
1  void Checkpointing()     { /*...*/ }
2  void calculateAverage() { /*...*/ }
3  void calculateMedia()    { /*...*/ }
4
5  int timingDelays[]  { 15, 5, 30 };
6  int timingMoments[] {  0, 0,  0 };
7  FUNC_PTR timingFunctions[] {
8   Checkpointing, calculateAverage, calculateMedia
9  };
10
11 void timingFunction() {
12  static bool firstRun = true;
13  if(firstRun) {
14   for(int i=0; i<3; i++)
15    timingMoments[i] = time(NULL);
16   firstRun = false;
17  }
18  for(int i=0; i<3; i++) {
19  int timeEdge = timingMoments[i] + timingDelays[i];
20   if(timeEdge > time(NULL))
```

```
21      continue;
22      (timingFunctions[i])();
23      timingMoments[i] = time(NULL);
24    }
25  }
26
27  int main(int argc, char **argv) {
28    boinc_init();
29    boinc_register_timer_callback(timingFunction);
30
31    // Keeps the application running...
32    for(int i=0; i<600; i++) sleep(1);
33  }
```

Listing 8.16: Code-implementation example for «Timing» & «TimingFlow» and their embedded actions.

«WaitForNetwork» and its Code-Mapping

Listing 8.17 shows an approach of how «WaitForNetwork» can be implemented. The structural feature is directly created in lines 2-11. Fig. 8.15 shows the two outgoing transitions. The true-guarded transition is implemented by lines 14-17, and the false-guarded by lines 19-22.

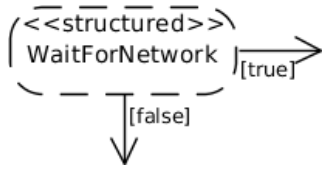

Figure 8.15: UML4BOINC's «WaitForNetwork» to query network connectivity.

```
1  /// Start: <<WaitForNetwork>>
2  #define TRIES 10
3  boinc_need_network();
4  bool netavailable = false;
5  for(int _try=0; _try < TRIES; _try++) {
6   if(boinc_network_poll()) {
7    netavailable = true;
8   } else {
9    netavailable = false;
10  }
11 }
12 /// End: <<WaitForNetwork>>
13
14 if(netavailable) {
15  /// Transition: [true]
16  /* ...do something on the network connection! */
17  std::cout << "␣NETWORK" << std::endl;
18 } else {
19  /// Transition: [false]
20  /* ...do something when no network is available! */
21  std::cout << "␣NO␣NETWORK" << std::endl;
22 }
```

Listing 8.17: Code-implementation example for «WaitForNetwork» to query network connectivity.

8.6 Services

Services are only present on the server-side of a BOINC project (Section 6.5.13). When they should be used by a BOINC project, it needs information about the individually added services: firstly, the location of the executable, its parameters for starting, and whether it is enabled or disabled; secondly, the application itself has to be implemented or a pre-implemented version has to be installed. Fig. 8.16 shows how these parts can be modelled with Visu@lGrid [184] diagrams (Section 6.4).

The services are modelled within the **(1)** Infrastructure and **(2)** Application diagrams; in case the service is a task the **(3)** Timing diagram is also relevant. The Task `db_dump` **(1.1)** is relevant in the following and these three diagrams enable the specification of the task configuration and implementation. Listing 8.18 shows the result of the generation process. Listing C.15 includes an excerpt of the relevant code-generation functionality of Visu@lGrid.

```
<tasks>
 <task>
  <cmd>db_dump -d 2 -dump_spec ../db_dump_spec.xml</cmd>
  <output>db_dump.out</output>
  <period>6 hours</period>
  <host>lmboinc-tasks</host>
  <disabled>0</disabled>
 </task>
</tasks>
```

Listing 8.18: BOINC's XML configuration for the BOINC task *db_dump*.

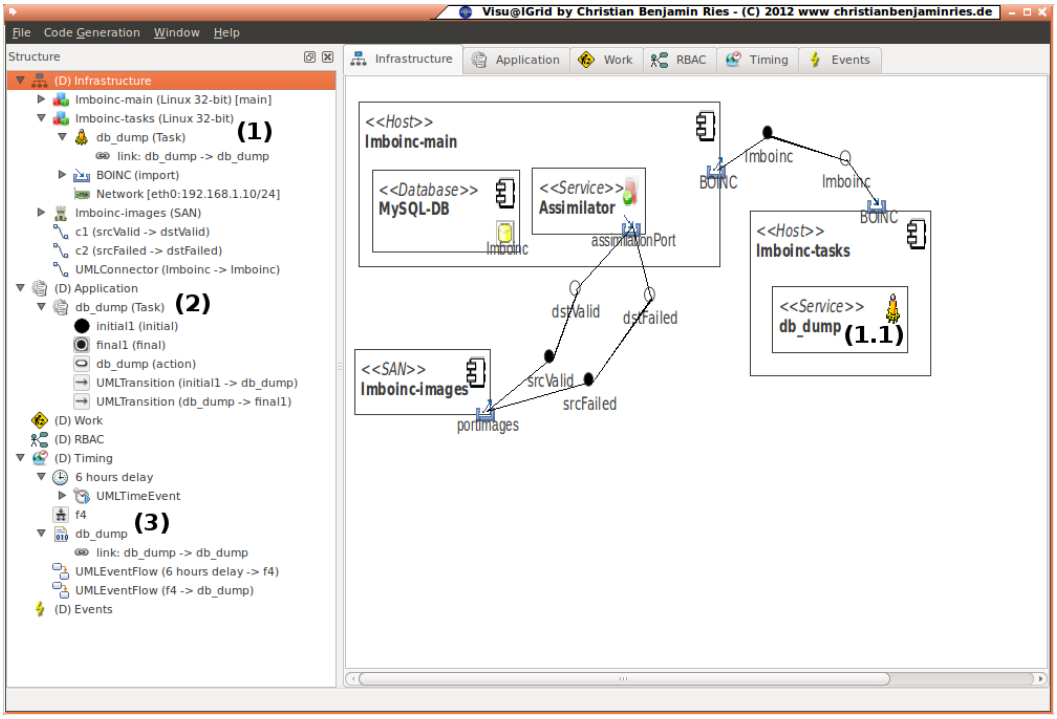

Figure 8.16: Visual modelling for the specification of BOINC services; the service configuration and implementation is generated next by use of UML4BOINC's code-generation approaches.

8.6.1 Visu@lGrid for BOINC

The tree-view situated on the left in Fig. 8.16 is based on UML4BOINC's specification. The following mapping is performed between Visu@lGrid's tree-view of Fig. 8.16 and Listing 8.18:

Line 2-8: Only Tasks which are part of a host are recognised. Every Task is embedded in the XML-block `tasks` with an individual XML-block `task`.

Line 3: This command is modelled by the Application diagram and can possess every action, i.e. position **(2)** of Fig. 8.16. The Task is modelled by an activity, exhibits one initial and final node, two transitions and the action for *db_dump*. A link which associates Task's execution location with the relevant implementation is created between the Infrastructure diagram (position **(1)**) and the activity, i.e. the host of the Task has to be executed as in line 6 of Listing 8.18.

Line 4: This value describes a tag-value of the Task element (Section 6.12), i.e. the sub-element *db_dump* of the tree-view element *(D) Application*. In particular it is specified by *(D) Timing* $\rightarrow$ *6 hours delay* $\rightarrow$ *UMLTimeEvent*.

Line 5: The period is specified by the Timing diagram at position **(3)**.

Line 6: The host of the executed Task is specified by the Infrastructure diagram.

Line 7: When a Task is enabled or disabled, it is specified by the tag-value *state* of «EventFlow» and within the Timing diagram at position **(3)** (Section 6.12.1).

This approach is used for all BOINC services with the exception that no modelling is necessary for the Timing diagram or for standard BOINC services such as the Feeder or Transitioner.

8.7 Work

BOINC's work processing is based on a small state-chain, namely *start* $\rightarrow$ *creation* $\rightarrow$ *processing* $\rightarrow$ *validation* $\rightarrow$ *assimilation* $\rightarrow$ *end*. Section 7.4 explains how the UML4BOINC stereotypes are applied in order to abstract this state-chain by a well-defined UML approach. Most of the tag-values of the stereotypes «Workunit», «Input» and «Output» are directly mapped to XML-configurations as shown in Listings 7.1 and 7.2. The file descriptions are based on the association *datafield* (Section 6.8) and this information is used to generate the structural parts of the workunits. The other stereotypes in the Work package create the behaviour of the workunit handling process and analyse how the input and output files have

to be used. Listing C.12 shows the relevant C++-implementation for verifying UML4BOINC's workunits stereotypes and semantics. The semantic model of this implementation is based on the UML4BOINC specification and subsequently generated by the generation process (Section 8.3). The implementation shows that the handling of workunits is separated into different areas:

Lines 130-165: The specification of one or more series,

Lines 168-197: the specification of datafields of workunits,

Lines 199-308: the specification of how individual workunits are structured,

Lines 310-346: which files of previous series are used by additional series,

Lines 348-371: that the series/workunits have to be created when the workunit handling process has started for the first time,

Lines 373-403: an iteration over all workunits and series to check whether they are completely finished and

Lines 414-529: when previous series/workunits are finished, series/workunits arise and need to be created and performed until they are finished.

Fig. 8.17 shows how series and workunits can be modelled by a visual approach provided by Visu@lGrid [184]. Different series can be created within the Work **(1)**. Every series contains a direct description of the workunit used and the input and output files. In this example the input file *lmboinc.xml* has four datafields to set-up the runtime parameter for the computation. The visualisation **(2)** shows the series with two input files and one output file. *images.zip* is a physical file which can be added manually or dynamically in this case and by the use of «InterfaceDataSource» (Section 6.9.3). The visual representation does not support this feature in the current version[3], i.e. it only supports the hard-coded link to a physical file at this stage. **(3)** and **(4)** show how the relation between the first four series and a fifth series can be described with Visu@lGrid. The output files of the series *s1* to *s4* are used as individual input files for the series *s5*. After the processing of the fifth series, the computational result is stored in *result.zip* of *s5*. The location is specified by the Infrastructure diagram and influenced by the association of the Assimilator to the destination storing place. Fig. 8.16 has an Assimilator with two specified targets: (a) *dstValid* and (b) *dstFailed*. When the computation of the workunits fails, it stores the results at *dstFailed* (long name is "destination Failed") but when they are validated correctly they are stored at *srcValid* (long name is "source Valid").

[3] Version 0.2 alpha

The implementation of the Validator and Assimilator is not performed in the Work but in the Application diagram. When an individual implementation or handling of workunits is necessary, then the series/workunits have to be modelled first. As a consequence, it is possible to possess a context sensitive menu within the application diagram which presents all specified input files and datafields automatically. The Validator and Assimilator code-generation approaches are shown in Sections 8.8 and 8.9.

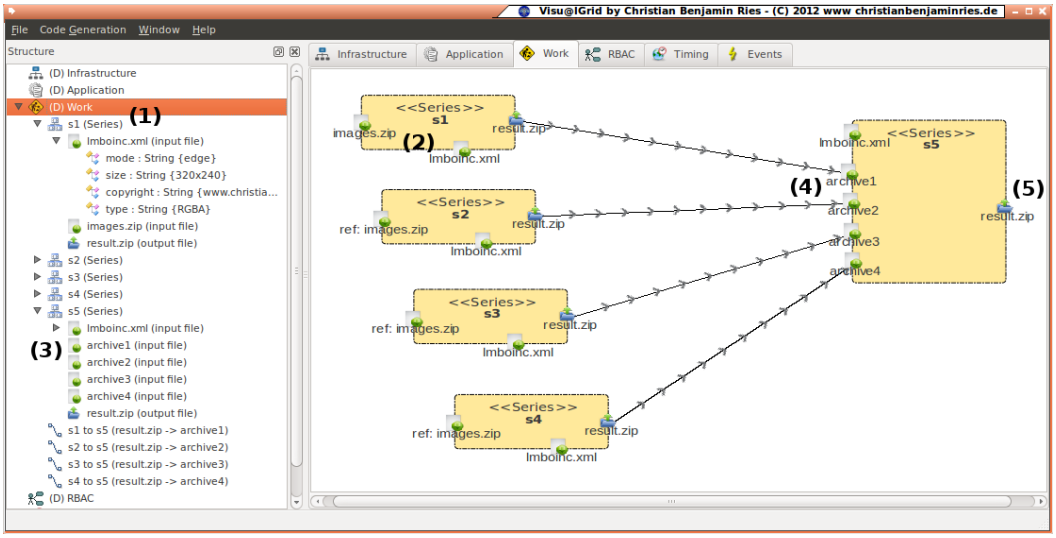

Figure 8.17: Visual modelling for the specification of BOINC workunits; The Series and the relation among themselves can be modelled with associations between input and output files.

8.8 Work Validation

Workunit specifications state, on the one hand, how output files have to be opened and, on the other, how the «InterfaceValidate» can be used. «InterfaceValidate» provides access to the output files for validation processes. Thus, the interface «InterfaceDataset» provides an universal way to open these output files. BOINC's validation framework exists on a high-level and low-level abstraction [91, Section 9.3]. This thesis covers only the high-level abstraction which is based on three steps: (i) *initialisation*, (ii) *comparing*, and (iii) *cleanup*. Only the second step has to be filled with functionality for a successful validation. As shown in Listing 8.19, the result has to be stored in the fifth parameter: (a) *false* or (b) *true*, i.e. the computational result is *not valid* or *is valid*. The first and second and third and fourth parameters are used to handle the result files and result raw datasets. Fig. 8.18 shows how Listing 8.19 can be modelled in UML by use of a UML Ac-

tivity. The parameters of the function `compare_results` are modelled as UML ParameterNodes. The comparing statement in line 5 is transformed into a UML DecisionNode and two outgoing transitions with equipment guards. These guards are the real validation parts and decide whether the computational results are valid or not, i.e. the output UML ParameterNode is valued with *false* or *true*. This approach can be added to the UML StateMachine and extended by «Validation» (Section 6.10.7). In particular, Fig. 6.15 presents, on the right-hand side, the possibility to add a user-defined validation process which can be filled by this approach.

```
1 int compare_results(RESULT & r1, void *data1,
2                      RESULT & r2, void *data2,
3                      bool & match)
4 {
5  match = (r1.cpu_time >= 10 && r2.cpu_time >= 10);
6  return 0;
7 }
```

Listing 8.19: Part of BOINC's high-level validation framework.

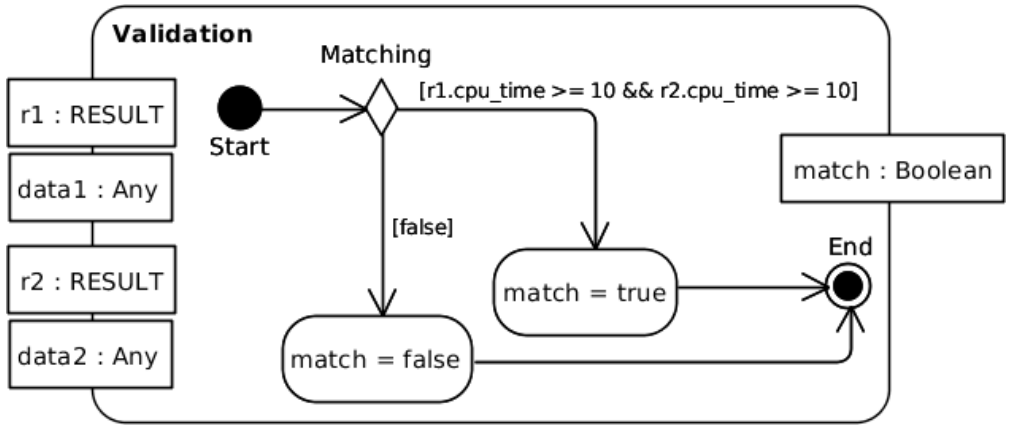

Figure 8.18: Part of BOINC's high-level validation framework in UML.

8.9 Work Assimilation

«InterfaceAssimilate» can be used for the assimilation of computational results (Section 6.9.1). It provides access to output files and then enables interface «InterfaceDataset» to open the output files. BOINC's assimilation framework provides a function which has to be filled with assimilation routines, i.e. by use of the standard implementation (Section 5.3.6) or custom specification for how output files and results have to be stored. Generally, BOINC's framework provides two targets for storing: (i) storing in a specific file-directory and (ii) storing in a database (Section 5.2). The destination for storing is specified within the code-generation process and by the Infrastructure diagram. The infrastructure specifi-

cation contains an «Assimilator» which can be extended by ports in order to use file-directories or database tables for storing.

8.10 Asynchronous Messages

This section gives an example of how asynchronous-messages can be transformed into code and by use of the relevant UML4BOINC stereotypes (Section 6.11). When asynchronous-messages are used on both sides, the message handling must be enabled in any case on the server- and client-side. Listing 8.20 shows the configuration necessary for the server-side and Listing 8.21 the BOINC API-calls for the client-side. The handling of trickle-messages is activated during the initialisation of the scientific application when on the client-side, i.e. the relevant option settings can be performed in the state “Initialization” (Section 7.3).

```
<boinc>
 <config>
  ...
  <msg_to_hosts/>
  ...
 </config>
 <tasks> ... </tasks>
 <daemons> ... </daemons>
</boinc>
```

Listing 8.20: XML-sequence for activating asynchronous-message handling on the server-side.

```
int main(int argc, char **argv) {
 BOINC_OPTIONS options;
 options.handle_trickle_ups = true;
 options.handle_trickle_downs = true;
 int returnValue = boinc_init_options(&options);
 ...
}
```

Listing 8.21: BOINC’s API-calls to enable asynchronous-message handling on the client-side.

8.10.1 Server-Side Handling

BOINC provides a default implementation for the handling of asynchronous-messages, i.e. an exemplary service which receives messages and sends them back to the sender; a kind of ping-pong system. This exemplary application iterates over a BOINC database table and queries unhandled asynchronous-messages. The queried messages are processed by a specific function and therefore named `handled_trickle(DB_MSG_FROM_HOST&)` (HT). The parameter is the database entry and contains information about the sender and the message sent,

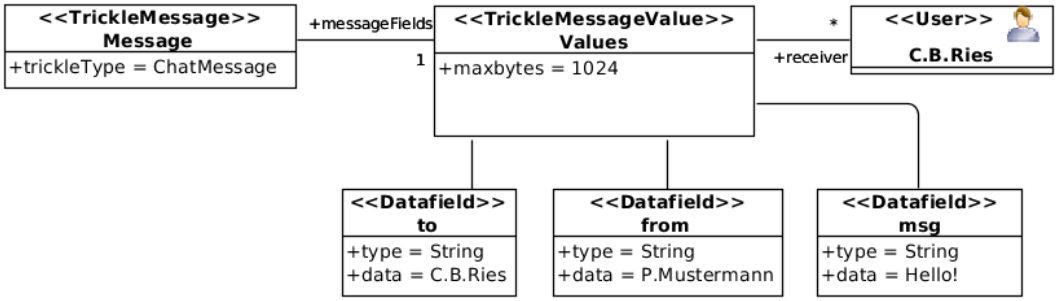

Figure 8.19: Instantiation of the UML4BOINC stereotypes for asynchronous-messages.

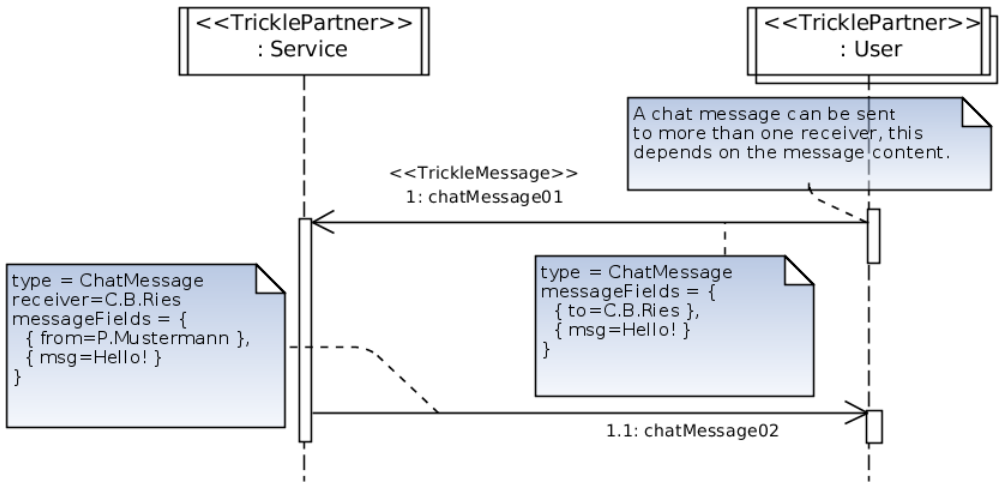

Figure 8.20: Asynchronous-messages between BOINC users and a BOINC project.

which is text-based. Fig. 8.19 shows how an asynchronous-message is specified by UML4BOINC. «TrickleMessage» specifies a chat message and contains one message value which is defined by three additional datafields: (i) *to*, (ii) *from*, and (iii) *msg*. This chat message has one receiver: *C.B.Ries*. The specification is not always available during runtime. Chat messages are transmitted primarily from users; the specification does not possess any association to receivers at this moment, only textual information about the planned receivers. The association is created on the server-side when message receivers are queried from the user database of a BOINC project. Fig. 8.20 shows this detail from another viewpoint. One user transmits the «TrickleMessage» *1: chatMessage01* to the BOINC project. The message contains only a description of the targeting message's receivers. The «TrickleMessage» *1.1: chatMessage02* has a different format. *messageFields* has modified the information into absolute information for the targeting receiver. The replacement of the targeting receiver information needs to be handled on the server-side. Additional UML Actions can be specified for this case. These actions are not defined in this thesis. Listing 8.22 shows the pseudocode of the relaying mecha-

nism for chat messages on the server-side. The full C++-implementation is shown in Listings C.16 to C.18. Information for the received message is filtered: (i) the sender host, (ii) the sending user, and (iii) the names of the target users. The received message is parsed into the format for the targeting users in the next step. Thus, the message is sent to all hosts of a user and accordingly to every workunit. As a matter of fact, BOINC's asynchronous-messages need information about the workunit and which message has to be handled. When a message shall target a specific user, the sender can not forsee when and where it will be read by the receiver. Accordingly, the server sends it to all workunits.

```
1  handle_trickle(MSG_FROM_HOST& mfh) {
2   hostFrom =  HOST(mfh)
3   userFrom =  USER(mfh)
4   usersTo  = USERS(mfh)
5
6   m = MAP_RECEIVED_MSG_TO_SEND_MSG(mfh)
7
8   do user = Send_Message_to_Users(usersTo)
9    do workunit = Send_Message_to_Workunit(user)
10    SEND_MESSAGE(workunit, m)
11   done
12  done
13 }
```

Listing 8.22: Pseudocode for the transfer of chat messages between users.

Listing 8.22 shows how little UML Action is needed in order to implement the pseudocode. It requires five actions and additional structural activities, i.e. a for-loop or do-while loop can be used to iterate over all users and workunits. Fig. 8.21 shows how this can be realised through UML. The received message is specified as a UML ObjectNode and used as input value for the three UML Actions: (i) `HOST`, (ii) `USER` and (iii) `USERS`. The first two are not used in this example. The third action is used in order to filter all users and query the information from the database. The user information is parsed to the first iterative element which consequently iterates over all users. The individual users are parsed to the second iterative; it enables iteration over all workunits of the specific users. Finally, a pre-parsed message is sent to a specific user's workunit.

8.10.2 Handling on the Client-Side

The handling of asynchronous-messages is less complex on the client- than on the server-side. The client only receives and transmits messages from and to a BOINC project. It must not query any other information from a database. The Transmission of asynchronous-messages is provided by «TrickleUp» (Section 6.6.25) and the reception is supplied by «Action» (Section 6.6.1). These two stereotypes can be combined. The handling of asynchronous-messages is introduced in Section 7.3 and the complete implementation is shown in Listing C.18.

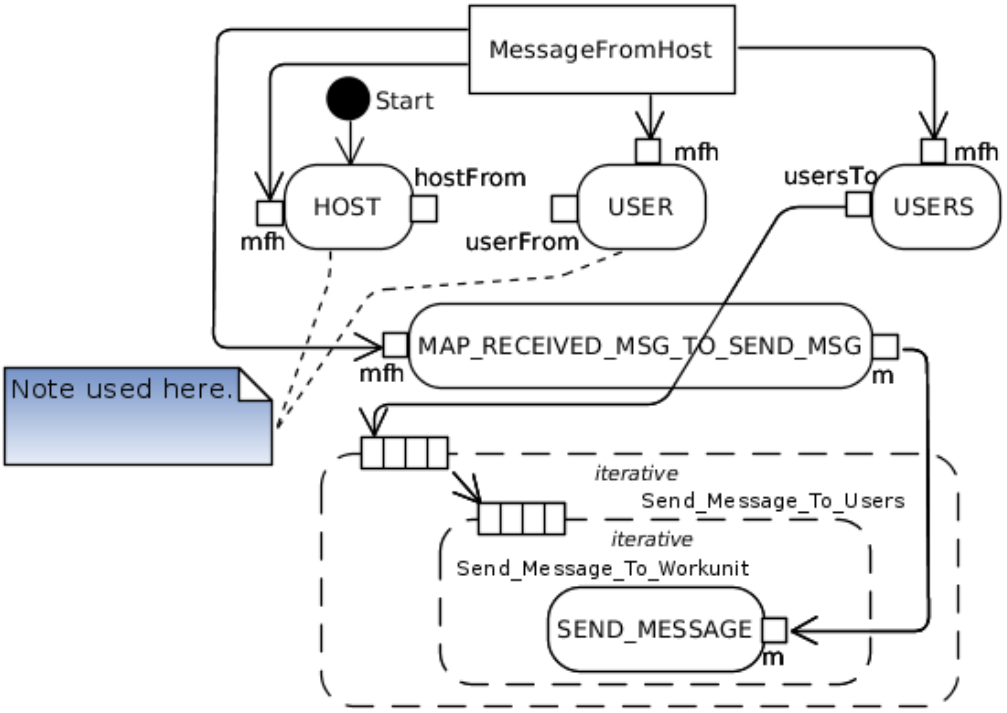

Figure 8.21: UML modelling for the handling of asynchronous-messages on the server-side.

8.11 Summary

This chapter presented a code-generation approach for the transformation of UML4BOINC's stereotypes into an implementation. At first, different code-generation approaches have been introduced, followed by examples of language recognition with ANTLR and how this is realised within this thesis. Subsequently, the direct transformation of models into executable code demonstrates how an Abstract-syntax Tree (AST) can be used for code-generation. In addition, it has been exemplified how a semantic model can be generated automatically through a specific Domain-specific Modelling Language (DSML). Accordingly a DSML – named after its author – Ries StateMachine (RSM) has been introduced; it enables the creation of a state machine which is based on a scientific application. Secondly, an object-orientated abstraction of BOINC's API has been presented briefly. All covered approaches and technologies are combined in order to establish a code-generation process which is used in the case study in Chapter 9. However, the code-generation transformation is not based on strict rules. Developers can decide how they implement its BOINC parts and scientific applications. The automatic checking of the OCL constraints did not occur in the code-generation approaches and within the implementations, because it is part of the future work.

Part III

Evaluation

CHAPTER 9

Case Studies

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.

DAVE PARNAS

9.1 Introduction

This chapter presents two case studies: (1) Spinhenge@home [22, 70, 165] in Section 9.2 and (2) LMBoinc [54, 186] in Section 9.3. The first study is only used in order to present how the code generation process is created and put in practice during this research. The second study introduces Visu@lGrid [184, 185] (see Section 8.6) which provides a graphical way for describing scientific applications.

9.2 Spinhenge@home

Spinhenge@home [165] has been established by Thomas Hilbig [70] and is a distributed computing project based on BOINC; it performs extensive numerical simulations concerning the physical characteristics of magnetic molecules. Spinhenge@home is a project of the Bielefeld University of Applied Sciences, the Department of Electrical Engineering and Computer Science and established in cooperation with the University of Osnabrück in Germany and Ames Laboratory in the USA. The project began testing on September 1, 2006. It employs the Metropolis Monte Carlo algorithm in order to calculate and simulate spin dynamics in nanoscale molecular magnets [168, Spinhenge@home]. However, in future these molecules will be used in localised tumour chemotherapy and developing of tiny memory-modules. As stated by C. Schröder with regard to the "recent careful magnetic measurements of the Fe30 [22] molecule done at the Ames Laboratory, some strange features have been observed. They could not have been detected beforehand because it requires a sophisticated technique in order to perform such measurements. [For understanding these features, C. Schröder's team] has proposed

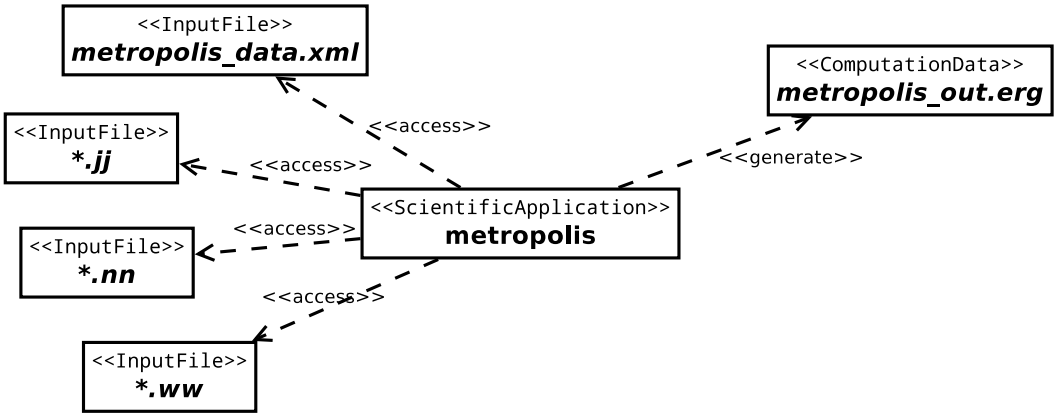

Figure 9.1: Spinhenge@home's input and output files dependencies; it needs four input files: (1) `metropolis_data.xml`, (2,3,4) a `*.jj`, `*.nn`, and a `*.ww` file, after successful execution one output file is created: `metropolis_out.erg`.

a new model description which takes small deviations from the ideal icosidodecahedron [170] molecule structure into account; it maps the structure onto the intramolecular interaction scenario. However, there is one unknown parameter which needs to be determined by subsequent comparison with the experimental data."

9.2.1 Codebase

In the beginning of the research for this thesis, the available Spinhenge@home's sources have been used as a base for proving that an object-orientated abstraction of BOINC's Application-programming interface (API) enables a compact way for implementing an own scientific application. Thus, Spinhenge@home's scientific application, the so-called "metropolis", requires four input files:

`metropolis_data.xml` This file contains the runtime configuration, e.g. the total number of Monte Carlo cycles or the number of spins of nanoscale molecular magnets.

`*.jj` Each line of this file contains a data sequence which describes the number of the nearest neighbour shells within a molecular magnet.

`*.nn` Each line of this file specifies the properties of the nearest neighbour shell. The file extension *.nn* denotes this, i.e. the nearest neighbour.

`*.ww` Each line of this file specifies the exchange interaction between the spins of a shell.

After the successful completion of the *metropolis*, the file `metropolis_out.erg` has been generated and contains the computation result. Fig. 9.1 visualises the input files and the output file dependencies.

9.2.2 3 Steps to Success

The code generation process is based on three steps. Fig. 9.2 illustrates these steps. On the left-hand side, an instance of `ScienceApp` has to be implemented in order to provide a code for generating a *Ries Statemachine* (RSM) description file. Section 9.2.4 presents an example for using `ScienceApp`. After compiling and calling the `ScienceApp` description, a new RSM file has been generated;it contains the state machine for supporting UML4BOINC's fundamental UML state machine which is introduced in Section 7.3. Within this case study the file of that state machine has been named `SpinhengeAtHome_main.rsm` and in the background and on the right-hand side, an excerpt of this file is shown. Listing C.9 on page 273 contains the complete example.

Next, the external tool `umlstatemachine` [183] — an implementation by the author of this thesis — is used for transforming the RSM description to a C++ implementation. The background of Fig. 9.2 presents an excerpt of this new file, whereas Listing C.10 contains the full C++ code. The described steps are the following:

Step 1 Implementation of a specific `ScienceApp` description.

Step 2 Compilation and call of the `ScienceApp` for generating the RSM description file.

Step 3 Transformation of the RSM description file into a C++ implementation which can be henceforth compiled and called.

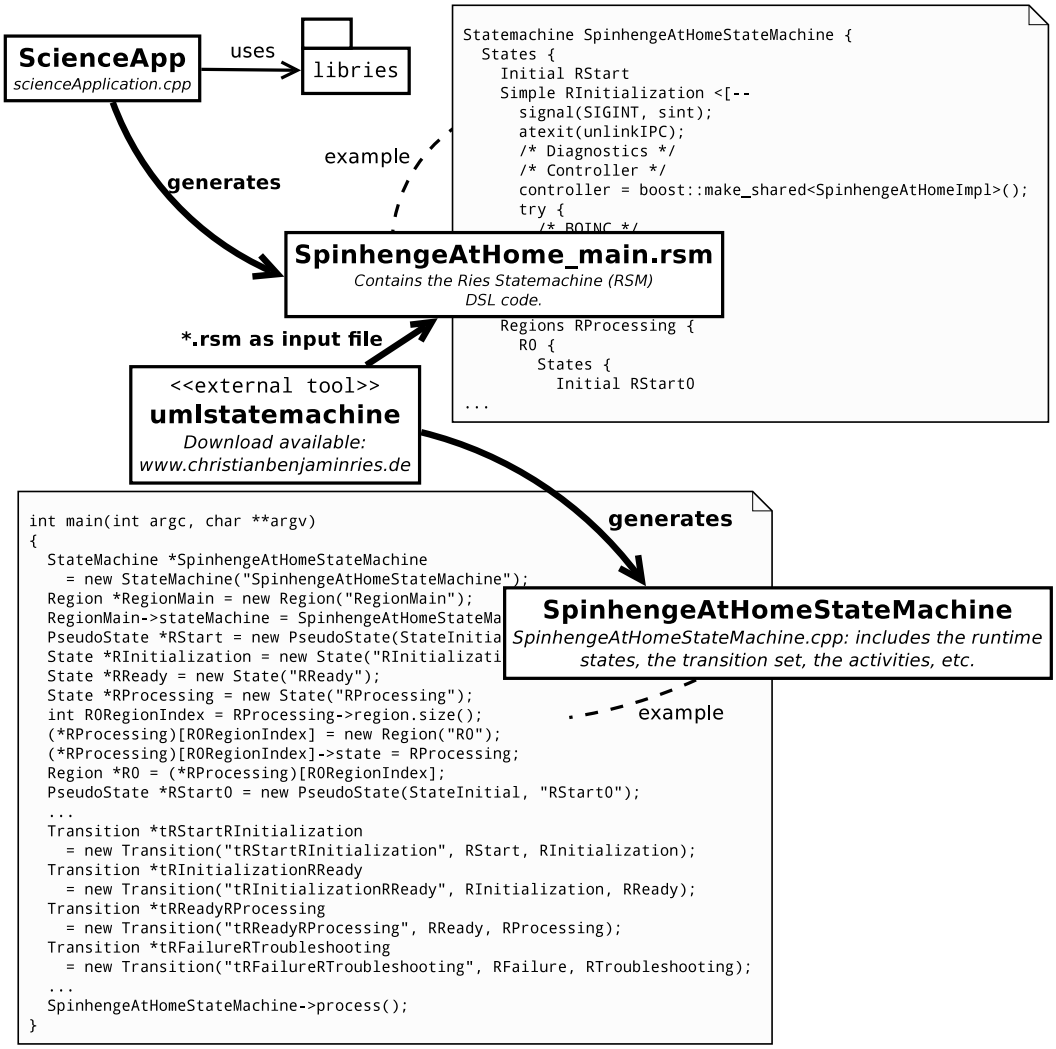

Figure 9.2: Three steps of creating Spinhenge@home's scientific application by help of a code generation process: (1) implementing of a specific `ScienceApp` description, (2) compiling and calling of the `ScienceApp` for generating the RSM description file, and (3) transforming of the RSM description to a C++ implementation.

9.2.3 Porting of the Codebase

The source code of Spinhenge@home's scientific application is closed. The following sections deal solely with the middleware layer which is provided by libries [185] — a software library by the author of this thesis. As introduced in Section 8.2.1, the code-generation process uses a semantic-model library implemented in C++ and three modelling approaches: (1) a Domain-specific Modeling Language (DSML), (2) a C++-Model, and (3) a Visual-Model. Not all of these three approaches have been used from the beginning of the research. Firstly, all models have been implemented manually as a direct C++-Model. The Spinhenge@home version 3.12 has been adopted for Linux systems and extended by an object-orientated middleware abstraction layer. Section 8.5 describes this abstraction layer in detail. By using the object-orientated layer, only the method for executing the working process has to be implemented. Listing 9.1 and 9.2 contain the specialisation `SpinhengeAtHomeImpl` of the base class `SpinhengeAtHome`; only the `doWork`-method has been implemented and calls the computation of Spinhenge@home's Metropolis Monte Carlo computation. Lines 33–38 create an instance of `SPIN_METROPOLIS_MC_BASE`, call its Monte Carlo computation and finally delete the created instance. Fig. 9.3 visualises the implemented code structure.

```
1  #pragma once
2  #ifndef __SPINHENGEATHOME_WORK_IMPL_H__
3  #define __SPINHENGEATHOME_WORK_IMPL_H__
4
5  // Generated
6  #include <spinhengeathome_controller.h>
7
8  class SpinhengeAtHomeImpl : public SpinhengeAtHome
9  {
10 public:
11   void doWork();
12 };
13
14 #endif
```

Listing 9.1: Implementation of the specialisation `SpinhengeAtHomeImpl` of the base class `SpinhengeAtHome` for calling Spinhenge@home's Metropolis Monte Carlo computation.

In the first lines of the `doWork`-method within Listing 9.2 the input and output files of the computation are defined. The used names describe virtual files which are by BOINC mapped to the physical files (see Section 5.3.7). The controller instance in line 10 is used by the state machine (see Section 9.2.4) for accessing the input and output files, and for controlling the application behaviour during execution, e.g. requests for stopping the execution are fetched by the state machine, filtered and forwarded to the controller. Between the lines 28 and 43 an excerpt

of Spinhenge@home's legacy-code is shown and executes the closed source part of Spinhenge@home's scientific application. Section 9.2.5 discusses the runtime behaviour of the state machine in more detail.

```
1  // C
2  #include <stdio.h>
3
4  // Implementation
5  #include <SpinhengeAtHome_work_impl.h>
6
7  // Spinhenge@home, original code by Thomas Hilbig.
8  #include <spin_monte_carlo.h>
9
10 boost::shared_ptr<Controller> controller
11   = boost::make_shared<SpinhengeAtHomeImpl>();
12
13 void SpinhengeAtHomeImpl::doWork()
14 {
15   Ries::BOINC::FileDecl fi1 = inputFiles()["metropolis_data.xml"];
16   Ries::BOINC::FileDecl fi2 = inputFiles()["param.jj"];
17   Ries::BOINC::FileDecl fi3 = inputFiles()["param.nn"];
18   Ries::BOINC::FileDecl fi4 = inputFiles()["param.ww"];
19
20   Ries::BOINC::FileDecl fo = outputFiles()["metropolis_out.erg"];
21
22   if(fi1 == NULL || fi2 == NULL || fi3 == NULL || fi4 == NULL)
23   {
24     std::cerr << "Could not find files: *.xml, *.[jj,nn,ww]" << std::endl;
25     handler()->finished(1);
26   }
27
28   // Begin of the modified legacy-code!
29   fprintf(stderr, "-----------------------------------\n");
30   fprintf(stderr, "worker: starting calculation ...\n");
31   fprintf(stderr, "-----------------------------------\n");
32
33   SPIN_METROPOLIS_MC_BASE *mc = new SPIN_METROPOLIS_MC_BASE(
34                                       fi1, fi2, fi3, fi4, fo
35                                     );
36   mc->monte_carlo_client();
37
38   delete (mc);
39
40   fprintf(stderr, "--------------------\n");
41   fprintf(stderr, "worker: finished!\n");
42   fprintf(stderr, "--------------------\n");
43   // End of the modified legacy-code!
44 }
```

Listing 9.2: Implementation for calling the closed source implementation of Spinhenge@home's scientific application.

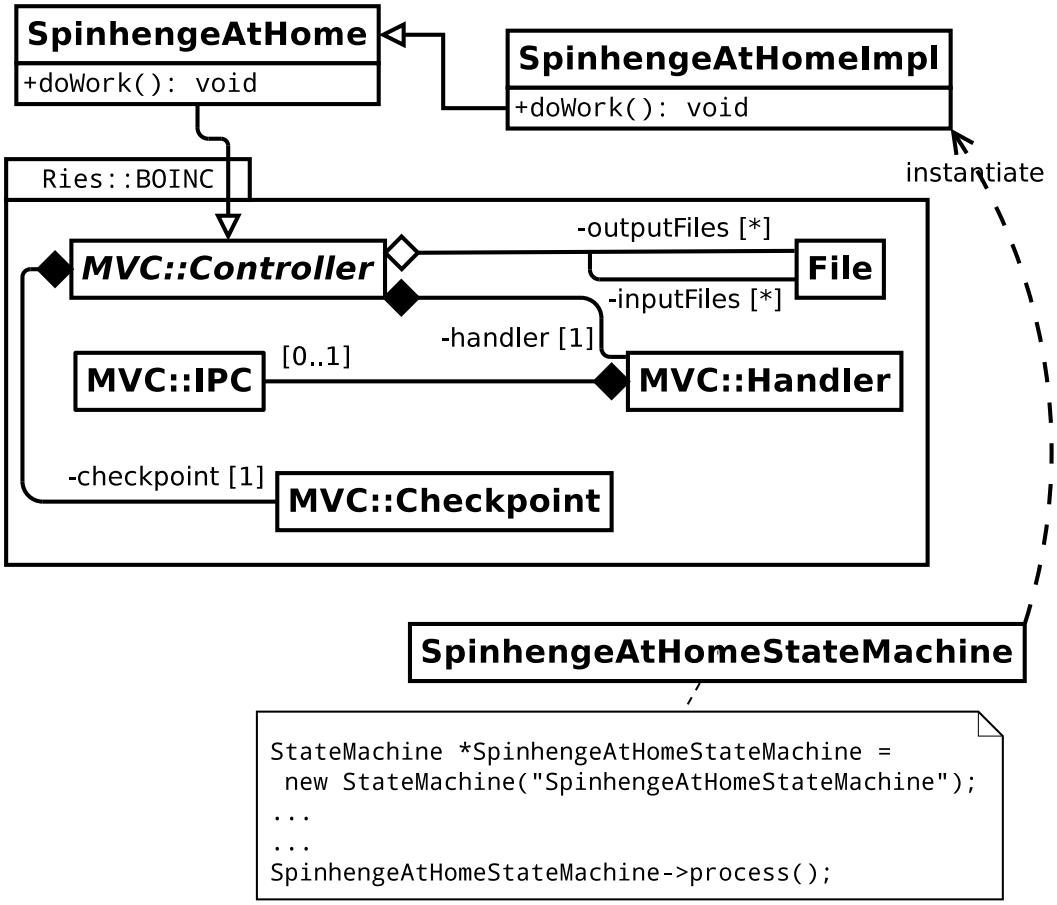

Figure 9.3: Class-hierarchy of Spinhenge@home's scientific application. The package `Ries::BOINC` is provided by *libries* [185]. The class `MVC::Controller` is specialised by `SpinhengeAtHome` which itself is generated by the state machine part in Section 9.2.4 and is specialised by `SpinhengeAtHomeImpl`. The real implementation of the scientific application is provided by the `doWork`-method of `SpinhengeAtHomeImpl`. In this first proving version it is implemented manually. On the bottom of this figure, the `SpinhengeAtHomeStateMachine` is presented and the note includes a listing of the generated state machine code (see Section 9.2.4 and Section C.3).

9.2.4 State Machine Support

The previous two Listings 9.1 and 9.2 do not contain a main-function used for starting the execution. During the compilation process, only object files are created which do not fulfil the requirements for linking these object files to a full executable file. In addition, the shape of the execution has to be implemented. Listing 9.3 contains a small set of C++ code which specifies how the code-generation process has to be executed.

Line 9 declares that the namespace `Ries::Generator` of *libries* has to be used.

Line 13 instantiates a `ScienceApp` instance which is a code generator for generating *Ries Statemachine* (RSM) code.

Lines 16–22 declare the input and output files of the Spinhenge@home's scientific application.

Lines 25–39 declare which variables and values have to be stored during the checkpoint process and which are restored after restarting the scientific application.

Lines 42–47 declare which variables and values have to be exchanged between processes, e.g. a screensaver instance for displaying these specific values.

Line 50 defines where the generated files are stored within the file system and generates them immediately.

libries contains a code-generator which generates a *Ries Statemachine* (RSM) (see Section 8.4) DSL code based on these information.

On page 273, the Listing C.9 contains the generated file; it has to be used by an additional code-generator named `umlstatemachine` [183] in order to generate C++ code. On page 276, Listing C.10 conveys the generated C++ file. This file provides the required `main`-function for executing the scientific application.

```
1  // Ries
2  #include <ries/antlr/antlr.h>
3
4  // C++
5  #include <iostream>
6  #include <vector>
7  #include <string>
8
9  using namespace Ries::Generator;
10
11 int main(int argc, char **argv)
12 {
13   ScienceApp app("SpinhengeAtHome");
14
15   // Input files
16   app. addInput( "data.xml", "metropolis_data.xml" );
17   app. addInput( "param.jj",           "param.jj" );
18   app. addInput( "param.nn",           "param.nn" );
19   app. addInput( "param.ww",           "param.ww" );
20
21   // Output files
22   app.addOutput( "result.zip", "metropolis_out.erg");
23
24   // Checkpoint data
25   app.addCheckpointVariable( "calc_flag",  0   );
26   app.addCheckpointVariable( "n_spins",    0   );
27   app.addCheckpointVariable( "t_index_in", 0.0f);
28
29   app.addCheckpointVariable<std::vector<double> >("sum_si_aver",   0.0f);
30   app.addCheckpointVariable<std::vector<double> >("sum_sisj_aver", 0.0f);
31   app.addCheckpointVariable<std::vector<std::vector<int> > > ("s0", 0);
32   app.addCheckpointVariable<std::vector<unsigned long> >("mt", 0);
33   app.addCheckpointVariable<std::vector<double> >("spin_min", 0.0f);
34   app.addCheckpointVariable<std::vector<double> >("spin_max", 0.0f);
35
36   app.addCheckpointVariable("E",         0.0f);
37   app.addCheckpointVariable("E_2",       0.0f);
38   app.addCheckpointVariable("progress", 0.0f);
39   app.addCheckpointVariable("mti",       0   );
40
41   // Inter-process communication
42   app.addIPCVariable("double", "cpuTime");
43   app.addIPCVariable("double", "fractionDone");
44   app.addIPCVariable("void*", "appInfo",
45     Ries::Generator::ScienceApp::InfoHandler);
46   app.addIPCVariable("void*", "initData",
47     Ries::Generator::ScienceApp::InitDataHandler);
48
49   // Directory for generating source code.
50   app.generate("src-gen/");
51
52   return 0;
53 }
```

Listing 9.3: Application for setting-up the *Ries Statemachine* (RSM) code generator.

9.2.5 Runtime

After creating Spinhenge@home's scientific application successfully, it can be executed manually or by distributing it with BOINC. Listing 9.4 shows the runtime behaviour when started manually. Section 8.5.1 introduces the used state machine and Fig. 7.2 visualises it. The naming of the different states has been adjusted, i.e. all states have a "R" as a first character followed by the state name. As defined by UML's state machine, the specification of each state possesses an `State::onEntry()` and `State::onExit()` method call. In case the state provides an implementation, the method `State::activity()` is called and executes its code implementation.

Listing 9.2 contains a handful print messages. The `doWork`-method is intended to be called within the state `Computing` (see Fig. 7.2) which is provided by the `SpinhengeAtHomeStateMachine`'s state `RComputing`. As presented in Listing 9.4, the runtime behaviour can be interpreted as follows

Line 20 The `RComputing` state is entered.

Line 21 The computation provided by `SpinhengeAtHome` and `SpinhengeAtHomeImpl` (see Fig. 9.3) starts.

Lines 22–27 The print messages of the `doWork`-method.

Lines 28–36 The computation is working in the background; the checkpointing and exchanging of information between processes is realised by two additional threads, as described by the UML state machine in Fig. 7.2.

Lines 37–39 Additional print messages which are written after the computation.

Line 43 The computation is ready and the state `RComputing` will be leaved.

Lines 44–47 The state machine will be finished and the scientific application will be exited.

```
1  <-- State::onExit() RInitialization
2  --> State::onEntry() RReady
3  --- State::activity() RReady
4  <-- State::onExit() RReady
5  --> State::onEntry() RProcessing
6  --> PseudoState::onEntry() RStart0
7  --> PseudoState::onExit() RStart0
8  --> State::onEntry() RIdleReport
9  --- State::activity() RIdleReport
10 --> PseudoState::onEntry() RStart1
11 --> PseudoState::onExit() RStart1
12 --> State::onEntry() RIdleCheckpointing
13 --- State::activity() RIdleCheckpointing
14 --> PseudoState::onEntry() RH2
15 --> PseudoState::onExit() RH2
16 --> State::onEntry() RReadCheckpoint
17 --- State::activity() RReadCheckpoint
18 --- State::activity() RProcessing
19 <-- State::onExit() RReadCheckpoint
20 --> State::onEntry() RComputing
21 --- State::activity() RComputing
22 ---------------------------------
23 worker: starting calculation ...
24 ---------------------------------
25 ---------------------------------
26 mc_client started successfully!
27 ---------------------------------
28 <-- State::onExit() RIdleCheckpointing
29 --> State::onEntry() RCheckpointing
30 --- State::activity() RCheckpointing
31 <-- State::onExit() RIdleReport
32 --> State::onEntry() RReporting
33 --- State::activity() RReporting
34 <-- State::onExit() RReporting
35 --> State::onEntry() RIdleReport
36 --- State::activity() RIdleReport
37 -----------------
38 worker: finished!
39 -----------------
40 <-- State::onExit() RCheckpointing
41 --> State::onEntry() RIdleCheckpointing
42 --- State::activity() RIdleCheckpointing
43 <-- State::onExit() RComputing
44 --> PseudoState::onEntry() RFinished
45 --> PseudoState::onExit() RFinished
46 --> FinalState::onEntry() RFinish
47 --- RegionMain is finished!
```

Listing 9.4: `SpinhengeAtHomeStateMachine`'s runtime behaviour.

9.3 LMBoinc

The following case study manipulates a video stream by basic image processing algorithms [186, 56, 55]. Fig. 9.4 shows the general idea of this case study and which video is added by *C.B.Ries*. The BOINC project is triggered by *inotify* [181] and *dataAvailable*; workunits are created on demand. In addition, it has to be clear which kind of data format is used during the workunit addition, i.e. ZIP-archives (ZIPs) are created in this case study. The ZIPs contain image sequences which are modeled by subsequent implementations. The manipulation algorithms and the graphical representations implemented and used by Visu@lGrid (Section 9.3.1) are mapped into UML4BOINC's semantics (Section 7.4): (1) the infrastructure is described in Section 9.3.3, (2) the computing core in Section 9.3.4 and (3) the creation of workunits in Section 9.3.5.

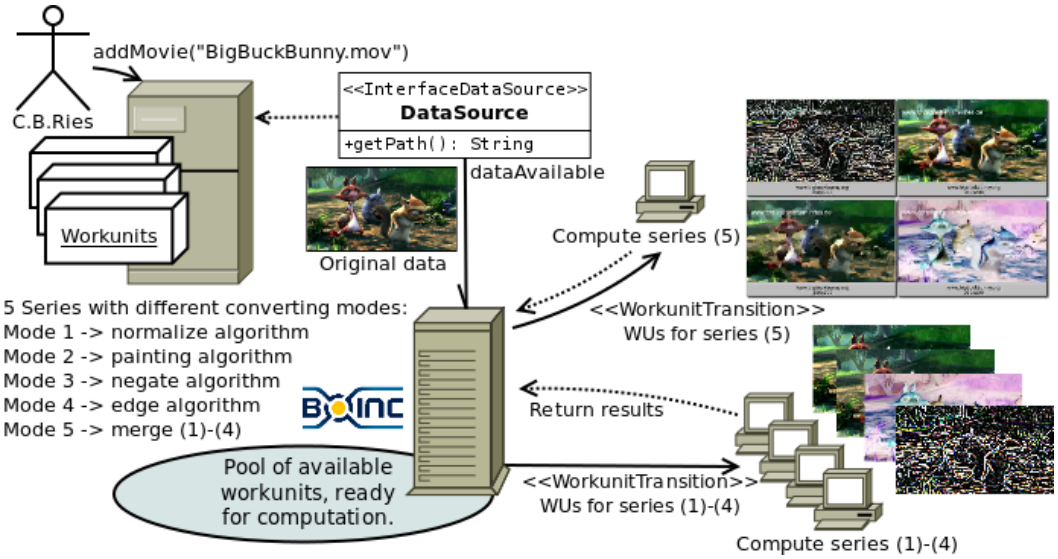

Figure 9.4: Case study for modifying a movie: *C.B.Ries* adds a new movie to a server; running applications prepare this movie and fragment it into image sequences. These images are thereupon zipped into ZIP-archives. *DataSource* notifies a BOINC project which automatically adds all announced workunits to the BOINC project. The addition of one ZIP-archive to the BOINC project implies four further workunits but with different configurations; each «Series» instance has a different mode for computation: *normalize*, *painting*, *negate* and *edge*. These workunits are processed and the computational results are used as an input for the fifth «Series»; again with a changed mode for computation: *merge*.

9.3.1 Visu@lGrid

Visu@lGrid [184] is an integrated development environment (IDE) which aims to support UML4BOINC. Fig. 9.5 shows Visu@lGrid and its different parts. In par-

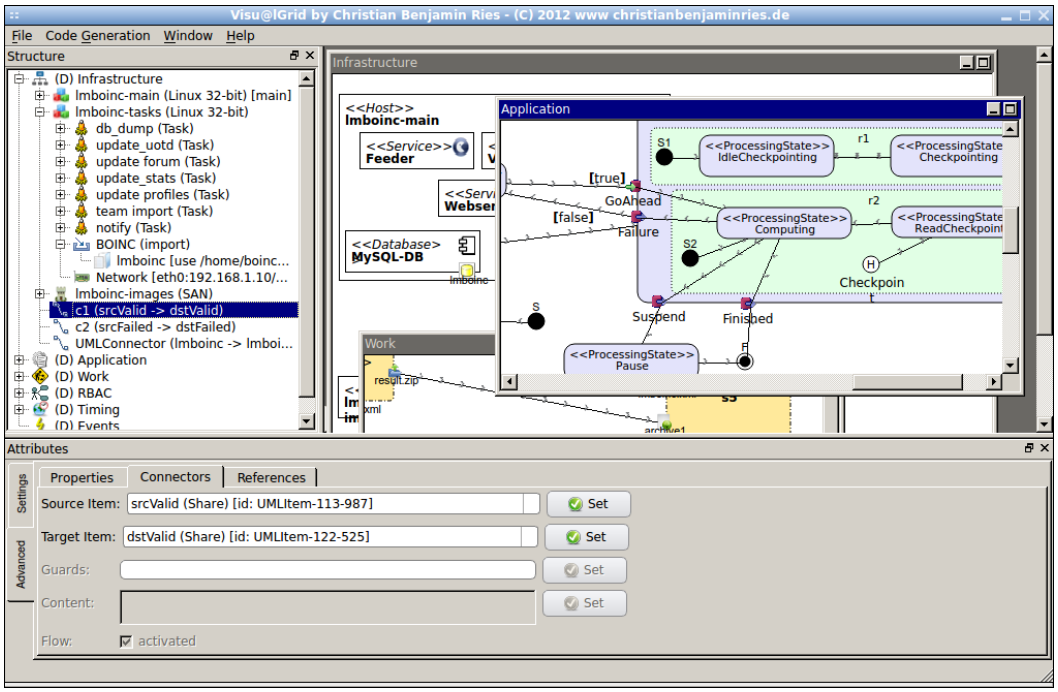

Figure 9.5: Visu@lGrid with its several parts: (i) the tree-view on the left-hand side, (ii) the diagram view on the right-hand side and (iii) the attribute setting and the preferences part. The diagram shows all UML4BOINC diagrams in a multiple document interface (MDI) view in order to provide simultaneous viewpoints. This figure presents three views which are relevant for LMBoinc: (1) the infrastructure, (2) the application, and (3) the work diagram.

ticular, three areas are created within Visu@lGrid: (1) the model tree-view on the left-hand side, (2) the graphical representation on the right-hand side and (3) a configuration area for setting the attributes of all created items and for enabling the set-up of connections between elements, i.e. a transition (based on a connection) which connects two actions in the application diagram. The structure of the tree-view is based on UML4BOINC, i.e. every diagram specification possesses its own sub-leafs and graphical representations. As a consequence, six diagrams are available. The diagrams on the right-hand side are covered in Sections 9.3.3, 9.3.4 and 9.3.5. The graphical representation and the notation of Visu@lGrid are based on UML. The stereotype names are not shown in all cases, e.g. «DB» is not shown for a UML Component extended by «Service». Visu@lGrid does not support the complete UML specification. It is a graphical user interface for handling Visu@lGrid's Domain-specific Modeling Language (DSML). Visu@lGrid does not provide the developer with all tool-support possibilities, e.g. context-aware sub-menus, the validation of the model and the complete code generation of all parts which are necessary for BOINC.

9.3.2 Video Processing

Visu@lGrid is used for the verification of UML4BOINC and supports a handful of UML4BOINC's stereotypes. The current version[1] provides the modelling of a specific scenario and the modification for a video stream. Visu@lGrid enables the modelling of an infrastructure, a scientific application, task configurations and the creation of workunits. The scientific application handles ZIPs which contain film sequences. In other words, it fragments a specific video into sequences. These sequences are packed into several ZIPs and distributed to clients. In addition, basic image processing algorithms are applied to the sequences; they return to the BOINC project, become validated and finally assimilated. Some figures of this case study were already presented in the previous chapters.

9.3.3 LMBoinc's Infrastructure

Fig. 9.6 shows some UML4BOINC stereotypes in the context of the BOINC project *LMBoinc*. Workunits and computational results include ZIPs with a specific number of sequences; accordingly, the scientific application *lmboinc* (Fig.9.8) has to be associated to one matching «Software» in order to make *zip* available during runtime. LMBoinc describes the whole BOINC project at that time and thereafter, whereas lmboinc only names the scientific application. LMBoinc uses three hosts on the server-side; they are described in Section 7.2:

lmboinc-main: This host has all necessary BOINC services to run a BOINC project.

lmboinc-tasks: This host is composed of seven BOINC services specified as tasks.

lmboinc-images: This is a SAN used for storing computational results.

Listing C.13 of Appendix C.5 presents the semantic model specification and Listing C.14 conveys the generated class-hierarchy of Visu@lGrid. This class-hierarchy is therefore traversed and can be used in order to generate different deployment configurations, e.g. a Puppet [203] configuration.

9.3.4 LMBoinc's Application

Fig. 9.7 shows LMBoinc's state machine. LMBoinc does not handle asynchronous-messages; as a consequence, only two regions are necessary for «Processing»:

[1] Version 0.2 alpha

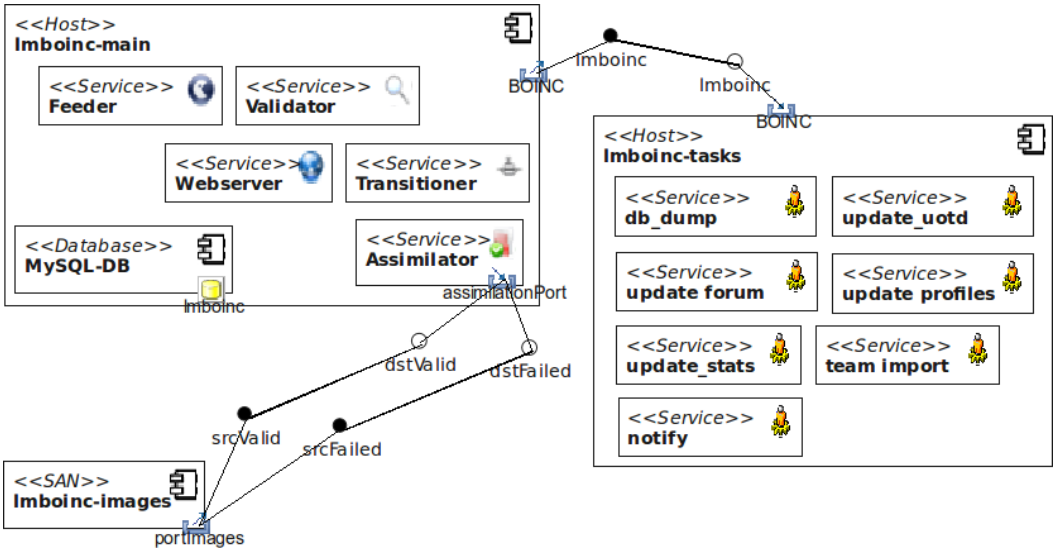

Figure 9.6: LMBoinc's infrastructure; two hosts, their BOINC services and a storage area network for computational results.

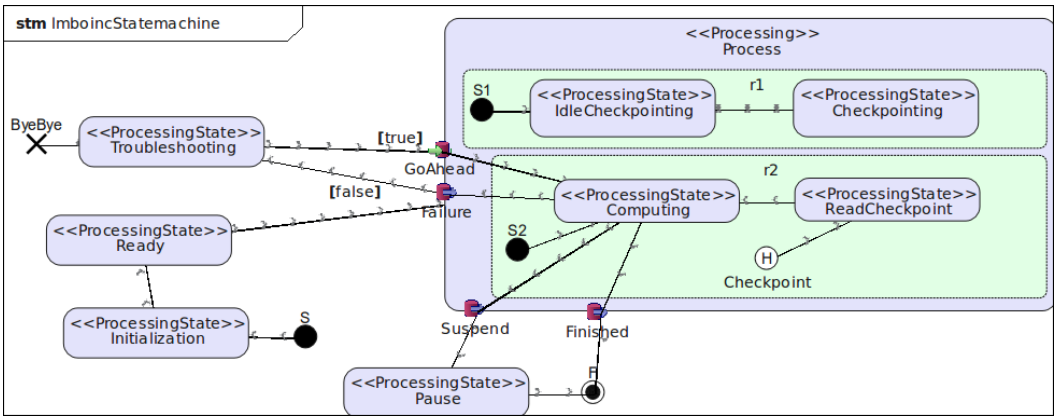

Figure 9.7: LMBoinc's state machine modelled with Visu@lGrid.

(1) for checkpoint creation and updating and (2) for performing the core computation. The code-generation process maps this model directly to the introduced RSM (Section 8.4). Every «ProcessingState» can possess its own linked activity. This linking is performed in the tree-view and resolved during code-generation. This study only simulates the linking of *Computing* and *Initialization* to two separate activities. Fig. 9.8 shows the activity of *Computing*. Fig. 9.9 visualises how states of the state machine are linked to activities. This approach is used for all items which can be extended by own implementations, e.g. the «Task» instances within the Timing diagram of Fig. 9.11 are linked to the relevant activities in Fig. 9.10. Listing 9.5 presents the generated code for the *Computing* state. This code includes calls which are provided by the Object-orientated Abstraction (OOA)

```
1  // Core computing of the scientific application.
2  Simple     RComputing
3  <|--
4    /* Files */
5    controller->addInputFile("A", "input.data/images.zip");
6    controller->addInputFile("C", "input.data/lmboinc.xml");
7    controller->addOutputFile("R", "result.zip");
8    controller->addHandler(handler);
9    controller->start();
10 --|>
```

Listing 9.5: *Computing* of the LMBoinc state machine.

(Section 8.5), i.e. `addInputFile` and `addOutputFile` are used to add alias names for accessing input/output files of workunits and to map the «InputFile» and «OutputFile» object nodes of the top left-hand side and right-hand side of Fig. 9.8. Lines 18 and 22 of Listing 9.6 show how these aliases are used. The checkpoint *lmboincCheckpoint* is handled by the internals of OOA. Lines 45 and 46 of Listing 9.6 demonstrate how the checkpoint values can be accessed. In addition, it conveys how an OOA's handler is added to an OOA's controller.

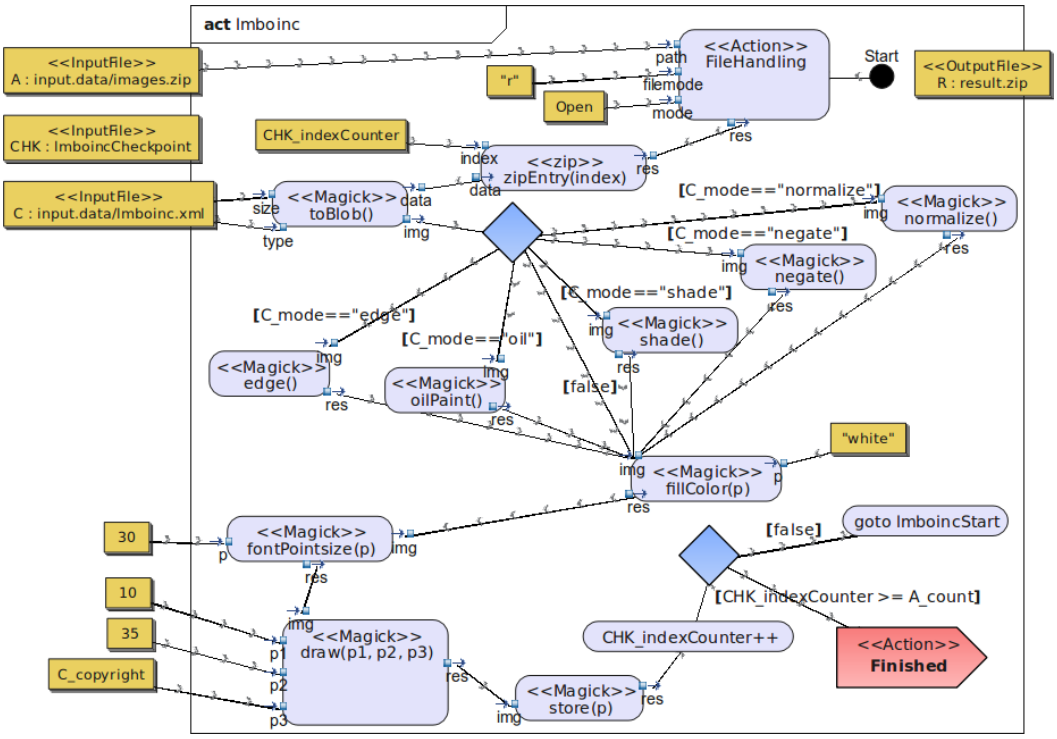

Figure 9.8: The model specification of LMBoinc's *Computing* activity.

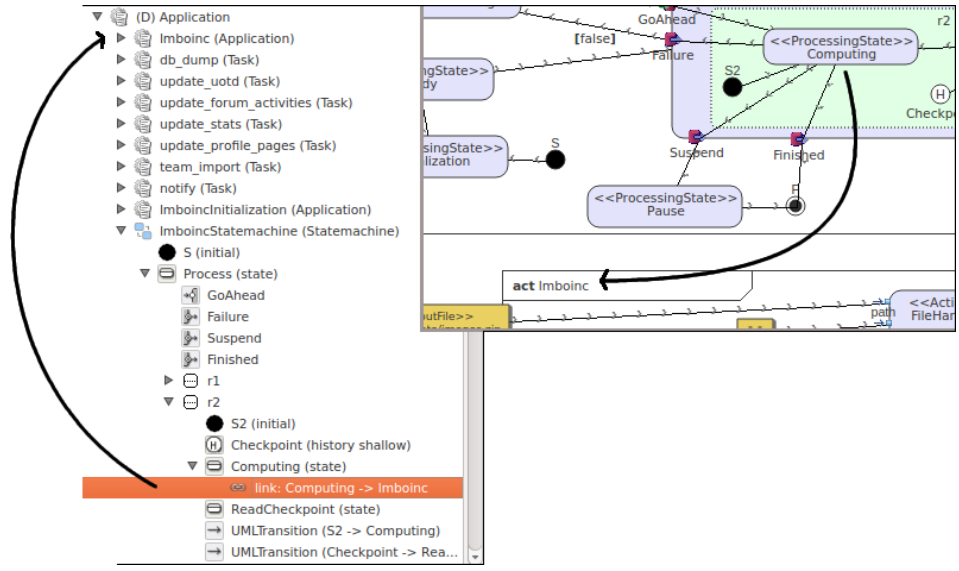

Figure 9.9: Link between the state *Computing* and the implementation *lmboinc*.

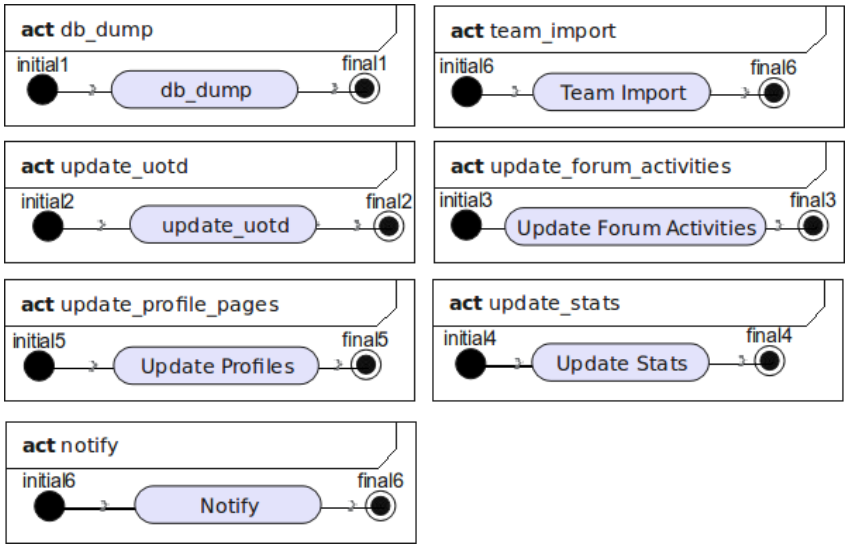

Figure 9.10: Part 1: LMBoinc's tasks specification.

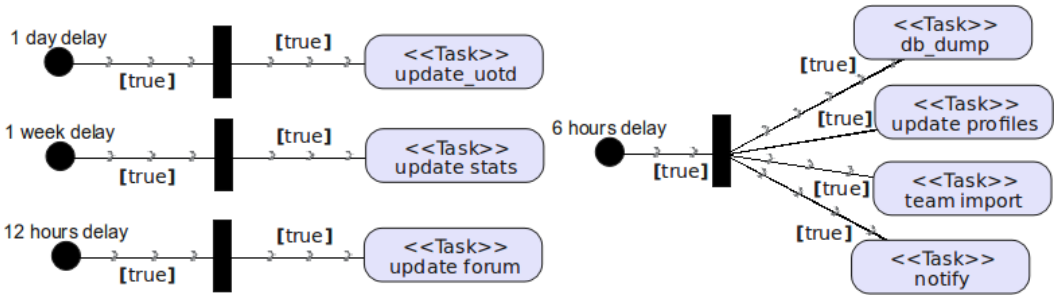

Figure 9.11: Part 2: LMBoinc's tasks specification; seven individual «Task» instances are specified and all of them are activated. The activation is specified by the transitions. The «Task» instances on the right-hand side are executed in a period of six hours, the instances on the left in several periods: (i) one day, (ii) one week, and (iii) twelve hours.

The activity *lmboinc* does not only contain UML4BOINC stereotypes in Fig. 9.8. The basic image processing algorithms and the handling of ZIPs are abstracted by the additional stereotypes «Magick» and «zip» which are not part of UML4BOINC. These stereotypes are created in order to enable the distinction of UML4BOINC's stereotypes. Besides, they resolve dependent software libraries which are necessary for execution. UML4BOINC's stereotype «Action» named *FileHandling* is presented at the top. The action mode is `Open` and used to access workunit input files. In particular, it is used in this study in order to open the ZIPs which contain film sequences. Lines 48 to 57 of Listing 9.6 show the generated code. The code-generation process checks the datatype of the input file and generates an appropriate handling code. The next action the `zipEntry` accesses is a ZIP entry addressed by the parameter `index`. This entry is converted into an ImageMagick's blob data format [197]. The following decisions are transformed into *if-else* statements and the guards of the transitions are used for the condition statements. The bottom input file on the left is lmboinc's configuration file which contains four variables:

mode: This variable specifies how the film sequences are manipulated. Six different values are supported in this study: (i) *edge*, (ii) *oil*, (iii) *shade*, (iv) *negate*, (v) *normalize*, and (vi) *merge*.

size: The image size which is used by `toBlob`.

type: The image type which is used by `toBlob`.

copyright: A string value which is written on every film sequence.

lmboinc accesses all film sequences sequentially and decides which image algorithm has to be used. Moreover it fills the image with a white background in order to increase the colour contrast and then sets the font size and writes the copyright text before storing the data blob on the hard disk drive. The mentioned `index` for ZIP access is increased and compared with the number of available film sequences in the ZIP file. When the increased index is greater than or equal to the number of sequences, then the execution is finished. Otherwise it will perform the next film sequence. The output file is not created in this activity but can be performed within the `onLeave` linked activity, when the exit pin *Finished* is used. lmboinc is only used for the first four series shown in Fig. 9.12. The fifth series has four input ZIPs and all of them contain film sequences merged to a single film sequence. As a consequence, lmboinc needs a modification, otherwise the creation of the fifth series requires a different activity. This additional activity is not included in Visu@lGrid's delivered example project [184].

```
#include <iostream>
#include <fstream>
#include <stdio.h>

// Third-party
#include <zip.h>
#include <ImageMagick/Magick++.h>

// Implementation
#include <lmboincStatemachine_work_impl.h>
#include <lmboincDatafield.h>

boost::shared_ptr<Controller> controller
  = boost::make_shared<lmboincStatemachineImpl>();

void lmboincStatemachineImpl::doWork()
{
Ries::BOINC::FileDecl A =  inputFiles()[ "A"];
if(A == NULL) {
std::cerr << "Could not find file: 'A'" << std::endl;
handler()->finished(1);}
Ries::BOINC::FileDecl C =  inputFiles()[ "C"];
if(C == NULL) {
std::cerr << "Could not find file: 'C'" << std::endl;
handler()->finished(1);}
std::ifstream ifs(C->fileName().c_str());
boost::archive::xml_iarchive ia(ifs);
LMBoincDatafield *ind;
ia >> BOOST_SERIALIZATION_NVP(ind);
ifs.close();
std::string *_mode = NULL, *_size = NULL, *_copyright = NULL, *_type = NULL;
Any inddataMode; Any inddataSize; Any inddataCopyright; Any inddataType;
for(auto field : ind->fields()) {
  Any inddata = field->data();
  if(field->name() == "mode") { inddataMode = field->data(); _mode = boost::get<
      std::string>(&inddataMode); }
  else if(field->name() == "size") { inddataSize = field->data(); _size = boost::
      get<std::string>(&inddataSize); }
  else if(field->name() == "copyright") { inddataCopyright = field->data();
      _copyright = boost::get<std::string>(&inddataCopyright); }
  else if(field->name() == "type") { inddataType = field->data(); _type = boost::
      get<std::string>(&inddataType); }
  else { /* unknown */ }
}
#define C_mode            *_mode
#define C_size            *_size
#define C_copyright *_copyright
#define C_type            *_type
int *indexCounterPtr = boost::any_cast<int>(indexCounter->value());
#define CHK_indexCounter (*indexCounterPtr)
lmboincStart:
std::string inputArchivePath;
struct zip *zipFileIn = NULL;
char zipError[512] = {'\0'};
  boinc_resolve_filename_s(A->fileName().c_str(), inputArchivePath);
zipFileIn = zip_open(inputArchivePath.c_str(), ZIP_CHECKCONS, NULL);
if (zipFileIn == NULL) {
zip_error_to_str(zipError, 512, 0, errno);
std::cerr << "zip failed: " << zipError << std::endl; }
int countFileIn = zip_get_num_files(zipFileIn);
#define A_count countFileIn
```

```
58 struct zip_file *zipReader = zip_fopen_index(zipFileIn, CHK_indexCounter,
       ZIP_FL_UNCHANGED);
59 if (zipReader == NULL) {
60 zip_error_to_str(zipError, 512, 0, errno);
61 std::cerr << "zip failed: " << zipError << std::endl; }
62 struct zip_stat zipFileInformation;
63 zip_stat_index(zipFileIn, CHK_indexCounter, ZIP_FL_UNCHANGED, &zipFileInformation)
       ;
64 char *data = new char[zipFileInformation.size];
65 zip_fread(zipReader, data, zipFileInformation.size);
66 Magick::Blob blob((void*)data, zipFileInformation.size);
67 Magick::Image img;
68 img.size(C_size);
69 img.magick(C_type);
70 img.read(blob);
71 if(C_mode=="edge") { img.edge(); }
72 else if(C_mode=="oil") {  }
73 else if(C_mode=="shade") { img.shade(); }
74 else if(C_mode=="negate") { img.negate(); }
75 else if(C_mode=="normalize") { img.normalize(); }
76 img.fillColor("white");
77 img.fontPointsize(30);
78 img.draw(Magick::DrawableText(10,35,C_copyright));
79 img.write(zipFileInformation.name);
80 CHK_indexCounter++;
81 if(CHK_indexCounter >= A_count) { // *************************
82 //   Execution is finished!
83 // *************************
84 return;
85  }
86 goto lmboincStart;
87 }
```

Listing 9.6: `doWork` contains the generated code of the model in Fig. 9.8; `doWork` is executed when the state machine in Fig. 9.7 reaches the *Computing* state.

Section 7.6 already described the semantic of the Timing diagram. Fig. 9.11 presents LMBoinc's tasks which are provided by BOINC and linked to the activities in Fig. 9.10. The activities can be extended or replaced by individual model descriptions. Listing 8.6 shows the result of Visu@lGrid's generated XML configuration which is added to BOINC's task configuration. The XML configuration contains an XML-tag for specifying the command for execution. This command has to be changed into the name of the modelled activity when a single model is used.

9.3.5 LMBoinc's Work

Fig. 9.12 shows five definitions of «Series» instances. This case study can only process the first four instances in more detail. As shown in Equation 9.1, the fifth «Series» requires the result of the first four «Series» instances [54]. All «Series» instances have a different runtime configuration, e.g. the images are manipulated by an edge algorithm in the fourth «Series». The fifth «Series» merges all previous results; as shown on the right-hand side of Fig. 9.4, the results of an image sequence

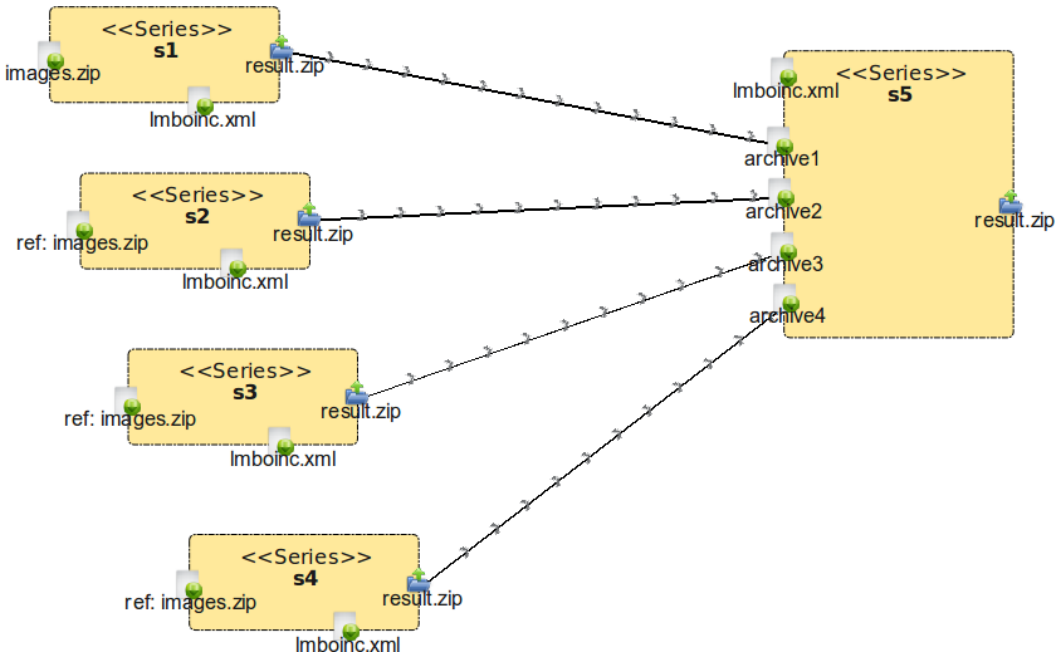

Figure 9.12: LMBoinc's work specification of five «Series» instances: (i) *s1 – s4* have two input files and one output file; (ii) *s5* has five input files and one output file. When the previous four series are performed, instance *s5* is created and started.

are added to a 2×2 raster image. The lower configuration shows the mode "edge" for image manipulation and the top assigns "merge". The fifth «Series» is not started until the other «Series» have finished. It is of relevance that the scientific application is always the same and that only the input files change. Just two files are necessary for the first four computations: (1) configuration of the mode and (2) the mentioned ZIPs and movie sequences. The last computation is altered and thus needs five files: (i) the same configuration as before but with different values and (ii) all four input files which are created by the other four computations. This case study describes a *dynamic series* (Section 7.4.3) where some workunits are available at the start of the BOINC project; additional workunits are created during runtime. In case one of the first four series is cancelled, the fifth series can never be started due to missing input data. This is solved by the state machine construction in Fig. 7.5 of Section 7.4.2 where the "Cancelation" state is responsible for stopping all related workunits and «Series» instances.

$$\begin{bmatrix} Series\ 1\ (normalize) \\ Series\ 2\ (painting) \\ Series\ 3\ (negate) \\ Series\ 4\ (edge) \end{bmatrix} \Rightarrow\ Series\ 5\ (merge) \tag{9.1}$$

Validation and assimilation is performed by BOINC's standard processes. As in Fig. 9.6, the two services Validator and Assimilator are linked to two activities

within the Application diagram. These activities call the BOINC applications for the trivial validation and archive assimilation (Section 5.3.5 and Section 5.3.6).

9.4 Summary

This chapter dealt with the two case studies *Spinhenge@home* and *LMBoinc*. The first one is a real BOINC project[2] for doing research in the field of nanoscale molecular magnets and the second one is a BOINC project for the manipulation of movie sequences by use of basic image processing algorithms. Secondly, the IDE Visu@lGrid for implementing LMBoinc has been presented. It is used for creating or applying the necessary code or configuration in order to set-up LMBoinc and to start the processing of individual computational series. In addition, Visu@lGrid can be applied as a graphical front-end for the creation of a new BOINC project. A manually implemented version of LMBoinc can be downloaded on the project website [186], whereas, an exemplary implementation of Visu@lGrid is given in the sources [184].

[2] http://spin.fh-bielefeld.de

CHAPTER 10

Conclusions

There are no failures — just experiences and your reactions to them.

TOM KRAUSE (1934)

10.1 Summary of this Thesis

In this thesis, a new Unified Modeling Language 2 (UML) Profile for the Berkeley Open Infrastructure for Network Computing (UML4BOINC) has been described and defined. The profile contains 95 new stereotypes which are organised in eight packages, which specify six diagrams for modelling a complete BOINC project. UML4BOINC is depicted by informal and formal constraints, i.e. the Object Constraint Language (OCL). An informal overview of the semantic and contextual relationship between the stereotypes and the BOINC-system is given. The diagrams are used to create different viewpoints of a BOINC-based project: (i) the infrastructure, (ii) the application, (iii) the sets of permission, (iv) the work handling, (v) timed activities and (vi) events. The infrastructure stereotypes and the specific diagrams can be used for the creation of the infrastructure on BOINC's server-side and to specify which scientific applications for which target platforms and architectures are used for distribution. Within the application diagram, stereotypes to allow developers' modelling of scientific applications can be used. This is followed by the presentation of the permission diagram and stereotype specifications for the creation of Role-based access control (RBAC) descriptions. After this, stereotypes for the specification of workunits are proposed. This proposal allows specifying of the structure of workunits, and how series of several autonomous or dependent series are handled by a BOINC-project. The last two diagrams are used for the specification of periodically executable tasks and event-handling aspects.

This thesis describes UML4BOINC's semantic and it has been demonstrated how the infrastructure of a BOINC project can be specified through the use of the

Infrastructure package. In an additional step, it is explained how the stereotype of the Application package is relevant in order to model a scientific application with different numbers of input/output files used. Furthermore, examples for the usage of the state machine for workunit handling and the lifetime of a scientific application have been given. In addition, the handling of asynchronous-messages and checkpointing have been presented thoroughly and the semantics for modelling activities and actions introduced. A demonstration of the semantic of work handling, how permission control can be modelled and how timed tasks can be executed is given.

This thesis presented a code-generation approach for the transformation of UML4BOINC's stereotypes into an implementation. At first, different code-generation approaches have been introduced followed by examples of language recognition with ANTLR and how this is realised. Subsequently, the direct transformation of models into executable code demonstrates how an Abstract-Syntax Tree (AST) can be used for code-generation. In addition, it has been demonstrated how a semantic model can be generated automatically through a specific Domain-Specific Modelling Language (DSML). Accordingly a DSML called the Ries StateMachine (RSM) has been introduced; it enables the creation of a state machine based on a scientific application. Secondly, an object-orientated abstraction of BOINC's Application Programming Interface (API) has been presented briefly and covered by the case study of a real BOINC project, namely Spinhenge@home[1]. All covered approaches and technologies are combined in order to establish a code-generation process which is used by the case study LMBoinc, which is used as a proof of concept. However, the code-generation transformation is not based on strict rules. Developers can decide how they implement its BOINC parts and scientific applications. The automatic checking of the OCL constraints did not occur in the code-generation approaches and within the implementations.

In summary, it can be stated that this profile is the first UML 2-based profile that defines the language especially for the specification and description of Berkeley Open Infrastructure for Network Computing (BOINC) project installations. The code generation engine has to be tailored to the general programming language C/C++ for supporting the whole BOINC API.

[1] http://spin.fh-bielefeld.de

10.2 Contributions

This thesis accounts for several emerged modelling approaches and technologies of how a BOINC-project can be specified, implemented and maintained.

10.2.1 Contributions to the BOINC Community

This thesis introduces a C++ abstraction framework for BOINC which offers a more comfortable approach for code-generation. In addition, the wrapper implementation for Scilab and COMSOL Multiphysics is available through *libries* [185]. One of the main contributions to the BOINC community is the prototype of the Integrated-development Environment (IDE), the so-called *Visu@lGrid* [184] for working with the new UML profile UML4BOINC.

10.2.2 Contributions to the Software Engineering Community

Some code-generation factories have been implemented during the research for this thesis. They are used in order to reduce the implementation effort for the creation of C++-based class-hierarchies. Thus, the Domain-specific Modeling Language (DSML) enables the definition of classes and dedicated class-hierarchies. Thereupon the DSMLs are parsed; the C++-code is generated immediately to be feasible for the implementation of semantic models. The current version supports the creation of C++ implementations. Additional work can be accomplished by other target languages such as Java. Furthermore, the UML 2 Statemachine code generator UML2STM4CPP has been presented [183]. This generator uses a small set of DSL statements in order to create statemachines with full support of UML's 2.4 Statemachine specification. The generator can be applied for Linux, Mac OS X, and Microsoft Windows operating systems. These two implementations are bundled in a larger software library, the so-called *libries* [185].

10.2.3 Contributions to the MDE Community

The UML Profile *UML4BOINC* for the development of BOINC-based projects has been introduced. UML4BOINC is based on UML version 2.4. UML itself is specialised by several stereotypes which enable the creation of UML-based model descriptions of distributed applications based on the BOINC framework. Different BOINC techniques are taken into account: (i) the creation of a scientific application by use of a small set of specialised UML Actions and UML Activities, (ii) the lifetime is monitored by a suitable set of UML StateMachines and dedicated UML States, (iii) different input and output file formats can be used, (iv) checkpointing is possible, (v) the work of a BOINC project can be created, monitored and handled,

(vi) individual sets of permissions can be established for users and administrators and (vii) asynchronous-messages can be used in order to control the scientific application performance or enable the communication between users.

10.3 Future Work

There exist various areas of future work for which this thesis may serve as a basis. Most of the implementations of the use-case scenarios are performed for prototyping. This work covers possible scopes only superficially. As is often the case, this thesis raises more questions than it answers. Thus, a selection of raised questions and open tasks is included and can be addressed by further research.

10.3.1 Code-Generation

Section 8.3 has presented how Abstract-syntax Trees (ASTs) are created with the help of ANTLR. The AST can be used for the code-generation of semantic-models. Additional ASTs can be created in order to generate BOINC parts. However, the current implementation and validation process of UML4BOINC does not provide any way or technique for the validation of the AST, the structure or actions. The current code-generation process is based on a visitor-pattern approach [83] and ignores the structural constraints. The AST has to be created error-free in order to enable a correct transformation into a valid semantic-model or BOINC parts. Open research questions are: (i) How can an AST be used to generate all parts of a BOINC project? (ii) Which kind of transformation technique approach can be used? (iii) How do the informal and OCL constraints fit in such a transformation?

10.3.2 Event Handling

Section 7.7 introduces an event handling mechanism for BOINC. This thesis covers only asynchronous-messages. It can be extended in future work in order to cover runtime problems or exceptions, e.g. when a host is not available anymore. When a host is not available the services do not exist anymore and a BOINC project stops unexpectedly. The handling of these events will enable developers to pursue a high level of availability of a BOINC project (Section 10.3.3).

10.3.3 High Availability

A problem can still arise during the execution of a BOINC project. Nevertheless, this thesis does not cover any approaches towards how exceptions such as a lost host can be handled, i.e. a BOINC-project's owned hosts can become unavailable

and all external hosts having the main project installation cannot work properly any longer. Section 6.4 introduces the Events package which can be used for handling such exceptions. In the course of this thesis only a small range is covered and reserved for future work. Some open questions are: (i) How can most of the possible exceptions during the BOINC runtime be modelled by the help of UML? (ii) How can load-balancing be modelled?

10.3.4 Integrated Development Environment

On the one hand, the prototype version of the Visu@lGrid Integrated Development Environment (VGIDE) has to be improved. On the other hand, more support for context-sensitive menus and automatic resolving of relevant information has to be implemented, e.g. a list of available targets has to be presented during the implementation of an Assimilator and especially when the developer specifies the location of computational results. In addition, the VGIDE can be used for developing and monitoring one or more BOINC projects. Thus, the modelled state machine diagrams can be depicted by visual notifiers, e.g. the current state of the workunit progress is highlighted. Moreover, the related activities and actions of states should be implemented through a well-functioning editor framework for different programming languages.

VGIDE supports the modelling of state machines and several hierarchical levels of states and regions. The code-generation process recognises only a level-depth of one, i.e. only top states, regions in these states, and states in these regions.

10.3.5 Network Security

Every «Host» and «SAN» has a minimum of one network-address card which is used for interconnection (Section 6.5.7). UML can be applied as a security description language which enables the creation of firewall-rules to restrict the communication pathways. A combination between UML and BOINC is possible in this context and can be used on the one hand on BOINC's server-side to restrict the access to the infrastructure; on the other hand, it can be employed on the client-side for enabling and handling the communication between more than one client (Section 10.3.6).

10.3.6 Peer-to-Peer Integration and Message-Passing

BOINC is a framework for solving large-scale computational problems often referred to as "embarrassing" problems, i.e. every computational task can be performed autonomously and without any communication to other clients. The data transfer rate of the internet and the last-mile to private consumers is becoming

faster [104]. It would be interesting to combine BOINC's approach with peer-to-peer and message-passing technologies. A scientific application can figure out automatically whether a cluster installation or multi-core processor is available on the client-side or an application can use adaptive algorithms to select the smartest execution model, i.e. the application is started as a single-process or uses an available cluster-environment and distributes the computation data to speed-up the execution.

10.4 Vision

In future software engineering praxis, the tools can be expected to become much more widely adopted in scientific and industry sectors. Nowadays, the range of different needs increases steadily, e.g. new protocols, hardware or devices, software libraries, usability issues etc. are being invented continuously. Every professional developer has to develop dexterity with such new elements in order to be dependably productive on a daily basis. Accordingly, software engineering is a promising way for handling the complexity of the real world.

As a rule, "Safety first" has to be realised thoroughly in industry. A recent report concerning security issues of industrial plants [171, 172] (e.g. the unsecured and remotely accessible software management of jails or the controlling stations of breweries) has evoked around half a dozen research questions, e.g. can access be prevented by software modelling approaches, or is there a process for ensuring security automatically during runtime by applying smart algorithms? The problems exist due to the fact that the developer has not considered the full set of ramifications because "time is money". Even worse, the developers frequently have no clue how to handle these complications. At this point, software engineering tools and abstraction of the complexity are promising approaches for resolving the security issues: the work presented in the present thesis illustrates a powerful approach that inherently puts security issues in the foreground, due to the need to distribute code (that is potentially safety- or security-sensitive) across processing units that are outside the user's control. This is fundamentally a much more sound basis that can be used for distributed development work in the future.

Exascale Computing is expected to become available within the next ten years [173, 174, 175] and thus it will influence every research field, e.g. high energy and nuclear physics, materials science and chemistry, life sciences, climate change prediction and many more. One exaflop is one quintillion (a number with 18 zeros) floating-point operations per second (FLOPs) and this will be achieved in massively parallel systems. Currently, most parallel computing systems are based on OpenMP or Message-passing Interface (MPI) software technologies,

using up to hundreds of thousands multi-core processors. With these technologies, the scaling of FLOPs and the complexity of the concurrency are based on the addition or removal of processor nodes. In contrast, exascale multi-core processors provide hundreds of thousands of sole cores and, in combination with additional clustered processors, a major challenge will be the smart organisation of how data has to be distributed, e.g. locally or remotely. In future, software engineering research will need to focus on handling this issue: whereas nowadays the software model must be optimised for computation, in the near future it will have to be optimised for minimisation of the data movements between nodes and processor cores. Furthermore, the software engineering tools must provide a way for highly fault-tolerant systems to ensure that when one exascale multi-core processor drops out, the computation will continue to work properly. In addition, heterogeneous systems need to pursue error detection and recovery, e.g., detecting and recovering from an error in a Graphical-Processing Unit (GPU) can involve hundreds of threads simultaneously and contain hundreds of cycles in drain pipelines for starting a recovery [65]. High-level software engineering tools for distributed systems, as presented in this work, will be essential for reliable programming of such systems.

Exascale Computing is conceived as being undertaken on dedicated co-located and well-secured systems, but there will also be great pressure to use 'free' resources such as Public-resource Computing (PRC) and Cloud Computing (CC). The technological requirements for these three approaches are quite similar, e.g. the system management must be highly efficient for managing small or large datasets between nodes. As stated by I. Wladawsky-Berger [176], one can actually regard PRC- and CC-based systems as a kind of exascale class of supercomputers which are designed to support parallel workloads, such as massive information analysis or huge numbers of sensors and mobile devices. Moreover, there can be a niche which enables the coexistence of these computing approaches but the software engineering tools and approaches must permit a simple access for handling them. The fundamental difference is that PRC and CC use vast quantities of processor nodes that are outside the ultimate control of the software user and hence much more rigorous methods to ensure safety and security have to be implemented: the approaches developed in the work reported in this thesis provide a strong basis for this.

Emergent design approaches use artificial intelligence algorithms and a large set of available functionalities or building blocks to generate an application from scratch, but this can only be initiated by describing a problem. For example, in the case of planning a new car, it is necessary to describe its shape and functional requirements: only then is it feasible for it to be generated fully and automatically.

Hence, the description of the shape and other requirements can be realised with Domain-Specific Modeling Languages (DSMLs) such as voice control (e.g. Star Trek's living voice system "Computer, one more wheel on the left." or like Siri's commands "Drive me to the next petrol station.") or plain-text: the generation process is like "code" generation in which "code" can contain any form, e.g. a visualisation of the new car or a printed 3D version. The tools presented in this thesis can be regarded as a service-orientated engineering approach in which a sub-device can be interpreted as a functionality or building block, e.g. the software for controlling the brakes or the air-conditioning system.

As presently conceivable, all future visions can be handled with the help of UML, additional DSMLs and code-generation approaches. As a matter of fact, the author of this thesis believes that the current deficiency in UML adoption is the result of inadequate tool support and the missing user-friendly software development: these deficiencies must be covered and resolved in the next years if the coming environment of massive and distributed computing systems is to be exploited effectively and reliably.

Part IV

Appendix

APPENDIX A

BOINC

This chapter includes additional information of the BOINC-framework, e.g. usable parameters for different BOINC services, configuration attributes, or information on BOINC project directory structures. The BOINC framework is permanently under development. During the writing of this line the latest development revision is 25753[1]. Anyone can contribute to the BOINC framework, the generally suggested way to connect the BOINC developers is BOINC's mailing list: *boinc_dev*.

A.1 Transitioner

Start parameters for BOINC's Transitioner:

–one_pass If this parameter is added, the Transitioner will exit after one finished run.

–d x This parameter defines the debug level, i.e. how many messages during runtime are logged. x can vary between 1 and 4. A higher value enables more log messages.

–mod n i This is a short form for modulo division. Each workunit result has an identifier, n and i are used to select specific workunit results which will be performed by Transitioner. The selection formula is: $(workunit\ identifier\ \%\ n) == i$. Here, $\%$ is the mathematical expression for modulo divisions in most programming languages. If this equation is true, then the workunit result is selected and subsequently performed by the Transitioner.

–sleep_interval x This parameter sets x seconds between two Transitioner executions.

A.2 Feeder

Start parameters for BOINC's Feeder:

[1] Last changes were on the 11th June 2012.

-d Similar to Transitioner (Section 5.3.3).

–sleep_interval x Similar to Transitioner (Section 5.3.3).

–allapps If set, all scientific applications are considered and the shared-memory segment is filled with workunits for all scientific applications.

–random_order Each workunit result has a database table field named "random" which is filled by an integer value during workunit creation. It is a random number. If this parameter is set, the selection is ordered. The default sort order is ascending, with smallest values first [156].

–priority_order Each workunit result has a database table field "priority". If this parameter is set, the selection of workunit results is based on the field value. The field values are ascending.

–priority_order_create_time This parameter is used to add a sorting of the creation time to the previous mentioned parameter. Only one of these parameters is necessary.

–purge_stale x This parameter removes workunit results which are longer than x minutes in the shared-memory segment and not assigned to a client.

–appids a1(,a2)* Only these scientific applications are recognised to feed with workunit results.

–mod n i Similar to Transitioner (Section 5.3.3).

–wmod n i Similar to the previous parameter, the difference is, that *wmod* selects workunits. As a result, a fine-grained selection can be performed by the use of *mod* and *wmod*.

A.3 Validator

Start parameters for BOINC's provided Validators:

–one_pass_N_WU N This parameter defines that *N* workunit results have to be validated.

–one_pass Similar to Transitioner (Section 5.3.3).

–mod n i Similar to Transitioner (Section 5.3.3).

–sleep_interval x Similar to Transitioner (Section 5.3.3).

–max_claimed_credit X If a client has performed a workunit too long, which results in a higher credit value than this defined value, the workunit result is invalid.

–max_granted_credit X This is the maximum allowed credit value for workunit performing.

–grant_claimed_credit A fixed credit value, irrespective of how much work one client has performed.

–update_credited_job Update the credits in the database after credit for a workunit result is granted.

–credit_from_wu The credits are defined in the workunit template file and therefore used for user credit assignments.

A.4 Assimilator

Start parameters for BOINC's provided Assimilator:

–app name Only workunits which are performed by the scientific application *name* are assimilated.

–sleep_interval x Similar to Transitioner (Section 5.3.3).

–mod n i Similar to Transitioner (Section 5.3.3).

–one_pass Similar to Transitioner (Section 5.3.3).

–one_pass_N_WU N Similar to Validator; but here for assimilation (Section 5.3.5).

-d N Similar to Transitioner (Section 5.3.3).

–dont_update_db Do the assimilation but do not update the database entries.

–noinsert Do not insert any values in the scientific application specific database.

APPENDIX B

UML4BOINC Packages & Stereotypes

B.1 Infrastructure Stereotypes

«ApplicationComponent» in Section 6.5.1:
Describes one scientific application, which is used for computations.

«Column» in Section 6.5.2:
A database column of a specific table.

«Connector» in Section 6.5.3:
Connects exported shares with imports and vice versa. An export could be used by multiple imports.

«Database» in Section 6.5.4:
Service to host BOINC's master/science database and optional replications to increase DB performance.

«DB» in Section 6.5.5:
Allow BOINC services to access database tables.

«Host» in Section 6.5.6:
Specifies one host within a BOINC project. A host owns some or all BOINC services.

«NIC» in Section 6.5.7:
Defines network settings for one host.

«OperatingSystem» in Section 6.5.8:
Unique literals for each planned or supported operating system, e.g. Linux32, Linux64, Windows32, or Windows64.

«PortExport» in Section 6.5.9:
Definition of network shared file directories between hosts.

«PortImport» in Section 6.5.9:
Definition of network shared file directories between hosts.

«Project» in Section 6.5.10:
One specific BOINC project.

«ReplicationAssociation» in Section 6.5.11:
Connects two «DB» contexts to enable a replication of one of them.

«SAN» in Section 6.5.12:
External host with capabilities to store data, e.g. workunits or computational results. Also known as storage area network (SAN).

«Service» in Section 6.5.13:
BOINC services on one host, e.g. feeder or Transitioner.

«ServiceType» in Section 6.5.18:
Type of a BOINC service, e.g. feeder or Transitioner.

«Share» in Section 6.5.19:
One specific directory which is ex- or imported. Exported directories are defined as provided interfaces, otherwise they are defined as required interfaces.

«ShareType» in Section 6.5.20:
Unique literal for each planned or supported share type, e.g. NFS or SAMBA.

«Software» in Section 6.5.21:
Describes additional software libraries which are necessary for the execution of services or a scientific application.

«SystemOperation» in Section 6.5.22:
Implementations of system or BOINC service functionalities, e.g. routines for a Validator or Assimilator.

«Table» in Section 6.5.23:
A describer of a database table to allow access to its rows.

«UserOperation» in Section 6.5.24:
Operations are executable by users.

B.2 Application Stereotypes

«Action» in Section 6.6.1
An abstraction of parts of BOINC's API.

«Atomic» in Section 6.6.2
An activity to execute atomic areas.

«Checkpoint» in Section 6.6.3
Specifies how checkpoints are structured.

«ComputationData» in Section 6.6.4
Provides input and output files as Parameters.

«ComputingState» in Section 6.6.5
Emits signals to describe the current computing state.

«Control» in Section 6.6.6
Receiving events to control the execution of a scientific application.

«Exchange» in Section 6.6.7
Input and output pin to describe inter-process communication.

«Exchanger» in Section 6.6.8
A data storing element for inter-process communication.

«FileActionMode» in Section 6.6.9
An enumeration for the specification of the execution behaviour of «Action» when *type* is *#FileHandling*.

«FileInformation» in Section 6.6.10
Provides information of a specific file.

«FileKind» in Section 6.6.11
An enumeration to specify a specific file.

«InputFile» in Section 6.6.12
Used to specify input files for the scientific application.

«IsStandalone» in Section 6.6.13
Used to check if the scientific application is executed in a BOINC-context.

«Processing» in Section 6.6.14
A state to describe different parts of a scientific application's behaviour, e.g. inter-process communication, checkpointing, asynchronous-messages, and computation core routines.

«ProcessingEntry» in Section 6.6.15
Entry point to start the computation.

«ProcessingExit» in Section 6.6.16
Exit point to leave the computation.

«ProcessingHistory» in Section 6.6.17
Used as start point when a computation is resumed and a checkpoint is available.

«ProcessingState» in Section 6.6.18
A general state for possible state execution states on the client-side.

«ProcessingTerminate» in Section 6.6.19
Terminates the execution of the client-side.

«ScientificApplication» in Section 6.6.20
An activity to specify scientific applications.

«Sync» in Section 6.6.21
Used to synchronise the inter-process communication.

«Timing» in Section 6.6.22
An activity to specify the execution of timed functionalities.

«TimingFlow» in Section 6.6.23
Used to connect nodes within «Timing».

«TimingSignal» in Section 6.6.24
Emits a signal for the execution of targeted functionalities.

«TrickleUp» in Section 6.6.25
Triggers the sending of an asynchronous-message.

«WaitForNetwork» in Section 6.6.26
An activity to wait for network connectivity.

B.3 Work Stereotypes

«Assimilation» in Section 6.10.1
A state machine to specify BOINC's assimilation process.

«Datafield» in Section 6.8.1
Specifies the format of input/output files.

«FileType» in Section 6.8.2
Specifies the input/output file format.

«Input» in Section 6.8.3
Specifies an input file of a workunit.

«InterfaceAssimilate» in Section 6.9.1
An interface to access output files and to allow storing of them at specific target places.

«InterfaceDataset» in Section 6.9.2
An interface to access datafields of input/output files.

«InterfaceDataSource» in Section 6.9.3
An interface which emits the creation of workunits.

«InterfaceValidate» in Section 6.9.4
An interface to access output files.

«MonitorFinished» in Section 6.10.2
Used to specify the handling of a «Workunit».

«OnClient» in Section 6.10.3
This state specifies the execution on the client-side.

«Output» in Section 6.8.4
Specifies an output file of a workunit.

«Range» in Section 6.8.5
Abstract class to specify the ranges of datafields of input files, when they are created.

«RangeRule» in Section 6.8.6
Specialisation of «Range», used to specify a specific creation rule.

«RangeSimple» in Section 6.8.7
Specialisation of «Range», used to specify a value between two bounds.

«Series» in Section 6.8.8
A set of workunits for one specific scientific application.

«SeriesRegion» in Section 6.10.4
Used to specify the handling of a «Series».

«SeriesState» in Section 6.10.5
Used to specify the handling of a «Series».

«SeriesTerminated» in Section 6.10.6
Used to specify the handling of a «Series».

«Validation» in Section 6.10.7
A state machine to specify BOINC's validation process.

«Workunit» in Section 6.8.9
A specific BOINC workunit for a scientific application.

«WorkunitAssociation» in Section 6.8.10
Used to specify the handling of a «Workunit».

«WorkunitEntry» in Section 6.10.8
Used to specify the handling of a «Workunit».

«WorkunitExit» in Section 6.10.9
Used to specify the handling of a «Workunit».

«WorkunitMonitor» in Section 6.10.10
Used to specify the handling of a «Workunit».

«WorkunitProcessing» in Section 6.10.11
Used to specify the handling of a «Workunit».

«WorkunitRegion» in Section 6.10.12
Used to specify the handling of a «Workunit».

«WorkunitState» extends `Class (from Kernel)` in Section 6.8.11: Used to specify the handling of a «Workunit».

«WorkunitState» extends `State (from BehaviorStateMachines)` in Section 6.10.13: Used to specify the handling of a «Workunit».

«WorkunitTransition» in Section 6.10.14
Used to specify the handling of a «Workunit».

B.4 RBAC Stereotypes

«Permission» in Section 6.7.1
Set of permissions for specific «Role» items.

«PermissionAssignment», «ResourceAssignment» See «UserAssignment».

«UserAssignment» in the Sections 6.7.2, 6.7.4, 6.7.7
Associates users, roles, permissions, and resources. «UserAssignment» assigns users to roles, «PermissionAssignment» assigns roles to permissions, and «ResourceAssignment» assigns permissions to resources.

«Resource» in Section 6.7.3
A user can only be assigned to resources when resources have this stereotype.

«User» in Section 6.7.6
A physical user which has different permissions within one BOINC project described by roles and permissions.

«UserRemote» in Section 6.7.8
Specifies a real user with remote log-in possibilities.

«Resource» in Section 6.7.3
Enables the RBAC functionalities for any item.

«Role» in Section 6.7.5
Defines roles for users and which permissions these roles include.

B.5 Timing Stereotypes

«EventFlow» in Section 6.12.1
Associates «Task», «TaskFork», and «TimeEvent».

«Task» in Section 6.12.2
Specifies a component which is used as a BOINC task.

«TaskFork» in Section 6.12.3
Used as a structure element for visualisation and to allow the traversing between the time event and it associated «Task» components.

«TimeEvent» in Section 6.12.4
Starting point for events.

«TimeExpression» in Section 6.12.5
Root of time expressions.

«TimeForTask» in Section 6.12.6
Specifies when a «Task» has to be executed.

B.6 Events Stereotypes

«TrickleMessage» in Section 6.11.1
Specifies an asynchronous-message.

«TrickleMessageValue» in Section 6.11.2
Specifies what an asynchronous-message contains.

«TricklePartner» in Section 6.11.3
Specifies which host/client/application has to handle a specific asynchronous-message.

«TricklePartnerType» in Section 6.11.4
Specifies the receiver of an asynchronous-message.

«TrickleType» in Section 6.11.5
Specifies the format of an asynchronous-message.

APPENDIX C

Code Generation

This chapter includes language syntax and grammar definitions which are used within this thesis. In addition, an implementation for the creation and handling of series and their workunits is given. Parts of the code-generation processes which are used within Visu@lGrid [184] are shown. Finally, an example implementation for asynchronous-message handling for the server- and client-side is given.

C.1 ANTLR's Abstract-syntax Tree for Summation

This section includes an ANTLR example for summation of integer values and the assignment to a specific identifier. Listing C.1 includes the syntax specification, the so-called grammar specification. Listing C.2 includes the Abstract-syntax Tree (AST) specification to handle the created AST of Listing C.1. Listing C.3 contains an example of input values which are lexically analysed and parsed by the ANTLR example. The main-function to start the ANTLR application is shown in Listing C.4. Finally, the make file to create the executable of this ANTLR example is shown in Listing C.5.

```
1  grammar Summation;
2  options {
3    output = AST; ASTLabelType=CommonTree;
4  }
5  tokens { ADD; ASSIGN; }
6  summation: leaf* -> leaf* ;
7  leaf: INT ('+' INT)* '=>' ID -> ^(ASSIGN ^(ADD INT*) ID);
8
9  INT:     '0'..'9'+ ;
10 ID:      ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
11 WS:      (' '|'\n'|'\r'|'\t')+ {$channel=HIDDEN;} ;
```

Listing C.1: Syntax specification – so-called grammar specification – for the summation of two or more integer values, the result is assigned to a specific identifier. An Abstract-syntax Tree (AST) is created on the right-hand side of line 7 and is started by the use of ->, Listing C.2 contains the grammar for the AST syntax; the filename is *Summation.g*

```
tree grammar SummationTree;
options {
  tokenVocab=Summation; ASTLabelType=CommonTree;
}
root: ^(ASSIGN ^(ADD INT*) ID) ;
```

Listing C.2: Grammar of the Abstract-syntax Tree (AST) for the summation of one or more integer values; the filename is *SummationTree.g*

```
10 + 11 => result_addition1
20 + 22 + 222 => result_addition2
```

Listing C.3: An example of valid input allowed by the grammar specification of Listing C.1; the filename is *Summation.test*

```
import java.io.*;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import org.antlr.stringtemplate.*;

public class Summation {
  public static void main(String[] args) throws Exception {
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    SummationLexer lexer = new SummationLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    SummationParser parser = new SummationParser(tokens);

    // NOTE: Use only [1] or [2], otherwise an exception raises.

    int mode = 2;

    // [1] Start
    if(mode == 1) {
      SummationParser.summation_return root = parser.summation();
      CommonTreeNodeStream nodes
                  = new CommonTreeNodeStream((Tree)root.tree);
      SummationTree walker = new SummationTree(nodes);
      walker.root();
    }
    // [1] End

    // [2] Start
    if(mode == 2) {
      CommonTree t  = (CommonTree) parser.summation().getTree();
      System.out.println(t.toStringTree());

    // Source: http://www.antlr.org/pipermail/antlr-interest/
    //                2009-March/033417.html
    // Accessed: 2nd June 2012

      DOTTreeGenerator gen = new DOTTreeGenerator();
      String outputName = "Summation.dot";
      System.out.println("Producing AST dot (graphviz) file");
      StringTemplate st = gen.toDOT(t, new CommonTreeAdaptor());

      FileWriter outputStream = new FileWriter(outputName);
      outputStream.write(st.toString());
      outputStream.close();

```

```
45        long pStart = System.currentTimeMillis();
46        Process proc = Runtime.getRuntime()
47                        .exec("dot -Tpng -o Summation.png Summation.dot");
48        long pStop = System.currentTimeMillis();
49        System.out.println("PNG graphic produced in "
50                           + (pStop - pStart) + "ms.");
51      }
52      // [2] End
53    }
54  }
```

Listing C.4: A Java-based implementation to call ANTLR's generated functionalities based on the grammar specification of Listing C.1 and the tree-grammar specification of Listing C.2; the filename is *Summation.java*

```
1  all:
2    java -cp antlrworks-1.4.3.jar org.antlr.Tool Summation*.g
3    javac -cp antlrworks-1.4.3.jar:stringtremplate-4.0.5.jar  *.java
4    java -cp antlrworks-1.4.3.jar:. Summation < Summation.test
5    convert Summation.png Summation.eps
6
7  clean:
8    rm -f *.class SummationLexer* SummationParser* \
9                  SummationTree.java *.tokens
```

Listing C.5: A make file to create the ANTLR-based lexical analysing and parsing functionalities, to compile the main-routines of Listing C.4, to call the summation application with the input values of Listing C.3, and finally the creation of an AST visualisation; the filename is *Makefile*

C.2 Statemachine's DSL-syntax

Listing C.6 contains the syntax specification, the so-called grammar specification of the *Ries Statemachine* (RSM). An example of a use of RSM is given in Listing 8.9. An additional Tester class is given in Listings C.7 and C.8, which can be combined with the example to fill it with behaviour.

```
1  RSM: RSMCodeAnchors*
2       STATEMACHINE stateMachineName=ID '{'
3         RSMStates RSMTransitions
4       '}' ;
5  RSMCodeAnchors : ANCHORS ccode=CCODE ;
6  RSMStates      : STATES '{' (RSMState | RSMRegions)* '}' ;
7  RSMState       : stateType=STATETYPES stateId=ID enterCCode=ONENTERCCODE?
8                   ccode=CCODE? exitCCode=ONEXITCCODE? ;
9  RSMRegions     : REGIONS regionId=ID '{' RSMRegion* '}' ;
10 RSMRegion      : regionId=ID '{' RSMStates* '}'
11                  (':' (stateType=(ENTRY|EXIT) stateId=ID )+ ':')? ;
12 RSMTransitions : TRANSITIONS '{' RSMTransition* '}' ;
13 RSMTransition  : owner=ID ':' source=ID ('+' (event=ID )*)?
14                 (guards=GUARDCODE)? ('/' behaviorcode=FUNCTIONCALL)?
15                 '==' target=ID ;
16
17 TOKENS {
```

```
  STATEMACHINE='Statemachine'; TRANSITIONS='Transitions';
  ANCHORS='Anchors'; REGIONS='Regions'; STATES='States';
  ENTRY='Entry'; EXIT='Exit';
}

STATETYPES    : ('Initial'|'Simple'|'Final'|'History'|'Terminate') ;
ID            : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
INT           : '0'..'9'+ ;
FLOAT         : ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
              | '.' ('0'..'9')+ EXPONENT?
              | ('0'..'9')+ EXPONENT ;
COMMENT       : '//' ~('\n'|'\r')* '\r'? '\n'
              | '/*' (.)* '*/' ;
WS            : ( ' ' | '\t' | '\r' | '\n' ) ;
STRING        : '"' ( ESC_SEQ | ~('\\'|'"') )* '"' ;
ONENTERCCODE  : '<[onEntry' (.)* '--]>' ;
ONEXITCCODE   : '<[onExit' (.)* '--]>' ;
CCODE         : '<[--' (.)* '--]>' ;
GUARDCODE     : '[' (.)* ']' ;
CHAR          : '\'' ( ESC_SEQ | ~('\''|'\\') ) '\'' ;
FUNCTIONCALL  : ID(('.'|'->') ID)* '('(.*)')';
```

Listing C.6: *UML 2 Statemachine for C++*'s DSL-syntax named *Ries Statemachine* (RSM)

```
#pragma once
#ifndef __TESTER_H__
#define __TESTER_H__
class Tester {
private:
  int _x;
public:
  Tester() : _x(8) {}
  void loop(int start, int end);
  void processing();
  void close();
  void setX(int value);
  int x() const;
  void show();
};
#endif
```

Listing C.7: C++-Header file of Tester; the filename is *Tester.h*

```
#include <iostream>
#include <Tester.h>

void Tester::loop(int start, int end) {
  for( ; start < end; start++) {
    std::cout << "{x} Tester::loop(" << start << ")" << std::endl;
  }
}

void Tester::processing() {
  std::cout << "{X} processing(): x=" << x() << std::endl;
}

void Tester::close() {
  std::cout << "{X} close()" << std::endl;
}

void Tester::show() {
```

```
19    std::cout << "{X} Tester::show() x:=" << x() << std::endl;
20  }
21
22  void Tester::setX(int value) { _x = value; }
23  int Tester::x() const        { return _x;  }
```

Listing C.8: C++-Source file of Tester; the filename is *Tester.cpp*

C.3 Spinhenge@home's generated State Machine

```
1   /*
2    * Christian Benjamin Ries, May 2013
3    * mail@christianbenjaminries.de
4    * http://www.christianbenjaminries.de
5    * Template version: 0.1
6    * DO NOT CHANGE SOMETHING HERE!!!
7    * THIS FILE IS GENERATED!!!
8    */
9   Anchors
10  <[--
11    /* Ries */
12    #include <ries/data/exceptions.h>
13    #include <ries/boinc/mvc/controller.h>
14    #include <ries/boinc/mvc/boinc_handler.h>
15    /* Generated */
16    #include <SpinhengeAtHome_controller.h>
17    #include <SpinhengeAtHome_work_impl.h>
18    #include <SpinhengeAtHome_ipc.h>
19    /* BOINC */
20    #include <diagnostics.h>
21    /* C++ */
22    #include <iostream>
23    /* C, Linux */
24    #include <unistd.h>
25    /* boost */
26    #include <boost/shared_ptr.hpp>
27    #include <boost/make_shared.hpp>
28    extern boost::shared_ptr<Ries::BOINC::MVC::Controller> controller;
29    /* Handler */
30    boost::shared_ptr<Ries::BOINC::MVC::Handler> handler(new Ries::BOINC::MVC::
          Handler());
31    void unlinkIPC()
32    {
33      unlink("SpinhengeAtHome_ipc");
34    }
35    void sint(int signr)
36    {
37      switch(signr)
38      {
39        case SIGINT:
40        {
41           unlinkIPC();
42           boinc_finish(1);
43        } break;
44        default:
45          std::cerr << "Signal " << signr << " unknown." << std::endl;
46      }
47    }
```

```
48    const char *SpinhengeAtHome_TODO_UNIQUE_IDENTIFIER3485995394_MAIN = "$(C)
          Christian Benjamin Ries [www.christianbenjaminries.de]$";
49    /* Actions */
50    void actionFinish()
51    {
52      controller->handler()->finished(0);
53    }
54  --]>
55  Statemachine SpinhengeAtHomeStateMachine
56  {
57    States {
58      Initial      RStart
59      Simple       RInitialization
60      <[--
61        signal(SIGINT, sint);
62        atexit(unlinkIPC);
63        /* Diagnostics */
64        /* Controller */
65        controller = boost::make_shared<SpinhengeAtHomeImpl>();
66        try
67        {
68          /* BOINC */
69          handler->init();
70          /* IPC */
71          if(VisualGrid::Generated::IPCInitialization("SpinhengeAtHome_ipc")) {
72            VisualGrid::Generated::IPCInstance()->setHandler(handler);
73            handler->registerTimerCallback(VisualGrid::Generated::IPCUpdate);
74            controller->setIPCEnabled(true);
75          }
76        } catch(Ries::Exceptions ex) {
77          std::cerr << "ex.what(): " << ex.what() << std::endl;
78          controller->handler()->finished(1);
79        }
80      --]>
81      Simple       RReady
82      Regions RProcessing
83      {
84        R0
85        {
86          States
87          {
88            Initial      RStart0
89            // ...
90            Simple       RIdleReport
91            <[--
92              sleep(5);
93            --]>
94            // ...
95            Simple       RReporting
96            <[--
97              /* Interprocess-communication */
98              boost::shared_ptr<SpinhengeAtHome> SpinhengeAtHomePtr =
99                boost::static_pointer_cast<SpinhengeAtHome>(controller);
100             SpinhengeAtHomePtr->doSharing();
101           --]>
102         }
103       }
104       R1
105       {
106         States
107         {
108           Initial      RStart1
```

```
109             Simple      RIdleCheckpointing
110             <[--
111               sleep(5);
112             --]>
113             Simple      RCheckpointing
114             <[--
115               controller->doCheckpoint();
116             --]>
117           }
118         }
119         R2
120         {
121           States
122           {
123             Initial     RStart2
124             History     RH2
125             Simple      RReadCheckpoint
126             // Core computing of the scientific application.
127             Simple      RComputing
128             <[--
129               /* Files */
130               controller->addInputFile("data.xml", "metropolis_data.xml");
131               controller->addInputFile("param.jj", "param.jj");
132               controller->addInputFile("param.nn", "param.nn");
133               controller->addInputFile("param.ww", "param.ww");
134               controller->addOutputFile("result.zip", "metropolis_out.erg");
135               controller->addHandler(handler);
136               controller->start();
137             --]>
138           }
139         } :
140           Entry       RGoAhead
141           Exit        RFailure
142           Exit        RSuspend
143           Exit        RFinished
144           :
145         }
146         // Create checkpoint...
147         Simple      RPause
148         <[-- --]>
149         // Error handling
150         Simple      RTroubleshooting
151         <[--
152           std::cerr << "{E} Arg, something went wrong!" << std::endl;
153         --]>
154         Final       RFinish
155         Terminate   RAbort
156       }
157       Transitions {
158 // Owner        source + event [guard] / action == target
159 // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
160 RegionMain: RStart == RInitialization
161 RegionMain: RInitialization == RReady
162 RegionMain: RReady == RProcessing
163 RegionMain: RFailure == RTroubleshooting
164 RegionMain: RTroubleshooting == RAbort
165 RegionMain: RTroubleshooting == RGoAhead
166 RegionMain: RGoAhead == RComputing
167 RegionMain: RSuspend == RPause
168 RegionMain: RPause == RFinish
169 RegionMain: RFinished / actionFinish() == RFinish
170 R0: RStart0 == RIdleReport
```

```
171 R0: RIdleReport == RReporting
172 R0: RReporting == RIdleReport
173 R1: RStart1 == RIdleCheckpointing
174 R1: RIdleCheckpointing == RCheckpointing
175 R1: RCheckpointing == RIdleCheckpointing
176 R2: RStart2 == RComputing
177 R2: RH2 == RReadCheckpoint
178 R2: RReadCheckpoint == RComputing
179 R2: RComputing + evComputingFailed == RFailure
180 R2: RComputing + evComputingSuspend == RSuspend
181 R2: RComputing + evComputingFinished  evComputingCancel == RFinished
182 // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
183   }
184 }
```

Listing C.9: The *Ries Statemachine* (RSM) code for Spinhenge@home described in Section 9.2.

```
1  // libries
2  #include <ries/uml/behavior.h>
3  #include <ries/uml/statemachine.h>
4  // C++
5  #include <iostream>
6  #ifdef _WIN32
7  // Windows
8  #include <windows.h>
9  #define SLEEP(x) Sleep(x*1000)
10 #else
11 // Linux
12 #include <unistd.h>
13 #define SLEEP(x) sleep(x)
14 #endif
15 using namespace Ries::UML;
16 /* Ries */
17     #include <ries/data/exceptions.h>
18     #include <ries/boinc/mvc/controller.h>
19     #include <ries/boinc/mvc/boinc_handler.h>
20     /* Generated */
21     #include <SpinhengeAtHome_controller.h>
22     #include <SpinhengeAtHome_work_impl.h>
23     #include <SpinhengeAtHome_ipc.h>
24     /* BOINC */
25     #include <diagnostics.h>
26     /* C++ */
27     #include <iostream>
28     /* C, Linux */
29     #include <unistd.h>
30     /* boost */
31     #include <boost/shared_ptr.hpp>
32     #include <boost/make_shared.hpp>
33     extern boost::shared_ptr<Ries::BOINC::MVC::Controller> controller;
34     /* Handler */
35     boost::shared_ptr<Ries::BOINC::MVC::Handler> handler(new Ries::BOINC::MVC::
           Handler());
36     void unlinkIPC()
37     {
38       unlink("SpinhengeAtHome_ipc");
39     }
40     void sint(int signr)
41     {
42       switch(signr)
```

```
43        {
44          case SIGINT:
45          {
46            unlinkIPC();
47            boinc_finish(1);
48          } break;
49          default:
50            std::cerr << "Signal " << signr << " unknown." << std::endl;
51        }
52      }
53      const char *SpinhengeAtHome_TODO_UNIQUE_IDENTIFIER3485995394_MAIN = "$(C)
            Christian Benjamin Ries [www.christianbenjaminries.de]$";
54      /* Actions */
55      void actionFinish()
56      {
57        controller->handler()->finished(0);
58      }
59
60  class EffecttRFinishedRFinish : public FunctionBehavior {
61  public:
62      EffecttRFinishedRFinish() : FunctionBehavior("EffecttRFinishedRFinish") {}
63      void execute() { actionFinish(); }
64  };
65  class BehaviorRCheckpointing : public Behavior {
66  public:
67      BehaviorRCheckpointing() : Behavior("BehaviorRCheckpointing") {}
68      void execute() {controller->doCheckpoint();}
69  };
70  class BehaviorRComputing : public Behavior {
71  public:
72      BehaviorRComputing() : Behavior("BehaviorRComputing") {}
73      void execute() {/* Files */
74        controller->addInputFile("data.xml", "metropolis_data.xml");
75        controller->addInputFile("param.jj", "param.jj");
76        controller->addInputFile("param.nn", "param.nn");
77        controller->addInputFile("param.ww", "param.ww");
78        controller->addOutputFile("result.zip", "*.res;*.dat");
79        controller->addHandler(handler);
80        controller->start();}
81  };
82  class BehaviorRIdleCheckpointing : public Behavior {
83  public:
84      BehaviorRIdleCheckpointing() : Behavior("BehaviorRIdleCheckpointing") {}
85      void execute() {sleep(5);}
86  };
87  class BehaviorRIdleReport : public Behavior {
88  public:
89      BehaviorRIdleReport() : Behavior("BehaviorRIdleReport") {}
90      void execute() {sleep(5);}
91  };
92  class BehaviorRInitialization : public Behavior {
93  public:
94      BehaviorRInitialization() : Behavior("BehaviorRInitialization") {}
95      void execute() {signal(SIGINT, sint);
96        atexit(unlinkIPC);
97        /* Diagnostics */
98        /* Controller */
99        controller = boost::make_shared<SpinhengeAtHomeImpl>();
100       try
101       {
102         /* BOINC */
103         handler->init();
```

```
104         /* IPC */
105         if(VisualGrid::Generated::IPCInitialization("SpinhengeAtHome_ipc")) {
106           VisualGrid::Generated::IPCInstance()->setHandler(handler);
107           handler->registerTimerCallback(VisualGrid::Generated::IPCUpdate);
108           controller->setIPCEnabled(true);
109         }
110       } catch(Ries::Exceptions ex) {
111         std::cerr << "ex.what(): " << ex.what() << std::endl;
112         controller->handler()->finished(1);
113       }}
114 };
115 class BehaviorRPause : public Behavior {
116 public:
117     BehaviorRPause() : Behavior("BehaviorRPause") {}
118     void execute() {}
119 };
120 class BehaviorRReporting : public Behavior {
121 public:
122     BehaviorRReporting() : Behavior("BehaviorRReporting") {}
123     void execute() {/* Interprocess-communication */
124       boost::shared_ptr<SpinhengeAtHome> SpinhengeAtHomePtr =
125         boost::static_pointer_cast<SpinhengeAtHome>(controller);
126       SpinhengeAtHomePtr->doSharing();}
127 };
128 class BehaviorRTroubleshooting : public Behavior {
129 public:
130     BehaviorRTroubleshooting() : Behavior("BehaviorRTroubleshooting") {}
131     void execute() {std::cerr << "{E} Arg, something went wrong!" << std::endl;}
132 };
133
134 int main(int argc, char **argv)
135 {
136   StateMachine *SpinhengeAtHomeStateMachine
137     = new StateMachine("SpinhengeAtHomeStateMachine");
138   Region *RegionMain = new Region("RegionMain");
139   RegionMain->stateMachine = SpinhengeAtHomeStateMachine;
140   PseudoState *RStart = new PseudoState(StateInitial, "RStart");
141   State *RInitialization = new State("RInitialization");
142   State *RReady = new State("RReady");
143   State *RProcessing = new State("RProcessing");
144   int R0RegionIndex = RProcessing->region.size();
145   (*RProcessing)[R0RegionIndex] = new Region("R0");
146   (*RProcessing)[R0RegionIndex]->state = RProcessing;
147   Region *R0 = (*RProcessing)[R0RegionIndex];
148   PseudoState *RStart0 = new PseudoState(StateInitial, "RStart0");
149   State *RIdleReport = new State("RIdleReport");
150   State *RReporting = new State("RReporting");
151   int R1RegionIndex = RProcessing->region.size();
152   (*RProcessing)[R1RegionIndex] = new Region("R1");
153   (*RProcessing)[R1RegionIndex]->state = RProcessing;
154   Region *R1 = (*RProcessing)[R1RegionIndex];
155   PseudoState *RStart1 = new PseudoState(StateInitial, "RStart1");
156   State *RIdleCheckpointing = new State("RIdleCheckpointing");
157   State *RCheckpointing = new State("RCheckpointing");
158   int R2RegionIndex = RProcessing->region.size();
159   (*RProcessing)[R2RegionIndex] = new Region("R2");
160   (*RProcessing)[R2RegionIndex]->state = RProcessing;
161   Region *R2 = (*RProcessing)[R2RegionIndex];
162   PseudoState *RStart2 = new PseudoState(StateInitial, "RStart2");
163   PseudoState *RH2 = new PseudoState(StateDeepHistory, "RH2");
164   State *RReadCheckpoint = new State("RReadCheckpoint");
165   State *RComputing = new State("RComputing");
```

```
166   PseudoState *RGoAhead = new PseudoState(StateEntry, "RGoAhead");
167   RGoAhead->state = RProcessing;
168   RProcessing->connectionPoint.push_back(RGoAhead);
169   ConnectionPointReference *cRefRGoAhead = new ConnectionPointReference("
          cRefRGoAhead");
170   cRefRGoAhead->state = RProcessing;
171   cRefRGoAhead->exit.push_back(RGoAhead);
172   RProcessing->connection.push_back(cRefRGoAhead);
173   PseudoState *RFailure = new PseudoState(StateExit, "RFailure");
174   RFailure->state = RProcessing;
175   RProcessing->connectionPoint.push_back(RFailure);
176   ConnectionPointReference *cRefRFailure = new ConnectionPointReference("
          cRefRFailure");
177   cRefRFailure->state = RProcessing;
178   cRefRFailure->exit.push_back(RFailure);
179   RProcessing->connection.push_back(cRefRFailure);
180   PseudoState *RSuspend = new PseudoState(StateExit, "RSuspend");
181   RSuspend->state = RProcessing;
182   RProcessing->connectionPoint.push_back(RSuspend);
183   ConnectionPointReference *cRefRSuspend = new ConnectionPointReference("
          cRefRSuspend");
184   cRefRSuspend->state = RProcessing;
185   cRefRSuspend->exit.push_back(RSuspend);
186   RProcessing->connection.push_back(cRefRSuspend);
187   PseudoState *RFinished = new PseudoState(StateExit, "RFinished");
188   RFinished->state = RProcessing;
189   RProcessing->connectionPoint.push_back(RFinished);
190   ConnectionPointReference *cRefRFinished = new ConnectionPointReference("
          cRefRFinished");
191   cRefRFinished->state = RProcessing;
192   cRefRFinished->exit.push_back(RFinished);
193   RProcessing->connection.push_back(cRefRFinished);
194   State *RPause = new State("RPause");
195   State *RTroubleshooting = new State("RTroubleshooting");
196   FinalState *RFinish = new FinalState("RFinish");
197   PseudoState *RAbort = new PseudoState(StateTerminate, "RAbort");
198
199   Trigger *evComputingCancel= new Trigger("evComputingCancel");
200   Trigger *evComputingFailed= new Trigger("evComputingFailed");
201   Trigger *evComputingFinished= new Trigger("evComputingFinished");
202   Trigger *evComputingSuspend= new Trigger("evComputingSuspend");
203
204   Transition *tRStartRInitialization = new Transition("tRStartRInitialization",
          RStart, RInitialization);
205   Transition *tRInitializationRReady = new Transition("tRInitializationRReady",
          RInitialization, RReady);
206   Transition *tRReadyRProcessing = new Transition("tRReadyRProcessing", RReady,
          RProcessing);
207   Transition *tRFailureRTroubleshooting = new Transition("
          tRFailureRTroubleshooting", RFailure, RTroubleshooting);
208   Transition *tRTroubleshootingRAbort = new Transition("tRTroubleshootingRAbort",
          RTroubleshooting, RAbort);
209   Transition *tRTroubleshootingRGoAhead = new Transition("
          tRTroubleshootingRGoAhead", RTroubleshooting, RGoAhead);
210   Transition *tRGoAheadRComputing = new Transition("tRGoAheadRComputing", RGoAhead
          , RComputing);
211   Transition *tRSuspendRPause = new Transition("tRSuspendRPause", RSuspend, RPause
          );
212   Transition *tRPauseRFinish = new Transition("tRPauseRFinish", RPause, RFinish);
213   Transition *tRFinishedRFinish = new Transition("tRFinishedRFinish", RFinished,
          RFinish);
```

```
214   EffecttRFinishedRFinish *instanceEffecttRFinishedRFinish = new
          EffecttRFinishedRFinish();
215   tRFinishedRFinish->add(instanceEffecttRFinishedRFinish);
216   Transition *tRStart0RIdleReport = new Transition("tRStart0RIdleReport", RStart0,
           RIdleReport);
217   Transition *tRIdleReportRReporting = new Transition("tRIdleReportRReporting",
          RIdleReport, RReporting);
218   Transition *tRReportingRIdleReport = new Transition("tRReportingRIdleReport",
          RReporting, RIdleReport);
219   Transition *tRStart1RIdleCheckpointing = new Transition("
          tRStart1RIdleCheckpointing", RStart1, RIdleCheckpointing);
220   Transition *tRIdleCheckpointingRCheckpointing = new Transition("
          tRIdleCheckpointingRCheckpointing", RIdleCheckpointing, RCheckpointing);
221   Transition *tRCheckpointingRIdleCheckpointing = new Transition("
          tRCheckpointingRIdleCheckpointing", RCheckpointing, RIdleCheckpointing);
222   Transition *tRStart2RComputing = new Transition("tRStart2RComputing", RStart2,
          RComputing);
223   Transition *tRH2RReadCheckpoint = new Transition("tRH2RReadCheckpoint", RH2,
          RReadCheckpoint);
224   Transition *tRReadCheckpointRComputing = new Transition("
          tRReadCheckpointRComputing", RReadCheckpoint, RComputing);
225   Transition *tRComputingRFailure = new Transition("tRComputingRFailure",
          RComputing, RFailure);
226   tRComputingRFailure->add(evComputingFailed);
227   Transition *tRComputingRSuspend = new Transition("tRComputingRSuspend",
          RComputing, RSuspend);
228   tRComputingRSuspend->add(evComputingSuspend);
229   Transition *tRComputingRFinished = new Transition("tRComputingRFinished",
          RComputing, RFinished);
230   tRComputingRFinished->add(evComputingFinished);
231   tRComputingRFinished->add(evComputingCancel);
232
233   RStart->container = RegionMain;
234   RInitialization->container = RegionMain;
235   RReady->container = RegionMain;
236   RProcessing->container = RegionMain;
237   RStart0->container = R0;
238   RIdleReport->container = R0;
239   RReporting->container = R0;
240   RStart1->container = R1;
241   RIdleCheckpointing->container = R1;
242   RCheckpointing->container = R1;
243   RStart2->container = R2;
244   RH2->container = R2;
245   RReadCheckpoint->container = R2;
246   RComputing->container = R2;
247   RPause->container = RegionMain;
248   RTroubleshooting->container = RegionMain;
249   RFinish->container = RegionMain;
250   RAbort->container = RegionMain;
251   tRStartRInitialization->container = RegionMain;
252   tRInitializationRReady->container = RegionMain;
253   tRReadyRProcessing->container = RegionMain;
254   tRFailureRTroubleshooting->container = RegionMain;
255   tRTroubleshootingRAbort->container = RegionMain;
256   tRTroubleshootingRGoAhead->container = RegionMain;
257   tRGoAheadRComputing->container = RegionMain;
258   tRSuspendRPause->container = RegionMain;
259   tRPauseRFinish->container = RegionMain;
260   tRFinishedRFinish->container = RegionMain;
261   tRStart0RIdleReport->container = R0;
262   tRIdleReportRReporting->container = R0;
```

```
263   tRReportingRIdleReport->container = R0;
264   tRStart1RIdleCheckpointing->container = R1;
265   tRIdleCheckpointingRCheckpointing->container = R1;
266   tRCheckpointingRIdleCheckpointing->container = R1;
267   tRStart2RComputing->container = R2;
268   tRH2RReadCheckpoint->container = R2;
269   tRReadCheckpointRComputing->container = R2;
270   tRComputingRFailure->container = R2;
271   tRComputingRSuspend->container = R2;
272   tRComputingRFinished->container = R2;
273
274   RegionMain->subvertex.push_back(RStart);
275   RegionMain->subvertex.push_back(RInitialization);
276   RegionMain->subvertex.push_back(RReady);
277   RegionMain->subvertex.push_back(RProcessing);
278   R0->subvertex.push_back(RStart0);
279   R0->subvertex.push_back(RIdleReport);
280   R0->subvertex.push_back(RReporting);
281   R1->subvertex.push_back(RStart1);
282   R1->subvertex.push_back(RIdleCheckpointing);
283   R1->subvertex.push_back(RCheckpointing);
284   R2->subvertex.push_back(RStart2);
285   R2->subvertex.push_back(RH2);
286   R2->subvertex.push_back(RReadCheckpoint);
287   R2->subvertex.push_back(RComputing);
288   RegionMain->subvertex.push_back(RPause);
289   RegionMain->subvertex.push_back(RTroubleshooting);
290   RegionMain->subvertex.push_back(RFinish);
291   RegionMain->subvertex.push_back(RAbort);
292
293   TriggerVector v;
294   v.push_back(evComputingCancel);
295   v.push_back(evComputingFailed);
296   v.push_back(evComputingFinished);
297   v.push_back(evComputingSuspend);
298
299   SpinhengeAtHomeStateMachine->region.push_back(RegionMain);
300
301   RCheckpointing->doActivity = new BehaviorRCheckpointing();
302   RComputing->doActivity = new BehaviorRComputing();
303   RIdleCheckpointing->doActivity = new BehaviorRIdleCheckpointing();
304   RIdleReport->doActivity = new BehaviorRIdleReport();
305   RInitialization->doActivity = new BehaviorRInitialization();
306   RPause->doActivity = new BehaviorRPause();
307   RReporting->doActivity = new BehaviorRReporting();
308   RTroubleshooting->doActivity = new BehaviorRTroubleshooting();
309
310   SpinhengeAtHomeStateMachine->process();
311
312   StateMachine *sm__ = SpinhengeAtHomeStateMachine;
313   bool terminated = false;
314   while(sm__->isRunning())
315   {
316     if(terminated) {
317       //std::cout << "Wait for termination..." << std::endl;
318       SLEEP(1); continue;
319     }
320   }
321   return 0;
322 }
```

Listing C.10: ...

C.4 Work Processing

Listing C.12 shows a use-case implementation for the creation and handling of series and workunits. This use-case is based on the Work package specifications (Section 6.8). The semantic-model used is shown in Listing C.11.

```
1  datatype String   "std::string";
2  datatype Boolean  "bool";
3  datatype Integer  "int";
4  datatype Float    "float";
5  datatype Binary   "void*";
6  datatype Any      "boost::variant<int, double, float, std::string, FileInfoBase
       *>";
7
8  FileType [ File, String, Integer, Double, Float ]
9
10 Application {
11   name               : String [1];
12   /* cut */
13   association series : Series [*];
14 }
15
16 WorkunitState {
17   state                : Integer;
18   maximum_sequence     : Integer = "1";
19   current_sequence     : Integer = "1";
20
21   check()              : String;
22
23   association workunit : Workunit [1];
24 }
25
26 InterfaceWorkunit {
27   create()      : Boolean;
28   cancel()      : Boolean;
29   information() : String;
30 }
31
32 Series  {
33   name                     : String;
34   deadline                 : Integer = "1";
35   activated                : Boolean = "false";
36   composition workunit     : Workunit [1..*];
37   association application : Application [1];
38   association ranges       : Range [0..*];
39 }
40
41 Range {
42   getValue()            : Any;
43   association series    : Series [*];
44   association datafield : Datafield [1];
45 }
46
47 RangeRule implements Range {
48   rule : String [1];
49 }
50
51 RangeSimple implements Range {
52   start : Float = "1.0f" [1];
53   stop  : Float = "1.0f" [1];
```

```
54    step  : Float = "1.0f" [1];
55  }
56
57  Workunit implements InterfaceWorkunit {
58    id                              : Integer;
59    rsc_fpops_est                   : Float;
60    rsc_fpops_bound                 : Float;
61    rsc_memory_bound                : Float;
62    rsc_disk_bound                  : Float;
63    delay_bound                     : Float;
64    min_quorum                      : Integer = "2";
65    target_nresults                 : Integer;
66    max_error_results               : Integer;
67    max_total_results               : Integer;
68    max_sucess_results              : Integer;
69
70    composition files               : Input [*];
71    composition results             : Output [*];
72
73    association series              : Series [1];
74    association workunitState       : WorkunitState [1];
75    association workunitAssociation : WorkunitAssociation [0..1];
76  }
77
78  WorkunitAssociation implements InterfaceWorkunit {
79    association workunit : Workunit [1];
80    association series    : Series [*];
81    association input     : Input [*];
82  }
83
84  InterfaceValidate  { }
85
86  InterfaceAssimilate {
87    association workunitAssociation : WorkunitAssociation [1..*];
88  }
89
90  InterfaceDataset {
91    open(   name : String) : Any;
92    exists( name : String) : Any;
93  }
94
95  Input implements InterfaceDataset {
96    name                  : String;
97    copyfile              : Boolean = "false";
98    sticky                : Boolean = "false";
99    nodelete              : Boolean = "true";
100   report_on_rpc         : Boolean = "false";
101   unique                : Boolean = "false";
102
103   composition datafield : Datafield [*];
104 }
105
106 Output implements InterfaceDataset {
107   name                  : String;
108   generated_locally     : Boolean = "true";
109   upload_when_present   : Boolean = "true";
110   max_nbytes            : Integer = "1024";
111   url                   : String;
112   copy_file             : Boolean = "false";
113   optional              : Boolean = "false";
114   no_validate           : Boolean = "false";
115   no_delete             : Boolean = "false";
```

```
116
117   composition datafield : Datafield [*];
118 }
119
120 Datafield {
121   name                  : String;
122   type                  : FileType;
123   data                  : Any;
124   optional              : Boolean = "false";
125
126   open()                : Any;
127   store(data : FileType) : Boolean;
128
129   association attributes : Datafield [0..*];
130   association fields     : Datafield [0..*];
131 }
```

Listing C.11: Excerpt of the semantic-model of the Work package (Section 6.8) which is used by the use-case implementation in Listing C.12.

```
1  // C++
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  #include <map>
6
7  // C, Linux
8  #include <unistd.h>
9
10 // libries
11 #include <ries/boinc/boinc.h>
12 #include <ries/string/string.h>
13 #include <ries/filesys/filesys.h>
14
15 // Generated
16 #include <Input.h>
17 #include <Output.h>
18 #include <Series.h>
19 #include <Project.h>
20 #include <RangeRule.h>
21 #include <Application.h>
22 #include <WorkunitAssociation.h>
23
24 // Ries
25 #include <wuFiles.h>
26 #include <boinc/boinc.h>
27 #include <lmboincOutput.h>
28 #include <lmboincWorkunit.h>
29 #include <lmboincDatafield.h>
30 #include <lmboincRangeRule.h>
31 #include <lmboincRangeSimple.h>
32 #include <InterfaceDataSource.h>
33
34 using namespace Generated;
35
36 void help_main_workunit() {
37  std::cout << "Help␣menu␣for␣mainLMBoincWorkunit\n";
38  std::cout << "++++++++++++++++++++++++++++++++\n";
39  std::cout << "␣—help␣|␣-h␣This␣help␣menu.\n";
40  std::cout << "␣—info␣␣␣WUID␣Shows␣information␣of␣one␣workunit.\n";
41  std::cout << "␣—cancel␣WUID␣(WUID)␣"
```

```
42            << " Cancels processing of one (or more between)\n"
43            << " specific workunit, both WUs are inclusive.\n";
44  exit(0);
45 }
46
47 int main_workunit( int argc, char **argv )
48 {
49  // Create temporary directory!
50  if(mkdir("/tmp/visualgrid", S_IRWXU|S_IRWXG|S_IROTH|S_IXOTH) >= 0) {
51   std::cout << "Temporary directory '/tmp/visualgrid' created!"
52             << std::endl;
53  }
54
55  int retval = config.parse_file(PROJECTDIR);
56  if(retval) {
57   std::cerr << ":C  Could not open BOINC's configuration"
58             << std::endl;
59   exit(1);
60  }
61  retval = boinc_db.open(config.db_name, config.db_host,
62                         config.db_user, config.db_passwd);
63  if(retval) {
64   std::cerr << ":C  Could not connect to database ("
65             << config.db_name << "@" << config.db_host << ") "
66             << std::endl;
67   exit(1);
68  }
69
70  for(int i=1; i<argc; i++) {
71   if(!strcmp(argv[i], "--help") || !(strcmp(argv[i], "-h"))) {
72    help_main_workunit();
73   }
74
75   if(!strcmp(argv[i], "--info")) {
76    if(argv[++i]) {
77     int wuid = atoi(argv[i]);
78     // lmboinc
79     LMBoincWorkunitState *ws = new LMBoincWorkunitState();
80     LMBoincWorkunit *w = new LMBoincWorkunit();
81     ws->setWorkunit(w);
82     w->setWorkunitState(ws);
83     w->setId(wuid);
84     std::cout << "State: " << w->workunitState()->check() << std::endl;
85     exit(0);
86    }
87   }
88
89   if(!strcmp(argv[i], "--cancel")) {
90    int start = -1;
91    int stop  = -1;
92
93    if(argv[++i]) start = atoi(argv[i]);
94    else {
95     std::cerr << ":C  Missing argument, one WUID is required!"
96               << std::endl;
97     help_main_workunit();
98    }
99    if(argv[++i]) stop = atoi(argv[i]);
100   if(stop <= 0) stop = start;
101
102      LMBoincWorkunit *w = new LMBoincWorkunit();
103      for( ; start <= stop; ++start) {
```

```
104         w->setId(start);
105         if(!w->cancel()) {
106           std::cerr << ":C  Cancelation of workunit(" << start << ")"
107                     << " failed." << std::endl;
108         } else {
109           std::cout << ":-) Cancelation of workunit(" << start << ")"
110                     << " successfully." << std::endl;
111         }
112       }
113       exit(0);
114     }
115   }
116
117   Project *p = new Project();
118   p->setName("LMBoinc@home");
119
120   Application *app = new Application();
121   app->setName("lmboinc");
122   p->addApplication(app);
123
124   LMBoincRangeRule *rr = new LMBoincRangeRule();
125
126   Series *series1 = new Series();
127   series1->setName("S1");
128   series1->setActivated(false);
129   series1->addRanges(rr);
130   series1->setApplication(app);
131   series1->setDeadline(time(NULL) + 24*3600);
132   rr->addSeries(series1);
133
134   Series *series2 = new Series();
135   series2->setName("S2");
136   series2->setActivated(false);
137   series2->addRanges(rr);
138   series2->setApplication(app);
139   series2->setDeadline(time(NULL) + 24*3600);
140   rr->addSeries(series2);
141
142   Series *series3 = new Series();
143   series3->setName("S3");
144   series3->setActivated(false);
145   series3->addRanges(rr);
146   series3->setApplication(app);
147   series3->setDeadline(time(NULL) + 24*3600);
148   rr->addSeries(series3);
149
150   Series *series4 = new Series();
151   series4->setName("S4");
152   series4->setActivated(false);
153   series4->addRanges(rr);
154   series4->setApplication(app);
155   series4->setDeadline(time(NULL) + 24*3600);
156   rr->addSeries(series4);
157
158   app->addSeries(series1);
159   app->addSeries(series2);
160   app->addSeries(series3);
161   app->addSeries(series4);
162
163   // Input: lmboinc.xml
164   LMBoincDatafield *datafieldIn00 = new LMBoincDatafield();
165   datafieldIn00->setName("configuration");
```

```
166   datafieldIn00->setType(FileType(FileType::String));
167   datafieldIn00->setData(std::string(""));
168
169   LMBoincDatafield *datafieldIn01 = new LMBoincDatafield();
170   datafieldIn01->setName("mode");
171   datafieldIn01->setType(FileType::String);
172   datafieldIn01->setData(std::string(""));
173   rr->setDatafield(datafieldIn01);
174
175   LMBoincDatafield *datafieldIn02 = new LMBoincDatafield();
176   datafieldIn02->setName("size");
177   datafieldIn02->setType(FileType::String);
178   datafieldIn02->setData(std::string("320x240"));
179
180   LMBoincDatafield *datafieldIn03 = new LMBoincDatafield();
181   datafieldIn03->setName("copyright");
182   datafieldIn03->setType(FileType::String);
183   datafieldIn03->setData(std::string("www.christianbenjaminries.de"));
184
185   LMBoincDatafield *datafieldIn04 = new LMBoincDatafield();
186   datafieldIn04->setName("type");
187   datafieldIn04->setType(FileType::String);
188   datafieldIn04->setData(std::string("RGBA"));
189
190   datafieldIn00->addFields(datafieldIn01);
191   datafieldIn00->addFields(datafieldIn02);
192   datafieldIn00->addFields(datafieldIn03);
193   datafieldIn00->addFields(datafieldIn04);
194
195   Input *input01 = new Input();
196   input01->setName("lmboinc.xml");
197   input01->setCopyfile(false);
198   input01->setSticky(false);
199   input01->setNodelete(false);
200   input01->setReport_on_rpc(false);
201   input01->setUnique(true);
202   input01->addDatafield(datafieldIn00);
203
204   // Input: archive_in.zip
205   LMBoincDatafield *datafieldIn05 = new LMBoincDatafield();
206   datafieldIn05->setName("data");
207   datafieldIn05->setType(FileType::File);
208   datafieldIn05->setData(std::string("/home/cr/inputfiles1/d400.zip"));
209
210   Input *input02 = new Input();
211   input02->setName("archive_in.zip");
212   input02->setCopyfile(false);
213   input02->setSticky(false);
214   input02->setNodelete(false);
215   input02->setReport_on_rpc(false);
216   input02->setUnique(false);
217   input02->addDatafield(datafieldIn05);
218
219   // Output: archive_out.zip
220   LMBoincDatafield *datafieldOut01 = new LMBoincDatafield();
221   datafieldOut01->setName("data");
222   datafieldOut01->setType(FileType::File);
223
224   Output *output01 = new Output();
225   output01->setName("archive_out.zip");
226   output01->setUrl("<UPLOAD_URL/>");
227   output01->setGenerated_locally(true);
```

```
228     output01->setUpload_when_present(true);
229     // 1e6 = 1MB, 1e7 = 10MB
230     output01->setMax_nbytes(1e9);
231     output01->setCopy_file(true);
232     output01->setOptional(false);
233     output01->setNo_validate(false);
234     output01->setNo_delete(false);
235     output01->addDatafield(datafieldOut01);
236
237     // lmboinc
238     LMBoincWorkunit *workSeries1 = new LMBoincWorkunit();
239     series1->addWorkunit(workSeries1);
240     workSeries1->setSeries(series1);
241     workSeries1->addFiles(input01);
242     workSeries1->addFiles(input02);
243     workSeries1->addResults(output01);
244     workSeries1->setRsc_fpops_est(1e10);
245     workSeries1->setRsc_fpops_bound(1e11);
246     workSeries1->setRsc_memory_bound(1e8);
247     workSeries1->setRsc_disk_bound(1e8);
248     workSeries1->setDelay_bound(7*24*60*60);
249     workSeries1->setMin_quorum(2);
250     workSeries1->setTarget_nresults(3);
251     workSeries1->setMax_error_results(1);
252     workSeries1->setMax_total_results(10);
253     workSeries1->setMax_sucess_results(10);
254
255     LMBoincWorkunit *workSeries2 = new LMBoincWorkunit();
256     series2->addWorkunit(workSeries2);
257     workSeries2->setSeries(series2);
258     workSeries2->addFiles(input01);
259     workSeries2->addFiles(input02);
260     workSeries2->addResults(output01);
261     workSeries2->setRsc_fpops_est(1e10);
262     workSeries2->setRsc_fpops_bound(1e11);
263     workSeries2->setRsc_memory_bound(1e8);
264     workSeries2->setRsc_disk_bound(1e8);
265     workSeries2->setDelay_bound(7*24*60*60);
266     workSeries2->setMin_quorum(2);
267     workSeries2->setTarget_nresults(3);
268     workSeries2->setMax_error_results(1);
269     workSeries2->setMax_total_results(10);
270     workSeries2->setMax_sucess_results(10);
271
272     LMBoincWorkunit *workSeries3 = new LMBoincWorkunit();
273     series3->addWorkunit(workSeries3);
274     workSeries3->setSeries(series3);
275     workSeries3->addFiles(input01);
276     workSeries3->addFiles(input02);
277     workSeries3->addResults(output01);
278     workSeries3->setRsc_fpops_est(1e10);
279     workSeries3->setRsc_fpops_bound(1e11);
280     workSeries3->setRsc_memory_bound(1e8);
281     workSeries3->setRsc_disk_bound(1e8);
282     workSeries3->setDelay_bound(7*24*60*60);
283     workSeries3->setMin_quorum(2);
284     workSeries3->setTarget_nresults(3);
285     workSeries3->setMax_error_results(1);
286     workSeries3->setMax_total_results(10);
287     workSeries3->setMax_sucess_results(10);
288
289     LMBoincWorkunit *workSeries4 = new LMBoincWorkunit();
```

```
290   series4->addWorkunit(workSeries4);
291   workSeries4->setSeries(series4);
292   workSeries4->addFiles(input01);
293   workSeries4->addFiles(input02);
294   workSeries4->addResults(output01);
295   workSeries4->setRsc_fpops_est(1e10);
296   workSeries4->setRsc_fpops_bound(1e11);
297   workSeries4->setRsc_memory_bound(1e8);
298   workSeries4->setRsc_disk_bound(1e8);
299   workSeries4->setDelay_bound(7*24*60*60);
300   workSeries4->setMin_quorum(2);
301   workSeries4->setTarget_nresults(3);
302   workSeries4->setMax_error_results(1);
303   workSeries4->setMax_total_results(10);
304   workSeries4->setMax_sucess_results(10);
305
306   Series *series5 = new Series();
307   series5->setName("S5");
308   series5->setActivated(false);
309   series5->setApplication(app);
310   series5->setDeadline(time(NULL) + 24*3600);
311
312   Generated::Input *s5in1 = new Generated::Input();
313   s5in1->setName("input1");
314   s5in1->setCopyfile(true);
315   s5in1->setSticky(false);
316   s5in1->setNodelete(false);
317   s5in1->setReport_on_rpc(true);
318   s5in1->setUnique(false);
319
320   Generated::Input *s5in2 = new Generated::Input();
321   s5in2->setName("input2");
322   s5in2->setCopyfile(true);
323   s5in2->setSticky(false);
324   s5in2->setNodelete(false);
325   s5in2->setReport_on_rpc(true);
326   s5in2->setUnique(false);
327
328   Generated::Input *s5in3 = new Generated::Input();
329   s5in3->setName("input3");
330   s5in3->setCopyfile(true);
331   s5in3->setSticky(false);
332   s5in3->setNodelete(false);
333   s5in3->setReport_on_rpc(true);
334   s5in3->setUnique(false);
335
336   Generated::Input *s5in4 = new Generated::Input();
337   s5in4->setName("input4");
338   s5in4->setCopyfile(true);
339   s5in4->setSticky(false);
340   s5in4->setNodelete(false);
341   s5in4->setReport_on_rpc(true);
342   s5in4->setUnique(false);
343
344   bool firstRun = true;
345   bool fifthWUCreated = false;
346
347   for(;;) {
348     if(firstRun) {
349       std::cout << "First␣run..." << std::endl;
350
351       series1->setActivated(true);
```

```
352       workSeries1->create();
353
354       series1->setActivated(false);
355       series2->setActivated(true);
356       workSeries2->create();
357
358       series2->setActivated(false);
359       series3->setActivated(true);
360       workSeries3->create();
361
362       series3->setActivated(false);
363       series4->setActivated(true);
364       workSeries4->create();
365
366       firstRun = false;
367     }
368
369     int series1NotFinished = 0;
370     for(auto it : series1->workunit()) {
371       LMBoincWorkunitState *s =
372         static_cast<LMBoincWorkunitState*>(it->workunitState());
373       if(s && s->check() != "FINISHED")
374         series1NotFinished++;
375     }
376
377     int series2NotFinished = 0;
378     for(auto it : series2->workunit()) {
379       LMBoincWorkunitState *s =
380         static_cast<LMBoincWorkunitState*>(it->workunitState());
381       if(s && s->check() != "FINISHED")
382         series2NotFinished++;
383     }
384
385     int series3NotFinished = 0;
386     for(auto it : series3->workunit()) {
387       LMBoincWorkunitState *s =
388         static_cast<LMBoincWorkunitState*>(it->workunitState());
389       if(s && s->check() != "FINISHED")
390         series3NotFinished++;
391     }
392
393     int series4NotFinished = 0;
394     for(auto it : series4->workunit()) {
395       LMBoincWorkunitState *s =
396         static_cast<LMBoincWorkunitState*>(it->workunitState());
397       if(s && s->check() != "FINISHED")
398         series4NotFinished++;
399     }
400 /*
401     std::cout << "Not finished for '" << series1->name() << "': "
402               << series1NotFinished << std::endl;
403     std::cout << "Not finished for '" << series2->name() << "': "
404               << series2NotFinished << std::endl;
405     std::cout << "Not finished for '" << series3->name() << "': "
406               << series3NotFinished << std::endl;
407     std::cout << "Not finished for '" << series4->name() << "': "
408               << series4NotFinished << std::endl;
409 */
410     if(series1NotFinished
411       +series2NotFinished
412       +series3NotFinished
413       +series4NotFinished == 0 && !fifthWUCreated
```

```
414     ) {
415       std::cout << "Create␣S5!" << std::endl;
416
417       // We know that we just have one workunit
418       // for each Serie, this makes queries easier.
419       // Otherwise, we have to iterate over all
420       // workunits of one series and have to merge
421       // them for this fifth serie.
422
423       std::stringstream squery;
424       // ***
425       LMBoincWorkunit *sw1 =
426         static_cast<LMBoincWorkunit*>(series1->workunit(0));
427       LMBoincOutputDB *odb1 = new LMBoincOutputDB();
428       squery.str("");
429       squery << "where␣wuid=" << sw1->id();
430       odb1->lookup(squery.str().c_str());
431
432       LMBoincDatafield *df1 = new LMBoincDatafield();
433       df1->setName("data");
434       df1->setType(FileType::File);
435       df1->setData(std::string(odb1->dbpath));
436       s5in1->addDatafield(df1);
437       // ***
438       LMBoincWorkunit *sw2 =
439         static_cast<LMBoincWorkunit*>(series2->workunit(0));
440       LMBoincOutputDB *odb2 = new LMBoincOutputDB();
441       squery.str("");
442       squery << "where␣wuid=" << sw2->id();
443       odb2->lookup(squery.str().c_str());
444
445       LMBoincDatafield *df2 = new LMBoincDatafield();
446       df2->setName("data");
447       df2->setType(FileType::File);
448       df2->setData(std::string(odb2->dbpath));
449       s5in2->addDatafield(df2);
450       // ***
451       LMBoincWorkunit *sw3 =
452         static_cast<LMBoincWorkunit*>(series3->workunit(0));
453       LMBoincOutputDB *odb3 = new LMBoincOutputDB();
454       squery.str("");
455       squery << "where␣wuid=" << sw3->id();
456       odb3->lookup(squery.str().c_str());
457
458       LMBoincDatafield *df3 = new LMBoincDatafield();
459       df3->setName("data");
460       df3->setType(FileType::File);
461       df3->setData(std::string(odb3->dbpath));
462       s5in3->addDatafield(df3);
463       // ***
464       LMBoincWorkunit *sw4 =
465         static_cast<LMBoincWorkunit*>(series4->workunit(0));
466       LMBoincOutputDB *odb4 = new LMBoincOutputDB();
467       squery.str("");
468       squery << "where␣wuid=" << sw4->id();
469       odb4->lookup(squery.str().c_str());
470
471       LMBoincDatafield *df4 = new LMBoincDatafield();
472       df4->setName("data");
473       df4->setType(FileType::File);
474       df4->setData(std::string(odb4->dbpath));
475       s5in4->addDatafield(df4);
```

```
476
477         // *** Create new workunit with four input files.
478
479         // ...lmboinc.xml with a different "mode":
480         datafieldIn01->setType(FileType::String);
481         datafieldIn01->setData(std::string("merge"));
482         // This datafield is embedded in "input01".
483
484         // ...archive_in.zip is used for a bash script.
485         LMBoincDatafield *datafieldDoImages = new LMBoincDatafield();
486         datafieldDoImages->setName("data");
487         datafieldDoImages->setType(FileType::File);
488         datafieldDoImages->setData(std::string(
489                                 "/home/cr/projects/lmboinc/test.xml"));
490         Input *input02 = new Input();
491         input02->setName("archive_in.zip");
492         input02->setCopyfile(false);
493         input02->setSticky(false);
494         input02->setNodelete(false);
495         input02->setReport_on_rpc(false);
496         input02->setUnique(false);
497         input02->addDatafield(datafieldDoImages);
498
499         LMBoincWorkunit *ws5 = new LMBoincWorkunit();
500         series5->addWorkunit(ws5);
501         ws5->setSeries(series5);
502         // lmboinc.xml with mode:=merge
503         ws5->addFiles(input01);
504         // archive_in.zip: just a dummy
505         ws5->addFiles(input02);
506         ws5->addFiles(s5in1);
507         ws5->addFiles(s5in2);
508         ws5->addFiles(s5in3);
509         ws5->addFiles(s5in4);
510         ws5->addResults(output01);
511         ws5->setRsc_fpops_est(1e10);
512         ws5->setRsc_fpops_bound(1e18);
513         ws5->setRsc_memory_bound(1e7);
514         ws5->setRsc_disk_bound(1e8);
515         ws5->setDelay_bound(7*24*60*60);
516         ws5->setMin_quorum(2);
517         ws5->setTarget_nresults(3);
518         ws5->setMax_error_results(1);
519         ws5->setMax_total_results(10);
520         ws5->setMax_sucess_results(10);
521
522         ws5->create();
523
524         fifthWUCreated = true;
525       }
526
527       //usleep(500);
528       sleep(1);
529     }
530 }
531
532 int main(int argc, char **argv) {
533   std::string appname = argv[0];
534   appname = Ries::filename(appname);
535   std::cout << "Start this application: " << appname << std::endl;
536   if(appname == "mainLMBoincWorkunit") {
537     return main_workunit(argc, argv);
```

```
538   } else
539   if(appname == "mainLMBoincValidator") {
540     return main_validate(argc, argv);
541   } else
542   if(appname == "mainLMBoincAssimilator") {
543     return Assimilator::main_assimilate(argc, argv);
544   } else {
545     std::cout << "This␣application␣is␣not␣mapped." << std::endl;
546     return 1;
547   }
548 }
```

Listing C.12: Use-Case for the workunit processing of *LMBoinc* [54, 186].

C.5 Infrastructure Deployment

The semantic-model used for the infrastructure deployment is included in Listing C.13. The generated class hierarchy of the case study in Chapter 9 is shown in Listing C.14.

```
1  /*
2   * Christian Benjamin Ries, March 2012
3   * http://www.christianbenjaminries.de
4   *
5   * class-infrastructure.vgsm (VisualGrid Semantic Model)
6   */
7
8  // Following operation is added to create
9  // polymorphic classes:
10 // __dummy() : Integer;
11
12 includes { "dummyHeader.h", "Operation.h" }
13
14 datatype String           "std::string";
15 datatype Boolean          "bool";
16 datatype Integer          "int";
17 datatype UnlimitedNatural "int";
18 datatype Float            "float";
19 datatype Binary           "void*";
20 datatype Any              "boost::variant<int, double, float, std::string>";
21
22 // E.g. one Linux or Unix
23 OperatingSystem [
24   Ubuntu = 1, UbuntuServer = 2
25 ] { /* empty */ }
26
27 // E.g. NFS, SAMBA
28 ShareType [
29     NFS, SAMBA
30 ] {
31   version : String [0..1];
32 }
33
34 // E.g. Feeder, Transitioner
35 ServiceType [
36   Webserver, Database, Task,
37   Feeder, Assimilator, Validator,
```

```
38    Transitioner, Scheduler
39  ]
40
41  TargetType   [ Windows, Linux, MacOSX ]
42  FileType     [ File, String, Integer, Double, Float ]
43
44  NamedElement {
45    name               : String         [1];
46    qualifiedName      : String         [1];
47    __dummy()          : Integer;
48    association owner : NamedElement  [1];
49  }
50
51  Package implements NamedElement {
52    association member : NamedElement [*];
53  }
54
55  Class implements NamedElement { }
56
57  Port implements NamedElement {
58    __dummy() : Integer;
59    association provided        : NamedElement [*];
60    association required        : NamedElement [*];
61  }
62
63  Component implements Class {
64    association packagedElement : NamedElement [*];
65  }
66
67  Association implements NamedElement {
68    __dummy() : Integer;
69    association source : NamedElement [1];
70    association target : NamedElement [1];
71  }
72
73  Connector implements Association {
74  }
75
76  Operation {
77    execcmd : String [1];
78    __dummy() : Integer;
79  }
80
81  //
82  // Profile Classes
83  //
84
85  // Chapter 7.3.38 ; Package (from Kernel)
86  Projects implements Package {
87    association projects      : Project [*];
88  }
89
90  // Chapter 7.3.38 ; Package (from Kernel)
91  Project implements Package {
92    name                      : String            [1];
93    description               : String            [1];
94    workingDirectory          : String            [0..1];
95    state                     : Boolean = "false" [1];
96    isResource                : Boolean           [1];
97
98    association application   : Application       [1..*];
99    association operations    : Operation         [*];
```

```
100 }
101
102 //
103 Host implements Component {
104   hostname                  : String           [1];
105   main                      : Boolean = "true" [1];
106   operatingSystem           : OperatingSystem  [1];
107   isResource                : Boolean          [1];
108   composition network       : NIC              [1..*];
109
110   association required : Share [*];
111   association provided : Share [*];
112 }
113
114 SAN implements Host { }
115
116 // Chapter 7.3.7 ; Class (from Kernel)
117 NIC implements Class {
118   device                    : String [1];
119   ipvalue                   : String [0..1];
120 }
121
122 // Chapter 9.3.12 ; Port (from Ports)
123 PortExport implements Port {
124   isProjects                : Boolean = "false" [0..1];
125   association exportTo      : Host              [*];
126 }
127
128 // Chapter 9.3.12 ; Port (from Ports)
129 PortImport implements Port { }
130
131 // Chapter 7.3.24 ; Interface (from Interfaces)
132 Share implements Class {
133   shareType                 : ShareType [1];
134   shareName                 : String    [0..1];
135   path                      : String    [1];
136   mountpoint                : String    [1];
137 }
138
139 // Chapter 8.3.1 ; Component (from BasicComponents)
140 Service implements Component
141 {
142   isResource                : Boolean          [1];
143
144   state                     : Boolean          [1];
145   serviceType               : ServiceType      [1];
146   command                   : Operation        [1];
147   outputFile                : String           [0..1];
148   debugMode                 : Integer = "2"    [0..1];
149
150 // Defined in a Task diagram.
151 //   // ServiceType::Task
152 //   period                    : String           [1];
153
154   // ServiceType::*
155   //mod                       : Integer          [2];
156   mod0                      : Integer          [1];
157   mod1                      : Integer          [1];
158   sleep                     : Integer          [1];
159   onepass                   : Boolean = "false" [1];
160
161   // ServiceType::Task (command:=file_deleter)
```

```
162   // ServiceType::Assimilator
163   // ServiceType::Validator
164   appname                    : String            [1];
165   apps                       : String            [1];
166
167   // ServiceType::Feeder
168   allapps                    : Boolean = "false" [1];
169   appids                     : Integer           [0..*];
170   randomOrder                : Boolean = "false" [1];
171   priorityOrder              : Boolean = "false" [1];
172   priorityOrderCreateTime    : Boolean = "false" [1];
173   purgeState                 : Boolean = "false" [1];
174   wmod                       : Integer           [2];
175   wmod0                      : Integer           [1];
176   wmod1                      : Integer           [1];
177
178   // ServiceType::Task (command:=file_deleter)
179   deleteAntiques             : Boolean           [1];
180   dontRetryError             : Boolean           [1];
181   inputOnly                  : Boolean           [1];
182   outputOnly                 : Boolean           [1];
183
184   // ServiceType::Task (command:=dbpurge)
185   ageDay                     : Integer           [0..1];
186   maxPurge                   : Integer           [0..1];
187   useZip                     : Boolean           [0..1];
188   useGZip                    : Boolean           [0..1];
189   noArchive                  : Boolean           [0..1];
190   maxWUperFile               : Integer           [0..1];
191
192   // ServiceType::Assimilator
193   updateDB                   : Boolean = "true"  [1];
194   insertDB                   : Boolean = "true"  [1];
195
196   // ServiceType::Validator
197   maxClaimedCredit           : Float             [0..1];
198   maxGrantedCredit           : Float             [0..1];
199   grantClaimedCredit         : Float             [0..1];
200   updateCreditJob            : Boolean           [0..1];
201   creditFromWU               : Boolean           [0..1];
202
203   // ServiceType::Webserver
204   instances                  : Integer = "5"     [1];
205   port                       : Integer = "80"    [1];
206   useSecure                  : Boolean = "false" [1];
207   isMaster                   : Boolean = "true"  [1];
208   isScheduler                : Boolean = "true"  [1];
209   isUpload                   : Boolean = "true"  [1];
210   isForum                    : Boolean = "true"  [1];
211   isDownload                 : Boolean = "true"  [1];
212
213   // Operations
214   start()                    : Boolean;
215   stop()                     : Boolean;
216   restart()                  : Boolean;
217
218   association required       : Share             [*];
219   association provided       : Share             [*];
220   association libraries      : Software          [*];
221 }
222
223 ServiceDynamic implements Service {
```

```
224   dynamic : Boolean = "false"                [1];
225   association hosts           : Host          [*];
226   association balancingRules  : LoadBalancer  [*];
227 }
228
229 // Chapter 7.3.7 ; Class (from Kernel)
230 LoadBalancer implements Class {
231   rules : Operation [*];
232 }
233
234 // Chapter 8.3.1 ; Component (from BasicComponents)
235 Database implements Service {
236   port        : Integer              [1];
237   isResource  : Boolean              [1];
238
239   association replication : ReplicationAssociation  [0..*];
240 }
241
242 DB implements Port {
243   isProjectDB : Boolean = "true" [1];
244   isScienceDB : Boolean = "true" [1];
245 }
246
247 // Chapter 7.3.24 ; Interface (from Interfaces)
248 Table implements Class {
249   name : String [1];
250 }
251
252 // Chapter 8.3.4 ; Connector (from BasicComponents)
253 // Actually this class is named "Connector", but this
254 // implementation failed when two classed named similar.
255 PortConnector implements Connector { }
256
257 // Chapter 7.3.3 ; Association (from Kernel)
258 ReplicationAssociation implements Association { }
259
260 // Chapter 8.3.1 ; Component (from BasicComponents)
261 Application implements Component
262 {
263   name        : String      [1];
264   description : String      [1];
265   targetType  : TargetType  [*];
266   isResource  : Boolean     [1];
267 }
268
269 // Chapter 7.3.34 ; NamedElement (from Kernel, Dependencies)
270 ConfigurationService implements NamedElement { }
271
272 Software {
273   fileName             : String    [0..1];
274   association software  : Software  [*];
275 }
```

Listing C.13: Excerpt of the semantic-model of the Infrastructure package (Section 6.5) which is used by the case study in Chapter 9.

```
1 #ifndef __GENERATE_VGIDE_INFRASTRUCTURE__
2 #define __GENERATE_VGIDE_INFRASTRUCTURE__
3 void generateInfrastructure(Project *ownerProject) {
4   Host *host1 = new Host();
5   host1->setOwner(ownerProject);
```

```
6    ownerProject->addMember(host1);
7    host1->setHostname("lmboinc-main");
8    host1->setOperatingSystem(1);
9    NIC *nic2 = new NIC();
10   nic2->setOwner(host1);
11   host1->addNetwork(nic2);
12   nic2->setDevice("eth0");
13   nic2->setIpvalue("192.168.1.9/24");
14   Database *db3 = new Database();
15   db3->setOwner(host1);
16   host1->addPackagedElement(db3);
17   db3->setName("MySQL-DB");
18   db3->setPort(3306);
19   db3->setState(false);
20   db3->setServiceType(11);
21   DB *dbContext4 = new DB();
22   dbContext4->setOwner(db3);
23   db3->addPackagedElement(dbContext4);
24   dbContext4->setIsProjectDB(true);
25   dbContext4->setIsScienceDB(false);
26   Service *service5 = new Service();
27   service5->setOwner(host1);
28   host1->addPackagedElement(service5);
29   service5->setName("Assimilator");
30   service5->setState(true);
31   service5->setServiceType(10);
32   service5->setSleep(5);
33   service5->setOnepass(false);
34   service5->setMod0(0);
35   service5->setMod1(0);
36   service5->setAppname("lmboinc");
37   PortImport *portImport5 = new PortImport();
38   service5->addPackagedElement(portImport5);
39   portImport5->setOwner(service5);
40   Share *shareImport7 = new Share();
41   shareImport7->setOwner(service5);
42   shareImport7->setShareName("dstValid");
43   shareImport7->setShareType(0);
44   shareImport7->setMountpoint("/mnt/lmboinc/valid");
45   service5->addRequired(shareImport7);
46   Share *shareImport8 = new Share();
47   shareImport8->setOwner(service5);
48   shareImport8->setShareName("dstFailed");
49   shareImport8->setShareType(0);
50   shareImport8->setMountpoint("/mnt/lmboinc/failed");
51   service5->addRequired(shareImport8);
52   Service *service9 = new Service();
53   service9->setOwner(host1);
54   host1->addPackagedElement(service9);
55   service9->setName("Feeder");
56   service9->setState(true);
57   service9->setServiceType(10);
58   service9->setDebugMode(2);
59   service9->setRandomOrder(false);
60   service9->setPriorityOrder(false);
61   service9->setPriorityOrderCreateTime(false);
62   service9->setAllapps(true);
63   service9->setWmod0(0);
64   service9->setWmod1(0);
65   service9->setSleep(5);
66   service9->setOnepass(false);
67   service9->setMod0(0);
```

```
68  service9->setMod1(0);
69  Service *service10 = new Service();
70  service10->setOwner(host1);
71  host1->addPackagedElement(service10);
72  service10->setName("Validator");
73  service10->setState(true);
74  service10->setServiceType(10);
75  service10->setMaxGrantedCredit(9.64289e-39);
76  service10->setUpdateCreditJob(1);
77  service10->setCreditFromWU(0);
78  service10->setGrantClaimedCredit(0);
79  service10->setMaxClaimedCredit(2.93891e-39);
80  service10->setSleep(5);
81  service10->setOnepass(false);
82  service10->setMod0(0);
83  service10->setMod1(0);
84  Service *service11 = new Service();
85  service11->setOwner(host1);
86  host1->addPackagedElement(service11);
87  service11->setName("Webserver");
88  service11->setState(true);
89  service11->setServiceType(10);
90  service11->setInstances(5);
91  service11->setPort(80);
92  service11->setUseSecure(false);
93  service11->setIsMaster(true);
94  service11->setIsScheduler(true);
95  service11->setIsDownload(true);
96  service11->setIsUpload(true);
97  service11->setIsForum(true);
98  Service *service12 = new Service();
99  service12->setOwner(host1);
100 host1->addPackagedElement(service12);
101 service12->setName("Transitioner");
102 service12->setState(true);
103 service12->setServiceType(10);
104 PortExport *portExport13 = new PortExport();
105 host1->addPackagedElement(portExport13);
106 portExport13->setOwner(host1);
107 Share *shareImport14 = new Share();
108 shareImport14->setOwner(portExport13);
109 shareImport14->setShareName("lmboinc");
110 shareImport14->setShareType(0);
111 shareImport14->setMountpoint("/home/boincadm/lmboinc");
112 portExport13->addRequired(shareImport14);
113 Host *host15 = new Host();
114 host15->setOwner(ownerProject);
115 ownerProject->addMember(host15);
116 host15->setHostname("lmboinc-tasks");
117 host15->setOperatingSystem(1);
118 NIC *nic16 = new NIC();
119 nic16->setOwner(host15);
120 host15->addNetwork(nic16);
121 nic16->setDevice("eth0");
122 nic16->setIpvalue("192.168.1.10/24");
123 Service *service17 = new Service();
124 service17->setOwner(host15);
125 host15->addPackagedElement(service17);
126 service17->setName("db_dump");
127 service17->setState(true);
128 service17->setServiceType(10);
129 Service *service18 = new Service();
```

```
130     service18->setOwner(host15);
131     host15->addPackagedElement(service18);
132     service18->setName("update_uotd");
133     service18->setState(true);
134     service18->setServiceType(10);
135     Service *service19 = new Service();
136     service19->setOwner(host15);
137     host15->addPackagedElement(service19);
138     service19->setName("update forum");
139     service19->setState(true);
140     service19->setServiceType(10);
141     Service *service20 = new Service();
142     service20->setOwner(host15);
143     host15->addPackagedElement(service20);
144     service20->setName("update_stats");
145     service20->setState(true);
146     service20->setServiceType(10);
147     Service *service21 = new Service();
148     service21->setOwner(host15);
149     host15->addPackagedElement(service21);
150     service21->setName("update profiles");
151     service21->setState(true);
152     service21->setServiceType(10);
153     Service *service22 = new Service();
154     service22->setOwner(host15);
155     host15->addPackagedElement(service22);
156     service22->setName("team import");
157     service22->setState(true);
158     service22->setServiceType(10);
159     Service *service23 = new Service();
160     service23->setOwner(host15);
161     host15->addPackagedElement(service23);
162     service23->setName("notify");
163     service23->setState(true);
164     service23->setServiceType(10);
165     PortImport *portImport23 = new PortImport();
166     host15->addPackagedElement(portImport23);
167     portImport23->setOwner(host15);
168     Share *shareImport25 = new Share();
169     shareImport25->setOwner(host15);
170     shareImport25->setShareName("lmboinc");
171     shareImport25->setShareType(0);
172     shareImport25->setMountpoint("/home/boincadm/lmboinc");
173     host15->addRequired(shareImport25);
174 }
175 #endif
```

Listing C.14: Class hierarchy for the deployment of the infrastructure of the case study (Chapter 9).

C.6 Generators

In this section excerpts of the code-generators are included. These are directly implemented in Visu@lGrid [184] and used to generate different parts of a BOINC project, e.g. configuration files. Listing C.15 contains a visitor-pattern [83] based approach for the creation of the configuration of different Task specifications.

```
/** @brief Queries the timing information and configuration of a specific
 *         task which has appName as reference.
 *  @param rootTiming The root of all timing information.
 *  @param appName Referenced application name, i.e. this application is
 *                 part of UMLDiagramTiming.
 *  @param period This parameter is used to store the period of the
 *                application execution, e.g. "7 days" or "6 hours".
 *  @return TRUE when application's execution is enabled, otherwise FALSE
 *          is returned.
 */
bool taskPeriod(
  UMLDiagramTiming *rootTiming,
  const QString & appName,
  QString & period
) {
  UMLTask *task = NULL;
  foreach(UMLTask *t, rootTiming->tasks) {
    if(!t) continue;
    if(t->references.count() < 1)
      continue;

    UMLReference *ref = t->references[0];
    if(ref->targetName() == appName) {
      task = t; break;
    }
  }

  if(!task)
    return false;

  UMLTaskFork *tfork = NULL;
  foreach(UMLEventFlow *eflow, rootTiming->connectors) {
    if(eflow->target->name() == task->name()) {
      // EventFlow is enabled, in this case the source UMLItem is used.
      if(eflow->state) {
        tfork = eflow->source;
        break;
      }
    }
  }

  if(!tfork)
    return false;

  UMLTimeForTask *tftask = NULL;
  foreach(UMLEventFlow *eflow, rootTiming->connectors) {
    if(eflow->target->name() == tfork->name()) {
      tftask = eflow->source;
      break;
    }
  }

  if(!tftask)
    return false;

  if(tftask->events.count() < 1)
    return false;

  /// TODO handle more than one time expressions
  UMLTimeExpression *expr = tftask->events[0]->expressions[0];
  if(!expr)
```

```
62      return false;
63
64    period = QString("%1 %2").arg(expr->timeValue()).arg(expr->unit());
65
66    return true;
67  }
68
69  QString generateTasks(
70    UMLDiagramInfrastructure *rootInfrastructure,
71    UMLDiagramApplication    *rootApplication,
72    UMLDiagramTiming         *rootTiming
73  ) {
74    QStringList out;
75    out << "<tasks>" << ENDL;
76
77    // Part of Infrastructure diagram
78    foreach(UMLHost *host, rootInfrastructure->hosts) {
79      if(!host) continue;
80      foreach(UMLService *service, host->services) {
81        if(!service) continue;
82        if(service->serviceType() == UMLService::ServiceTypeTask) {
83          // Task
84          if(service->references.count() < 1) continue;
85
86          UMLReference *ref = service->references[0];
87          if(!ref) continue;
88
89          foreach(UMLApplication *app, rootApplication->applications) {
90            if(!app) continue;
91            if(app->name() == ref->targetName()) {
92              UMLAction *taskCommand = NULL;
93              foreach(UMLAction *act, app->actions) {
94                if(act->actionType() == UMLAction::ActionTypeAction) {
95                  taskCommand = act;
96                  break;
97                }
98              }
99              if(taskCommand)
100             {
101               // Got command...
102
103               QString period;
104               bool taskEnabled = taskPeriod(rootTiming,
105                                             app->name(), period);
106
107               out << " <task>" << ENDL;
108               out << "  <cmd>" << taskCommand->implementation()
109                   << "</cmd>" << ENDL;
110               out << "  <output>" << service->outputFile()
111                   << "</output>" << ENDL;
112               out << "  <period>" << period << "</period>" << ENDL;
113               out << "  <host>" << host->name() << "</host>" << ENDL;
114               out << "  <disabled>" << (taskEnabled?"0":"1")
115                   << "</disabled>" << ENDL;
116               /// TODO always run
117               out << " </task>" << ENDL;
118             }
119           }
120         }
121       }
122     }
123   }
```

```
124
125   out << "</tasks>" << ENDL;
126
127   return out.join("");
128 }
```

Listing C.15: Excerpt of the code-generation functionalities of Visu@lGrid [184] for the creation of different Task specifications.

C.7 Processing asynchronous-messages

In Listing C.16 the structure for the asynchronous-message is given. Listing C.17 includes the server-side handler for asynchronous-messages with the variety "msg". It receives messages, queries information from the database, and relays the received message to specific hosts and users. Listing C.18 includes a scientific application mainly used as the handler for asynchronous-messages on the client-side.

```
1  struct CHATMSG {
2    std::string msgFrom;
3    std::string msgTo;
4    std::string message;
5    CHATMSG();
6
7    /** @return Error code: 0 (success), 1 (no data), 2 (wrong msg format)
8     */
9    int parse(std::string xmlmsg);
10   void show();
11 };
```

Listing C.16: Structure for the asynchronous-message; the filename is *chatMsg.h*

```
1  // BOINC
2  #include <util.h>
3  #include <boinc_db.h>
4  #include <str_util.h>
5  #include <error_numbers.h>
6
7  // BOINC Scheduler
8  #include <sched_config.h>
9  #include <sched_util.h>
10
11 // C++
12 #include <iostream>
13 #include <sstream>
14 #include <string>
15 #include <vector>
16
17 // Trickle-Handler for Chat@home
18 #include <chatMsg.h>
19
20 char variety[] = "msg";
21
22 /**
23  * @param mfh Information of one Trickle-Message from a host.
```

```
24  * @return If successfully 0 is returned, otherwise a higher value.
25  */
26 int handle_trickle(MSG_FROM_HOST& mfh) {
27   DB_HOST hostFrom;
28   DB_USER userFrom;
29   hostFrom.lookup_id(mfh.hostid);
30   userFrom.lookup_id(hostFrom.userid);
31
32   CHATMSG m;
33   switch(m.parse(mfh.xml)) {
34     case 1: /* Fehlerbehandlung */ break;
35     case 0: {
36       m.show();
37
38       DB_MSG_TO_HOST mth;
39       mth.clear();
40       mth.handled = false;
41       mth.create_time = time(0);
42
43       // iterate all host-ids
44       char q[1024] = {'\0'};
45       snprintf(q, 1024, "WHERE␣name='%s'", m.msgTo.c_str());
46       DB_USER userTo;
47       if(userTo.lookup(q)) {
48         // Could not find user.
49         /* Fehlerbehandlung, Nachricht zum Sender. */
50         fprintf(stderr, "Could␣not␣find␣user:␣%s\n", m.msgTo.c_str());
51         return 0;
52       }
53
54       DB_HOST userHost;
55       char qHost[1024] = {'\0'};
56       snprintf(qHost, 1024, "WHERE␣userid=%d␣ORDER␣BY␣userid␣ASC", userTo.id);
57       std::cerr << "qHost=" << qHost << std::endl;
58
59       while(userHost.enumerate(qHost) == 0) {
60         mth.hostid = userHost.id;
61
62         DB_RESULT userResult;
63         char qResult[1024] = {'\0'};
64         snprintf(qResult, 1024,
65                  "WHERE␣hostid=%d␣AND␣userid=%d␣"
66                      "AND␣received_time=0␣"
67                      "AND␣server_state=%d␣"
68                      "AND␣outcome=%d␣ORDER␣BY␣name␣ASC",
69                  userHost.id, userTo.id,
70                  RESULT_SERVER_STATE_IN_PROGRESS /* 4 */,
71                  RESULT_OUTCOME_INIT /* 0 */
72         );
73         std::cerr << "qResult=" << qResult << std::endl;
74
75         while(userResult.enumerate(qResult) == 0) {
76           // Check if result is in progress and not cancelled.
77           DB_WORKUNIT resultWU;
78           char qWU[64] = {'\0'};
79           snprintf(qWU, 64, "WHERE␣id=%d", userResult.workunitid);
80           std::cerr << "qWU=" << qWU << std::endl;
81           resultWU.lookup(qWU);
82           if(resultWU.error_mask != 0) {
83             continue;
84           }
85
```

```
86            sprintf(mth.variety, "%s", variety);
87            char msgOut[1024] = {'\0'};
88            snprintf(
89              msgOut, 1024,
90              "<trickle_down>\n"
91              "<result_name>%s</result_name>\n"
92              "<time>%d</time>\n"
93              "<from>%s</from>\n"
94              "<chat>%s</chat>\n"
95              "</trickle_down>\n",
96              userResult.name, (int)time(NULL),
97              userFrom.name, m.message.c_str())
98            ;
99            sprintf(mth.xml, "%s", msgOut);
100           std::cerr << "[begin]" << std::endl;
101           std::cerr << "mth.xml=" << msgOut << "[end]" << std::endl;
102
103           int retval = mth.insert();
104
105           if(retval) {
106             /* Fehlerbehandlung */
107             continue;
108           }
109         }
110       }
111       userHost.end_enumerate();
112     } break;
113   }
114   return 0;
115 }
116
117 /**
118  * Handles trickles messages when of is received.
119  * @return TRUE when one Trickle-Message found,
120  *         otherwise FALSE.
121  */
122 bool do_trickle_scan() {
123   bool found=false;
124
125   char buf[256] = {'\0'};
126   sprintf(buf, "where variety='%s' and handled=0", variety);
127
128   while (1) {
129     DB_MSG_FROM_HOST mfh;
130     int retval = mfh.enumerate(buf);
131
132     if(retval) {
133       if(retval != ERR_DB_NOT_FOUND) {
134         fprintf(stderr, "lost DB conn\n");
135         exit(1);
136       }
137       break;
138     }
139     retval = handle_trickle(mfh);
140     if(!retval) {
141       mfh.handled = true;
142       mfh.update();
143     }
144     found = true;
145   }
146   return found;
147 }
```

```
148
149 /**
150  * @param one_pass Defines if handler will exit after first run.
151  * @return 0 if anything worked perfect.
152  */
153 int main_loop(bool one_pass) {
154   bool did_something;
155
156   int retval = boinc_db.open(
157     config.db_name, config.db_host, config.db_user, config.db_passwd
158   );
159   if(retval) {
160     log_messages.printf(MSG_CRITICAL, "boinc_db.open␣failed:␣%d\n", retval);
161     exit(1);
162   }
163
164   while(1) {
165     check_stop_daemons();
166     did_something = do_trickle_scan();
167     if (!did_something) {
168       sleep(5);
169     }
170   }
171   return 0;
172 }
173
174 /**
175  */
176 void usage() {
177   // ...
178 }
179
180 int main(int argc, char** argv) {
181   int i, retval;
182   bool one_pass = false;
183
184   check_stop_daemons();
185
186   for (i=1; i<argc; i++) {
187     if (!strcmp(argv[i], "-d")) {
188       if (!argv[++i]) {
189         log_messages.printf(MSG_CRITICAL,
190           "%s␣requires␣an␣argument\n\n", argv[--i]);
191           usage(); exit(1);
192       }
193       int dl = atoi(argv[i]);
194       log_messages.set_debug_level(dl);
195       if (dl == 4) g_print_queries = true;
196     } else {
197       log_messages.printf(MSG_CRITICAL,
198         "unknown␣command␣line␣argument:␣%s\n\n", argv[i]);
199         usage(); exit(1);
200     }
201   }
202
203   retval = config.parse_file();
204   if (retval) {
205     log_messages.printf(MSG_CRITICAL,
206       "Can't␣parse␣config.xml:␣%s\n", boincerror(retval)
207     );
208     exit(1);
209   }
```

```
210
211   install_stop_signal_handler();
212   main_loop(one_pass);
213 }
```

Listing C.17: Use-Case for the asynchronous-message processing; the filename is *TrickleChatHandler.cpp*

```
1  // C++
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  // C
7  #include <stdio.h>
8  #include <errno.h>
9
10 // BOINC
11 #include <boinc_api.h>
12 #include <filesys.h>
13 #include <util.h>
14
15 // Trickle-Handler for Chat@home
16 #include <chatMsg.h>
17
18 int main(int argc, char **argv)
19 {
20   BOINC_OPTIONS options;
21   options.handle_trickle_ups = true;
22   options.handle_trickle_downs = true;
23
24   int returnValue = boinc_init_options(&options);
25   if (returnValue) {
26     std::cerr << "boinc_init returned " << returnValue << std::endl;
27     exit(returnValue);
28   }
29
30   boinc_touch_file("trickleChat");
31
32   while(1) {
33     char bName[64] = {'\0'};
34     char bMsg[512]  = {'\0'};
35
36     if(boinc_file_exists("msgChat")) {
37       FILE *f = boinc_fopen("msgChat", "r");
38       if(f) {
39         char *cp = NULL;
40         cp = fgets(bName, 64, f);
41         cp[strlen(cp)-1] = '\0';
42         cp = fgets(bMsg, 512, f);
43         cp[strlen(cp)-1] = '\0';
44         fclose(f);
45       }
46       boinc_delete_file("msgChat");
47     }
48
49     if(strlen(bName) > 0 && strlen(bMsg) > 0) {
50       char msgUp[1024] = {'\0'};
51       snprintf(
52         msgUp, 1024,
53         "<to>%s</to>\n<chat>%s</chat>\n",
```

```
54          bName, bMsg
55        );
56
57        int r = boinc_send_trickle_up((char*)"msg", msgUp);
58        if(r) {
59          fprintf(stderr, "Could not send trickle up message: %d\n", r);
60        }
61
62      }
63
64      char trickleFilename[32] = {'\0'};
65      int retval = boinc_receive_trickle_down(trickleFilename, 32);
66
67      std::string trickleMsg;
68      FILE *trickleFile = boinc_fopen(trickleFilename, "r");
69      if(trickleFile) {
70        char b[512] = {'\0'};
71        while(fgets(b, 512, trickleFile)) {
72          trickleMsg += b;
73        }
74        fclose(trickleFile);
75        boinc_delete_file(trickleFilename);
76      }
77
78      if(retval) {
79        CHATMSG m;
80        switch(m.parse(trickleMsg)) {
81          case 2:
82          case 1: /* Fehlerbehandlung */ break;
83          case 0: {
84            m.show();
85            FILE *f = boinc_fopen("msgChatIn", "w");
86            if(f) {
87              fprintf(f, "%s\n", m.msgFrom.c_str());
88              fprintf(f, "%s\n", m.message.c_str());
89              fclose(f);
90            } else {
91              fprintf(stderr, "Could not open msgChatIn file for writting.\n");
92              fprintf(stderr, "Got message(%s) from %s\n",
93                m.message.c_str(), m.msgFrom.c_str());
94            }
95          } break;
96        }
97      }
98
99      std::cerr.flush(),
100     boinc_sleep(1);
101   }
102
103   boinc_fraction_done(1);
104   boinc_finish(0);
105 }
```

Listing C.18: Use-Case for the asynchronous-message processing; the filename is *TrickleChatHandler.cpp*

APPENDIX D

Visu@lGrid

In this appendix Visu@lGrid's tree structure is presented. The several items are mapped to UML4BOINC's elements. In particular, the tree-view represents UML4BOINC's hierarchy, e.g. «Service» is associated by «Host» and in the tree-view «Service» is a child of «Host». In case, an association has to connect items, the associated is at the top context of its planned to used hierarchy-level, e.g. two «Host» items can be connected via «Share», and therefore «Connector» is a child of «Host» which can access the relevant contexts.

```
1  (D) RBAC
2    <<User>>
3    <<UserRemote>>
4    <<Controller>>    // TODO
5    <<Role>>
6    <<Permission>>
7    <<Resource>>
8      Reference (to <<UserOperation>>
           from Infrastructure)
9      Reference (to <<SystemOperation>>
           from Infrastructure)
10   Assignment
```

Listing D.1: RBAC's tree-view

```
1  (D) Timing
2    <<TimeForTask>>
3      <<TimeEvent>>
4        <<TimeExpression>>
5    <<TaskFork>>
6    <<Task>>
7      Reference (to <<
           ScientificApplication>> from
           Application)
8    <<EventFlow>>
```

Listing D.2: Timing's tree-view

```
1  (D) Events
2    <<TricklePartner>>
3    <<TrickleMessage>>
4      <<TrickleMessageValue>> (embedded
           in <<TrickleMessage>>)
```

Listing D.3: Event's tree-view

```
(D) Infrastructure
  <<Host>>
    <<Service>>
      <<PortImport>>
        <<Share>>
    <<Database>>
      <<DB>>
        <<Table>>
          <<Column>>
    <<Software>>
    <<PortExport>>
      <<Share>>
    <<PortImport>>
      <<Share>>
    <<NIC>>
  <<SAN>>
    <<PortExport>>
      <<Share>>
    <<NIC>>
  <<Connector>>
  <<ReplicationAssociation>>  // not
      implemented
```

Listing D.4: Infrastructure's tree-view

```
(D) Work
  <<Series>>
    <<Input>>
      <<Datafield>>
    <<Output>>
      <<Datafield>>
  <<WorkunitAssociation>>
```

Listing D.5: Work's tree-view

```
(D) Application
  <<ScientificApplication>>
    Action (initial)
    Action (action)
      input : InputPin
      output : InputPin
    Action (decision)
    Action (bar, horizontally)
    Action (bar, vertically)
    Action (send signal)
      input : InputPin
    Action (accept signal)
      output : InputPin
    Action (object node)
    Action (final)
    Action (flow end)
    Action (terminate)
    Transition
  Statemachine
    State (initial)
    State (state)
      Pseudostate (entryPoint)
        Reference (onEntry)
        Reference (onExit)
      Pseudostate (exitPoint)
        Reference (onEntry)
        Reference (onExit)
      Region
        Statemachine*
      Reference (doActivity)
    State (decision)
    State (bar, horizontally)
    State (bar, vertically)
    State (history deep)
    State (history shallow)
    State (final)
    State (flow end)
    State (terminate)
  <<Checkpoint>>
    <<Datafield>> (from Work)
  <<Exchanger>>
    <<Datafield>> (from Work)
```

Listing D.6: Application's tree-view

APPENDIX E

Papers

In this chapter all published papers are included.

Contents

Code Generation Approaches for an Automatic Transformation of the Unified Modeling Language to the Berkeley Open Infrastructure for Network Computing Framework

Christian Benjamin Ries and Vic Grout
Creative and Applied Research for the Digital Society (CARDS)
Glyndŵr University, United Kingdom
www.christianbenjaminries.de || *www.visualgrid.org*

***Abstract*—This paper describes the verification process of the UML4BOINC stereotypes and the semantic of the stereotypes. Several ways enable the verification and this paper presents three different ways: (i) specifications of Domain-specific Modeling Languages (DSMLs), (ii) the use of C++-Models, and (iii) the use of Visual-Models created with Visu@lGrid [12], [18]. As a consequence, specific code-generators for the transformation of these models are implemented into applicable parts for a Berkeley Open Infrastructure for Network Computing (BOINC) project [1]. As for the understanding of how the transformation is realised, a brief introduction about the language-recognition and the way how code-generators can be implemented by use of ANTLR (ANother Tool for Language Recognition) [11] is given. This paper does not cover all transformations because most of them can vary, i.e. they depend on the target language (e.g. C++) and how tool-vendors handle semantic-models.**

***Keywords*-DSML, BOINC, UML, ANTLR, AST**

I. Introduction

RUnning of a Berkeley Open Infrastructure for Network Computing (BOINC) project can be a very time-consuming task. Despite the fact that it is necessary to implement a scientific application (SAPP) [20] and to establish a fully operable server infrastructure [18], moreover it is necessary to describe how SAPP and all BOINC components handle computational jobs. A SAPP has be implemented by developers and for individual projects several implementations has to be repeated. By the help of UML (Unified Modeling Language) [23] one can cope this work by doing it with the help of a prominent visual programming language. The idea in this paper is to have UML extension (so-called stereotypes) and code-generator (CG), which have to support developers with the ability to generate all required implementation. BOINC is chosen as the first distribution framework because of the fact that some effort is spend in implementing an integrated development environment – so-called Visu@lGrid – which should allow the fully implementation based on a modeling approach of a BOINC framework. This research project has started in December 2009 and has been finished by a PhD thesis which covers a world new UML profile named UML4BOINC [14].

Section II gives a briefly introduction how code generation works. Section III describes how the CG methodology in this paper is supported towards to fulfil the task of supporting BOINC developers by describing. In Section IV the support of state machine of the UML version 2.4 is scoped. An abstraction of BOINC's SAPP and services is shown in the Sections V–X. In the last section a conclusion is given.

II. Code Generation

A. Approaches

The code-generation (CG) generator-backend is often realised by one of the following three approaches [4]:

- **Patterns:** This approach allows specifying a search pattern on the model graph. A specific output is generated for every match.
- **Templates:** As the name suggests, code-files are the basis in the target language. Expressions of a macro language are inserted or replacement patterns are used for specifying the generator instructions.
- **Tree-traversing:** This kind of code generation approach specifies a predetermined iteration path over the Abstract-syntax Tree (AST). Generation instructions are defined for every node type or leaf and executed when a node or leaf is passed.

These approaches can be combined. Accordingly, this paper applies a combination of the second and third approach, i.e. when the AST is created, the leafs of the AST are transformed into code-fragments and merged in existing template files. Fig. 1 shows the CG processing parts which are used; in particular seven parts supports the verification. Section III introduces the first part (1); the result of it allows the generation of C++-code which is used to create semantic models. Such a semantic model can be created in several ways: (3) with a Domain-specific Modeling Language (DSML) description, (4) as a direct C++ implementation based on the previous created semantic model and (5) as a visual description. These approaches end up in (6) and describe a BOINC project. The description is transformed in different parts which are relevant for BOINC, e.g. configurations, implementations of services and scientific applications or

the creation of workunits. For the steps between (1), (2), (6), and the creation of BOINC parts, the software library (7) *libries* is used [13].

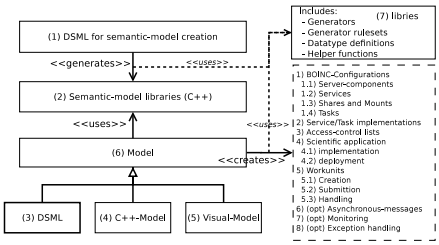

Figure 1. Architecture of the CG process; different models are used to generate a BOINC project with its used components and configurations.

B. ANother Tool for Language Recognition (ANTLR)

ANTLR [11] is a parser generator used to implement DSL interpreters, compilers and other translators, e.g. for the handling of specific configuration file formats. Two steps are necessary for language recognition: *lexical analysis* and *parsing* [8]. Fig. 2 shows the general language recognition process. An input stream as `1 + 21 => res` consists of different bytes. This stream can be filtered by a *lexer* which splits the bytes into belonging tokens, based on an user-defined syntax or grammar: a number is defined as an `INT` (integer), the plus sign as `ADD` (addition) and `=>` is used to assign (`ASSIGN`) 'something' to `res`, the underscores are whitespaces and are not recognized. The input stream is filtered and transformed into several tokens. Every token has its own meaning, e.g. `ADD` can be used to sum-up two integers. In addition, the chain of different tokens can have a different meaning (so-called semantic). The parser is used to interpret the tokens, i.e. either an optional AST can be created and analyzed; a direct interpretation is possible.

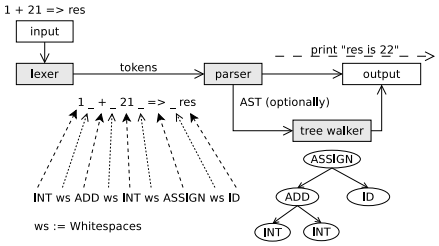

Figure 2. General process description of language recognition.

C. Abstract-syntax Tree (AST)

Generally, an AST is an abstract representation of any formatted input or token stream; it is optional, a direct interpretation can be done. The AST is created by adding mapping rules directly to the syntax of the new language:

```
multiplication: v1=INT '*' v2=INT->^(MUL $v1 $v2);
```

The operator `->` introduces the tree construction rules. The body of the rule is a list where the first element is the node type (`MUL`) and followed by the child nodes which are the factors for the multiplication in this case [5]. By means of this approach two syntax trees are defined, i.e. an input syntax on the left-hand side and an AST-syntax on the right-hand side.

D. ANTLR and the Extended Backus-Naur Form (EBNF)

The EBNF (Extended Backus-Naur Form) is a syntactic metalanguage which is a notation for defining the syntax of a language through a set of rules. Each rule names a part of the language (called a non-terminal symbol of a language) and then defines its possible forms. A terminal symbol of a language is an atom that cannot be split into smaller components of the language [7]. The EBNF has a small set of syntax rules and ANTLR applies most of the EBNFs with modifications as shown in Table I.

	EBNF	ANTLR
'a' zero or once	[a]	a?
'a' zero or more	{a}	a*
'a' once or more	a {a}	a+
'a' or 'b'	a | b & a | b	
('a' or 'b') and 'c'	(a | b) c & (a | b) c	

Table I
EXCERPT OF THE EBNF'S AND ANTLR'S SYNTAX [11], [7].

Fig. 2 shows a small summation (s1) of two integer values and an assignment to the variable *res*. The second summation (s2) allows unlimited summations, e.g. `20 + 22 + 222 => res`. Listing 1 states the EBNF-syntax of this small command. Any token is defined in the last five lines, i.e. integers only consist of numbers. The last line defines whitespaces (WSs) and a special command forwards these characters to a hidden channel.

```
s1: INT ADD INT ASSIGN ID ;
s2: (INT (ADD INT)* ASSIGN ID)+ ;

ADD:    '+';
ASSIGN: '=>';
INT:    '0'..'9'+;
ID: ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
WS: (' '|'\n'|'\r'|'\t')+ {$channel=HIDDEN;};
```

Listing 1. EBNF-syntax for the addition of two integers and assignment.

Finally, the ANTLR syntax enables the direct AST creation. Listing 2 shows how the AST on the bottom right-hand side of Fig. 2 can be described. The ANTLR operator `->` is used

to map the EBNF-syntax to a valid AST-syntax. The result of this mapping and its dataset is shown in Fig. 3.

```
summation: leaf* -> leaf* ;
leaf: INT ('+' INT)* '=>' ID ->
    ^(ASSIGN ^(ADD INT*) ID);
```

Listing 2. Definition of the AST of the summation.

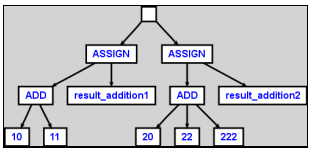

Figure 3. AST for the summation of unlimited integer values.

E. Result

Fig. 2 shows the parser during the language-recognition and with two outgoing transitions: (1) a direct transition to output and (2) a detour of AST creation are subsequently finished by an output generation. The first approach is also called “embedded translation” because it populates the semantic model from the embedding code to the parser; it implements semantic model at appropriate points in the parse [5]. The use of an additional AST can be more explicit and easier to maintain [10].

III. Generation of the Semantic Models

The research for this paper required the implementation of different approaches, in order to verify and modify models in an iterative way. All models are based on an object-oriented hierarchy; the creation is supported by a self-defined DSML and allows a compact specifying of all necessary information: (i) datatype specifications, (ii) creation of enumerations, (iii) classes with attributes and methods, (iv) derivations of classes and (v) different kinds of associations between classes and their different multiplicities, i.e. compositions and aggregations.

Listing 3 shows an example of the created DSML. Line 1 specifies additional header files. Lines 2-3 indicate a mapping of used datatypes within the DSML which are therefore mapped in the target’s programming language datatypes. Line 4 specifies an enumeration usable as a datatype. The last lines specify three classes: *Project*, *Host* and *NIC*. *Project* has one attribute *name* and one association to one or more *Host* instances. *Host* itself is a composition of one or more *NIC* instances. The CG produces setter and getter methods and routines automatically in order to check the multiplicities during the adherence of instances to associations or compositions. Finally, every class gets a factory and destroyer functionality [6].

```
includes { "General.h" }
datatype String  "std::string";
datatype Integer "int";
OperatingSystem[Ubuntu=1, UbuntuServer=2] { }
Project { name : String [1];
          association hosts : Host [1..*];  }
Host    { composition network : NIC [1..*];
          // association : NIC [1..*];       }
NIC     { device : String [1];
          ipvalue : String [0..1];          }
```

Listing 3. DSML description for the creation of a class-hierarchy.

IV. Generation of Statemachines

UML’s StateMachine package has been used for the research of this paper. There are some state machine implementations for the combination with C++ [2], [3], [9] which are partly based on the UML specification. All of them are restricted in their use (e.g. no support for orthogonal regions, no entry and exit points). Concerning this, the *UML 2 Statemachine for C++* (UML2STM4CPP) [21] and UML’s StateMachine in version 2.4 support have been created and applied. UML2STM4CPP’s DSL *Ries Statemachine* (RSM) enables the creation of hierarchy state machine constructs. Fig. 4 shows a small state machine which is based on an initialisation state, choice state, two system states, and a final state. Listing 4 shows an excerpt of the DSL definition for this state machine.

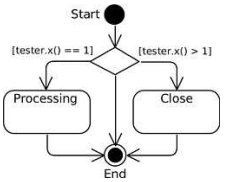

Figure 4. UML Pseudostates with guarded and default transition.

Listing 4 shows the DSL definition for the state machine in Fig. 4. The state machine has two areas: (1) `States{...}` defines states and regions, and (2) `Transitions{...}` defines transitions, guards and events between states and regions. The DSL defines all used nodes, e.g. decision node, initial node, action node, and final node. Finally, the DSL is an equipment with extra code anchors:

- `<[onEntry CODE-]>` is executed on entering.
- `<[-CODE-]>` is the main part.
- `<[onExit CODE-]>` is executed on state’s leaving.

The combination of these three anchors with the first anchor in lines 7-9 allows fulfilling the state machine with arbitrary behaviour.

```
Anchors <[-- #include <Tester.h>
            Tester tester; int x;--]>
```

```
Statemachine Pseudostates {
  States {
    Initial Start
    Simple Processing
      <[onEntry /* do some action */ ---]>
      <[---      tester.processing(); ---]>
      <[onExit  /* do some action */ ---]>
    Simple Close
    Final End
  }
  Transitions {
     R: Start [tester.x() == 1]
              / tester.setX(2) == Processing
     R: Start [tester.x() > 1]
              / tester.setX(3) == Close
     R: Start == End
     R: Processing == End
     R: Close / tester.show()   == End
  }
}
```

Listing 4. A UML2STM4CPP example of the state machine in Fig. 4.

V. Scientific Application

BOINC offers an Application-programming Interface (API) for the procedural implementation of scientific applications. UML is mainly an object-orientated specification and modeling approach. Generally, some tasks have to be done whenever a new scientific application (SAPP) has to be implemented. In order to avoid these incidents, an Object-orientated Programming (OOP) abstraction is implemented and set up on the top of BOINC's API [15], [20]. Fig. 5 shows the abstraction layers of this approach. The OOP layer provides a top-down approach where most of BOINC's functionalities are merged in objects and used as building blocks, e.g. BOINC's checkpoint mechanism is abstracted by the class `Ries::BOINC::MVC::Checkpoint` [13]. The biggest issue of a SAPP is the computation's core

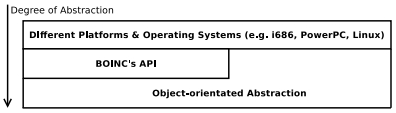

Figure 5. Object-orientated Programming layer for BOINC.

which provides the "intelligence" and creates engineering or scientific results. Fig. 6 shows an excerpt of the OOP approach. In this case, two scientific applications are given: (i) *LMBoinc* [19] and (ii) *Spinhenge* [22]. In both cases, only the `doWork` method has to be implemented. The computation is called within *Main* and responsible for the instantiation of the BOINC framework and the OOP layers. *Main* is instantiated by BOINC's client and has access to the `MVC::Handler` which provides information about the computation state.

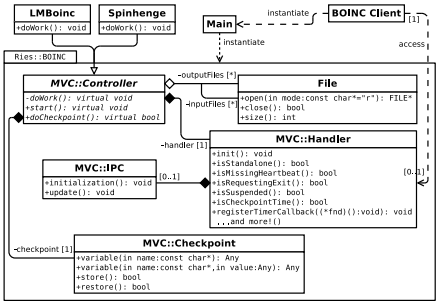

Figure 6. Object-orientated Abstraction of BOINC's API.

A. State Machine of a Scientific Application

The fundamental idea of the state machine for the SAPP is to reduce the complexity based on the fact that only relevant functions are used for creating various state nodes, e.g. a state "Initialization" is used for initializing BOINC's runtime environment or "Computing" is used for processing the core computation; thus, it will call `doWork()`. SAPP's state machine possesses three Regions:

- **R1** Is used for the handling of asynchronous-messages,
- **R2** is used for the checkpointing process, and
- **R2** specifies the computation of the SAPP.

The Transitions of the processing states and the three regions can be triggered by external events and are able to call actions when they are used.

B. Mapping of Activities and Actions

Most of UML4BOINC's stereotypes can be mapped directly to BOINC's API-calls or with less code-complexity. The current section illustrates how the CG from UML to implementation code can be done.

1) «Atomic»: Atomic functions cannot be interrupted and have to be finished before subsequent actions or activities are executed. Listing 5 shows a small code-snippet. Line 2 opens and line 5 closes the atomic area; thus, any action or activity is called and cannot be interrupted between these calls. Fig. 7 shows a UML model consisting of an atomic area. It executes an activity with an initialization and final node and then executes the "transferMoney()" action. Exceptions cannot interrupt the execution.

```
/// Start: <<Atomic>>
boinc_begin_critical_section();
  transferMoney(); /*...*/
boinc_end_critical_section();
/// End: <<Atomic>>
```

Listing 5. Code-implementation example of BOINC's atomic mechanism.

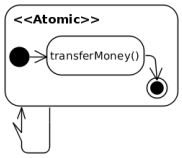

Figure 7. Code-mapping for UML4BOINC's «Atomic».

2) «Action» & type=#FractionDone: This type of «Action» is used in order to inform the BOINC client how much work has been accomplished. It is a value between zero and one, i.e. a percentage descriptor. The mapping of «Action» in BOINC's API is a single line: boinc_fraction_done (fdone). *fdone* can be an input pin and requires a floating-point value (PrimitiveType::Real [23]) which has to be calculated by the scientific application developer.

3) «Action» & type=#FileInfo: It enables querying information of specific files. Listing 6 implements «FileInformation» directly as a C++-class, namely tag-values as attributes and the operations as methods. «Action» is mapped directly: (a) the input pin parameter *path* is used as `query`'s parameter, and (b) the output pin *finfo* as a new variable which assigns the returning value of `query`.

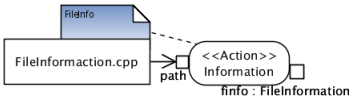

Figure 8. «Action» to query file information.

```
class FileInformation {
public:
 enum FileKind {Unknown=0, File=1,
                Directory=2, Symlink=3};
 static FileKind type;
 static std::string cwd, absolutePath;
 static double filesize, dirsize;
 static double totalSpace, freeSpace;

 static FileInformation query(const char *path) {
  FileInformation fi;
  if(is_file(path))      type=FileKind::File;
  else if(is_dir(path)) type=FileKind::Directory;
  else if(is_symlink(path))
                        type=FileKind::Symlink;
  else                  type=Unknown;
  char buffer[4096] = {'\0'};
  boinc_getcwd(buffer); cwd = buffer;
  relative_to_absolute(path, buffer);
  absolutePath = buffer;
  file_size(path, filesize);
  dir_size(path, dirsize);
  get_filesystem_info(totalSpace, freeSpace);
  return fi;
 }
};
```

```
/* ...initialise default values! */
int main(int argc, char **argv) {
 FileInformation finfo =
    FileInformation::query("FileInformation.cpp");
 /*...*/
}
```

Listing 6. «FileInformation» and «Action» for file information queries.

4) «Action» & type=#Locking: BOINC provides a functionality of preventing access to specific files, i.e. the BOINC-structure `FILE_LOCK` is used for locking and unlocking a file. Every file needs its own locking instance. Fig. 9 shows «Action» with one input pin to lock/unlock the file "FileInformation.cpp". The state of the lock/unlock-call is stored in the output pin *state* after the execution. When something goes wrong, an (exception named) *ExceptionLocking* which can be caught and handled individually, will be raised. Every locking mechanism is only usable in its context. When a global lock has to be used for specific files, it is necessary to define its lock instance in a global way.

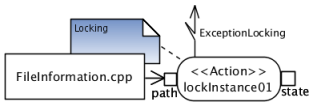

Figure 9. «Action» for locking/unlocking files.

5) «Action» & type=#FileHandling: Fig. 10 and Fig. 11 show how UML modeling can be realized when UML4BOINC's «Action» and *type=#FileHandling* are used. «Action»'s characteristic is specified by the input pin *mode*; it states how many input pins can exist in one action. The notes in Fig. 10 and Fig. 11 describe the modes with one and two additional input pins. These actions and individual characteristics can be directly transformed into C++-implementations. Table II shows the related mapping of *mode*'s value to the suitable BOINC API-call. This is a trivial approach and some modes can need more specific implementation, e.g. the mode `Open` can be used for virtual filenames or/and for filenames with immediate access. In case the filename is a virtual name, it is necessary to resolve the physical filename. Listing 7 shows a general approach for file opening. Various problems can arise when the file is in a specific file format, e.g. it is a ZIP-archive and the opening differs from mentioned approaches. Opening approaches can be handled directly by the CG or implemented in the OOP abstraction layer. In this regard, CG is mostly a tool-specific task and can be handled in different ways.

```
std::string inputDataIn;
boinc_resolve_filename_s(path, inputDataIn);
int res = boinc_fopen(inputDataIn, filemode);
```

Listing 7. General code-implementation for file opening.

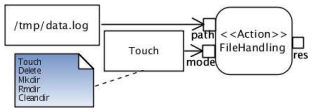

Figure 10. «Action» for file handling with two parameters.

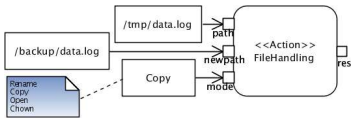

Figure 11. «Action» for file handling with three parameters.

Mode	BOINC's Function Call
one additional input pin	
Touch	int res = boinc_touch_file (path);
Delete	int res = boinc_delete_file (path);
Mkdir	int res = boinc_mkdir(path);
Rmdir	int res = boinc_rmdir(path);
Cleandir	int res = clean_out_dir (path);
two additional input pins	
Copy	int res = boinc_copy(path, newpath);
Chown	int res = boinc_chown(path, owner);
Rename	int res = boinc_rename(path, newpath);
Open	FILE *res = boinc_fopen(path, filemode);

Table II
MAPPING OF UML TO BOINC'S API.

6) «Action» & type=#TrickleDown: Asynchronous-messages are not handled immediately. The SAPP is responsible for collecting all messages. The collection process can be done in the first region of the state machine. The messages are provided by the output pin of «Action». Listing 8 shows the receiving of trickle-messages and a suggestion for handling them.

```
char trickleFilename[32] = {'\0'};
int retval = boinc_receive_trickle_down(
    trickleFilename, 32);
FILE *trickleFile = boinc_fopen(trickleFilename, "
    r");
if(trickleFile) {
 /// (1) read in the trickle-message content
 /// (2) parse the content
 /// (3) create a specific TrickleMessage instance
 ///     and fill this instance with content of
     the message
}
```

Listing 8. Approach of how the trickle-message handlers can be implemented on the client-side.

7) «TrickleUp»: Fig. 12 shows UML4BOINC's stereotype «TrickleUp» for sending messages from every host to a BOINC project. Generally, this stereotype can only be used by clients. Listing 9 shows an example of how the stereotype can be implemented. A vector container for storing string values is defined and provides the planned content of the message which is created in lines 5-10. The sending itself is done in line 11. The stereotype does not provide an output pin for handling the return value of the sending function. Thus, it is checked in order to enable error treatment. However, the reception of messages and their handling on the server-side require more work. Thus, the content of the message must be parsed and handled individually.

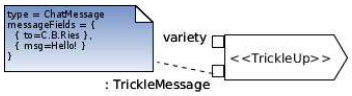

Figure 12. Code-mapping for UML4BOINC's «TrickleUp».

```
std::vector<std::string> messageFields;
/* ... fill messageFields with data! */
if(messageFields.size() >= 3) {
 char msgUp[1024] = {'\0'};
 snprintf(msgUp, 1024,
  "<to>%s</to>\n<from>%s</from>\n<msg>%s</msg>\n",
  messageFields[0],
  messageFields[1],
  messageFields[2]
}
int ret = boinc_send_trickle_up(
            (char*)variety, msgUp);
if(ret) { /* Error-handling, ... */ }
```

Listing 9. Trickle-message handling on the client-side.

Listing 10 shows how the set-up for activating message handling has to be done by programming when on client-side. The handling can be activated during the initialization of the SAPP, i.e. in the state "Initialization". Section X describes the server-side.

```
int main(int argc, char **argv) {
 BOINC_OPTIONS options;
 options.handle_trickle_ups = true;
 options.handle_trickle_downs = true;
 int returnValue = boinc_init_options(&options);
 ...
}
```

Listing 10. Enabling of BOINC's asynchronous-messages on clients.

8) «Timing»: Periodically executed function-calls can be implemented in a SAPP, e.g. to calculate an average value or to handle input-/output calls. BOINC provides a mechanism which enables adding of specific functions. Accordingly, a function can be used as callback-function by BOINC's internal. UML4BOINC specifies «Timing» in order to add several different function calls as shown in Fig. 13. Only «Timing» has to be added to BOINC's callback mechanism. Listing 11 shows how the generated code can look like. The first three lines are dummy implementations and can be replaced by more complex action flows. The last execution is compared to its individual delay value (Line 5) by `timingFunction`. When the current time possesses less than the last `timingMoments`

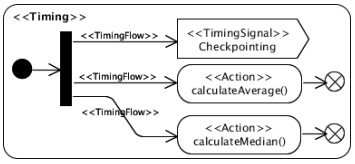

Figure 13. Code-mapping for UML4BOINC's «Timing».

value plus the `timingDelays` value, then the action is executed. This implementation approach does not allow executing the three functions in an orthogonal way. Finally, the `timingFunction` has to be registered for execution within BOINC's runtime environment and this is done in line 29.

```
void Checkpointing()     { /*...*/ }
void calculateAverage() { /*...*/ }
void calculateMedia()    { /*...*/ }

int timingDelays[]  { 15, 5, 30 };
int timingMoments[] {  0, 0,  0 };
FUNC_PTR timingFunctions[] {
 Checkpointing, calculateAverage, calculateMedia
};

void timingFunction() {
 static bool firstRun = true;
 if(firstRun) {
  for(int i=0; i<3; i++)
   timingMoments[i] = time(NULL);
  firstRun = false;
 }
 for(int i=0; i<3; i++) {
 int timeEdge = timingMoments[i]+timingDelays[i];
  if(timeEdge > time(NULL))
   continue;
   (timingFunctions[i])();
   timingMoments[i] = time(NULL);
 }
}

int main(int argc, char **argv) {
 boinc_init();
 boinc_register_timer_callback(timingFunction);

 // Keeps the application running...
 for(int i=0; i<600; i++) sleep(1);
}
```

Listing 11. «Timing» & «TimingFlow» and their embedded actions.

9) «WaitForNetwork»: Fig. 14 shows the two outgoing transitions. Listing 12 shows an approach of how «WaitForNetwork» can be implemented. The structural feature is directly created in lines 2-7. The true-guarded transition is implemented by lines 12-13, and the false-guarded by lines 15-16.

```
/// Start: <<WaitForNetwork>>
#define TRIES 10
boinc_need_network();
bool netavailable = false;
for(int _try=0; _try < TRIES; _try++) {
 netavailable = (boinc_network_poll()?true:false);
}
/// End: <<WaitForNetwork>>

if(netavailable) {
 /// Transition: [true]
 std::cout << " NETWORK" << std::endl;
} else {
 /// Transition: [false]
 std::cout << " NO NETWORK" << std::endl;
}
```

Listing 12. «WaitForNetwork» to query network connectivity.

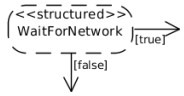

Figure 14. «WaitForNetwork» to query network connectivity.

VI. Services

Services are only present on the server-side of a BOINC project; it needs information about the individually services: firstly, the location of the executability, its parameters for starting, and whether it is enabled or disabled; secondly, the application itself has to be implemented or a pre-implemented version has to be installed. The services are modeled within the (a) Infrastructure and (b) Application diagrams; in case the service is a task the (c) Timing diagram is also relevant. This part is covered by additional work [17].

VII. Work

BOINC's work processing is based on a small state-chain, namely *start* → *creation* → *processing* → *validation* → *assimilation* → *end* [16]. Fig. 15 shows how series and workunits can be modeled by a visual approach provided by Visu@lGrid [12], [18]. Different series can be created within Work branch **(1)**. Every series contains a direct description of the used workunit, the input and output files. Any file can have individual datafields to set-up the runtime parameter for the computation. The visualization **(2)** shows the series with input/output files. *images.zip* is a physical file which can be added manually or dynamically in this case and by the use of «InterfaceDataSource». **(3)** and **(4)** show how the relation between the first four series and a fifth series can be described. A detailed explanation can be found in additional work [16].

VIII. Work Validation

Workunit specifications state on the one hand how output files have to be opened and on the other hand how the «InterfaceValidate» can be used. It provides access to the output files for validation processes. Thus, the interface «InterfaceDataset» provides an universal way to open these output files. BOINC's validation framework exists on

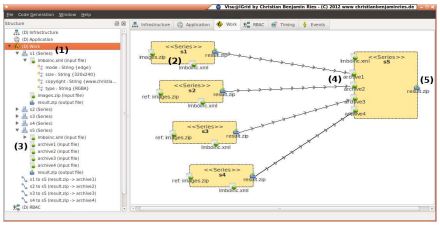

Figure 15. Visual modeling for the specification of BOINC workunits; The Series and the relation among themselves can be modeled with associations between input and output files.

a high-level and low-level abstraction [15, Section 9.3]. This paper covers only the high-level abstraction which is based on three steps: (i) *initialization*, (ii) *comparing*, and (iii) *cleanup*. Only the second step has to be filled with functionality for a successful validation. As shown in Listing 13, the result has to be stored in the fifth parameter: (a) *false* or (b) *true*, i.e. the computational result is *not valid* or *is valid*. The first & second and third & fourth parameters are used to handle the result files and result raw datasets. Fig. **??** shows how the Listing 13 can be modeled in UML by use of a UML Activity. The parameters of the function `compare_results` are modeled as UML ParameterNodes. The comparing statement in line 4 is transformed into a UML DecisionNode and two outgoing transitions with equipment guards. These guards are the real validation parts and decide whether the computational results are valid or not, i.e. the output UML ParameterNode is valued with *false* or *true*. This approach can be added to the UML StateMachine and extended by «Validation».

```
int compare_results(RESULT & r1, void *data1,
                    RESULT & r2, void *data2,
                    bool & match) {
 match = (r1.cpu_time >= 10 && r2.cpu_time >= 10);
 return 0;
}
```

Listing 13. Part of BOINC's high-level validation framework.

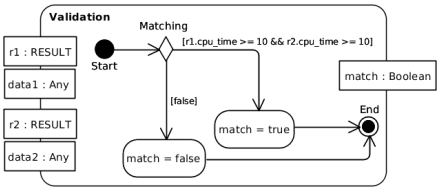

Figure 16. Part of BOINC's high-level validation framework in UML.

IX. Work Assimilation

The «InterfaceAssimilate» can be used for the assimilation of computational results. It provides access to output files and then enables interface «InterfaceDataset» to open the output files. BOINC's assimilation framework provides a function which has to be filled with assimilation routines, i.e. by use of the standard implementation or an own specification and how output files and results have to be stored. Generally, BOINC's framework provides two targets for storing: (i) storing in a specific file-directory and (ii) storing in a database. The destination of storing is specified within the CG process and by the Infrastructure diagram. The infrastructure specification contains an «Assimilator» which can be extended by ports in order to use file-directories or database tables for storing.

X. Asynchronous Messages

This section gives an example of how asynchronous-messages can be transformed into code and by use of the relevant UML4BOINC stereotypes on server-side. When asynchronous-messages are used on both sides, the message handling must be enabled in any case on server- and client-side. On the server-side a specific XML configuration has to be set.

A. Server-Side Handling

BOINC provides a default implementation for the handling of asynchronous-messages (AMs), i.e. a kind of ping-pong system. This exemplary application iterates over a BOINC database table and queries unhandled AMs. The queried messages are processed by a specific function and therefore named `handled_trickle(DB_MSG_FROM_HOST&)`. The parameter is the database entry and contains information about the sender and the message send which is text-based. Fig. 17 shows how an AM is specified by UML4BOINC. «TrickleMessage» specifies a chat message and contains one message value which is defined by three additional datafields: (i) *to*, (ii) *from*, and (iii) *msg*. This chat message has one receiver: *C.B.Ries*. The specification is not always available during runtime. Chat messages are transmitted primarily from users; the specification does not possess any association to receivers at this moment, only textual information about the planned receivers. The association is created on server-side when message receivers are queried from the user database of a BOINC project. Fig. 18 shows this detail through another viewpoint. One user transmits the «TrickleMessage» *1: chatMessage01* to the BOINC project. The message contains only a description of the targeting message's receivers. The «TrickleMessage» *1.1: chatMessage02* has a different format. *messageFields* has modified the information into an absolute information of the targeting receiver. The replacement of the targeting receiver information needs to be handled on the server-side. Additional UML Actions can be specified for this case; it is

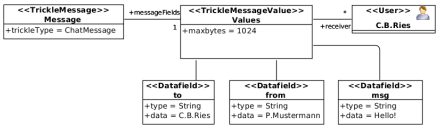

Figure 17. Instantiation of the stereotypes for asynchronous-messages.

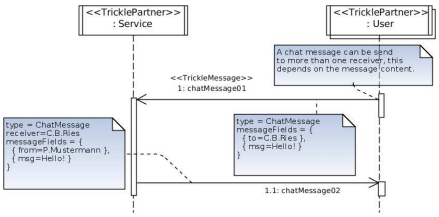

Figure 18. Asynchronous-messages within a BOINC project.

not done in this research. Listing 14 shows the pseudocode of the relaying mechanism for chat messages on the server-side. Information of the received message is filtered: (i) the sender host, (ii) the sending user, and (iii) the names of the target users. The received message is parsed into the format for the targeting users in the next step. Thus, the message is send to all hosts of an user and accordingly to every workunit. As a matter of fact, BOINC's AMs need the information about the workunit and which message has to be handled. When a message shall target a specific user, the sender can not foresee when and where it will be read by the receiver. According to that, the server sends it to all workunits.

```
handle_trickle(MSG_FROM_HOST& mfh) {
 hostFrom = HOST(mfh)
 userFrom = USER(mfh)
 usersTo  = USERS(mfh)

 m = MAP_RECEIVED_MSG_TO_SEND_MSG(mfh)

 do user = Send_Message_to_Users(usersTo)
  do workunit = Send_Message_to_Workunit(user)
   SEND_MESSAGE(workunit, m)
  done
 done
}
```

Listing 14. Pseudocode for the trasnfer of chat messages between users.

Listing 14 shows how little UML Action is needed in order to implement the pseudocode. It requires five actions and additional structural activities, i.e. a for-loop or do-while loop can be used to iterate over all datasets. Fig. 19 shows how this can be realized through UML. The received message is specified as a UML ObjectNode and used as input value for the three UML Actions: (i) HOST, (ii) USER and (iii) USERS. The first two are not used in this example. The third action is used in order to filter all users and query the information from the database. The user information is parsed to the first iterative element which consequently iterates over all users. The individual users are parsed to the second iterative; it enables an iteration over all workunits of the specific users. Finally, a pre-parsed message is send to a specific user's workunit. The handling of AMs is

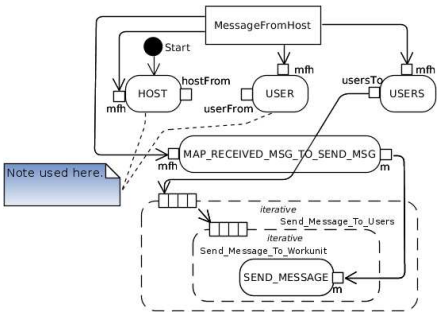

Figure 19. UML modeling for asynchronous-messages on the server-side.

less complex on the client- than on the server-side. The client only receives and transmits messages from and to a BOINC project. It must not query any other information from a database. The Transmission of AMs is provided by «TrickleUp» and the reception is supplied by «Action».

XI. Conclusion

This paper presented a CG approach for the transformation of UML4BOINC's stereotypes into an implementation. At first, different CG approaches have been introduced followed by examples of language recognition with ANTLR. Subsequently, the direct transformation of models into executable code demonstrates how an AST can be used for CG. In addition, it has been exemplified how a semantic model can be generated automatically through a specific DSML. Accordingly a DSML named RSM has been introduced; it enables the creation of a state machine which is based on a scientific application. Secondly, an object-orientated abstraction of BOINC's API has been presented briefly. However, the CG transformation is not based on strict rules. Developers can decide how they implement its BOINC parts and scientific applications.

References

[1] D.P.Anderson, "BOINC: A system for public-resource computing and storage," in *Grid Computing 2004 Proceedings Fifth IEEEACM International Workshop* (R. Buyya, ed.), pp. 4-10, IEEE Computer Soc, 2004.

[2] A. H. Dönni, “The Boost Statechart Library,” http://www.boost.org/doc/libs/1_46_0/libs/statechart/doc/-index.html [Online. Last accessed: 10th December 2012], 2007.

[3] C. J. Henry, “Meta State Machine (MSM),” http://www.boost.org/doc/libs/1_49_0/libs/msm/doc/HTML/-index.html [Online. Last accessed: 23rd November 2012], 2010.

[4] M. Feikas, “How to represent Models, Languages and Transformations,” *System*, 2006.

[5] M. Fowler, *Domain-Specific Languages*, vol. 5658 of Lecture Notes in Computer Science. Addison-Wesley Professional, 2010.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissies, *Design Patterns*, vol. 47 of *Addison Wesley Professional Computing Series*. Addison Wesley, 1995.

[7] “Information technology – Syntactic metalanguage – Extended BNF.” 1996

[8] T. A. Morgensen, “Basics of Compiler Design Extended edition,” *Analysis*, 2008

[9] I. A. Niaz, *Automatic Code Generation From UML Class and Statechart Diagrams. Dissertation*, University of Tsukuba, 2005

[10] T. Parr, “Translators should use tree grammars,” tech rep., University of San Francisco, San Francisco, CA, 2004.

[11] T. Parr, *The Definitive ANTLR Reference – Building Domain-Specific Languages*. Pragmatic Programmers, ISBN: 978-0978739256, 2007.

[12] C. B. Ries, “Visu@lGrid – Integrated Development Environment for BOINC,” http://visualgrid.sourceforge.net [Online. Last accessed: 2012]

[13] C. B. Ries, “libries – Research library for Visu@lGrid,” http://libries.sourceforge.net [Online. Last accessed: 2012]

[14] C. B. Ries, "‘A Modelling Language Approach for the Development of Distributed Applications based on the Berkeley Open Infrastructure for Network Computing,"’ Ph.D. dissertation, Glyndŵr University, Wrexham (Wales, UK), 2013

[15] C. B. Ries. “BOINC - Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing.” Berlin Heidelberg: Springer-Verlag, 2012

[16] C. B. Ries, C. Schröder, and V.Grout. “Model-based Generation of Workunits, Computation Sequences, Series and Service Interfaces for BOINC based Projects,” in *Proc. SERP’12 (Worldcomp)*, Las Vegas (NV) 2012

[17] C. B. Ries, C. Schröder, and V. Grout. “Approach of a UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. ICCAIE*, 2011, pp. 483-488

[18] C. B. Ries, C. Schröder, and V. Grout. “Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. SEIN*, 2011, pp. 67-76

[19] C. B. Ries and C. Schröder. “Public Resource Computing mit Boinc.” Linux-Magazin, vol. 3, pp. 106-110, March 2011. Internet: lmboinc.sourceforge.net

[20] C. B. Ries, T. Hilbig, and C. Schröder. “A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework,” in *Proc. IEEE-IMCSIT*, 2010, pp. 663-670

[21] C. B. Ries, “UML 2 Statemachine for C++,” https://sourceforge.net/projects/uml2stm4cpp/ [Online. Last accessed: 2013]

[22] “Spinhenge@home.” http://spin.fh-bielefeld.de [Online. Last accessed: 15th December 2012]

[23] Object Management Group. “OMG Unified Modeling Language (OMG UML) Superstructure.” formal/2010-05-05, May, 2010.

Model-based Generation of Workunits, Computation Sequences, Series and Service Interfaces for BOINC based Projects

Christian Benjamin Ries
Computational Materials Science and Engineering (CMSE)
University of Applied Sciences
Bielefeld, Germany
www.visualgrid.org

Christian Schröder
Computational Materials Science and Engineering (CMSE)
University of Applied Sciences
Bielefeld, Germany
Christian.Schroeder@fh-bielefeld.de

Vic Grout
Creative and Applied Research for the Digital Society (CARDS)
Glyndŵr University, United Kingdom
v.grout@glyndwr.ac.uk

***Abstract*—Berkeley Open Infrastructure for Network Computing (BOINC) is a popular Grid Computing (GC) framework which allows the creation of high performance computing installations by means of Public Resource Computing (PRC). With BOINC's help one can solve large scale and complex computational problems. A fundamental element of BOINC is its so-called workunits (WUs), each computer works on its own WUs independently from each other and sends back its result to BOINC's project server. Handling of WUs is a challenging process: (1) the order of used input files is important, (2) even more contributory components has to know how these input files are structured and on which data format are they based for an accurate WU processing. Small modifications can have a high impact to a BOINC project. Indeed scientific applications, BOINC's components, and third-party applications all have to be adjusted to have a correctly running project with desired the functionality. This can be a highly error-prone and time-consuming task. In this paper we present a Unified Modeling Language (UML) model to give a high abstraction for BOINC's WU handling. Only a model description and a corresponding code-generator are necessary to construct a WU handling infrastructure with less development and implementation effort: (a) one model to fit most WU cases and (b) essential interfaces for WU access.**

***Keywords*—BOINC, Code Generation, Modelling, UML, Work**

I. Introduction

SEt-up of a Berkeley Open Infrastructure for Network Computing (BOINC) project can be a challenging and sophisticating task. Despite the fact that it is necessary to implement a scientific application (SAPP) [10] and to establish a fully operable server infrastructure [7], moreover it is necessary to describe how SAPP and all BOINC components handle computational jobs. Here, participating clients retrieve a project specific SAPP from a BOINC project (BP) server along with so-called workunits (WUs), i.e. a number of parameter usually provided in data files of ASCII or binary format that are optionally needed by the application to perform specific tasks. The idea in this paper is to have a Unified Modeling Language (UML) model and code-generation (CG) facilities, which have to support developers with the ability to generate all required WU configurations, interfaces for opening and accessing WUs, and creating one or more computational series and sequences, i.e. different computational jobs with varied runtime configurations.

This project is funded by the German Federal Ministry of Education and Research.

A. Unified Modeling Language & Object Constraint Language

One of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability [15]. Since version 2.2, UML has 14 different diagram types subdivided in three categories: (1) structure diagrams, (2) behavior diagrams, and (3) interaction diagrams. In this paper we use the *Class* and *State Machine* diagrams. Class diagrams are used to specify system related elements, e.g. a class can describe a SAPP. An instantiated class element is seen as an object and mostly it is an executable instance. UML state machines help to model discrete behavior through finite state-transitions systems. It can be used to visualize the current state of one system, and orthogonal regions allow to model client-server state-machines where each side is working independently. The Object Constraint Language (OCL) is used to express constraints and properties of UML model elements [16].

B. BOINC's Workunit System

BOINC uses a fine-grained file based system to set-up WUs for a BOINC project (BP). WUs are packages with descriptions of input and output data needed by the SAPP to perform specific tasks [1]. Before a WU can be added to a BP, it is necessary to create several input files with planned to use datasets for one computation. Two additional template files are required: (1) an input template to describe which files are used as input, how they are ordered and which flags for them are set, and (2) a result template to describe how output files must be named by the SAPP, or how big in bytes they can be [3].

C. Research Topics

To make the handling of WUs easier some questions arise and we will work on them within this paper.

- Which UML elements are necessary to create a model for WU creation?
- How can we model a sequential queue for WU progressing? The answer to this question should make it possible to have WUs with the need of pre-processed results by one or more other WUs.
- How can BOINC's validator and assimilator access result's data on a higher abstraction level? In addition, is it possible to have only one interface or description which makes it possible to allow access by all BOINC components? Here, BOINC's validator is responsible for validating a WU and developers of a BP can implement their own validator routines. The default behavior of BOINC's assimilator is storing of results within a file system.
- How can we track the lifetime of WUs when they are used in different scenarios, e.g. one WU is used within a sequential performed queue?

This paper can be seen as the conjunction of previous work [6], in which BOINC's services are described with UML to be deployable on server farms. This is why «Application» is added in Fig. 1 where previous work is followed in this paper.

The remainder of this paper is organised as follows. Section II describes the problematic of BOINC's architecture to handle different defined WUs for varied kinds of computations. Next, Section III proposes our idea as to how we can fill the gap of BOINC's problematic to utilize it with an easier and less vulnerable interface. In Section IV we use our UML model and apply it to a small case-study. Finally, Section V concludes this paper and Section VI suggests future work.

II. Problematic of BOINC's Architecture

WUs are packages with descriptions of input and output data [1]. These WUs are fundamental pieces for BOINC and contain information on how these data are defined and formatted, i.e. binary data or plain text and functionalities to describe how several data items can be used. The flexibility to define arbitrary structured WUs and input files can be a complex issue. It has been shown, that WUs within a BP are crucial elements and are essential for the BP success [9], [11]. At the time a BP is being established it must be defined how all BOINC components have to handle WUs, otherwise WUs will stop immediately wrongly configured and, as a result, without proper working components. A BOINC administrator needs answers to several questions before a BP can be set-up as a fully operable system. Certainly we think about our computational concern and how we can solve this problem firstly. In this paper we will not discuss this difficulty, previous work has focused on this field of activity [5], [10]. In this paper we will discuss a solution for the following questions:

- Is all information about WU's structure available at the beginning of it's use or are they gathered continuously during BOINC's runtime? Here, it is also important to define how continuously created WUs differ from each other. It is necessary to know if their content differs and if they must be restructured or not, e.g. if different sigma values have to be set for statistical computations.
- How should WUs be opened and how should all potentially contained sub-elements be handled by a SAPP? Are the nested data defined as plain-text, or as encrypted text, or maybe a binary format?
- It is not only the WU input files that are important. The result files are also essential for the success of a computation. In the later BOINC process they must be validated and subsequently stored by an assimilator to make results usable for particular later cases.
- The assimilation process is used to store results, but what if one WU does not have enough results? E.g. one WU is distributed to three hosts, a minimum of two results must be returned but in one scenario two hosts are too late — deadline is reached — and only one result is available. In this case, BOINC's transitioner will flag the missing results as *overdue*, then directly flagged as *ready for assimilation* by BOINC's validator [2] and after this the assimilation process could create a duplicated WU for a retry. This can be done periodically until the WU is completely returned and successfully validated, or after some failed tries the available results can be stored in a database or on the file system which can be defined for failed results.
- Under some circumstances a computation relies on different sets of runtime parameters or they must adhere to a sequence of different runs, i.e. a result of a WU must be used as input for another WUs. In this case, the results must be converted to the right format of a new WU and it can be necessary to modify mentioned attributes for the different purposes of a WU, e.g. an unit conversion can be required before a WU result is usable for subsequent computations.

BOINC's architecture relies heavily on a fragile methodology; if one or more software components are misconfigured or disabled the WU handling chain will be stopped on the failed element, i.e. if the validator is not working properly no validation of returned WU results is executed and as a consequence the WU will never complete.

BOINC's WU consists of two template files, additional input files and, during computation, created output files. Template files are based on an XML [12] format and therefore they are not really human readable and XML-tags can be misspelled very easily. More important is the fact that all input files must be described within this template file and must have a specific order. In the header of the input template the numbering of input files is defined. After this part each file has optional attributes, e.g. a file is sticky and will not be deleted after one computation on one host. A similar approach is used for the description of result files. These files and the part of BOINC's framework for WU creation are elementary and every BOINC

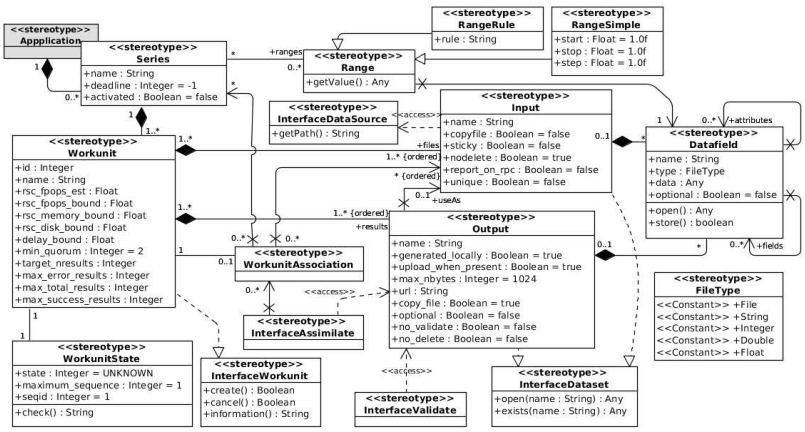

Fig. 1. Unified Modeling Language (UML) class diagram to abstract workunit structures. In the top area different *Ranges* for values within *Input*'s *Datafields* can be assigned. On the right hand-side *Datafield* allows to configure an arbitrary format for input and output files, the enumeration *FileTypes* provides different standard formats. In case one *Datafield* has *#File* as value for *type* no associated *attributes* or *fields* are allowed. Three stereotyped interfaces helps to access input and output files: (1) «InterfaceDataset», (2) «InterfaceValidate», and (3) «InterfaceAssimilate». Associations enable one to set-up different workunit processing scenarios: (1) static processing, (2) continuous processing, and (3) dynamic processing as seen in Fig. 2.

administrator or developer must give attention to this process. Several steps are required to add WUs for one BP: (1) one or more input files must be copied to BOINC's download hierarchy, (2) mentioned template files must be created, and (3) all input files must be arranged in the right order when BOINC's functionalities for WU creation are called. Each change within one of these steps has an impact on the other steps and must be adapted.

III. Modeling of Work Packages for BOINC

For computations with BOINC it is necessary to have one or more WUs which contain descriptive information on how to execute these computations. WUs could contain several files, e.g. additional configurations, data sets, definitions of algorithms and arbitrary extra files. Fig. 1 shows one part of our UML definition for WU definition. Here, we can define WU's input and output files and multiple data fields for these files.

The three stereotypes «Workunit», «Input», and «Output» are directly based on BOINC's WU system. All tag-values of these three stereotypes are directly mapped to attributes of BOINC's templates, the only exception is tag-value *unique*. If *unique* is *true* all input files are renamed to be unique within a BP. The other presented stereotypes are extensions to fulfil our UML model.

«Workunit» must be associated by «Series» and that must be associated to «Application» [6]. This association unites previous work with this paper.

A. Input-/Output Files and Datafields

«Input» is used to describe input files and «Output» describes result files. A WU can own several file instances and each of them can have distinct «Datafields». «Datafields» are used to describe data for input files, the data format is not restricted and for this reason two methods are defined: (1) *open()* is used to access data, and (2) *store()* is used to add datasets. The reason for these methods is, that the embedded data can have different formats, i.e. values have to be encrypted during saving or specific embedded function calls must be used during data access in case a file is packed as a ZIP-archive [18]. These functions can set by a developer to supply special opening and storing methods for currently unknown data types. There is no reason to allow a «Datafield» to be used by «Input» and «Output» at the same time, as a consequence only one owner of the root «Datafield» is allowed. This root and all other instances of «Datafield» have two associations which can be used to create tree structures with several pieces of information for a WU embedded in a «Input» file. With this methodology different structures are possible, e.g. a XML structure can be created as shown in Listing 1. The use of these associations is restricted, if one «Datafield» is associated by *attributes*, then it can not have additional associations. Each «Datafield» has the tag-values *name*, *type*, *data*, and *optional*. *Name* must be user-defined at any time when a «Datafield» is used, the other tag-values are optional and their use depends on the task. Listing 1 shows the use of the first three tag-values:

name *person*, *interests*, and *topics* are names,

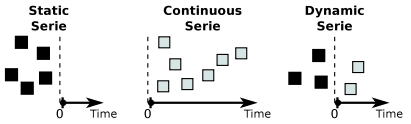

Fig. 2. Our UML model allows us to define «Series» in three different ways: (1) all WUs must be available before a BP is started, (2) during runtime WUs are created and added continuously, and (3) a mix of the first and second; some WUs are available at the beginning and during runtime additional WUs are added to one BP.

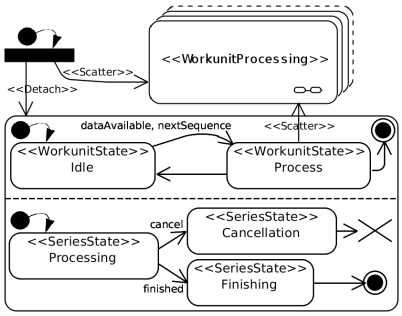

Fig. 3. First part of our UML statechart diagram to monitor instances of «Workunit» and «Series». State's top region is responsible for WU monitoring and creates new WUs when data is available or next WU in a sequence has to be performed. The bottom region monitors a «Series» and handles canceling events for a «Series» instance or if its finished and results can be merged.

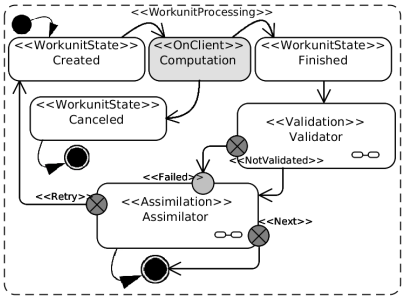

Fig. 4. Second part of our UML statechart diagram for WUs. During computation of one WU, clients can decide to cancel current WU, and therefore it has a changed WU state. When it is finished it will be validated, if this validation failed the exit pseudostate is used. The followed assimilation state can decide to retry this WU and a new WU is created with same «Input» values. If this WU is in a sequence, "Next" is used otherwise the statechart is finished.

type *"C.B.Ries"* and *"Research, Sport"* are of the enumeration type *FileType::String*, and
data mentioned string values are the real embedded information.

```
<person name="C.B.Ries">
  <interests topics="Research, Sport"/>
</person>
```

Listing 1. Example of «Datafield» usage to define a XML structure.

B. Rule-based Creation of Datafield Values

During WU creation data fields of input files or the input files themselves can be specified by «Range». Therefore values can be generated by «Range» specializations: (a) «RangeRule» and (b) «RangeSimple». With «RangeRule» a rule-set for value creation can be defined. For this purpose the tag-value *rule* can be filled with a user-defined rule, e.g. each WU within a specific «Series» can have a corresponding mode for algorithms. «Range» defines an operation *getValue()* which is used to query the related «Datafield» value. As shown in Fig. 1 each rule can only modify one «Datafield». «RangeSimple» is used to have a range-loop for one specific «Datafield», for this reason three tag-values are defined: (1) *start*, (2) *stop*, and (3) *step*. In combination with different additional rules each call of *getValue()* can increment the lower-bound value *start* by *step* to the upper-bound *stop*. One «Range» can be owned by several «Series», as a consequence it must be possible to retrieve which «Series» is calling *getValue()*. For this reason tag-value *activated* is specified. When this tag-value is valued by *true*, the association between «Range» and «Series» can be used to query the currently used «Series». This allows «Range» to access all information of a «Series» with associated «Workunits».

C. Continuous Creation of Workunits for Series

As seen in Fig. 2 a BP can have different scenarios for WU creation:

- **Static** All WUs are created before a «Series» will be created. Only these known WUs are handled by a BP.
- **Continuous** This configuration has no WUs at the beginning of a «Series». WUs are created on demand, e.g. when new data packages are available or when a time slot is reached.
- **Dynamic** In this configuration the previous two possibilities are merged.

These three approaches are supported by our model. «InterfaceDataSource» (IDS) is an interface which is implemented by a «Service» component for WU creation [6]. This component could have several connections to data sources. When these data sources signals new available data packages, IDS provides with the help of *getPath()* a file path which is usable for «Input» and a corresponding «Datafield» has *#File* as *type*.

Fig. 3 shows the first statechart diagram for our modelling approach. Depending on your BOINC scenario you can define how WUs are created. For all mentioned scenarios the statechart will always start at the initial point in the top-left corner. Immediately the process is subdivided into two parts with two transitions stereotyped by «Scatter» and «Detach». The BOINC's WUs are independently processed on

the client side from other processes. In addition to this WUs are structure elements and that's why they are not conceived to have a behavior. Other components have to deal with them and as a consequence these components can have behaviour definitions. While all WUs are public within BOINC's domain any component has access and can modify them.

«Detach» creates two orthogonal regions for the lifetime monitoring of one «Series» and all associated WUs. The top region is responsible for WU monitoring and the bottom region monitors the current «Series». The transition between "Idle" and "Process" is triggered by *dataAvailable* and *nextSequence*. In this transition *dataAvailable* is called by the IDS, and thereupon the file path is used to define a new WU. Fig. 4 shows the statechart of a single WU. In that statechart "Assimilation" has a "Next" named exit pseudostate and *nextSequence* is triggered when this exit is entered. As a result a new WU is created. It is defined that this exit pseudostate can only be used when a WU is part of a sequence as described in the next section. As at the initial point of this statechart, all available and new WUs are scattered and within this statechart are monitored. The top region is left when no more WUs are in process. The bottom region monitors the lifetime of a «Series» and "Processing" is only left under two circumstances: (1) the «Series» has to be canceled and (2) processing is complete and all results can be merged, which can be done in "Finishing", e.g. an average over all results of a monte-carlo simulation can be calculated.

D. Sequences of Workunits

In [4] a system for remote creation of chained WUs is shown, where one result can be used as input for other WUs. In our model we can handle a similar task. «WorkunitAssociation» enables one to define a «Series» with sequential computations. «InterfaceAssimilate» delegates these computations and can have an association to «WorkunitAssociation». As mentioned in the previous section the "Next" exit pseudostate in Fig. 4 is used when one WU is assimilated and has additional WUs to be performed. The following pseudocode demonstrates how the assimilation process can decide if one WU is in a sequence and if a WU follows:

```
Let ws As workunitAssociation.workunit.workunitState
If ws.seqid < ws.maximum_sequence Then
 For ro In Output
  Set workunitAssociation.input = ro
    Where
      workunitAssociation.input.name = ro.useAs.name
 EndFor
EndIf
ws.seqid = ws.seqid + 1
```

A new WU is filled with «Datafield» values of one «Output» when they are associated by *useAs*. As a consequence *useAs* must only be set when «Output» is used for one «Input» configuration. The fact is, when no *useAs* is available then it makes no sense to check for a sequence.

In the case when all results of a WU are required, BOINC's assimilator can create additional WUs with a duplicated configuration, e.g. a WU is missed to complete a sequence and is too often canceled by BOINC clients or WU's *delay_bound*[1] is reached. For this occurrence the WU can be copied and added to a «Series».

When a WU is part of a sequence, the WU's name has a special format to distinguish WUs. A similar approach for rBOINC is used [4]. rBOINC defines a specialized WU name and we modify this format to **"NNN-SEQ-XX-YY"**:

- **NNN** is the name of the WU, and
- **SEQ** is a start pattern for a sequence description.

Here the embedded string **"-XX-YY-"** is defined as follows: **XX** is the current sequence id and **YY** is used for the maximum number of sequences. This WU name format is used to select sequenced WUs in section III-F.

E. State of Workunit Computation

«WorkunitState» is associated by «Workunit» (WU) and from the beginning of its existence the state of a WU can be queried at any time. The tag-value *state* holds the current state and can be valued with the following variables:

- **CREATED** WU is created.
- **FAILED** WU has failed and can not be finished.
- **COMPUTATION** WU is in progress and one or more clients work on it.
- **DONE** Enough clients have worked on one WU and it can be moved to the validation and assimilation process.
- **VALIDATION** WU has to be validated.
- **ASSIMILATION** WU has to be assimilated.
- **CANCELED** WU is canceled by an administrator or by other processes, e.g. when sequenced WUs have failed or are canceled.
- **FINISHED** WU is finished and ready for later use, e.g. to create a new «Series» or to use their computational results.

For the UML model it is important what the state of a WU is, as a matter of fact the state value is responsible for deciding which actions are performed during the WU processing, i.e. when a WU fails the assimilation process has to decide if it should be performed again. The accessory method *check()* is used to query the current state of a WU and returns a descriptive text value, i.e. the string contains the current state with additional more precise information such as the *timestamp* of the last check or how long a WU is currently processing. The other two tag-values *maximum_sequence* and *seqid* are used for the «WorkunitAssociation» in the next section.

F. Cancelation of Series and Workunits

A WU can be canceled at any time. This is done by *InterfaceWorkunit::cancel()* and it is necessary to cancel WUs which are related to a cancelled WU, i.e. when the current WU is cancelled and has associated WUs, these must be also cancelled because they can never be processed with missing «Input» values. «WorkunitAssociation» has an additional OCL operation to query which WUs are in the current sequence:

[1] Deadline of one workunit.

```
WorkunitAssociation::querySequencedWorkunits(
  seqid : Integer, name : String) : Set(Input);
querySequencedWorkunits =
  self.series->select( s | s.workunit->select(
      w | w.workunitState.seqid > seqid
        AND
        -- NNN-SEQ-XX-YY
         w.name.substring(1,
              w.name.strpos("-SEQ")) = name
    )
  )
```

With this OCL statement, WUs can be selected which are later defined within a sequence and as a consequence they can be cancelled. Following cancelled WUs can have other associated WUs and they must be cancelled. The call of *Workunit::cancel()* is used to cancel one WU. «Series» can be cancelled with this concept, it is enough to call *cancel()* during the iteration of all associated WUs.

G. Service Interfaces

BOINC has several components which need to access «Input», «Output» and their embedded «Datafield» fields. Fig. 1 shows three interfaces to access them: (1) «InterfaceDataset», (2) «InterfaceValidate» and (3) «InterfaceAssimilate». With the help of this structure all functionalities can be generated and this makes the access more comfortable. Changes in the model are automatically resolved and interfaces are always valid for use.

IV. Case-Study

Our case-study modifies a movie, i.e. a movie is fragmented in single image sequences and basic image processing algorithm are applied to these sequences, some results could be seen on the project website [8]. Fig. 5 shows our use-case where one video is added by *C.B.Ries*, with the help of inotify [13] one BP is triggered by *dataAvailable* and WUs are created on demand. During WU adding it has to be clear which kind of data format is used, i.e. in our use-case we add a complete movie and subsequent implementations has to prepare this movie for WU creation. In this scenario we create ZIP-archives automatically and fill them with a number of image sequences.

Five «Series» are defined, in this case only the first fourth can be processed immediately. As shown in Equation 1 the fifth «Series» needs the result of the first four «Series».

$$\begin{bmatrix} Series\ 1\ (normalize) \\ Series\ 2\ (painting) \\ Series\ 3\ (negate) \\ Series\ 4\ (edge) \end{bmatrix} \Rightarrow\ Series\ 5\ (merge) \quad (1)$$

All «Series» instances have a different runtime configuration, e.g. in «Series» number four the image is manipulated by an edge algorithm. The fifth «Series» merges all previous results where for each image sequence they are added to a 2×2 raster image. On the right-hand side of Fig. 5 these different configurations are shown where the bottom configuration shows the mode "edge" for image manipulation and in the top configuration "merge" is assigned.

The fifth «Series» is not started until the other «Series» has finished. It is important to notice that the SAPP is always the same, only the input files are changed. In the first four computations only two files are necessary: (1) configuration for the mode of configuration and (2) the mentioned ZIP-archive with movie sequences. The last computation is altered and needs five files: (1) configuration as before with different values and (2) all four input files which are created by the other four computations.

This use-case describes a *dynamic series* where some WUs available on BP's start and additional WUs created during runtime. In the case that one of the first four series is canceled, the fifth series can never be started because of missing input data. This is solved by our statechart construction in Fig. 3 where the "Cancelation" state is responsible for cancelling all related WUs and «Series» instances.

V. Conclusion

In this paper we describe a UML model for WU creation and how the lifetime of WU series and individual WUs can be monitored. We have shown that only one model description is necessary to allow BOINC related components to access WU's input and output files, i.e. BOINC's validator and assimilator must not be changed to access the WU, all necessary code elements can be generated with one model description. With our model it is possible to set-up different computational scenarios where WUs are generated statically, continuously or mixed by these two approaches. WUs can be added to a process before a BP is started or they can be added on-demand during the runtime of a BP. With the help of UML statechart diagrams we can set-up a BP configuration where we can define how *overdue* or *absent* WUs are handled. It is possible to recreate them or if it is wished, the related computational series and related WUs are aborted.

The proposed UML modeling approach can help to reduce errors during administration of BPs. As a matter of fact, in traditional BPs it is necessary to reconfigure and reimplement several parts when only one configuration is changed, i.e. if the format of computational results is changed then all related components such as BOINC's validator and assimilator have to be similarly changed. Furthermore, adding or removing input files for one computation has an impact on several BOINC parts: (1) non well readable XML input files must be changed, (2) the call of BOINC's WU creation tools has to be altered, (3) altered files has to be copied to BOINC's download hierarchy, and (4) (maybe) input files have to be generated or prepared. Our modeling approach solves all these problems with the help of UML and OCL.

VI. Future Work

WU's performance can have restrictions, e.g. the use of floating-point operations or allocation of hard disk space can be limited. Currently it is not clear if UML can help to detect perfectly fitted values for this purpose. During our use-case tests we noted a large number of failed WUs because of

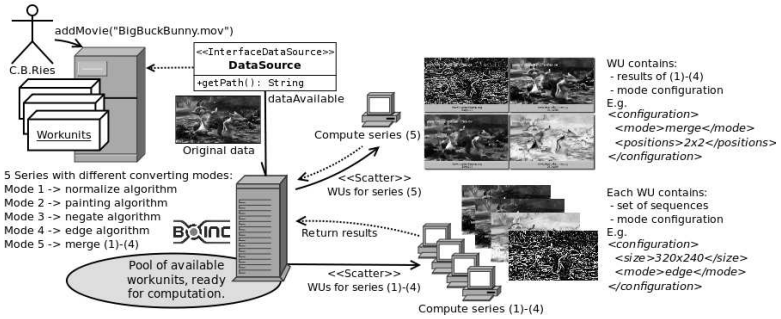

Fig. 5. Use-case to modify a movie: *C.B.Ries* adds a new movie to one server, running applications prepare this movie and fragment it into image sequences, thereupon these images are zipped into ZIP-archives. *DataSource* notifies a BP which automatically adds all announced WUs to this BP. Adding of one ZIP-archive implies four added WUs to this BP but with different configurations, each «Series» instance has a different mode for computation: *normalize*, *painting*, *negate*, and *edge*. These WUs are processed and the computational results are used as input for a fifth «Series», again with a changed mode for computation: *merge*.

wrongly adjusted boundary values for the restrictions mentioned. The set-up of this values has to be more precise and at best automatic. In future work we will work on this objective.

During the writing of this paper implementation with the support of this model are hard-coded and can not be changed — they are not flexible during runtime. One idea is to add a Domain-specific language (DSL) to describe WUs, series and sequences of computation. It could be possible to interpret this DSL during the runtime of a BP, to change the behavior of this BP and to generate code for all necessary components for WU handling on-demand.

Additional thought should also be spent on how our presented model can be used for conventional supercomputers, where other technologies like Message Parsing Interface (MPI) [14] or OpenMP [17] are used. It should be clear that MPI has to process workunits like BOINC, although admittedly with less input files; instead it uses more numerical values which are communicated between all involved computation nodes. The fact is that BOINC can be perfectly used to solve embarrassing parallel computational problems with less communication between all involved nodes. MPI enables one to use a distributed computing environment with several autonomous interacting nodes to achieve a common goal.

REFERENCES

[1] D. P. Anderson, C. Christensen, and B. Allen. “Designing a Runtime System for Volunteer Computing.” in *Proc. ACM/IEEE SC*, 2006, Article No. 126

[2] BOINC. “Backend program logic,” Internet: http://boinc.berkeley.edu/trac/wiki/BackendLogic [Version 2]

[3] BOINC. “Submitting jobs,” Internet: http://boinc.berkeley.edu/trac/wiki/JobSubmission [Version 19]

[4] T. Giorgino, M. J. Harvey and G. De Fabritiis. “Distributed computing as a virtual supercomputer: Tools to run and manage large-scale BOINC simulations”. *Computer Physics Communications*, vol. 181, February, 2010

[5] C. B. Ries. “BOINC - Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing.” Berlin Heidelberg: Springer-Verlag, 2012

[6] C. B. Ries, C. Schröder, and V. Grout. “Approach of a UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. ICCAIE*, 2011, pp. 483-488

[7] C. B. Ries, C. Schröder, and V. Grout. “Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. SEIN*, 2011, pp. 67-76

[8] C. B. Ries and C. Schröder. “Public Resource Computing mit Boinc.” Linux-Magazin, vol. 3, pp. 106-110, March 2011. Internet: lm-boinc.sourceforge.net

[9] C. B. Ries and C. Schröder. “ComsolGrid - A Framework For Performing Large-Scale Parameter Studies Using Comsol Multiphysics and Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. COMSOL Conf.*, Paris, 2010

[10] C. B. Ries, T. Hilbig, and C. Schröder. “A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework,” in *Proc. IEEE-IMCSIT*, 2010, pp. 663-670

[11] C. B. Ries. “ComsolGrid - Konzeption, Entwicklung und Implementierung eines Frameworks zur Kopplung von COMSOL Multiphysics und BOINC um hoch-skalierbare Parameterstudien zu erstellen.” M.Sc. thesis, University of Applied Sciences Bielefeld, Germany, 2010.

[12] W3C. “Extensible Markup Language (XML) 1.0 (Fifth Edition),” Internet: http://www.w3.org/TR/REC-xml/

[13] J. McCutchan, R. Love, and A. Griffis. “inotify - monitoring file system events,” Linux man pages(7)

[14] Message Passing Interface. “The Message Passing Interface (MPI) standard,” Internet: http://www.mcs.anl.gov/research/projects/mpi/ [18th February 2012]

[15] Object Management Group. “OMG Unified Modeling Language (OMG UML) Superstructure.” formal/2010-05-05, May, 2010.

[16] Object Management Group. “Object Constraint Language.” Version 2.2, Feb., 2010

[17] OpenMP. “The OpenMP API Specification for Parallel Programming,” Internet: http://www.openmp.org [18th February 2012]

[18] PKWARE. “APPNOTE.TXT - .ZIP File Format Specification.” Version 6.3.2, Sept., 2007

Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC)

Christian Benjamin Ries[1], Christian Schröder[1], and Vic Grout[2]

[1]Computational Materials Science and Engineering
University of Applied Sciences Bielefeld, Germany
e-mails: [christian_benjamin.ries, christian.schroeder]@fh-bielefeld.de
[2]Centre for Applied Internet Research
Glyndŵr University, United Kingdom
e-mail: v.grout@glyndwr.ac.uk

Abstract

Model Driven Engineering (MDE) is a software development process based on domain specific models and is applicable in any computer engineering challenges which make processes more transparent to different users in the individual domains. The Unified Modeling Language (UML) can be used as a powerful tool within MDE to describe complex architectures which are more intelligible and precise. In this paper, we will discuss the first steps of a new MDE process - with help of the UML, Domain-specific modeling languages (DSML), and a generated Integrated Development Environment (IDE) - to offer the opportunity to model a whole Berkeley Open Infrastructure for Network Computing (BOINC) project with less time and maintenance effort. Within this MDE process we defined six diagram types, each for a unique purpose for a better separation of modeling aspects. These research and development effort lowers the barrier for entry to create an own Volunteer Computing project based on BOINC.

Keywords

BOINC, Model/View/Controller, Model/View/Delegate, Integrated Development Environment, UML, Code Generation

1. Introduction

Over the past five decades, software researchers and developers have been creating abstractions that help them program in terms of their design intent rather than the underlying computing environment, e.g. CPU, memory, and network devices are more understandable by use of Application-programming interfaces (APIs) (Schmidt, 2006). In this paper we use Berkeley Open Infrastructure for Network Computing (BOINC) to build up a world-wide computer cluster based on resources released by volunteers (Anderson *et.al.* 2006). We will define an abstraction layer on top of BOINC and encapsulate BOINC's architecture and functionalities to create a Model Driven Engineering (MDE) process. We will discuss how we can generate an

Integrated Development Environment (IDE) by use of a Unified Modeling Language (UML) Profile (OMG UML, 2011), three Domain-specific modeling language (DSML) specifications, and a template-based Code-Generation (CG) within our research project Visu@lGrid (VG). VG covers research towards an all-in-one IDE to model a whole BOINC project with the help of UML, code generation facilities, and separation of related functionalities to specific diagrams. Some effort to define and implement a MDE environment with support of textual and graphical elements is spend for tool-chains within Eclipse, e.g. Eclipse Modeling Framework (EMF), UML2Tools, Xtext (Eclipse, 2011). EMF is one prominent example but different in its way of modeling. EMF tends to take a bottom-up approach whereas UML tends to take a top-down approach (Gerber and Raymond, 2004). Additionally, EMF is the core component of the Eclipse Graphical Modeling Framework (GMF) and it has be shown that GMF lacks in support of evolution, and that the GMF infrastructure has a number of limitations, some of them related to co-evolution, i.e. changes in the EMF model would make the GMF editor unusable and must be changed in most parts (Ruscio, 2010). In our work we present an approach we will discuss how we could handle this gap by the use of a specific model specification by use of a UML Profile and DSML's to create an IDE with the minimum support to define a BP.

The remainder of this paper is organized as follows. Section 2 gives an overview of the VG, the problem domains, and how this paper uses well established architectural software implementation patterns. Next, Section 3 covers the main content of this paper and describes our approach to model a BOINC project (BP) with help of a UML Profile and our approach to generate an IDE for BOINC development. In Section 4 some open tasks and research questions are listed. Finally, Section 5 concludes this paper.

2. Visu@lGrid for BOINC

BOINC is a framework for solving large scale and complex computational problems by means of Public Resource Computing (PRC). BOINC is based on the principle of PRC where the computational effort is distributed in separated so-called workunits onto a large number of computers connected by the Internet. Volunteers provide their released computer resources, download progress workunits, and send back the result to one BP (Anderson, 2004). On the one hand, installing, configuring, and maintaining a BOINC based project is a highly sophisticated task (Ries *et.al.* 2010) and a lot of experience regarding the underlying communication and operating system technologies are needed for most application. On the other hand to handle these challenges, VG would fill these gaps.

VG is a research project with the goal to have specifications and definitions how to create a BOINC project within a Model Driven Engineering process. Furthermore, an IDE based on Code Generation (CG) facilities and an UML Profile (Ries *et.al.* 2011) should be created to have an abstraction of all BOINC functionalities. This abstraction makes it feasible to have a non error-prone process to create and run BP's with minimum work and time effort. Figure 1 shows one viewpoint of the underlying architecture of VG. In fact, VG is based on three different projects: (1) *Visu@lGridML*, (2) *Visu@lGridCG*, and (3) *Visu@lGridIDE*. Some work is done in

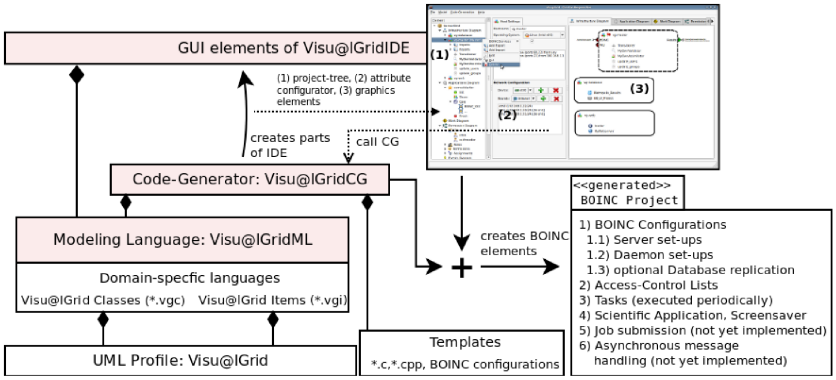

Figure 1: Workflow of an IDE generation, followed by a generation of a BOINC project based on the UML Profile Visu@lGrid, two DSMLs, and templates files.

any project (Ries *et.al.* 2010).

Visu@lGridML (VGML) includes DSML definitions and specifications. These values are twofold:

(1) VGML defines a UML Profile to create models which can describe a whole BP with support to specify the architecture and functionalities of one BP, i.e. how many hosts are used, how should BOINC daemons work, which scientific application should be distributed and which kind of workunits should be deployed., and
(2) VGML defines DSML's which are used by Visu@lGridCG to generate an IDE with support of an architectural tree-view, window forms to edit attributes, and a graphical view of a BP and associations between BOINC components, e.g. results of a computation should be stored on one specific host or in a database.

The IDE itself will be equipment with a module to optionally export a BP description which could be used by other code generators to create all necessary scripts, configuration files, and code implementations of a BP. Fig. 1 shows the first generated version of Visu@lGridIDE in the top-right area.

2.1. UML Profile: Visu@lGrid

UML Profile is a kind of UML extension mechanism. It specializes some of the language elements, imposes new restrictions on them while respecting the UML metamodel and leaving the original semantics of the UML elements unchanged. Icons and symbols can be specified and applied for these specialized elements. The Object Management Group (OMG) maintains some common and widely accepted profiles, such as SysML (OMG SysML, 2010). Stereotypes extend standard UML meta-classes. We have already specified a UML Profile called **UML Profile**

Visu@lGrid (UMLVG). UMLVG defines 50 stereotypes which are applicable to model a complete BP (Ries *et.al.* 2011). Current version of UMLVG defines no diagram types which could be used to encapsulate different focused models, e.g. a BP has an infrastructure based on one or more hosts and different sets of local or remote network-access restrictions for users with different owned rule-sets, e.g. permissions to administrate a whole BP or users with permissions just to request complete workunits.

2.2 Software Patterns: Model/View/Controller, Model/View/Delegate

One software pattern is the Model/View/Controller (MVC) architecture which is often used in Graphical User Interfaces (GUIs) to separate one data Model from graphical screen representations, so-called *View*. Between the *Model* and *View* is the *Controller* placed to handle the way the GUI reacts to user input. Requests by the *View* and changes of the *Model* should be synchronized to have an always up-to-date state. We already defined and implemented a DSML with an additional framework to enrich BOINC with a more handy and easier to use API (Ries *et.al.* 2010b). Our DSML approach makes it possible to decrease the lines of code for one fully executable BOINC application by approximately 77 percent based on a MVC architectural framework. In contrast, Qt uses a Model/View/Delegator (MVD) concept, where the MVC-Controller is replaced by a *Delegator* (Qt, 2011). The *Model* and *View* are still used. The *Model* class is responsible to offer and edit data items. The *View* renders items of the *Model*. How the items are rendered is defined by the *Delegator*. Between the *Model*, *View*, and *Delegator* one or more data items are exchanged and the underlying Qt framework keeps all views of one model synchronized. In our VGIDE we use more than one view of a model, i.e. Fig. 2 shows on the left-hand side an architectural tree view and on the right-hand side a graphical view of the same model. Unfortunately Qt's version 4.7 implementation does not support direct synchronization of an architectural tree model with a graphical representation. One work covers an approach to handle this gap (Schile, 2008) and this idea is partially used in our implementation.

3. Visu@lGrid Integrated Development Environment

UMLVG defines no diagrams; as a consequence some more effort is required to fill this gap. Here, main goals are:

(1) the approach should be easy to apply by persons who are not familiar with UML, and
(2) it should be possible to recreate the IDE at each time if it is necessary without lose of already done implementations, i.e. if descriptions of BOINC's validator or assimilator are done, they should be used in the new generated VGIDE.

Furthermore, the view of the hierarchy tree of one BP within the IDE should support context-sensitive menus to create or manipulate the structure of a BP and changes of the hierarchy should be synchronized by the underlying MVD implementation to have an always correct graphical representation of a BP, i.e. several levels of a hierarchy could only own associated elements, e.g. a host owns shares, an application

owns input files.

3.1. Idea

We created two new DSML specifications which are used to generate the structure of an IDE with support of contextual menus, automatic generation of forms to manipulate the attributes of UMLVG elements, synchronization of the underlying data structure with the hierarchy tree and graphical representation, and facilities to export the structure which could be used by external code generators. Fig. 2 shows part of these DSML's and the relation to our VGIDE. The DSML's have different file-extensions: (1) *VisualGrid Classes* ***(*.vgc)***, and (2) *VisualGrid Items* ***(*.vgi)***. VGC is used as an extension of the UMLVG to describe in which type of diagram the elements are shown, and VGI describes how the elements are visualized within the graphical view of VGIDE.

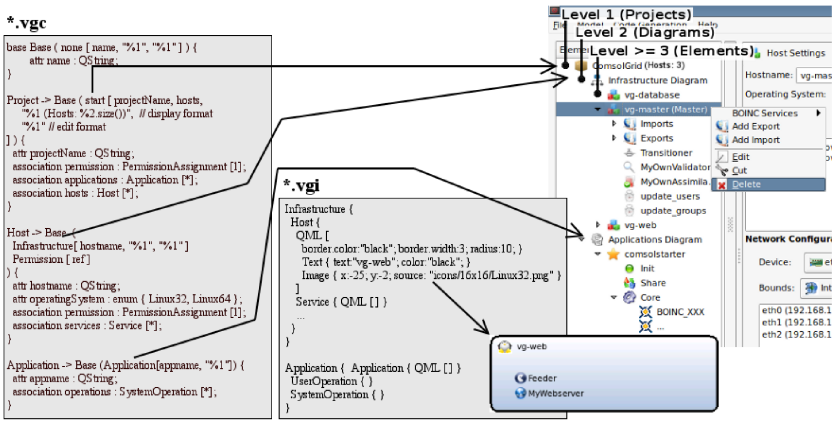

Figure 2: Conceptual idea and first realization of two DSMLs to generate a hierarchy tree and graphical representation of UMLVG elements.

VGC covers three important aspects:

(1) Description how elements are related to each other; described by the notation *"class name → class name of parent"* and by each element owned associations.

(2) Description in which diagram the elements are shown (see Section 3.2). Elements could be added to one or more diagram, e.g. *Host* is added to the Infrastructure and Permission Diagram. Within the Permission Diagram Host is just a reference and could not be manipulated, but set of permissions could be associated with it, i.e. one user could be set-up to have permission to remote log-in on this host.

(3) Description which information and how they are shown in the hierarchically tree. Used strings in the tree are described by format strings: *"%1 (Hosts: %2.size())"*. During execution the placeholders - *"%1"* and *"%2"* - are

replaced by predefined variables, i.e. *"projectName"* and *"hosts"* in this case. The type of these variables are checked automatically by VGCG, i.e. *"projectName"* is a kind of string and could be used directly, *"hosts"* is an association and has a multiplicity *"[*]"* to keep zero-or-more elements with type of *Host* on which methods could be executed.

VGI is used for the representation of graphical items related to elements in the VGC and could be described by other DSML specifications, e.g. in this example we use Qt's Modeling Language (QML, 2011) to specify a graphical node for one host. QML makes it possible to embed descriptions of graphical nodes within our DSML and to replace them during runtime.

3.2. Concept

As mentioned, UMLVG defines no types of diagrams, this is done by VGC. Here, we defined six diagrams for different purposes:

- **Infrastructure Diagram:** This diagram is used to specify the infrastructure of one BP. We can add different hosts, Storage Area Networks (SAN), BOINC services and periodically executed tasks. Fig. 1 includes an example of a generated IDE with a diagram of three hosts with few added BOINC Services (BS's), i.e. BOINC daemons, databases, crontab tasks, webserver installation, and other applications. This diagram defines the relation between hosts and where computational results should be stored. The infrastructure diagram is a kind of UML Deployment diagram.
  Applications Diagram: Within this diagram logical parts are specified, how an application would be executed or which third-party libraries are used. Furthermore, it is possible to add a wrapper application (Ries and Schröder, 2010b) which would make it feasible to use de-facto industry standard software components. This type of diagram is a kind of UML Statechart and Sequence diagram. The structure of this diagram could be more restricted, restrictions are depended by the type of application. On the one hand if a legacy-application like COMSOL Multiphysics (Ries *et.al.* 2010) should be used, it is not necessary to define a whole state or sequence graph, because the underlying wrapper-implementations handle the execution. On the other hand if an individual implementation is required, the structure of this BOINC application (BA) could be separated into four main sections: **(1) init**, **(2) share**, **(3) core**, and **(4) finish**, described in Table 2. Within the UMLVG these sections are defined by UML Classes which are associated to one or more UML Classifier.
- **Work Diagram:** This diagram is used to define the structure of workunits. We can specify which file aliases must be used within an Application Diagram and how the structure of input and output is. This type of diagram is a kind of UML Class diagram.
- **Permission Diagram:** Role-Based Access Lists (RBAL) are defined in this type of diagram. Within any users, roles and permissions could be added, cross-references are used to associate set of permissions to resources, i.e. users are not associated directly to resources like hosts, they are associated

to roles, these roles to set of permissions and finally to resources (Cirit and Buzluca, 2009).

- **Events Diagram:** BOINC could handle asynchronous communication to and from BOINC server to volunteers. The type of event could be defined in this diagram and the implementation could be referenced to elements within our mentioned Applications Diagram. This type of diagram is a kind of UML Sequence diagram. Unlimited events could be added and each is a start of a sequence to handle separated events, e.g. if one host becomes unreachable the BOINC services can be started on other hosts.
- **Timing Diagram:** As mentioned for Infrastructure Diagram, a BP could have periodically executed tasks. They are more specified in this Timing Diagram where we could add time-ranges for execution timing. This type of diagram is a kind of UML Class diagram.

Table 1 covers definitions of a static structure of our hierarchy tree model. The first three levels must be available in new BP's. It is not necessary that level 0 is shown. All leaves in this tree could be exported and external leaves are importable. Restrictions are present in the underlying UMLVG, i.e. exports are always possible and leaves of this tree which are imported must be valid on the point-of-interest.

Tree Level	Description
Level 0	First column of one tree describes the model root element and is hidden as default and owns project nodes.
Level 1	Describes separate BP's within one IDE. Fig. 2 shows one BP named *"ComsolGrid"*.
Level 2	Specifies default diagrams to support a full Model Driven Architecture (MDA) process to create a new BP.
Level >= 3	Each diagram could only stand for context relevant specifications, i.e. an infrastructure diagram specifies how a BP will be deployed. More hierarchically levels are used for contextual relevant elements and attributes, e.g. a host could contain ex-/imports and BOINC services. Each additional leaf hold dependent elements, e.g. a network interface card for users make no sense. Which elements are possible to add is defined by the UMLVG and VGC in Fig. 2. Only elements which are defined for one diagram are possible to add.

Table 1: Columns definition of a Visu@lGrid Tree Model

Our IDE is based on a clear structure matched with COMSOL Multiphysics (COMSOL, 2010). Figure 1 shows on the top-right hand-side our IDE. Three areas with well-defined purposes are shown: (1) the column of the left-hand side shows our hierarchy tree based on UMLVG, (2) the column in the middle shows contextual sensitive attributes/settings (AS) for selected elements within the tree, and (3) last column on the right-hand side shows our graphical representation of the hierarchy equal to or higher than level 2. AS's are based on our predefined UMLVG

stereotypes. They could be extended if higher fine-grained AS's are needed. The form to manipulate element attributes is generated by the VGCG and widgets are selected on the data type, e.g. string values are editable by text fields, associations are editable by selection lists.

Application Sections	Description
Init	Bas could be initialized in different ways, this depends on the target processor (e.g. multicore processors or Graphics Processing Units) and some special diagnostics features must be used, e.g. for debugging purposes.
Share	In any circumstances where it is needed to exchange values between more than one executed application, it can be defined which values are exchanged and how the data is retrieved by one application, i.e. an application have to set values for variables to exchange with external applications.
Core	Predefined BOINC functionalities (included in BOINC's framework) could be added here or own implementations are addable, i.e. textual (DSL, C/C++-Code), graphical (UML) implementations. Some work is done here, we already defined a way of a MVC concept and how Aspect-Oriented Programming (AOP) can be used (Ries *et.al.* 2010a).
Finish	Clean-up the execution environment, i.e. closing open files or database connections.

Table 2: Predefined section for the application implementation.

3.3. Benefit

With this code-generation and IDE facilities we have tool support for a wide range of different field of application. Different DSML specifications allow creation of IDE's for specific areas. This approach fills the gap between lightweight tools and heavy tools like Eclipse Modeling Framework which are not practicable in projects with a lot of changes of the underlying model from one day to another (Ruscio, 2010). It is only necessary to change the code-generation process to create executable projects. This work enables us to work further within the research project Visu@lGrid and it is a good starting point to work toward a code generator for full support to generate a whole BP.

4. Future Work

Future work will be focused on the implementation of a parser for the Object-constraint language (OMG OCL, 2011) to set-up rules between associations of UMLVG elements.

Additional work has to be done on the underlying architecture and how models can be converted into executable code, i.e. a modeled BP can be exported to XML files

and this is the end of the workflow currently. We have to work toward code generators to transform these XML files into code of scientific applications, validator and assimilator for computational results and tools for a practicable maintaining of a BP.

5. Conclusion

In this paper we have presented an approach to extend an already defined UML Profile for BOINC by Domain-specific modeling languages to make it easier and more intelligible to define and understand available BP's. We mentioned two Domain-specific modeling languages and added six diagram types with special purposes to specify BOINC's infrastructure, applications, how workunits must be used, who has permissions for different areas, how asynchronous communication is handled, and when selected tasks should be executed. Furthermore, we have defined a first structure of an IDE for Visu@lGrid which is automated generated, where each area has a well-defined purpose. With this effort the barrier to work with BOINC is lowered. The maintaining effort of one BOINC Project is less, because it is only necessary to manipulate the model within VGIDE, to export it to XML and then to create a BOINC Project by a not currently available code generator.

6. Acknowledgement

This work has been fully funded by the German Federal Ministry of Education and Research.

7. References

Anderson, D.P., Christensen, C., and Allen, B. (2006), Designing a Runtime System for Volunteer Computing, UC Berkeley Space Sciences Laboratory, Dept. of Physics, University of Oxford and Physics Dept., University of Wisconsin – Milwaukee

Anderson, D.P. (2004) BOINC: A System for Public-Resource Computing and Storage, *5th IEEE/ACM International Workshop on Grid Computing*

Cirit, C. and Buzluca, F. (2009), A UML Profile for Role-Based Access Control, *ACM SIN*

COMSOL Multiphysics Press Release Web Site (2010), "COMSOL Multiphysics Version 4.0 ab sofort verfügbar", http://www.comsol.de/press/news/article/648/, (Accessed 28 June 2011)

Eclipse Modeling Tools (2011), http://www.eclipse.org/home/categories/index.php ?category=modeling, (Accessed 20 August 2011)

Gerber, A. and Raymond, K. (2004), MOF to EMF: There and Back Again, *Cooperative Research Center for Enterprise Distributed Systems (DSTC)*, University of Queensland, Brisbane, Australia

Object Management Group (2011), OMG Unified Modeling Language (OMG UML), Superstructure, http://www.omg.org/spec/UML/2.3/Superstructure/

Object Management Group (2011), OMG Object Constraint Language (OMG OCL), http://www.omg.org/spec/OCL/2.0/

Object Management Group (2010), OMG Systems Modeling Language (OMG SysML), http://www.omg.org/spec/SysML

Qt Reference Documentation (Version 4.7), http://doc.qt.nokia.com/latest/model-view-programming.html, (Accessed 28 June 2011)

Qt QML (Version 4.7), Introduction to the QML Language, http://doc.qt.nokia.com/4.7-snapshot/qdeclarativeintroduction.html (Accessed 22 August 2011)

Ries, C.B., Schröder, C., and Grout V. (2011), UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC), submitted to *IEEE Conference on Computer Applications & Industrial Electronics (ICCAIE)*

Ries, C.B., Hilbig, T., and Schröder, C. (2010a), A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework, *Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT)*

Ries, C.B. and Schröder, C. (2010b), ComsolGrid - A framework for performing large-scale parameter studies using Comsol Multiphysics and Berkeley Open Infrastructure for Network Computing (BOINC), *Proceedings of the COMSOL Conference*

Ruscio, D.D, Lämmel, R., Pierantonio, A. (2010) Automated co-evolution of GMF editor models, *CoRR*, Volume abs/1006.5761, June 2010

Schiele, J. (2008), Being Qt in Theoretical Computer Science - Design and implementation of a program for editing and viewing of finite automates, *Studienarbeit*

Schmidt, Douglas C. (2006), Model-Driven Engineering, *IEEE Computer Society*, Volume 39, Number 2, February 2006, pp25-31

Approach of a UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC)

Christian Benjamin Ries
Computational Materials
Science and Engineering (CMSE)
University of Applied Sciences
Bielefeld, Germany
www.visualgrid.org

Christian Schröder
Computational Materials
Science and Engineering (CMSE)
University of Applied Sciences
Bielefeld, Germany
Christian.Schroeder@fh-bielefeld.de

Vic Grout
Centre for Applied
Internet Research (CAIR)
Glyndŵr University, United Kingdom
v.grout@glyndwr.ac.uk

***Abstract*—Despite the current enormous hype and popularity of Grid Computing environments like Amazons EC2 or Microsoft's Windows Azure , there exist open-source and free of cost software frameworks which allow to create high performance computing installation by means of Public Resource Computing (PRC). One PRC framework is BOINC (Berkeley Open Infrastructure for Network Computing) for solving large scale and complex computational problems. Each computer works on its own workunits independently from each other and sends back its result to a project server. Installing, configuring, and maintaining a BOINC based project however is a highly sophisticated task. Scientists and developers need a lot of experience regarding the underlying communication and operating system technologies, even if only a handful of BOINC related functions are actually needed for most applications. In this paper we present a Unified Modeling Language (UML) profile for BOINC called Visu@IGrid profile (VGP). A BOINC project installation for one or more hosts, a role-based access control, and modeling of scientific application is feasible by use of VGP. Based on our approach we provide a specification that allows the creation of BOINC projects with less development and implementation effort.**

***Keywords*—BOINC, UML, Profile, Stereotypes, Tag-Values**

I. Introduction

PUblic Resource Computing (PRC) technologies allow realising low-cost high-performance computing projects in certain application areas. Berkeley Open Infrastructure for Network Computing (BOINC) is a very prominent framework based in the principles of PRC and is based on a server-client communication infrastructure mechanism. Here, the client retrieves a project specific application from the server along with so-called workunit (WU), i.e. a number of parameter usually provided in data files of simple ASCII or binary format that are optionally needed by the application to perform specific tasks. Each BOINC project (BP) has its own infrastructure, i.e. few daemons, tasks, hosts, and scientific application (SAPP) for different target platforms.

This paper presents a Unified Modeling Language (UML) profile for Visu@IGrid (VG), called Visu@IGrid profile (VGP). VG is a research project with the goal to have specifications and definitions how to create a BP within a Model-driven engineering (MDE) process and graphical elements. This is the reason why we call it visual. VGP is used as a core specification which makes it possible to create UML models for a complete BP set-up with all components for a fully predictable infrastructure to have the possibility to use code-generation (CG) facilities. We show an approach which makes it possible to define how a BP will be installed on different ranges of unlimited count of hosts. Furthermore, we will show how to configure a complete role-based access control (RBAC) for different software components and system functionalities, e.g. database access or log-in on one host. There exists some promising RBAC approaches which will be adopted by VGP [4]. We have already presented a first modeling domain-specific language (DSL), i.e. textual and graphical model elements [11]. Additional, we have shown that only a handful BOINC functions are necessary to create a complete usable BP.

The remainder of the paper is organised as follows. Section 2 gives an overview of BOINC's problem domain. Next, Section 3 describes the UML modeling principles and which UML extensions are defined for VGP. In Section 4 some open tasks and research questions are refereed which we will focus on in additional work. Finally, Section 5 concludes this paper.

This project is funded by the German Federal Ministry of Education and Research.

II. BOINC: Usage and Challenges

BOINC is based on a simplified architecture, i.e. each task is separated in autonomous running applications like BOINC daemons or scripts and we call these BOINC services (BS). In fact, few steps allow to create a new BP: (1) download and install prerequisite software packages, (2) download the BOINC sources, (3) configure and build the BOINC software, and (4) call scripts for an automatic rudimentary and not really usable BP installation process [19]. Next, you can modify each part of your new BP, e.g.

- which and where BS should be executed, i.e. in case more than one hosts are used it is possible to define specific BS's should be executed with different or same start-up parameters on individual hosts,
- if a database uses a replication for read-only queries to speed-up BP's performance,
- which host has the main installation, i.e. if individual hosts are used on server-side, then one host has the BP

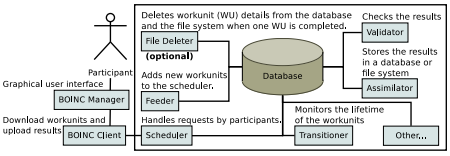

Fig. 1. Client and server components of BOINC's infrastructure

installation and has to distribute it to supplementary hosts. More detailed adjustments are possible. As in [11] mentioned, there exists typical errors that can occur due to manual editing of the BOINC server configuration files. As a consequence of these errors one can expect a significant effect on the system's integrity and application performance. In practice, wrong configured BS's do not work correctly, i.e. a BOINC Assimilator (BA) will not store results for later use or a BOINC Validator (BV) is not able to validate results received by participants.

On one hand if only one host is planned to use, it could be an uncomplicated proceeding from download of the BOINC sources to a running BP. In this case, all work is done on one host, i.e. BOINC sources could be accessed directly, BS's must be started on this host, database and web-server installations are also directly available, and only for this host relevant values are added to BOINC's configuration. Additionally, only one network interface card (NIC) is necessary to be configured and exclusive ports of Hypertext-Transfer Protocol [Secure] (HTTP[S]) must be allowed for network traffic. On the other hand it could be an enormous challenge to set-up a BP with more than one host and when daemons, tasks, database and web-server installations are distributed for execution to individual hosts. First established BP - Seti@Home (SAH), which is one of the most driven BP - has minimum count of 13 hosts on server side, and each host has a different functionality[1]. SAH has three databases[2] (DB) on respective host, and each DB has a different purpose but it is necessary that all BS's can interact with them.

A. Special Interest Groups and Architecture

Different special interested groups could administrate one BP, i.e. "administrators" configure and maintain the system installation, and "scientists" could create new WU's and add new SAPP versions. SAPP's could be implemented in different ways, e.g. scripts, hand-written code in different programming languages (C/C++ and Phyton are supported by default), or we can use ISV (Independent Software Vendor)-Applications or legacy applications [3], [7], e.g. in [10] we implemented a wrapper to handle COMSOL Multiphysics [5] computations on one computer. To a greater extent, an application can be written from scratch. This approach is the most difficult of all and fortunately BOINC's application programming interface (API) is not changing frequently with a new version. Fact is, one can show that only 23 different BOINC functions are necessary to implement a successfully running research relevant distributed SAPP [11]. We have to note, that we do not mention something about new target devices which could be implemented with additional application logic or extended functionalities, e.g. to support graphics processing units (GPU) with the help of BOINC's API. When a BP is installed and a SAPP is implemented, new problems and questions will arise:

- How can you manage errors or how is it possible to upgrade one SAPP during runtime?
- How could we replace software components on particular hosts or a whole host itself with same functionalities?
- How could we restrict access to groups of users and only for selected functions and system facilities?

Mentioned steps could be executed on different hosts, and it is highly error-prone to install and configure all required software components and interface settings and to keep an always valid configuration and stable system when for example software has to upgrade or a host has to be replaced. In addition, it is too cumbersome to create a fast and easy to use BP, just only for tests or quality assurance processes.

B. Proceeding

As seen on the left hand side of Fig. 1, participants need to download the BOINC Manager[3] (BM), register for one or more BP's, and let the BOINC Client (BC) organizes all system calls, in-/output handling, allocating compute resources, so-called scheduling [1] and handling of communication messages to and from individual BP's.

Fact is, with a minimum of five different BOINC daemons it is feasible to start-up a fully operable BP. Fig. 1 shows on the right hand side these five daemons with an optional sixth one. Scheduler is the interface for participant requests and supports two kinds of requests: (1) to send out WU's, and (2) to handle with computation results [1], [6]. Frequently, the shared-memory (SHM) based queue for schedule requests is refilled by the Feeder. File Deleter will keep the DB and file system clean of expired or not longer needed datasets and files. Lifetime of WU's is tracked by the Transitioner and this makes it necessary that all BS's have access to BOINC DB's, i.e.

- when an user creates WU's, information about these WU's are stored within BOINC's DB and they are marked as not *yet processed*,
- a request to the Scheduler will select few WU's and will change the state of these WU's to *processing*,
- after this request, different proceedings are possible: (1) each WU has a time-slot in which they should proceed and when time-slots upper limit - so-called deadline - is reached this WU will be marked as *out of date* and could be transmitted to participants again or will be deleted

[1]http://setiathome.berkeley.edu/sah_status.html - accessed 22.05.2011

[2]Components of SETI@home: (1) BOINC master database which is the BP back-end database, (2) BOINC replica database for read-only access, and (3) SETI@home science database to store results.

[3]A GUI to handle all BP's where volunteers are registered.

later[4], and (2) WU's are returned inside time-slots and the Scheduler will mark them in BOINC DB's as *ready for validation*,
- the BV will validate the result and mark each WU result as *valid* or *invalid*, and
- the BA will store valid and/or invalid results in an additional DB or within the file system hierarchy and mark assimilated WU's as *finished* and *ready for deletion*.

In Fig. 1 it could be seen, that each daemon needs a valid configuration to access BOINC's DB. Furthermore, a lot of knowledge is required to define runtime parameters of each daemon. Default values are fine, but in some cases it is more valuable to define various parameter sets, e.g. if load-balancing is necessary to increase server capacity. A file is used to define which SAPP's are offered by one BP. In addition, this file contains information about planned or supported target platforms, e.g. windows_intelx86[5] or x86_64-pc-linux-gnu[6]. Definitions in this file are not strict, it is necessary to have a SAPP within BP's file hierarchy, otherwise not used platforms are unrelated to SAPP's.

III. UML Profile: Visu@lGrid

UML profile is a kind of UML extension mechanism [4], [15]. It specializes some of the language elements, imposes new restrictions on them while respecting the UML metamodel and leaving the original semantics of the UML elements unchanged. Icons and symbols can be specified and applied for these specialized elements. The Object Management Group (OMG) maintains some common and widely accepted profiles, such as Systems Modeling Language (SysML) [16] and UML profile For Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [17]. Stereotypes extend standard UML metaclasses. Furthermore, we can exchange UML models with the XML Metadata Interchange (XMI) language from one integrated development environment (IDE) to another.

UML profiles are defined in terms of three basic mechanisms: *stereotypes*, *tagged values*, and *constraints*. A stereotype defines how an existing metaclass may be extended. A tagged value is an additional meta-attribute, it could be seen as a variable. It has a name and a type, and is member of a specific stereotype. Constraints can be associated with stereotypes, they could be informal in plain-text or more specific defined with Object Constraint Language (OCL) [18]. OCL is part of the UML, and is used to express constraints and properties of model elements. Currently, only informal constraints are defined for VGP, because a high effort for implementation is needed to realize a fully support for all VGP requirements.

Most of the current OCL tools are academic tools and were developed by a team of a single university [2]. Although the quality of tools has improved considerably over the last years, it is not a surprise that these OCL tools cannot compete in terms of usability and the functionality they offer with IDE's for writing implementations.

Our UML profile approach is currently based on nine UML metaclasses and Table I lists these with our specified stereotypes which are used in this paper. With them we can set-up an automatic code generation (CG) of a BP. Here, VGP is currently based on 31 stereotypes which are usable to define BP's for a single host or a bundle of hosts, where each host runs different BS's. Our principles for VGP are directly based on the BOINC architecture. This means we are strongly focused on top-down analysis of BOINC's functionality, communication and runtime behavior, i.e. BP's are seen as UML packages with embedded UML components for hosts. BS's are seen as UML components with functionalities to outside world. Static configurations are UML properties which could be defined as UML classes, i.e. configuration of network interface cards (NICs) has fixed values. Main benefit of our VGP profile are based on the advantages of UML itself. With UML it is possible to specify Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs) of processes and implementations. Here, we follow this methodology and define a model with a mix of PIM and PSM:

- On server side elements are fixed for one specific target platform[7] and PSM is used,
- creation and maintaining of workunits or scientific applications is not related to underlying operating system (OS) functionalities which means that PIM will be used, and
- participant's side could be highly heterogeneous where a PIM is reasonable.

A. Concept Idea behind Visu@lGrid Profile

As mentioned, our VGP concept is based on BOINC's implementation concept itself. Here, we make use of the opportunity that UML elements could be nested in other ones. With this in mind, we define a hierarchy with UML for a BP and each level in the hierarchy must have the potential to be replaceable. In our view, UML is the best choice to support this. Two engineering points of views exist: (1) outer view, and (2) inner view. Clearly, the outer view is the easiest to understand and it describes the world of participants / volunteers which are registered to one or more BP's and is illustrated on the left hand side of Fig. 1. In contrast, the inner view is more complex and needs more attention to specify the infrastructure and behavior on server-side.

B. UML Profile Itself

Fig. 2 shows an overview of all currently defined stereotypes, how they relate to each other stereotypes, and which UML metaclasses are extended. Constraints are not included, they are partly informal specified in Table I. «Projects» is our root of VGP and could own components extended by «SAN» and packages extended by «Project». The idea is,

[4]This behavior depends on sets of parameters during WU creation.
[5]Microsoft Windows running on an Intel x86-compatible CPU
[6]Linux running on an AMD x86_64 or Intel EM64T CPU
[7]BOINC supports only Unix platforms.

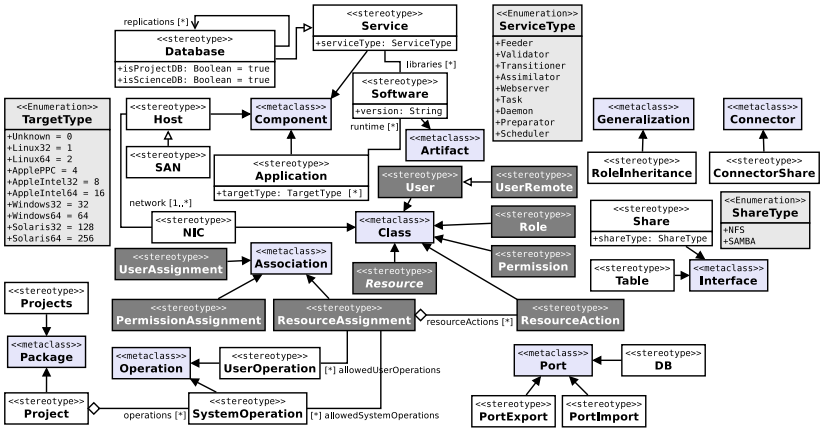

Fig. 2. VGP stereotypes to restrict the use of UML modeling aspects. This is an overview diagram and Table I names all metaclasses which are extended by these stereotypes.

that «Project» can be imported and exported any time within an IDE [8]. BS's can be added with components extended by «Service», which must be typed by «ServiceType». The enumeration does not provide any case of different BS types, for this purpose enumeration literals *Task* and *Daemon* are added and must be used if supplementary BS's are implemented, configured and wished to use for one BP. When a BS needs advanced software or libraries for runtime, they can be added in different versions by «Software». «Database» is a specialization of «Service» and can be used to add three different types of databases: (1) a BP database as shown in Fig. 1, (2) a science database (SDB) to store computational results or other datasets, and (3) database replications for the first and second database type. DB instances can be determined by two tag-values: *isProjectDB* and *isScienceDB*. First one is true when it is BP's database, and second one is true when this it is a SDB. DB replications are associated by *replications*, but the mentioned two tag-values are not necessary to specify while they can be retrieved automatically by check of owner's type. SAPP's can be modeled with components extended by «Application» and all planned supported target platforms must be added to *targetType*. Again, for runtime required software packages and libraries can be specified.

All hosts within one BP can share data among each other and for this reason, hosts get ports extended by «PortExport» and «PortImport» which must be associated to each other with interfaces extended by «Share». One «PortExport» with a provided «Share» can be used by more than one «PortImport», i.e. a «PortExport» is a global definition within one BP. If authorized access is wished, both ports must be extended by «Resource» and associated to a set of «Permissions» (described later). Same methodology is specified for databases and database-tables, i.e. «DB» is like mentioned two kind of ports and «Table» is like «Share». These principles are based on the UML specification to create components which are fully replaceable when all ports of another component are equal.

In Fig. 2, stereotypes with a dark-gray background color are used to define RBAC for one BP. It is not explicit defined where RBAC should be defined. Few ways are possible: (1) global definition in «Projects» or (2) fixed definitions for each «Project». We suggest to define them within «Project». In this case it would be possible to exchange BP's more effortless, otherwise it is necessary to create new RBAC or to duplicate existing ones. In [4] the authors give an idea of an approach to define an user control mechanism where each user could be added to roles and individual resources, our approach extends this idea. We define that permission rules could only associated to elements which are extended by «Resource». Individual users can be defined with classes extended by «User», roles are defined with classes extended by «Role», and a set of permissions are defined with classes extended by «Permissions». Based on this there is a strict coherence how associations must be used, i.e. «User» instances must be associated to «Roles» and these to «Permissions». Finally, «Permissions» could be associated with elements extended by «Resource». Fig. 3 shows a first example how our VGP could be applied to models.

C. Case-Study LMBoinc

Fig. 3 shows some applied stereotypes in a context of a real BP named *LMBoinc*. LMBoinc modifies a video stream, i.e. a video is fragmented in sequences and basic image processing algorithm are applied to these sequences, some results could

be seen on the project website [9]. WU's and computational results within LMBoinc include ZIP-archives (ZIP) with a specific number of sequences, wherefore the SAPP *lmboinc* is associated to one matching «Software» to make zip available during runtime. LMBoinc uses three hosts on server-side:

- **lmboinc-db:** This host has the BP database "DBProject" and SDB "DBScience" installed. SDB is used to store metadata of computational results, i.e. dimension, filesize, and filenames of sequences. Furthermore, BP's database has an additional replication to speed-up database queries.
- **lmboinc-services:** This host is composed by four BS's. The BA is associated to mentioned SDB and has an additional association to the third host and stores results within Webserver's file system, i.e. when a port is owned by one BS, additional constraints define which directories are usable for external use. Here, the exported directory is below the root directory for web-pages used by this webserver instance and network file system (NFS) is used between these hosts.
- **lmboinc-web:** Because of the provided Webserver component, participants just see this host of one BP. As a consequence, Feeder and Scheduler must be added to this host, due to the mentioned fact that Feeder is responsible to refill the shared-memory with information of next WU's which are selected for computation. When one participant requests new WU's, Feeder will check the shared-memory and send out informations where participants could download these WU's with HTTP requests. Therefore, necessary ZIP's must be below Webserver's document root directory, to make them accessible. *lmboinc-services* owns a BS to create WU's which are added to this download area.

In the top-right area three users are added with particular role associations. *Ries* is associated to role *Admin* and has permissions to access all hosts locally and remotely, additionally he is allowed to handle WU's for the SAPP *lmboinc*. The set of permissions for the association between *HandleWork* and *Preparator* include two operations: *createWU* and *cancelWU*. These two operations are implemented within *Preparator* and can be called by users when they get this permission. How this could be done must be provided by CG. Schroeder owns two roles, first is *Scientist* to interact with WU's as mentioned, and second role is *WebAdmin*. *WebAdmin* allows only to moderate posts within a BP related web forum. User *Grout* has only one role and the permission to moderate forum entries.

As defined in the top-left area, this BP targets Linux and Windows for 32 bit platforms where *lmboinc* will be distributed to. For more supported platforms it is required to extend the tag-value *targetType* by additional enumeration literals of «TargetType».

IV. Conclusion

We have proposed VGP, which provides an early approach to handle system and runtime relevant problems to implement a PRC project based on BOINC. Our approach is just the beginning of a MDE process to set-up a complete BP and to handle all necessary administration and maintaining tasks. In this paper, we have shown how our VGP can be applied to a real model within a cluster environment, how data can be shared between this cluster domain, where BS's are installed, which SAPP is used and which target platforms are supported. We have just shown an extract of VGP, due to the fact of the restriction of upper-page limit of this paper. A draft version of VGP is available and will be frequently changed [13]. An entire model would contain all design model elements and specifications to create a whole BP with a scientific application.

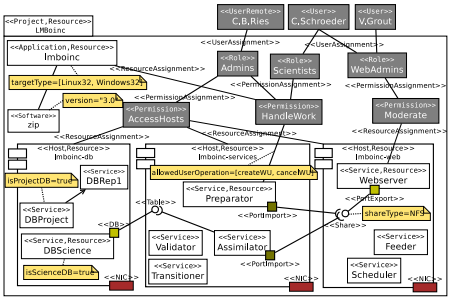

Fig. 3. A BP with three users, three hosts with different added BS and NIC's, a SAPP called *lmboinc* [9], and a RBAC set-up.

V. Future Work

Future work will focus on different approaches to make it possible to define processes for WU creation and maintaining of failed processed WU's. In general, currently it is not clear how we could monitor WU's lifetime, i.e. how many WU's are still in queue or if failed WU's should be processed again or not.

Additional work must be done for modeling concepts of adding and configuring of available ISV applications, e.g. Scilab. Some work is done and could be integrated in or used for VGP [8], [10], [11], [12]. As well some effort has to be spend on how we can define security restrictions on network traffic on «NIC» elements.

Currently, periodically executed tasks can be added by «Service» and *Task* assigned to service type. The period when one task should be executed is specified by a string value and is based on the format of the UNIX cron daemon. Future work will cover an approach how to describe this period with SysML timing diagrams [16] to make it more predictable.

Asynchronous messages by participants to one BP and vice versa are currently not supported, but suitable BS's can be added with «Service». How to react on incoming messages on client- and server-side is undefined. Additionally, it is unclear how we should react on unexpected exceptions, e.g. one BS is not responding anymore, or if there is not enough space to store computational results.

TABLE I
EXTRACT OF OUR UML STEREOTYPES WITHIN VGP

Stereotype name; extended Metaclass	**Description**	**Stereotype name; extended Metaclass**	**Description**
Projects; Package (from Kernel)	Top-level package which owns all sub-elements, i.e. hosts or users.	Host; Component (from BasicComponent)	Specifies one host within a BP, a host owns some or all BS.
Project; Package (from Kernel)	One specific BP.	NIC; Class (from Kernel)	Defines network settings for one host.
SAN; Component (from BasicComponent)	External host with capabilities to store data, e.g. WU's or computational results. Also known as storage area network (SAN).	OperatingSystem; Enumeration (from Kernel)	Unique literals for each planned or supported operating system, e.g. Linux32, Linux64, Windows32, or Windows64.
PortExport, PortImport; Port (from Ports)	Definition of network shared file directories between hosts.	ShareType; Enumeration (from Kernel)	Unique literal for each planned or supported share type, e.g. NFS or SAMBA [14].
Share; Interface (from Interfaces)	One specific directory which is ex- or imported. Exported directories are defined as provided interfaces, otherwise they are defined as required interfaces.	UserAssignment, PermissionAssignment, ResourceAssignment; Association (from Kernel)	Associates users, roles, permissions, and resources. UserAssignment assigns users to roles, PermissionAssignment assigns roles to permissions, and ResourceAssignment assigns permissions to resources.
User, UserRemote; Class (from Kernel)	A physical user which has different permissions within one BP described by roles and permissions.	Software; Artifact (from Artifacts, Nodes)	Software packages or libraries which are required during runtime, i.e. BA needs unzip to extract results out of an ZIP.
Resource; Class (from Kernel)	An user could only be assigned to resources when resources have this stereotype.	UserOperation; Operation (from Kernel, Interfaces)	Operations are executable by users.
ResourceAction; Class (from Kernel)	Actions which one user in a role or permission list could call.	Role, Permission; Class (from Kernel)	Defines roles for users and which permissions these roles include.
DB; Port (from Ports), Table; Interface (from Interfaces)	Allow BS's to access database tables.	SystemOperation; Operation (from Kernel, Interfaces)	Implementations of system or BS functionalities, e.g. routines for BV or BA.
Service; Component (from BasicComponents)	BS's on one host, e.g. Feeder and Transitioner.	ServiceType; Enumeration (from Kernel)	Type of a BS, e.g. Feeder or Transitioner.
Database; Component (from BasicComponents)	BOINC's master/science database and optional replications to increase DB performance.	ConnectorShare; Connector (from BasicComponents)	Connects exported shares with imports and vice versa. An export could be used by multiple imports.
Application; Component (from BasicComponents)	Describes one SAPP, which is used for computations.	Platform; Enumeration (from Kernel)	Literals for each target platform, e.g. Linux 32 bit or Windows 32 bit.

REFERENCES

[1] D. P. Anderson, C. Christensen, and B. Allen. "Designing a Runtime System for Volunteer Computing." in *Proc. ACM/IEEE SC*, 2006, Article No. 126

[2] T. Baar, D. Chiorean, A. Correa, M. Gogolla, H. Hußmann et.al. "Tool Support for OCL and Related Formalisms - Needs and Trends" in *Lecture Notes in Computer Science*, vol. 3844. J.-M. Bruel, Ed. Berlin Heidelberg: Springer-Verlag, 2006, pp.1-9

[3] O. Baskova, O. Gatsenko, G. Fedak, O. Lodygensky, and Y. Gordienko. "Porting Multiparametric MATLAB Application for Image and Video Processing to Desktop Grid for High-Performance Distributed Computing," presented at the 25th Int. Supercomputing Conf. (ISC), Hamburg, Germany, 2010

[4] Ç. Cirit and F. Buzluca. "A UML Profile for Role-Based Access Control," in *Proc. ACM SIN'09*, 2009, pp. 83-92

[5] "Multiphysics Modeling and Simulation Software – COMSOL." Internet: http://www.comsol.com [Oct. 29, 2011]

[6] D. Werthimer, J. Cobb, M. Lebofsky, D. Anderson, and E. Korpela. "SETI@HOME - massively distributed computing for SETI". *Computing in Science and Engineering*, vol. 3, pp. 78-83, Jan. 2001

[7] A. C. Marosi, Z. Balaton, and P. Kacsuk. "GenWrapper: A Generic Wrapper for Running Legacy Applications on Desktop Grids," in *Proc. IEEE-IPDPS*, 2009, pp. 1-6

[8] C. B. Ries, C. Schröder, and V. Grout. "Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC)," in *Proc. SEIN*, 2011, pp. 67-76

[9] C. B. Ries and C. Schröder. "Public Resource Computing mit Boinc." Linux-Magazin, vol. 3, pp. 106-110, March 2011. Internet: lm-boinc.sourceforge.net

[10] C. B. Ries and C. Schröder. "ComsolGrid - A Framework For Performing Large-Scale Parameter Studies Using Comsol Multiphysics and Berkeley Open Infrastructure for Network Computing (BOINC)," in *Proc. COMSOL Conf.*, Paris, 2010

[11] C. B. Ries, T. Hilbig, and C. Schröder. "A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework," in *Proc. IEEE-IMCSIT*, 2010, pp. 663-670

[12] C. B. Ries. "ComsolGrid - Konzeption, Entwicklung und Implementierung eines Frameworks zur Kopplung von COMSOL Multiphysics und BOINC um hoch-skalierbare Parameterstudien zu erstellen." M.Sc. thesis, University of Applied Sciences Bielefeld, Germany, 2010.

[13] C. B. Ries. "UML Profile for Visu@lGrid (draft)." Internet: www.visualgrid.de/research/drafts/UMLProfileVG-draft.pdf

[14] "Samba – opening windows to a wider world." Internet: www.samba.org [Oct. 29, 2011]

[15] Object Management Group. "OMG Unified Modeling Language (OMG UML) Superstructure." formal/2010-05-05, May, 2010.

[16] Object Management Group. "OMG Systems Modeling Language (OMG SysML)." formal/2010-06-01, June, 2010

[17] Object Management Group. "OMG Profile For MARTE: Modeling And Analysis Of Real-time Embedded Systems." Version 1.1, June, 2011

[18] Object Management Group. "Object Constraint Language." Version 2.2, Feb., 2010

[19] "Setting up a BOINC server." Internet: boinc.berkeley.edu/trac/wiki/ServerIntro, July 25, 2011 [Oct. 29, 2011]

A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework

Christian Benjamin Ries
University of Applied Sciences Bielefeld
Computational Materials Science & Engineering
Wilhelm-Bertelsmann-Str. 10,
33602 Bielefeld, Germany
Christian_Benjamin.Ries@fh-bielefeld.de

Thomas Hilbig, Christian Schröder
University of Applied Sciences Bielefeld
Computational Materials Science & Engineering
Wilhelm-Bertelsmann-Str. 10,
33602 Bielefeld, Germany
{Thomas.Hilbig, Christian.Schroeder}@fh-bielefeld.de

***Abstract*—BOINC (*Boinc Open Infrastructure for Network Computing*) is a framework for solving large scale and complex computational problems by means of public resource computing. Here, the computational effort is distributed onto a large number of computers connected by the Internet. Each computer works on its own workunits independently from each other and sends back its result to a project server. There are quite a few BOINC-based projects in the world. Installing, configuring, and maintaining a BOINC based project however is a highly sophisticated task. Scientists and developers need a lot of experience regarding the underlying communication and operating system technologies, even if only a handful of BOINC related functions are actually needed for most applications. This limits the application of BOINC in scientific computing although there is an ever growing need for computational power in this field. In this paper we present a new approach for *model-based development* of BOINC projects based on the specification of a high level abstraction language as well as a suitable development environment. This approach borrows standardized modeling concepts from the well-known *Unified Modeling Language (UML)* and *Object Constraint Language (OCL)*.**

I. Introduction

VOLUNTEER computing technologies allow to realize low-cost high-performance computing projects in certain application areas. A very prominent framework based on the principle of Public-Resource Computing (PRC) is BOINC (*Boinc Open Infrastructure for Network Computing*). BOINC provides an Application Programming Interface (API) with about one hundred functions of different categories, e.g. *filesystem operations*, *process controlling* and *status message handling* [1], [2]. A few of the most important functions are listed in section I-B. PRC is based on a server-client communication infrastructure mechanism. Here, the client retrieves a project specific application from the server along with a so-called workunit, i.e. a number of parameters usually provided in data files of simple ASCII or binary format that are optionally needed by the application to perform a specific task. BOINC is strongly focused on autonomic applications, each BOINC project has its own server, applications and tasks. The client executes the application, i.e. performs the calculations and sends its results back to the server which assembles these into a "global" result or stores these results at specific places.

This project is funded by the German Federal Ministry of Education and Research

The setup of a BOINC project heavily relies on the use of file-based scripting techniques. For instance, the programming language *Python* is utilized for the creation of a standard BOINC server infrastructure with database initialization, website and administration interface configuration and an optional BOINC test application. A few scripts are implemented using GNU BASH to sign executable files with encryption keys and yet another script uses the C shell (csh) to monitor network traffic. Extensible Markup Language (XML) files are used for the runtime server configuration. *All* files have to be edited *manually* by the project developer, scientist or administrator which bears the risk of making a large number of typical errors. For example, wrong spelling of parameter names or simulation relevant values as shown in I-A would have a significant effect on the system's integrity and the application's performance [8]. One way to cope with these problems is to provide a tool support which allows to automate the manipulation, generation, and checking of all necessary scripts.

In this paper we discuss a model-based approach for the development of BOINC projects that makes it possible to implement application specific changes while always keeping a valid configuration of the BOINC infrastructure. We give an idea on how to create a proper high-level domain specific language (DSL) in order to develop and maintain a complete BOINC application. This DSL forms the basis of an easy-to-use programming environment along with a high-level programming language and a suitable development process. Additionally, we present a way to model a complete BOINC installation including the client application for different target computer architectures and processor types. Aspects of single- and multicore processor units (CPU) and graphics processing units (GPU) are also discussed.

A. Typical errors during the BOINC server configuration process

The following list shows examples of typcial errors that can occur due to manual editing of the BOINC server configuration files. As a consequence of these errors one can expect a significant effect on the system's integrity and application performance.

- *uldl_dir_fanout* is the parameter that contains the number of subdirectories inside the upload and download directories on the server computer. A wrong value set here may dramatically slow down the system's server performance because of too many hard disk drive accesses.
- *shmem_key* names the allocated memory that is needed for the interprocess communication (IPC) between all BOINC applications on every BOINC project server. It is required that this value is unique, never changed during the runtime and is used by all BOINC server applications.
- *msg_to_host* must be included in the BOINC server configuration to enable sending of trickle-down messages[1] to the BOINC client nodes.
- *tasks* describes a set of parameters for applications which should execute in a cycle period, i.e. a crontab. Suitable values are needed to avoid problems like extremely large logging files or a outdated statistics, i.e. how many workunits are left for working or have errors during the computation.
- *daemon* contains a set of command descriptions. It is useful to start more than one daemon process to get a good load balancing of user requests, e.g. when the BOINC projects are much in demand.

One of our goals is the automatic determination of these most important parameters for different target computer architectures and processor types [20].

B. The BOINC Application Programming Interface

BOINC offers few example applications in which the number of lines of code range from 38 to 308. The first one only includes some elementary functions and no BOINC specific commands, e.g. a for-loop which just keeps the processor busy for one second. The second one is a more useful example since it contains BOINC specific function calls, e.g. how to retrieve the name of the checkpoint file or the actual processing state. Implementing a complex scientific application using BOINC is far more complicated and requires a broad experience of the developer. However, one can show that only 23 different BOINC functions are necessary to create a successfully running research relevant distributed computing application [23], [21].

II. State of the Art

Model-driven engineering (MDE) is becoming the dominant software engineering paradigm to specify, develop and maintain software systems. For example, Brunelière *et al.* [6] propose a *Modeling as a Service* (MaaS) initiative for cloud computing projects with an emphasis on topics like scalability, tool interoperability, and the definition of modeling mash-ups as a combination of MDE services from different vendors. Moreover, the definition of domain-specific languages (DSL) [10], [26], [27] along with the development of tools which support the developer during the DSL conception [9], [17], [26] are currently under investigation. However, none of the above mentioned approaches is related to public resource computing nor can it be directly used for the modeling process of BOINC projects. In the following chapters we therefore propose a first modeling language approach for the abstraction of the BOINC framework.

[1]Trickle messages are asynchronous, ordered, and reliable messages between the BOINC server/clients and let applications communicate with the server during the execution of a workunit.

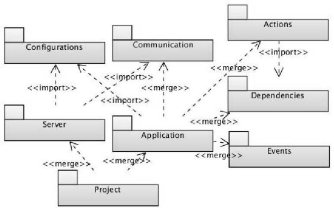

Fig. 1. Logical packages and dependencies of the BOINC functionalities

III. Abstraction of the BOINC Framework

The BOINC framework offers many very useful functionalities that help the developer to create his application. As a first step towards our modeling language approach we have subdivided the BOINC functionalities into different logical packages as shown in Fig. 1. Each package contains functions that cover a specific aspect during the development process [22] and can be used independently from functions of other packages which minimizes the number of dependencies. The whole BOINC project including the server installation components and application specific implementations is contained in the package *Project*. This package directly depends on the packages *Server* and *Application*. The package *Application* contains all application specific implementations and depends on the following child packages:

- *Events* - This package describes the abstractions for all possible events that can occur during the execution, e.g. exceptions like *'File not found'* or *'Segmentation fault'*. It contains routines for clearly defined error handling.
- *Actions* - Every execution statement is gathered within this package including wrapper routines to call third-party applications, i.e Matlab, or other domain-specific simulation tools.
- *Dependencies* - All libraries that are needed for the whole project are contained in this package. This includes the BOINC libraries as well as application specific runtime libraries and external sources.

- *Communication* - This package includes the BOINC core client component which is the interface between the client installations and project servers and performs the information exchange between them.
- *Configurations* - In this package one keeps all system relevant parameters coded in XML files.

The complete server installation and maintenance process is provided by the package *Server*. This package describes the relationships between all components of a complete BOINC server installation, i.e. all configuration files, a list of the parameters for the workunits, description of installed applications with the corresponding architecture and processor targets. The package *Server* imports its required information from the contents of the *Configurations* and *Communication* packages.

IV. The modeling language approach for the abstraction of BOINC

Nowadays, models and model-based techniques are the fundamental means by which engineers are able to cope with otherwise unmanageable complexity and reduce design risk. In particular, software models have the distinct advantage that they can be *evolved* from high-level views of possible designs into actual implementations.

The *Unified Modeling Language (UML)* – a widely adopted, widely supported and customizable industry standard – plays a key role in modern software development. With the possibility of creating standardized UML profiles it provides the fundamentals for a true engineering-oriented approach to the construction of software. That is, system models can be used to understand and assess designs and predict design risks in meaningful (e.g., quantifiable) ways. Full automatic code generation from UML models facilitates preservation of proven model properties in the final implementation.

In our approach to a model-based development of BOINC projects we focus on the use of quasi standard software tools available within the Eclipse development environment. These tools enable us to develop all necessary components, like diagram editors, including graphical representations of modeling elements, code generators, etc. in one and the same development environment. Specifically, the code generation should be realized using a template engine which could also be used to generate important documentation files.

A. Graphical modeling environment

Throughout our project we exclusively use the Eclipse Modeling Framework (EMF) as released by the Eclipse Model Development Tools (MDT) project [26]. A detailed description of all components can be found in [9]. A key feature of our approach is the definition of a suitable standardized UML2 profile [22]. Here, we make use of the EMF-based implementation of the *UML2* and *UML2 Tools* subproject. EMF specifically allows implementing of constraints based on the Object Constraint Language (OCL) [18].

Fig. 2 gives an overview about a simplified modeling process. Here, the developer uses graphical modeling elements within diagrams to design an application as described in Sec. IV-B below. For a more detailed modeling of the application logic the developer is required to use a textual DSL as described in Sec. IV-D. The *Graphical Model* will be implemented using the Graphical Modeling Framework (GMF). GMF includes EMF, the Graphical Editing Framework (GEF) and Draw2D. GEF is a framework built upon the Model-View-Controller (MVC) pattern and handles the view and logical components with own instances. The controller handles the logic between these instances. The *Textual Model* and the *Graphical Model* are synchronized, i.e. every change in one of the models causes changes in the other one and each model will automatically be adjusted. Both models will be transformed into one (yet to be defined) so-called *VisualGridML Genmodel*. This can be done by exploiting the model-to-model transformation framework Query/View/Transformation (QVT). The QVT operation mapping language is capable of dealing with multiple input and output models and also supports OCL statements.

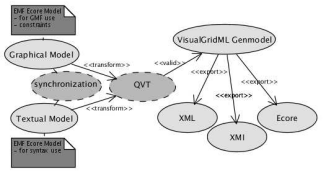

Fig. 2. Simplified view of the modeling process

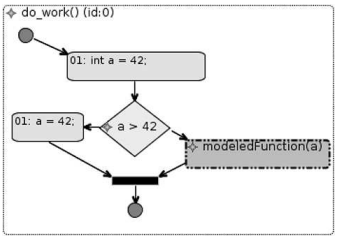

Fig. 3. Example flowchart diagram for instruction sequences.

One of the key issues of our approach is to define a proper *VisualGridML Genmodel*. The *VisualGridML Genmodel* can be exported into different other formats, e.g. XML, XML Metadata Interchange (XMI), or Ecore. The *VisualGridML Genmodel* will only be generated when the previously created models are valid. In order to support the developer during the verification and validation process adequate error messages and output comments will be created.

B. Graphical modeling elements

The proper definition of suitable graphical modeling elements is vital for our modeling language approach for the

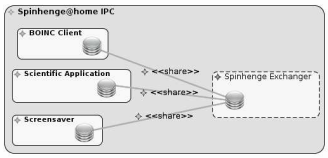

Fig. 4. Interprocess Communication within a BOINC application

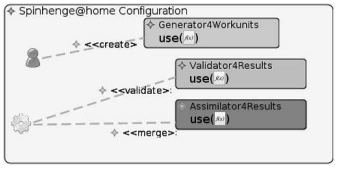

Fig. 5. Elements for configuration purpose

abstraction of BOINC. In the following we present some examples along with sample modeling fragments. Fig. 3 shows a flowchart of a high-level description of instruction sequences. The graphical notation is adapted from the UML2 flowchart definition [17, Fig. 12.36].

Fig. 3 shows an example that uses seven graphical modeling elements. As mentioned above these modeling elements are defined in the *Actions* package and have the following meaning [22]:

- *Start*, which describes the entry point of an application.
- *Stop*, which defines the end of execution, i.e. after that no other instructions will be executed and the application will shut down.
- *Action* is a modeling element which can execute native C/C++ instructions.
- *Decision* describes an `if-else` condition.
- *Join* merges two or more subdivided instruction sequences.
- *Execute* executes external C/C++ functions which are implemented in header and source files.

In Fig. 4 we show an adapted version of a UML2 collaboration diagram [17, Tab. 9.1, 9.2]. The dashed rounded rectangle on the right hand side specifies a shared data space. This data space is defined using one ore more port descriptors. Different ports could include different data descriptions and could also be connected with different so-called handlers. The three elements on the left hand side are examples of such handlers. Each handler handles a specific functionality and is connected with one application implementation. This example contains handlers for the *BOINC Core Client*[2], a scientific application, and a component for a screensaver session [1]. The handler can have only one data port. Data ports can only be connected to data ports of shared data space. Connections between ports may also be named as shown in this example.

In Fig. 5 we show an adapted version of a UML2 Use-Case diagram [17, Fig. 16.10] which contains elements of the *Configuration* package [22]. The elements on the left hand side are the actors for the use cases of the right hand side. The use cases contain a reference to predefined functions which can be modeled with a flowchart diagram. The dashed line connects the use cases with the appropriate actor and defines the type of execution. The lower actor describes the BOINC server which could be a cronjob. The upper actor describes a user. In this example the user defines workunits.

C. Aspect-Oriented Programming and Feature-Oriented Programming

Aspect-Oriented Programming (AOP) aims at separating and modularizing cross-cutting concerns [12]. The idea behind AOP is to implement so-called cross-cutting concerns as *aspects* and the core (non-cross-cutting) features are implemented as *components* [5]. Using *pointcuts*[3] and *advices*[4], an aspect weaver glues aspects and components at *join points* together. Fig. 6 shows on the left hand side (1) two aspects (A1 and A2) which extend the class definitions (C1 and C2). In AOP aspects can be added to the programming logic where ever functions are called. It is also possible to replace any functionality dynamically or to define the order of execution by precedence.

In Feature-Oriented Programming (FOP) the program functionalities can extended during the compilation and execution process, e.g. two or more functions can be combined to create extented features [7]. Fig. 6 shows on the right hand side (2) a simple overview of FOP. Here, F4 inherits the properties and methods of F1. Additionally, F6 refines F4, which can be done during runtime or while the compiling process creates the application.

Aspects and features in their current representation are intended for solving problems at different levels of abstraction [4], [14], [16]. Whereas aspects in AspectC++ [24] act on the level of classes and objects in order to modularize cross-cutting concerns, features act on the software architecture level [3].

For our approach, we are expecting that AOP and FOP are suitable methodologies for dynamic binding of applications as stated in Sec. IV-E. For example, the BOINC project result file format must be defined and created manually by the scientist or developer for a specific scientific application. This file can be generated during the code generation process as soon as definitions of the BOINC validator and BOINC assimilator are existing. After that, the BOINC project can be deployed with error-free validator and assimilator configurations.

D. Domain-specific language for BOINC

Domain-specific languages (DSL) are language definitions tailored to the development needs of specific problem domains

[2]The BOINC Core Client communicates with schedulers, uploads and downloads files, and executes and coordinates applications.

[3]The point of concern to execute an *advice*.

[4]Additional code that should apply to the existing model.

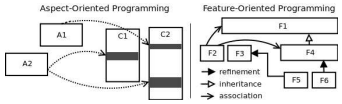

Fig. 6. (1) Two aspects extend two classes, (2) Two features refine two other features

[10]. For example the Structured Query Language (SQL) is designed for database queries. Another vital part of our modeling language approach to BOINC is a proper definition of a DSL for BOINC applications. Here, a key issue is the use of Xtext which allows to create an application by code generation. Xtext offers in combination with Xpand a template-based code generation engine [26]. By using Xtext and Xpand is has been shown that it is possible to create a complete BOINC application [21].

Generally speaking, all BOINC functionalities can be defined using a set of specific language elements. Our model-driven approach allows to develop applications independently of the target type, i.e. for CPU or GPU targets. Furthermore, BOINC offers various diagnostic parameters which enable or disable checks, e.g. *memory leaks* or *heap violations*. With the help of DSL statements these specific options can enabled or disabled for a subsequent automatic code generation. For example, the following code fragment defines a single processor environment which enables the above mentioned diagnostic flags:

```
target cpu single;

diagnostics {
 dumpcallstack
 heapcheck
 memoryleakcheck
 redirectstderr
 tracetostderr
}
```

In order to create applications with multicore or GPU computing support the following statement can be used:

```
target cpu mode multi with 10;
  // or
target gpu dim(10) block(4);
```

The first line describes a multi-thread application with up to 10 threads. The last statement enables the support of GPU computing. In GPU computing the process is splitted into *dim* threads, executed in 4 *blocks* [13]. It is also possible to include manual implementations or third-party libraries. The following examples show this in more detail. The required dependencies to third-party libraries or functionalities could be also defined with only a few lines of code. This code is used for the "make" process of an executable application. On Linux or Unix like operating systems a makefile is generated whereas on Windows systems it is possible to generate different project files which can be imported by integrated development environments like Visual Studio or Eclipse. Using this feature one can realize a *platform-independent* approach.

```
includes AppInclude {
 "~/boincadm/framework"
 "~/boincadm/src/api"
 "~/boincadm/src/lib"
}

libraries AppLibrary {
 "/lib", "pthread"
 "~/boincadm/framework", "visualgrid"
 "~/boincadm/src/api", "boinc_api"
 "~/boincadm/src/lib", "boinc"
}
```

It is common to use parameter files for the BOINC applications. The native way of doing this is to create *mapping files* on the server with a few parameters. Whenever clients connect to the server, they retrieve the application and all necessary parameter files. The following DSL fragment describes this procedure using the `infile` statement. The content is specified as an XML tree and could be iterated with a reference, e.g. `ObjectName1`. As a consequence of, binary data must be base64 encrypted [25]. Each reference contains the parameter as a *struct* or *class* definition.

```
infile "metropolis_data.xml"
                as ObjectName1;
infile "param.jj" as ObjectName2;
infile "param.nn" as ObjectName3;
infile "param.ww" as ObjectName4;
```

After the execution of the client application the results are stored in *result files* defined by the statement `outfile` and are uploaded to the server.

```
outfile "metropolis_out.erg"
                as ObjectResult1;
```

In general there exist several ways using a modeling process to develop a certain client application. As a matter of fact, every developer differs in his way to develop applications and it is necessary that the DSL supports this variety. For example, it is possible to link third-party libraries to the application using DSL statements. Furthermore, the application code can also be directly implemented into the `worker` part. The following DSL fragment contains an example which is used in [21]. Here, the `worker` starts the environment for execution instructions with the name `Spinhenge`. The statement `exec` defines pointcut expressions which are used by the AOP weaver process to deploy the scientific application. In this definition, the pointcut expression describes the working function in Fig. 8.

```
worker Spinhenge {
 exec "void %::Spinhenge::doWork(...)";
}
```

This statement could be replaced by other instructions, e.g.

```
worker Spinhenge {
 cpp {
  int a = 42;
 }
 action(modeledFunction(a));
}
```

Here, `cpp` starts an inline code area which contains native C/C++ statements. The variable `a` is available right after its definition and optional initialization within the context. It can be used by DSL defined functions like `modeledFunction(variable)`. Third-party applications can be executed using the DSL statement `wrapper`. It allows to call an external application, so called *legacy application*, and only one call is allowed to realize. Optional parameters for the application can be set in the *Configuration* package.

```
worker Spinhenge {
  wrapper("Matlab", "Argv[1] Argv[2]" [,
       weight, checkpoint_filename,
       fraction_done_filename, ...]);
}
```

These optional parameters are defined by the wrapper interfaces [11], [15], [19].

```
screensaver Spinhenge {
  render "% Screensaver
             ::Spinhenge::doWork(...)";
}
```

During the execution of an application different events can occur, e.g. events which could also be logged for diagnostic analysis in the above mentioned definitions. Furthermore an exception handling is described by the following DSL definition:

```
handle TypeOfException (: optionalName) {
 /* to be defined handler */
}
```

Predefined exception handlers are reusable by other exceptions. This is enabled by using the `ref` statement which uses the optional name `optionalName` of the previous example.

```
handle AnotherTypeOfException
   ref optionalName;
```

The BOINC framework uses interprocess communication (IPC) to exchange data between different application instances, e.g. scientific application, and screensaver. The native way to use IPC needs the definition of shared variables, e.g. in a C/C++ `struct` or `class` definition and furthermore different functions which handles these variables. A strict well-formed definition would be easier to use and reduces the effort of changing code parts in the source files. The DSL statement `exchange` describes the structure of the IPC with C/C++ language elements like datatypes which are usable in every application. The keyword `feature` defines an AOP pointcut expression which is used to assign values to the variable on the left hand side, e.g. `update_time` keeps the delta in milliseconds between the value updates. The listed AOP pointcut expressions are implemented in the *Visual Grid Framework* which is described in Sec. IV-E.

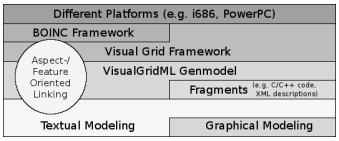

Fig. 7. *Visual Grid Framework* Abstraction Layer

```
exchange {
 double update_time :
   feature "% Boinc::updateTime(...)";
 double fraction_done :
   feature "% Boinc::fractionDone(...)";
 double cpu_time :
   feature "% Boinc::cpuTime(...)";
}
```

To handle BOINC Trickle Messages the `TrickleUp` and `TrickleDown` commands are stated in the following listing. The `TrickleUp` needs two AOP pointcut expressions to check if a trickle up must handled and the trickle up handler itself.

```
 TrickleUp "bool checkTrickleUp(...)"
     do "% handleTrickleUp(...)";
 TrickleDown "% handleTrickleDown(...)";
```

The handler defined by `TrickleDown` is called frequently to manage the incoming messages by the BOINC server, e.g. command to abort one workunit or informations of the current BOINC credit points.

Furthermore, general descriptive information about the application can be defined with the statement `info`.

```
info {
 author="Christian Benjamin Ries";
 email="cries@fh-bielefeld.de";
 license="FH Bielefeld";
 description="Spinhenge@home Example";
 project="Spinhenge@home";
 version="3.16";
}
```

E. Visual Grid Framework

Fig. 7 shows the proposed *Visual Grid Framework* layer which is defined between the layer that describes the underlying computer hardware, the BOINC framework, and the *VisualGridML Genmodel*. The *Visual Grid Framework* offers a less complex access to the BOINC functionalities and handles the creation of applications for different platform targets, e.g. Intel 686 based 32-bit or 64-bit architectures.

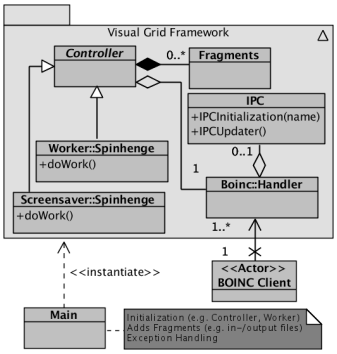

Fig. 8. Composition of the Visual Grid Framework Approach

The left side describes one approach to generate the function calls and logical program parts between the textual-/graphical modeling tools, fragments, *VisualGridML Genmodel*, *Visual Grid Framework*, and BOINC Framework. In Fig. 8 we show how the *Visual Grid Layer* works.

The `Main` class is generated by the code generation process and contains the implementation of the starting routines. This routine instantiates the interface to the BOINC framework as well as to the scientific application. The entry point of the scientific application is called using the *doWork()* functions of the *Worker::Spinhenge* and *Screensaver::Spinhenge* classes. These two classes are specializations of the abstract *Controller* class. The *Controller* class keeps track of all necessary data management, e.g. input, and output data files, checkpoint definitions, etc. The *IPC* class (Interprocess-Communication) is completely generated and implements the initialization and update routines. This class is used by the *Worker::Spinhenge* and *Screensaver::Spinhenge* classes for the exchange of data. The external *BOINC Client* class is an actor and therefore represents just the interface to the client. The BOINC Client has the full control of the executed applications and processes.

As a consequence, there exist two ways to generate an application within the *Visual Grid Framework*, (1) creation of a class which is derived from the abstract *Controller* class, (2) the definition of AOP *joinpoints* and *advices* as defined by the DSL in Sec. IV-D.

As a first test of our approach we have created an application which is similar to the BOINC sample to perform a transformation of lowercase to uppercase texts. The BOINC sample is based on approximately 400 lines of code, including the makefile, C++ header, source files, and 31 BOINC specific function calls. In contrast to this, our generated application contains only about 90 lines of code, including 60 lines of DSL code, and 30 lines of application specific code which performs the transformation. All defined dependencies, e.g. for the initialization of the process or exception handling, are resolved by generating files. Furthermore, a makefile is generated which on execution builds the complete application for the defined client platform architecture.

V. Conclusion

We have presented a first modeling language approach for developing Public Resource Computing applications on the basis of BOINC. We have demonstrated that the complete BOINC framework can be divided into a few logical packages that provide the necessary graphical and textual model elements to allow for a model-based development of applications with subsequent source code generation. Our approach is technically realized by using standardized and well-defined technologies [26]. Thus far, we have just implemented a small part of the BOINC functionalities. Key features like graphical and textual modeling elements, the *VisualGridML Genmodel* have been defined to an extent that we could show the general feasibility of our approach. However, further investigations with regard to the abstraction of the system architecture, dependencies, error checking, and the transformation to different target languages are needed. We have successfully performed a first test of our approach by modeling an existing BOINC application with just a few lines of DSL code using external libraries for the core computational routines and abstraction of the BOINC functionalities. However, most steps of our modeling process are still performed manually, and we are currently working on the creation of a unified development environment that supports the wide range of technologies, including a one and only graphical modeling framework which minimizes the need of textual modeling fragments, automatic dependencies resolving, completely error-free code generation, and higher support for legacy applications.

References

[1] D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, 5th IEEE/ACM International Workshop on Grid Computing. November 8, 2004, Pittsburgh, USA
[2] D. P. Anderson, C. Christensen, and B. Allen, *Designing a Runtime System for Volunteer Computing*, IEEE Computer, 2006
[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, *FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++*, Technical Report, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005
[4] S. Apel, T. Leich, and G. Saake, *Aspectual Mixin Layers: Aspects and Features in Concert*, In Proceedings of International Conference on Software Engineering (ICSE), 2006
[5] S. Apel and D Batory, *When to Use Features and Aspects? A Case Study*. In Proceedings of ACM SIGPLAN 5th International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, October 2006
[6] H. Brunelière, J. Cabot, and F. Houault, *Combining Model-Driven Engineering and Cloud Computing*, AtlanMod, INRIA RBA Center & EMN, France, Nantes, 2010
[7] D. Batory, J. N. Sarvela, and A Rauschmayer, *Scaling Step-Wise Refinement*, IEEE Transactions on Software Engineering (TSE), 30(6), 2004
[8] T. Estrada, M. Taufer, and D. P. Anderson, *Performance Prediction and Analysis of BOINC Projects: An Empirical Study with EmBOINC*, in J Grid Computing, Springer, 2009
[9] R. C. Gronback, E. Gamma, L. Nackmann, and J. Wiegand, *Eclipse Modeling Project, A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley, 2009, ISBN: 978-0-321-53407-1

[10] A. Hessellung, *Domain-Specific Multimodeling*, IT University of Copenhagen, Denmark, 2008
[11] P. Kacsuk, J. Kovacs, Z. Farkas, A. C. Marosi, G. Gombas and Z. Balaton, *SZTAKI Desktop Grid (SZDG): A Flexible and Scalable Desktop Grid System*, Journal of Grid Computing, 2009
[12] G. Kiczales et al., *Aspect-Oriented Programming*, In Proceedings of European Conference ob Object-Oriented Programming (ECOOP), 1997
[13] D. Kirk, and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*, Morgan Kaufman Publ Inc, 2010, ISBN: 978-0123814722
[14] K. Lieberherr, D. H. Lorenz, and J. Ovlinger, *Aspectual Collaborations: Combining Modules and Aspects*, The Computer Journal, 46(5), 2003
[15] A. C. Marosi, Z. Balaton, and P. Kacsuk, *GenWrapper: A Generic Wrapper for Running Lagacy Applications on Desktop Grids*, 3rd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2009), 2009 May, Rome, Italy
[16] M. Mezini and K. Ostermann, *Variability Management with Feature-Oriented Programming ans Aspects*, In Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2004
[17] OMG Adopted Specification formal/2009-02-02, *OMG Unified Modeling LanguageTM (OMG UML)*, Superstructure, Version 2.2, OMG, 2009
[18] OMG Adopted Specification formal/2010-02-01, *OMG Object Constraint Language*, Version 2.2, OMG, 2010
[19] C. B. Ries, and C. Schröder, *ComsolGrid - A framework for performing large-scale parameter studies using Comsol Multiphysics and BOINC*, COMSOL Conference, Paris, France, 2010
[20] C. B. Ries, *Performance measuring and automatic calibration of BOINC installations*, University of Applied Sciences Bielefeld, Germany, unpublished
[21] C. B. Ries, T. Hilbig, C. Schröder et al., *Spinhenge@home - Monte Carlo Metropolis*, Version 3.16, University of Applied Sciences Bielefeld, Germany, http://spin.fh-bielefeld.de
[22] C. B. Ries, T. Hilbig, and C. Schröder, *UML 2.2 Profile: Visu@lGridML*, University of Applied Sciences Bielefeld, Germany, unpublished
[23] C. Schröder, "Spinhenge@home - in search of tomorrow's nanomagnetic application", *to appear in Distributed & Grid Computing - Science Made Transparent for Everyone. Principles, Applications and Supporting Communities*, 2010
[24] O. Spinczyk, D. Lohmann, and M. Urban, *Advances in AOP with AspectC++*, Software Methodologies, Tools and Techniques (SoMeT 2005), IOS Press, September, 2005, Tokyo, Japan
[25] T. Imamura, B. Dillaway, and E. Simon, *XML Encryption Syntax and Processing*, W3C, December, 2002, http://www.w3.org/TR/xmlenc-core
[26] Xtext - programming language framework, Xpand - a template language, http://www.eclipse.org/modeling/mdt
[27] U. Zdun, *Concepts for Model-Driven Design and Evolution of Domain-Specific Languages*, In Proceedings of the International Workshop on Software Factories OOPSLA, pp. 1-6, October, 2005

Excerpt from the Proceedings of the COMSOL Conference 2010 Paris

ComsolGrid – A framework for performing large-scale parameter studies using COMSOL Multiphysics and the Berkeley Open Infrastructure for Network Computing (BOINC)

Christian Benjamin Ries[1] and Christian Schröder[1]
[1]Department of Engineering Sciences and Mathematics, University of Applied Sciences Bielefeld, Wilhelm-Bertelsmann-Straße 10, 33602 Bielefeld, Germany

Abstract: BOINC *(Berkeley Open Infrastructure for Network Computing)* is an open-source framework for solving large-scale computational problems by means of public resource computing (PRC). In contrast to massive parallel computing, PRC applications are distributed onto a large number of heterogeneous client computers connected by the Internet where each computer is assigned an individual task that can be solved independently without the need of communication upon the clients. Nowadays even small companies hold remarkable computer resources which are not accessible as a whole but distributed over the numerous computers which belong to the standard working environment in today's companies. Over the day, these computers are rarely operating at full capacity and hence valuable computational power is just wasted.
Here, we present a new approach for performing large-scale parameter studies using COMSOL Multiphysics based on the BOINC technology which can be used to utilize idle computer resources that are connected within a companywide intranet. Based on our approach we provide a tool chain called ComsolGrid that allows the COMSOL Multiphysics user to define parameter study by means of a high-level description. ComsolGrid then automatically performs the complete parameter study.

Keywords: BOINC, Grid Computing, large-scale Parametric Studies

1. Introduction

With the help of BOINC [1, 6] it is possible to create and run self-made scientific applications heterogeneous computer networks. However, one can also run legacy applications like Matlab [8] by using so-called wrapper functions within BOINC for performing parameter studies on distributed computer networks [2]. Based on this idea, we have created a framework called ComsolGrid to perform distributed COMSOL Multiphysics *(hereafter referred to as COMSOL)* simulations. The key idea is to run COMSOL simulations with predefined parameter values on different computers. The COMSOL simulation model and parameter values are packed in so-called *workunits* which a server computer sends to the client computers. Each client performs a single simulation according to the parameter set given and returns the result to the server computer. The BOINC project server keeps tracks of the

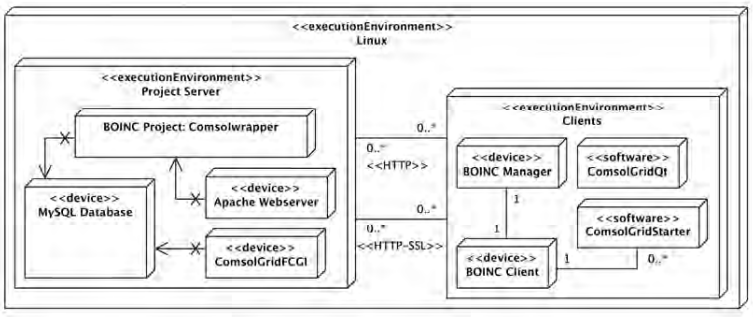

Figure 1: Software architecture of *ComsolGrid*.

workunits and checks whether each result is valid or not (within the so-called *validation* process). Validated results are stored in specified locations on the server computer during the so-called *assimilation process*.

A COMSOL simulation can be started using the within a *batch mode* [7], i.e. a command-line based program call which bypasses the use of the common graphical user interface (GUI). The command-line parameters in the *batch mode* allow the user to control the simulation process. For example, it is possible to define the target processor type, i.e. 32-/64-bit architecture. Furthermore, one can define the parameter names and parameter values which should be used by the COMSOL simulation. This enables one to perform a parameter study based on one specific simulation model by varying the parameter values for each simulation in a distributed computer environment. Within ComsolGrid the complete configuration and communication management is defined by data structures based on the Extensible Markup Language (XML).

2. ComsolGrid – Components, Architecture, and Definitions

Fig. 1 shows the software architecture of ComsolGrid and gives an overview of the developed software components. There is one component which describes the complete infrastructure of ComsolGrid. Currently, only Linux installations are supported. On the left hand side the project server is shown. This server is responsible for the correct handling of user requests, the creation process of new workunits and the validation and assimilation of completed simulations. On the right hand side the client environment is shown. This environment includes the user interfaces to enable the maintenance process; in order to create new parameter studies and to supervise the server system conditions clients can perform COMSOL parameter simulations as well.

As shown in Fig. 1, there are eight applications defined: **(1)** the BOINC project itself, which is named *"Comsolwrapper"*, **(2)** a MySQL database to store information about users and host computers of the network, **(3)** an Apache web server installation which provides the interface to the client computers for the exchange of information (distributing workunits, gathering results), **(4)** *ComsolGridFCGI* which is described in more detail in Section 2.1, **(5)** the BOINC Manager providing the GUI on each BOINC client computer, **(6)** the BOINC client which performs the actual COMSOL simulation, **(7)** *ComsolGridStarter* is an application which enables the BOINC Manager to start, stop, and pause COMSOL simulations, (see section 2.2), and **(8)** the tool *ComsolGridQt*, which provides the GUI for managing the complete parameter study (see section 2.3).

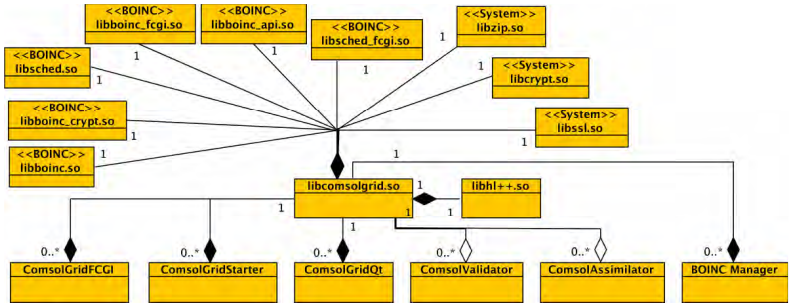

Figure 2: Architecture of software libraries used in the *ComsolGrid framework*. Most of them are implemented by the BOINC framework (indicated by the stereotype: *BOINC*), four libraries are external software libraries (*libhl++.so*, indicated by the stereotype: *System*). The library *libcomsolgrid.so* in a composition of the core functions which are used by all ComsolGrid applications listed at the bottom of the figure. For ComsolGrid the BOINC Manager has been modified in order to handle COMSOL specific parameters.

The server and client processes can have two network communication channels. One channel is encrypted using the Secure Socket Layer (SSL) and the other one is based on plain-text. The BOINC clients do not need an encrypted channel to receive new workunits. For this purpose, BOINC has implemented a hash key verification system. The BOINC project creates a hash key from a combination of the user's personal password and e-mail address and is only valid for one machine. Only this hash key is exchanged between the BOINC clients and the project server and does not enable permissions to administrate the BOINC server. The encrypted channel is used by the user to create new parameter studies.

Fig. 2 shows the software architecture of the *ComsolGrid framework*. At the top the BOINC dependencies to internal and external software libraries are shown. These libraries are used by the software library *libcomsolgrid.so*. Furthermore, *libcomsolgrid.so* is used by the ComsolGrid applications which are implemented for the *ComsolGrid framework* and are described in the following sections 2.1 to 2.4.

2.1 ComsolGridFCGI – Server interface for Parameter Studies

The *ComsolGridFCGI* is an interface for the

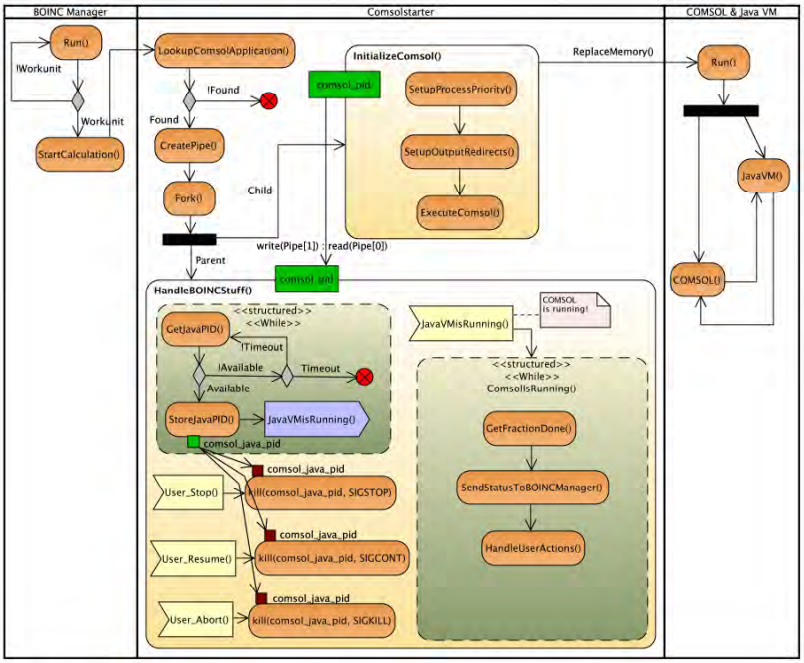

Figure 3: Architecture of the *ComsolGrid* wrapper: **(BOINC Manager)** checks for existing workunits and calls *Comsolstarter*, **(Comsolstarter)** is our wrapper implementation, **(COMSOL & Java VM)** is the native COMSOL Multiphysics application process.

user. The interface is installed on the project server computer and is directly connected to the MySQL database using the FastCGI library [3]. FastCGI provides an easy to use application programming interface to implement server applications with a full support of web communication by plain-text communication protocols like Hypertext Transfer Protocol (HTTP).

Each framework user must be assigned a correct user role. User roles are an extension to the BOINC framework. We have defined three roles: **(1)** *Administrator*, **(2)** *Developer*, and **(3)** *Scientist*. Users assigned the role *Administrator* can start, stop and maintain the BOINC project server. The *Developer* role defines a user who can do everything, just like a Linux root administrator. Users assigned the role *Scientist* can create and run new parameter studies; they are allowed to define or change the COMSOL simulation model, parameter names and values, and they can upload this configuration to the BOINC project server. In Section 2.4 we describe details of the GUI which allows a high-level description of a new parameter study.

2.2 ComsolGridStarter - BOINC Wrapper for COMSOL Multiphysics

The *ComsolGridStarter* is a wrapper routine handling the BOINC client and BOINC user commands which are used to control a COMSOL application. The BOINC user can send commands using the BOINC Manager; these commands are sent via predefined BOINC communication channels [1]. There exists one channel between the BOINC Manager and the BOINC client and another one between the BOINC client and each running scientific application.

At the bottom of the Fig. 3, the user commands are visualized. Any user of a client computer can stop, resume, and abort the COMSOL simulation on that machine. The stop request is mapped to the Linux signal **stop** *(SIGSTOP)*. The Linux signal **stop** suspends one process. Suspended processes can be resumed by the Linux signal for **continuation** *(SIGCONT)* and **aborted** by the Linux kill signal *(SIGKILL)*. However, in principle one can configure ComsolGrid in a way that the simulations cannot be controlled by the user in order to ensure that a certain amount of computing power of a client machine is always used for ComsolGrid simulations.

We have implemented our own wrapper routine since all available wrapper routines [4, 5, 6] could not handle the COMSOL process tree. In fact, on Linux systems COMSOL internally starts several instances of the Java virtual machine to perform a simulation. Table 1 show the process trees of the COMSOL major versions 3.5x and 4.x. The available wrapper routines can only handle the process identification numbers (PID) of the COMSOL master processes with the PID 30521 and 15895. As a result, these processes can be controlled by operating system signals; however their child processes cannot be manipulated directly by the BOINC Client. As a consequence we have implemented a bridge between the BOINC client and the COMSOL process tree. Fig. 3 describes three processes: **(1)** *BOINC Manager*, **(2)** *Comsolstarter*, and **(3)** *COMSOL & Java VM*. The first one is responsible for the communication between the BOINC user and the BOINC client and starts the *ComsolGridStarter* for performing a simulation based on a given workunit. The second one is the most important and more complex process. *ComsolGridStarter* first checks for a valid COMSOL installation on the client computer and then executes it. It is necessary to start the correct version of COMSOL, because the model files are usually not downward compatibly with other COMSOL major version, i.e. a COMSOL model of the major version 4.x cannot be opened by COMSOL version 3.x. The routine *IntializeComsol()* starts a COMSOL process and passes its PID to the routine *HandleBOINCStuff()*. *HandleBOINCStuff()* polls the state of the internally started Java virtual machines when it becomes valid. The term valid means that the PID of the Java virtual machine is available, e.g. table 1 shows the required PIDs: (1) **comsol_java_pid= 30674** for the process named *java*, and (2) **comsol_java_pid=15919** for the process named *comsollauncher*. If one of them becomes valid, the PID is stored for later use by the handler of the user commands. The wrapper runs until the COMSOL process is finished. With these PIDs, the complete COMSOL process becomes controllable.

During runtime, the wrapper tries to retrieve information about the progress' progress and sends the value to the BOINC Client. This value

is then transferred to the BOINC Manager and displayed as shown in fig. 4.

The third process on the right hand side in fig. 3 shows the native COMSOL process. When it starts, it forks the COMSOL process and the simulation execution is supported by Java virtual machines.

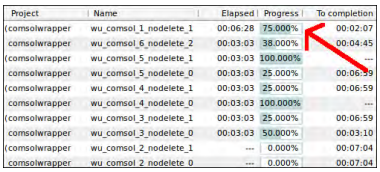

Figure 4: Progress in percent of one COMSOL Multiphysics simulation.

Our wrapper routine is fully configurable using an XML configuration file. It is possible to define the current processor architecture or which license should be used. In addition, we can map the output and input channels to files, i.e. for debugging purposes.

2.3 ComsolGridQt – User interface for COMSOL Multiphysics user

We have implemented a GUI *ComsolGridQt* which provides a tool that enables the user to define a new parameter study by means of a high-level description. As shown in fig. 5 the GUI guides the user through the process of creating a complete parameter study. When run for the first time, the user must add the authentication data, i.e. user name, e-mail address, and password. In the next step it is necessary to define the web address of the server interface *ComsolGridFCGI*, e.g. *127.0.0.1/comsolfcgi*. After that basic information about the project server is requested; the user receives a list of the available BOINC projects, data of the server condition, and information of the supported COMSOL versions and plattform.

The GUI includes some well-defined elements for adding COMSOL specific simulation model files, changing the order of these files, and adding the BOINC input/result template files. These files contain XML structures. When a new model is added, the GUI automatically determines the parameter names which are defined in the model and adds these names as column descriptions to a table in the *Parameter* tab. After that the user can define parameter values for each parameter name. After every change of these values, it is automatically checked whether the values are valid or not. Four values are required for a valid state. The format string of these values is:

"START:STOP:STEP:DEFAULT"

This format is modified with regard to the COMSOL *range(…)* function. Each of these values can keep one data item as a floating-point value. The user can add arbitrarily many new rows of data items.

The input template and result template files are not generated but must be created manually. In our cases, we are using only one COMSOL simulation model for each parameter study with different parameter ranges. As a result of this, we have only two files and reuse them for each new parameter study.

Table 1: Example of a process tree of the two COMSOL Multiphysics major versions: (1) 3.5x, and (2) 4.x.

(1) COMSOL Multiphysics 3.5x process tree:

```
su(30443) --- bash(30452) --- comsol(30521) --- java(30674) -+- {java}(30675}
                                                             |- {java}(30676)
                                                             `- …
```

(2) COMSOL Multiphysics 4.x process tree:

```
su(13564) --- bash(13576) -+- comsol(15895) --- comsollauncher(15919) -+- {comsollauncher}(15020)
                           |                                           |- {comsollauncher}(15021)
                           |                                           `- …
```

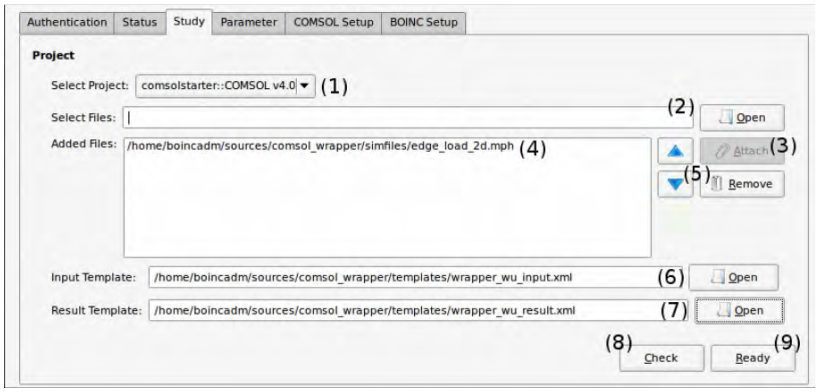

Figure 5: The GUI *ComsolGridQt* provides a high-level description of a parameter study: **(1)** select the available BOINC project, **(2, 3, 4, 5)** select, add, and arrange COMSOL Multiphysics files and define the according parameter data ranges using the *Parameter* tab, **(6, 7)** add the input template and result template file, and **(8, 9)** check your configuration and upload the files and parameter data ranges to the BOINC project server.

Fig. 6 shows an example for a complete specification of a workunit. As one can see, the workunit uses three external components: **(1)** the *COMSOL model file (*.mph)*, **(2)** the *Input Template* file, and **(3)** the *Result Template* file. The *Input Template* file defines the set of the COMSOL simulation files, the according file of the parameter values and the BOINC logical names of these files, e.g. *comsol.mph*, and *comsol.txt* (described later). It does not matter which name a COMSOL simulation file has been assigned, it is always mapped to the name *comsol.mph*. The *comsol.txt* file contains the parameter values.

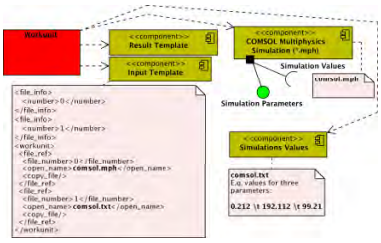

Figure 6: Components of a ComsolGrid workunit

Each row of this file describes a set of parameter values separated by tabulators. The file *comsol.txt* is created for each workunit by *ComsolGridFCGI* whenever *ComsolGridQt* uploads a new parameter study from the server.

2.4. ComsolGrid Validator and Assimilator

Two more applications are needed for the complete simulation process, namely the *comsol_bitwise_validator* and *comsol_copydir_assimilator*. Both applications are basically copies of the original BOINC validator and assimilator functions. The validator is responsible for checking whether or not a maximum processing time is reached or whether two results for the same workunit parameters are equal; if so, the result is valid, otherwise it is not valid and will be discarded. This procedure is a standard check used for public projects but can be dropped completely for projects using trustworthy computer networks like company intranets. The assimilator copies all valid results to a specified target directory that is accessible by users assigned the role *Scientist*.

3. Proof of Concept

We have set up a test installation and successfully performed a parameter study using a number of workunits. For this installation, we exported one COMSOL installation by using the Network File System (NFS) to other computers. The computers are equipped with the BOINC Client and BOINC Manager software only. Each machine is registered to the above mentioned BOINC project *"Comsolwrapper"*. Whenever a BOINC client requests a new workunit, the BOINC project searches for parameter values that have not been processed already, and sends these in form of a workunit to the requesting BOINC client.

Our test installation uses the model library example *falling_sand.mph*, which is provided by the COMSOL default installation *(Model Library / COMSOL Multiphysics / Fluid Dynamics / falling_sand)*. We have added two parameter names for the test study. Fig. 7 shows the parameter names: **(1)** *objWidth*, and **(2)** *objHeight*.

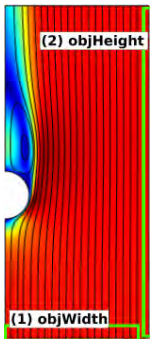

Figure 7: COMSOL Simulation model with the two parameters **(1)** *objWidth*, and **(2)** *objHeight* used for a parameter study with ComsolGrid.

The parameters have values ranging between 0.001 to 0.015 with a step size of 0.0005 for *objWidth* and 0.001 to 0.025 with a step size of 0.001 for *objHeight*. Currently there is no multiple variation scheme implemented; i.e. when one parameter is varied, the other parameters are fixed. We have created 54 workunits that have been validated using the bitwise validator which needs two results. Therefore, a total of 108 workunits have been processed by the client computers.

The parameter study took approximately 209 minutes with a success rate of 95 percent per run. It is feasible to reach 100 percent, but in our case five percent have been lost due to a malfunction of one computer where an X11 process has crashed.

4. Conclusions

In this article we have presented a new approach to perform large-scale parameter studies with COMSOL Multiphysics on a heterogeneous computer network using a tool chain based on the public resource computing framework BOINC. Our *ComsolGrid* approach enables one user to create parameter studies with an easy to use GUI. The implementation of our wrapper routine is very generic and therefore usable with other legacy applications as well. Further work is focused on the realization of a version for Mac OS X and Windows.

8. References

1. D. P. Anderson, C. Christensen, and B. Allen, *Designing a Runtime System for Volunteer Computing*, UC Berkeley Space Sciences Laboratory, Dept. of Physics, University of Oxford, and Physics Dept., University of Wisconsin - Milwaukee (2006)

2. O. Baskova, O. Gatsenko, G. Fedak, O. Lodygensky, and Y. Gordienko, *Porting Multiparametric MATLAB Application for Image and Video Processing to Desktop Grid for High-Performance Distributed Computing*, International Supercomputing Conference (ISC), Hamburg, Germany (2010)

3. M. R. Brown, *FastCGI Specification*, Open Market, Inc., 245 First Street, Cambridge, MA 02142 U.S.A, April, 1996, URL: http://www.fastcgi.com

4. D. Gonzales, F. Vega, L. Trujillo, G. Olague, M. Cardenas, L. Araujo, P. Castillo, K. Sharman, and A. Silva, *Interpreted Applications within BOINC Infrastructure*, May (2008)

5. A. C. Marosi, Z. Balaton, and P. Kacsuk, *GenWrapper: A Generic Wrapper for Running Legacy Applications on Desktop Grids*, 3rd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2009), Rome, Italy (2009)

6. BOINC – Open-source software for volunteer computing and grid computing, URL: http://boinc.berkeley.edu

7. COMSOL Installations and Operations Guide, Version 3.5a, 4.0, 4.0a, included in COMSOL Multiphysics installations

8. MathWorks, Accelerating the pace of engineering and science, http://www.mathworks.com

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. "SETI@home: an experiment in public-resource computing". *Communications of the ACM*, vol. 45, pp. 55–61, 2002. (Cited on pages 53 and 54.)

[2] G. Booch, A. W. Brown, S. Iyengar, J. Rumbaeugh, and B. Selic. "An MDA Manifesto." *Business Process Trends, MDA Journal*, 2004. (Cited on pages 32, 33 and 36.)

[3] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H.-L. Truong. "ASKALON: a tool set for cluster and Grid computing." *Concurrency and Computation Practice and Experience*, vol. 17, pp. 143–169, 2005 (Cited on page 27.)

[4] S. Danforth and C. Tomlinson. "Type theories and object-oriented programming." *ACM Computing Surveys (CSUR)*, vol. 20, pp. 93–127, 1988. (Cited on page 38.)

[5] I. Foster and C. Kesselman. "Globus: A Metacomputing Infrastructure Toolkit." *International Journal of High Performance Computing Applications*, vol. 11, pp. 115–128, 1997. (Cited on pages 10 and 25.)

[6] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. "Model-driven development using UML 2.0: promises and pitfalls." *Computer – IEEE Computer Society*, vol. 39, pp. 59–66, 2006. (Cited on pages 39 and 44.)

[7] L. Fuentes-Fernández and A. Vallecillo-Moreno. "An Introduction to UML Profiles." *European Journal for the Informatics Professional*, vol. V, pp. 6–13, 2004. (Cited on pages 35, 42, 44, 45, 46 and 47.)

[8] G. Giachetti, B. Marín, and O. Pastor. "Using UML as a Domain-Specific Modeling Language : A Proposal for Automatic Generation of UML Profiles." *Integration The Vlsi Journal*, pp. 110–124, 2009. (Cited on pages 3, 19 and 47.)

[9] T. Giorgino, M. J. Harvey, and G. De Fabritiis. "Distributed computing as a virtual supercomputer: Tools to run and manage large-scale BOINC simulations." *Computer Physics Communications*, vol. 181, pp. 1402–1409, 2010. (Cited on pages 63 and 142.)

[10] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. "PlanetLab: An Overlay Testbed for Broad-Coverage Services". *ACM SIGCOMM Computer Communication Review*, vol. 33, Issue 3, pp. 3–12, 2003. (Cited on page 25.)

[11] B. Henderson-Sellers. "UML? the Good, the Bad or the Ugly? Perspectives from a panel of experts." *Software & Systems Modeling*, vol. 4, pp. 4–13, 2005. (Cited on pages 22, 23 and 30.)

[12] P. Kacsuk, J. Kovacs, Z. Farkas, A. C. Marosi, G. Gombas, and Z. Balaton. "SZTAKI Desktop Grid (SZDG): A Flexible and Scalable Desktop Grid System." *Journal of Grid Computing*, vol. 7, pp. 439–461, 2009. (Cited on pages 55, 58, 59, 61, 62, 70 and 72.)

[13] S. Kelly and J.-P. Tolvanen. "Domain-Specific Modeling." *IEEE Software*, vol. 26, pp. 39–46, 2009. (Cited on page 33.)

[14] M. Kircher, P. Jain, A. Corsaro, and D. Levine. "Distributed eXtreme Programming." *Challenges*, pp. 66–71, 2001. (Cited on page 6.)

[15] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. "SETI@home-massively distributed computing for SETI." *Computing in Science Engineering*, vol. 3, pp. 78–83, 2001. (Cited on pages 53 and 54.)

[16] F. Lagarde et.al. "Improving UML Profile Design Practices by Leveraging Conceptual Domain Models." *ACM Press – Optimization*, Internet: `http://portal.acm.org/citation.cfm?doid=1321631.1321705`, pp. 445–448, 2007. (Cited on page 20.)

[17] C. Larman and V. R. Basili. "Iterative and Incremental Development: A Brief History." *Computer*, vol. 36, pp. 47–56, 2003. (Cited on page 6.)

[18] A. C. Marosi, Z. Balaton, and P. Kacsuk. "GenWrapper: A generic wrapper for running legacy applications on desktop grids." *Sciences – New York*, Internet: `http://www.computer.org/portal/web/csdl/doi/10.1109/IPDPS.2009.5161136` 2009. (Cited on pages 15, 16, 59 and 72.)

[19] M. Mernik, J. Heering, and A. M. Sloane. "When and how to develop domain-specific languages." *ACM Computing Surveys*, vol. 37, pp. 316–344, 2005. (Cited on pages 33 and 34.)

[20] T. O. Meservy and K. D. Fenstermacher. "Transforming software development: an MDA road map." *IEEE Computer Society Press*, vol. 38, pp. 52–58, 2005. (Cited on page 4.)

[21] T. A. Mogensen. “Basics of Compiler Design Extended edition.” Internet: `http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/`, 2008. (Cited on page 191.)

[22] A. Müller, M. Luban, C. Schröder, R. Modler, P. Kögerler, M. Axenovich, J. Schnack, P. Canfield, S. Bud’ko, and N. Harrison. “Classical and quantum magnetism in giant Keplerate magnetic molecules.” *ChemPhysChem*, vol. 2, pp. 517–521, 2001. (Cited on page 223.)

[23] A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, M. Herrick, B. P. Clarke, and A. S. Grimshaw. “The Legion Grid Portal.” *Portal*, vol. 14, pp. 1365–1394, 2002. (Cited on page 25.)

[24] C. B. Ries and C. Schröder. “Public Resource Computing mit Boinc”. Linux-Magazin, vol. 3, pp. 106–110, March 2011 (Cited on pages 70 and 79.)

[25] F. Truyen. “The Fast Guide to Model Driven Architecture The Basics of Model Driven Architecture.” Internet: `http://corba.com/mda/mda_files/Cephas_MDA_Fast_Guide.pdf`, 2006. (Cited on pages 30 and 31.)

[26] C. Türker and M. Gertz. “Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems.” *The VLDB Journal*, vol. 10, pp. 241–269, 2001. (Cited on page 47.)

[27] L. Zhang, Y. Lu, and F. Xu. “Unified modelling and analysis of collaboration business process based on Petri nets and Pi calculus.” *Software, IET*, vol. 4, pp. 303–317, 2010. (Cited on page 21.)

[28] D. P. Anderson. “Volunteer Computing: Planting the Flag,” in *Proc. PCGrid 2007 Workshop*, 2007. (Cited on pages 53, 54 and 55.)

[29] D. P. Anderson and G. Fedak. “The Computational and Storage Potential of Volunteer Computing,” in *Proc. Sixth IEEE International Symposium on Cluster Computing and the Grid CCGRID06*, 2006. (Cited on page 54.)

[30] D. P. Anderson, E. Korpela, and R. Walton. “High-Performance Task Distribution for Volunteer Computing,” in *Proc. First International Conference on eScience and Grid Computing eScience05*, 2005, pp. 196–203. (Cited on pages 56, 60, 61, 62 and 74.)

[31] D. P. Anderson. “BOINC: A System for Public-Resource Computing and Storage,” in *Proc. 5th IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 4–10. (Cited on pages 53, 54, 55, 56, 60 and 177.)

[32] O. Baskova, O. Gatsenko, G. Fedak, O. Lodygensky, and Y. Gordienko. "Porting Multiparametric MATLAB Application for Image and Video Processing to Desktop Grid for High-Performance Distributed Computing," in *Proc. 25th Int. Supercomputing Conf. (ISC)*, 2010. (Cited on pages 15 and 16.)

[33] M. Black and W. Edgar. "Exploring mobile devices as Grid resources: Using an x86 virtual machine to run BOINC on an iPhone," in *Proc. 10th IEEE ACM International Conference on Grid Computing*, 2009, pp. 9–16. (Cited on page 53.)

[34] B. Boehm, W. Brown, and R. Turner. "Spiral development of software-intensive systems of systems," in *Proc. 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 706–707. (Cited on page 6.)

[35] C. Christensen, T. Aina, and D. Stainforth. "The Challenge of Volunteer Computing with Lengthy Climate Model Simulations," in *Proc. First International Conference on eScience and Grid Computing eScience05*, 2005, pp. 8–15. (Cited on page 58.)

[36] C. Cirit and F. Buzluca. "A UML profile for role-based access control," in *Proc. 2nd international conference on Security of information and networks SIN'09*, 2009. (Cited on pages 81, 127 and 183.)

[37] F. Costa, L. Silva, and M. Dahlin. "Volunteer Cloud Computing: MapReduce over the Internet," in *Proc. IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2011, pp. 1855–1862. (Cited on page 26.)

[38] V. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. "Volunteer Computing and Desktop Cloud: The Cloud@Home Paradigm," in *Proc. Eighth IEEE International Symposium on Network Computing and Applications*, 2009, pp. 134–139. (Cited on page 26.)

[39] V. D. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. "Cloud @ Home : Bridging the Gap between Volunteer and Cloud Computing," in *Proc. ICIS*, 2009, pp. 423–432- (Cited on page 26.)

[40] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. Sixth Symposium on Operating System Design and Implementation*, 2004. (Cited on page 26.)

[41] T. Fahringer and S. Pllana. "Teuta: Tool Support for Performance Modeling of Distributed and Parallel Applications," in *Proc. Computational Science, ICCS*, 2004. (Cited on page 27.)

[42] T. Fahringer, S. Pllana and A. Villazon. "A-GWL: Abstract Grid Workflow Language," in *Proc. 4th International Conference on Computational Science ICCS*, 2004, pp. 42–49. (Cited on page 27.)

[43] G. Fedak, C. Germain, V. Neri, and F. Cappello. "XtremWeb: A Generic Global Computing System," in *Proc. First IEEEACM International Symposium on Cluster Computing and the Grid*, 2001, 582–587. (Cited on pages 24 and 25.)

[44] M. Feilkas. "How to represent Models, Languages and Transformations," Internet: `http://serv2.ist.psu.edu:8080/viewdoc/summary?doi=10.1.1.104.7999`, 2006, pp. 169–176. (Cited on pages 33 and 189.)

[45] M. Fowler. "Language Workbenches : The Killer-App for Domain Specific Languages ?", 2005, pp. 1–27. (Cited on pages 5 and 19.)

[46] R. France and B. Rumpe. "Model-driven Development of Complex Software: A Research Roadmap," in *Proc. Future of Software Engineering FOSE 07*, 2007, pp. 37–54. (Cited on pages 3 and 29.)

[47] A. Gerber and K. Raymond. "MOF to EMF : There and Back Again," in *Proc. OOPSLA Workshop on Eclipse Technology eXchange Eclipse*, 2003, pp. 60–64. (Cited on page 20.)

[48] C. Germain-Renaud and N. Playez. "Result checking in global computing systems," in *Proc. 17th annual international conference on Supercomputing*, 2003, pp. 226–233. (Cited on page 56.)

[49] D. L. Gonzalez et.al. "Increasing GP Computing Power for Free via Desktop GRID Computing and Virtualization," in *Proc. 17th Euromicro International Conference on Parallel Distributed and Network based Processing*, 2009, pp. 419–423. (Cited on page 58.)

[50] A. S. Grimshar and A. Natrajan. "Legion: Lessons Learned Building a Grid Operating System," in *Proc. IEEE*, 2005, pp. 589–603. (Cited on page 25.)

[51] S. Pllana, J. Qin, and T. Fahringer. "Teuta: A Tool for UML Based Composition of Scientific Grid Workflows." in *Proc. 1st Austrian Grid Symposium*, 2005. (Cited on page 175.)

[52] S. Pllana, J. Qin, and T. Fahringer. "Teuta: A Tool for UML Based Composition of Scientific Grid Workflows." in *Proc. 1st Austrian Grid Symposium*, 2005. (Cited on page 27.)

[53] S. Pllana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic. "Towards an UML Based Graphical Representation of Grid Workflow Applications." in *Proc. European Across Grids Conference*, 2004. (Cited on page 27.)

[54] C. B. Ries, C. Schröder, and V. Grout. "Model-based Generation of Workunits, Computation Sequences, Series and Service Interfaces for BOINC based Projects," in *Proc. International Conference on Software Engineering Research and Practice (SERP12)*, 2012 (Cited on pages 82, 180, 223, 243 and 293.)

[55] C. B. Ries, C. Schröder, and V. Grout. "Approach of a UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC)," in *Proc. IEEE Conference on Computer Applications and Industrial Electronics (ICCAIE 2011)*, 2011. (Cited on pages 4, 81, 146 and 234.)

[56] C. B. Ries, C. Schröder, and V. Grout. "Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC)," in *Proc. Seventh Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN)*, 2011, pp. 67–76. (Cited on pages 70 and 234.)

[57] C. B. Ries and C. Schröder, "ComsolGrid – A framework for performing large-scale parameter studies using COMSOL Multiphysics and the Berkeley Open Infrastructure for Network Computing (BOINC)," in *Proc. COMSOL Multiphysics Conference*, 2010. (Cited on pages 6, 15, 16, 59, 63, 70 and 81.)

[58] C. B. Ries, T. Hilbig, and C. Schröder, "A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework," in *Proc. International Multiconference on Computer Science and Information Technology (IMCSIT)*, 2010, pp. 663–670. (Cited on pages 13, 14, 15, 22, 58, 81 and 200.)

[59] P. G. Ríos, P. Fonseca, P. Pablo, and O. P. Díaz. "Legion : An extensible lightweight web framework for easy BOINC task submission , monitoring and result retrieval using web services." in *Proc. CLCAR*, 2011. (Cited on page 63.)

[60] B. Selic. "A Systematic Approach to Domain-Specific Language Design Using UML," in *Proc. 10th IEEE International Symposium on Object and Component Oriented RealTime Distributed Computing (ISORC07)*, 2007, pp. 2–9. (Cited on page 46.)

[61] S. H. Kim and J. W. Jeon. "Programming LEGO mindstorms NXT with visual programming," in *Proc. International Conference on Control Automation and Systems*, 2007, pp. 2468–2472. (Cited on page 4.)

[62] M. Soukup and J. Soukup. "Graphical Tools and Language Evolution," in *Proc. ATEM*, 2007. (Cited on page 4.)

[63] M. Wimmer, A. Schauerhuber, M. Strommer, W. Schwinger, and G. Kappel. "A Semi-automatic Approach for Bridging DSLs with UML," in *Proc. 7th OOPSLA Workshop on Domain Specific Modeling*, 2007, pp. 97–104. (Cited on page 20.)

[64] U. Zdun. "Concepts for Model-Driven Design and Evolution of Domain-Specific Languages," in *Proc. International Workshop on Software Factories at OOPSLA*, 2005, pp. 1–6. (Cited on pages 33 and 34.)

[65] "SOS 14 – Challenges in Exascale Computing," in *Proc. 13th Workshop on Distributed Supercomputing*, 2013. (Cited on page 253.)

[66] R. I. Bull. "Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization." PhD, University of Victoria, Canada, 2008. (Cited on page 79.)

[67] F. da Costa et.al. "Extension of BOINC middleware to a Peer-to-Peer Architecture." MSc, University of Coimbra, Portugal, 2007. (Cited on pages 55 and 57.)

[68] T. Friese. "Service-Oriented Ad Hoc Grid Computing." PhD, Philipps-Universität Marburg, Germany, 2006. (Cited on page 27.)

[69] A. Hessellund. "Domain-Specific Multimodeling." PhD, IT University of Copenhagen, Denmark, 2009. (Cited on page 33.)

[70] T. Hilbig. "Entwicklung einer Applikation für verteiltes Rechnen auf Basis der Berkeley Open Infrastructure for Network Computing (BOINC) zur Durchführung klassischer Monte Carlo Spin Dynamik Simulationen für magnetische Nanostrukturen." Diploma, Bielefeld University of Applied Sciences, Germany, 2006. (Cited on pages 15, 55 and 223.)

[71] K. R. Mudgal. "A Multi-platform Job Submission System for Epidemiological Simulation Models." MSc, Virginia Polytechnic Institute and State University, United States, 2011. (Cited on pages 56, 58, 61, 62, 63 and 68.)

[72] I. A. Niaz. "Automatic Code Generation From UML Class and Statechart Diagrams." PhD, University of Tsukuba, Japan, 2005. (Cited on page 197.)

[73] C. B. Ries. “ComsolGrid – Konzeption, Entwicklung und Implementierung eines Frameworks zur Kopplung von COMSOL Multiphysics und BOINC um hoch-skalierbare Parameterstudien zu erstellen.” MSc, Bielefeld University of Applied Sciences, Germany, 2010. (Cited on pages 4, 6, 15, 16, 59 and 81.)

[74] C. Werner. “A UML Profile for Communicating Systems.” PhD, Georg-August-Universität zu Göttingen, Germany, 2006. (Cited on pages 6, 35, 77 and 78.)

[75] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. (Cited on page 6.)

[76] S. Berner, M. Glinz, and S. Joos. “A Classification of Stereotypes for Object-Oriented Modeling Languages” in *UML99 – The Unified Modeling Language*, Springer Verlag, 1999. (Cited on pages 44 and 45.)

[77] J. Cabot and E. Teniente. “Constraint Support in MDA Tools: A Survey” in *Model Driven Architecture – Foundations and Applications Foundations and Applications*, Springer, 2006, pp. 256–267. (Cited on pages 24 and 47.)

[78] S. K- Card, J. D. Mackinlay, and B. Schneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999. (Cited on page 79.)

[79] B. Case. “SPARC Architecture” in *A Guide to RISC Microprocessors*, Academic Press, 1992, pp. 33–45. (Cited on page 58.)

[80] L. Engelhardt and C. Schröder. “Simulating computationally complex magnetic molecules” in *Molecular Cluster Magnets*, World Scientific Publishers, 2011, pp. 241–296. (Cited on page 173.)

[81] R. Eshuis and R. Wieringa. “Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets” in *Weber et al*, Springer, 2002, pp. 321–351. (Cited on page 20.)

[82] M. Fowler. *Domain-Specific Languages*. Addison Wesley Professional, 2010. (Cited on pages 192 and 194.)

[83] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. vol. 47, Addison Wesley Professional, 1995. (Cited on pages 79, 196, 250 and 300.)

[84] C. R. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Longman, 2009. (Cited on page 21.)

[85] R. Herken. “Alan Turing and the Turing machine” in *The Universal Turing Machine A HalfCentury Survey*, Oxford University Press, 1988, pp. 3–15. (Cited on page 34.)

[86] D. Kirk and Wen-Mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010. (Cited on pages 18 and 74.)

[87] T. Lodderstedt, D. Basin and J. Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security” in *UML 2002 The Unified Modeling Language*, 2002, pp. 426–441. (Cited on page 30.)

[88] R.Milner. *Communicating and Mobile Systems – The Pi Calculus*. Cambridge University Press, Cambridge, UK, 1999. (Cited on page 21.)

[89] T. Parr. *The Definitive ANTLR Reference – Building Domain-Specific Languages*. Pragmatic Programmers, 2007. (Cited on pages 189, 190 and 193.)

[90] R. Pucella. *Review of Communicating and Mobile Systems: The π-calculus*. Department of Computer Science, Cornell University, 2000. (Cited on page 21.)

[91] C. B. Ries. *BOINC – Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing*. Berlin Heidelberg: Springer, 2012. (Cited on pages 6, 11, 56, 62, 63 and 215.)

[92] K. Schwaber and B. Gloger. *Scrum: Produkte zuverlässig und schnell entwickeln. Mit beigehefteter Scrum-Checkliste 2010*. Carl Hanser Verlag GmbH & CO. KG, 2010. (Cited on page 6.)

[93] D. A. Stainforth et.al. “climateprediction.net: a global community for research in climate physics” in *Environmental Online Communication*, Springer, 2004, pp. 101–112. (Cited on page 57.)

[94] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall International, 2001. (Cited on pages 71 and 120.)

[95] C. Wienands and M. Golm. “Anatomy of a Visual Domain-Specific Language Project in an Industrial Context” in *Model Driven Engineering Languages and Systems*, Springer, 2009, pp. 453–467. (Cited on page 22.)

[96] J. Bruck and K. Hussey. “Customizing UML: Which Technique is Right for You?” Internet: `http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html`, 2007. (Cited on pages 10, 37 and 44.)

[97] W. Heinickel and T. Feldmann. “CORBA / UML Success Story - Zuercher Kantonalbank Banking/Finance.” *KLEIN+STEKL, GmbH*, 2002. (Cited on page 13.)

[98] T. Parr. “Translators Should Use Tree Grammars.” *University of San Francisco, CA: United States*, Internet: `http://antlr.org/article/1100569809276/use.tree.grammars.tml`, 2004. (Cited on page 195.)

[99] A. Senac, D. Sevilla, and G. Martínez. “EMF4CPP: a C++Ecore Implementation.” *University of Murcia, Spain*, 2010. (Cited on page 21.)

[100] S. Temme. “Apache Performance Tuning Part One: Scaling Up.” 2006. (Cited on page 99.)

[101] Gentleware. “Selecting a UML tool.” *Whitepaper*, 2006. (Cited on page 24.)

[102] I-Logix Inc. “Objective Control Cuts Development Time, Enhances Customer Communication and Increases Component Re-use with I-Logix’ Rhapsody.” 2002. (Cited on page 13.)

[103] Object Management Group (OMG). “MDA Guide Version 1.0.1.” 2003. (Cited on pages 31 and 33.)

[104] Keynote – The Mobile & Internet Performance Authority. “Turn the Green Light on the Last Mile: Optimizing End-User Experience in a Web 2.0 World.” 2008. (Cited on page 252.)

[105] Siemens. “Success Story: Siemens RailCom and Model Driven Architecture.” 2006. (Cited on page 13.)

[106] Unisys Corporation. “Moving up the model-driven development maturity curve.” 2009. (Cited on page 10.)

[107] “V-Modell XT.” Internet: `http://v-modell.iabg.de/v-modell-xt-html-english/index.html` (Cited on page 6.)

[108] S. Shepler, M. Eisler, and D. Noveck. “Network File System (NFS) Version 4 Minor Version 1 Protocol.” RFC5661. (Cited on pages 59 and 60.)

[109] The Internet Society, Network Working Group, Network Appliance, Inc. “NFS version 4 Protocol.” RFC3010. (Cited on page 84.)

[110] W3C. “Hypertext Transfer Protocol – HTTP/1.1,” RFC2616. (Cited on pages 15 and 16.)

[111] S. Deering and R. Hinden. “Internet Protocol, Version 6 (IPv6) Specification.” RFC2460. (Cited on page 89.)

[112] R. Rivest. “The MD5 Message-Digest Algorithm.” RFC1321. (Cited on page 62.)

[113] M. del Rey. “Internet Protocol.” RFC791. (Cited on page 89.)

[114] J. E. White. “A High-Level Framework for Network-Based Resource Sharing.” RFC707. (Cited on page 70.)

[115] OpenMP Architecture Review Board. “The OpenMP API specification for parallel programming.” Version 3.1 Complete Specification, 2011. (Cited on pages 71 and 74.)

[116] International Organization for Standardization (ISO) & American National Standards Institute (ANSI). “ISO/IEC 9075:2011.” 2011. (Cited on pages 16, 24 and 47.)

[117] W3C. “Extensible Markup Language (XML) 1.0.” Internet: `http://www.w3.org/TR/REC-xml/`, 2008. (Cited on page 58.)

[118] International Organization for Standardization. “ISO/IEC C++ 2003.” Internet: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110`, 2003. (Cited on pages 7 and 56.)

[119] ISO/IEC. “Information technology – Syntactic metalanguage - Extended BNF.” 1996. (Cited on page 193.)

[120] IEEE. “IEEE POSIC 1003.1c.” 1995. (Cited on page 71.)

[121] University of Tennessee. “MPI: A Message-Passing Interface Standard.” Internet: `http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/mpi-report.htm`, 1995. (Cited on pages 25 and 71.)

[122] Khronos Group. “OpenCL - The open standard for parallel programming of heterogeneous systems.” Internet: `http://www.khronos.org/opencl/`. (Cited on pages 18 and 74.)

[123] IEEE. “IEEE 802.3 CSMA/CD (ETHERNET).” Internet: `http://www.ieee802.org/3/` (Cited on page 57.)

[124] Object Management Group (OMG). “MDA – The Architecture of Choice for a Changing World.” Internet: `http://www.omg.org/mda/`. (Cited on pages 10, 11, 30, 31, 32, 35 and 47.)

[125] Object Management Group (OMG). "OMG Unified Modeling Language TM (OMG UML), Superstructure, Version 2.4." 2011. (Cited on pages 6, 13, 23, 35, 36, 37, 39, 40, 41, 43, 44, 80, 84, 85, 86, 87, 88, 89, 90, 92, 93, 95, 96, 100, 101, 102, 105, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 122, 123, 124, 125, 128, 129, 130, 131, 134, 135, 136, 137, 138, 139, 140, 142, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 158, 159, 161, 162, 163, 164, 165, 166, 171, 172, 175, 187, 197, 198 and 207.)

[126] Object Management Group (OMG). "OMG Unified Modeling Language TM (OMG UML), Infrastructure, Version 2.4." 2011. (Cited on pages 36, 40, 115, 134, 135 and 203.)

[127] Object Management Group (OMG). "OMG Object Constraint Language (OCL) Version 2.2." 2010. (Cited on pages 5, 24, 37, 48, 49, 50, 51 and 113.)

[128] Object Management Group (OMG). "SysML Specification." Internet: http://www.sysml.org/specs/, 2010. (Cited on page 23.)

[129] Object Management Group (OMG). "MOF Core specification." 2006. (Cited on pages 43 and 47.)

[130] J. S. Ashley Mills. "ANTLR." Internet: http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/antlr/antlr.html#ANTLR-Notation, 2005. (Cited on page 193.)

[131] P. D. Buck. "Redundancy and Errors – Unofficial BOINC Wiki." Internet: http://www.boinc-wiki.info/Redundancy_and_Errors [Online. Last accessed: 22nd April 2012] (Cited on pages 62, 65 and 67.)

[132] A. Etengoff. "Tesla GPUs accelerate engineering sims." Internet: http://www.tgdaily.com/hardware-features/51644-tesla-gpus-accelerate-engineering-sims [Online. Last accessed: 1st June 2012] (Cited on page 18.)

[133] A. Gurtovoy and D. Abrahams. "The BOOST MPL Library." Internet: http://www.boost.org/doc/libs/1_49_0/libs/mpl/doc/index.html [Online. Last accessed: 12th April 2012] Gurtovoy, Aleksey and Abrahams, David (Cited on page 21.)

[134] IBM. "Types of modeling diagrams." Internet: http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=/com.ibm.xtools.modeler.doc/topics/

rdiagtype.html [Online. Last accessed: 21st April 2012] (Cited on pages 41 and 42.)

[135] IBM. "IBM Software - Rational Rhapsody." Internet: http://www-01.ibm.com/software/awdtools/rhapsody/ [Online. Last accessed: 12th April 2012] (Cited on page 23.)

[136] B. Kiers. "Stack Overflow - ANTLR, how to convert BNF,EBNF data in ANTLR?" Internet: http://stackoverflow.com/questions/3230563/antlr-how-to-convert-bnf-ebnf-data-in-antlr [Online. Last accessed: 2nd June 2012] (Cited on page 193.)

[137] J. Luber. "Quick Starter on Parser Grammars - No Past Experience Required." Internet: http://www.antlr.org/wiki/display/ANTLR3/Quick+Starter+on+Parser+Grammars+-+No+Past+Experience+Required [Online. Last accessed: 2nd June 2012] (Cited on page 193.)

[138] P. McJones. "History of FORTRAN and FORTRAN II." Internet: http://www.softwarepreservation.org/projects/FORTRAN/, 2005 [Online. Last accessed: 20th April 2012] (Cited on pages 31, 56 and 59.)

[139] C.A Petri and W. Reisig "Petri net." Scholarpedia, 3(4):6477. (Cited on page 20.)

[140] S. Tatár and H. Greve. "GPU-Supercomputer ermöglicht chinesischen Forschern weltweit erste Simulation des kompletten H1N1-Virus." Internet: http://www.nvidia.de/object/tesla-gpu-improves-drugs-20111110-de.html [Online. Last accessed: 1st June 2012] (Cited on page 18.)

[141] B. Stroustrop. "The C++ Programming Language." Internet: http://www2.research.att.com/~bs/C++.html [Online. Last accessed: 13th April 2012] (Cited on pages 7 and 59.)

[142] J. van Haan. "MDE - Model Driven Engineering – reference guide." Internet: http://www.theenterprisearchitect.eu/archive/2009/01/15/mde---model-driven-engineering----reference-guide, 2009 [Online. Last accessed: 14th June 2012] (Cited on page 10.)

[143] "PlanetLab – An open platform for developing, deploying, and accessing planetary-scale services." Internet: https://www.planet-lab.org/hosting [Online. Last accessed: 19th May 2013] (Cited on page 25.)

[144] "ASKALON – Workflow Composition." Internet: http://www.dps.uibk.ac.at/projects/teuta/ [Online. Last accessed: 12th April 2012] (Cited on page 27.)

[145] "C++ language gets high marks on performance with new ISO/IEC standard." Internet: http://www.iso.org/iso/pressrelease.htm?refid=Ref1472 [Online. Last accessed: 13th April 2012] (Cited on page 7.)

[146] "CERNVM/Vboxwrapper Test Project." Internet: http://boinc.berkeley.edu/vbox/ [Online. Last accessed: 23rd April 2012] (Cited on page 72.)

[147] "Choosing BOINC projects." Internet: http://boinc.berkeley.edu/projects.php [Online. Last accessed: 14th June 2012] (Cited on page 13.)

[148] "Condor – High Throughput Computing." Internet: http://research.cs.wisc.edu/condor/ [Online. Last accessed: 17th April 2012] (Cited on page 24.)

[149] "Dresden OCL Toolkit." Internet: http://www.dresden-ocl.org/index.php/DresdenOCL [Online. Last accessed: 13th April 2012] (Cited on page 24.)

[150] "DynaPing (beendet)." Internet: http://www.rechenkraft.net/wiki/index.php?title=DynaPing_(beendet) [Online. Last accessed: 23rd April 2012] (Cited on page 70.)

[151] "EGEE." Internet: http://egee1.eu-egee.org [Online. Last accessed: 17th April 2012] (Cited on page 25.)

[152] "Features of the BOINC System." Internet: http://www.boinc-wiki.info/Features_of_the_BOINC_System [Online. Last accessed: 23rd April 2012] (Cited on page 56.)

[153] "The DoDAF Architecture Framework Version 2.02." Internet: http://dodcio.defense.gov/dodaf20.aspx [Online. Last accessed: 14th April 2012] (Cited on page 23.)

[154] "IberCivis." Internet: http://registro.ibercivis.es [Online. Last accessed: 11th April 2012] (Cited on page 59.)

[155] "Linux Man Pages." Internet: http://www.linuxmanpages.com/ [Online. Last accessed: 12th June 2012] (Cited on page 185.)

[156] "MySQL 5.0 Reference Manual :: 3.3.4.4 Sorting Rows." Internet: http://dev.mysql.com/doc/refman/5.0/en/sorting-rows.html [Online. Last accessed: 22nd April 2012] (Cited on page 258.)

[157] "Native BOINC for Android." Internet: https://github.com/matszpk/native-boinc-for-android [Online. Last accessed: 17th April 2012] (Cited on page 53.)

[158] "NorduGrid – Grid Solutions for Wide Area Computing and Data Handling." Internet: http://www.nordugrid.org/ [Online. Last accessed: 17th April 2012] (Cited on page 25.)

[159] "Oxford Dictionaries Online." Internet: http://oxforddictionaries.com [Online. Last accessed: 2012] (Cited on pages 9 and 11.)

[160] "Project Tasks – BOINC." Internet: http://boinc.berkeley.edu/trac/wiki/ProjectTasks [Online. Last accessed: 2011-11-06] (Cited on page 163.)

[161] "PVM: Parallel Virtual Machine." Internet: http://www.csm.ornl.gov/pvm/pvm_home.html [Online. Last accessed: 17th April 2012] (Cited on page 25.)

[162] "Projektübersicht – Rechenkraft." Internet: http://www.rechenkraft.net/wiki/index.php?title=Projekt%C3%BCbersicht [Online. Last accessed: 14th June 2012] (Cited on page 13.)

[163] "SimpleOCL." Internet: http://soft.vub.ac.be/soft/research/mdd/simpleocl [Online. Last accessed: 12th April 2012] (Cited on page 24.)

[164] "XtremWeb – Computing on Large Scale Distributed Systems." Internet: http://www.xtremweb.net/index.html [Online. Last accessed: 17th April 2012] (Cited on page 24.)

[165] "Spinhenge@home." Internet: http://spin.fh-bielefeld.de/ [Online. Last accessed: 23rd April 2012] (Cited on pages 6, 15, 70, 84 and 223.)

[166] "Waterfall Model – Advantages, Examples, Phases and more abour Software Development." Internet: http://www.waterfall-model.com [Online. Last accessed: 1st June 2012] (Cited on page 6.)

[167] "Welcome to the BOINC Trac Wiki – BOINC." Internet: http://boinc.berkeley.edu/trac/wiki [Online. Last accessed: 26th March 2012] (Cited on pages 6, 14, 16, 63, 64, 70, 71, 72, 88, 97, 98, 135 and 136.)

[168] "Wikipedia, the free encyclopedia." Internet: http://en.wikipedia.org/wiki/Main_Page [Online. Last accessed: 31st May 2012] (Cited on pages 180 and 223.)

[169] "Worldwide LHC Computing Grid." Internet: http://lcg.web.cern.ch/lcg [Online. Last accessed: 17th April 2012] (Cited on page 25.)

[170] "Icosidodecahedron." Internet: http://mathworld.wolfram.com/Icosidodecahedron.html [Online. Last accessed: 24th May 2013] (Cited on page 224.)

[171] "Kritische Schwachstelle in hunderten Industrieanlagen." Internet: http://www.heise.de/newsticker/meldung/Kritische-Schwachstelle-in-hunderten-Industrieanlagen-1854385.html [Online. Last accessed: 26th May 2013] (Cited on page 252.)

[172] "Hunderte Industrieanlagen ungesichert im Internet – Egal ob Brauerei oder Heizkraftwerk." Internet: http://www.heise-medien.de/presse/Hunderte-Industrieanlagen-ungesichert-im-Internet-1854654.html [Online. Last accessed: 26th May 2013] (Cited on page 252.)

[173] "Exascale – Newsletter des Forschungszentrums Jülich zum Supercomputing." Internet: http://www.fz-juelich.de/portal/DE/Presse/Publikationen/exascale-newsletter/_node.html [Online. Last accessed: 26th May 2013] (Cited on page 252.)

[174] "IBM, Forschungszentrum Juelich Found Joint Exascale Innovation Center." Internet: http://www.hpcwire.com/hpcwire/2010-03-23/ibm_forschungszentrum_juelich_found_joint_exascale_innovation_center.html [Online. Last accessed: 26th May 2013] (Cited on page 252.)

[175] "Exascale-Computer fÃ¼r JÃ¼lich in der Forschung." Internet: http://www.heise.de/ix/meldung/Exascale-Computer-fuer-Juelich-in-der-Forschung-961303.html [Online. Last accessed: 26th May 2013] (Cited on page 252.)

[176] I. Wladawsky-Berger. "Opinion – Challenges to exascale computing." Internet: http://www.isgtw.org/feature/opinion-challenges-exascale-computing [Online. Last accessed: 26th May 2013] (Cited on page 253.)

[177] AUTOSAR. "AUTomotive Open System ARchitecture (AUTOSAR) - Main Requirements." Internet: http://www.autosar.org, 2011 (Cited on page 23.)

[178] A. H. Dönni. "The Boost Statechart Library." Internet: http://www.boost.org/doc/libs/1_46_0/libs/statechart/doc/index.html, 2007 [Online. Last accessed: 12th April 2012] (Cited on page 197.)

[179] Google. "google-blockly - A visual programming language." Internet: http://code.google.com/p/google-blockly/, 2012 (Cited on page 4.)

[180] C. J. Henry. "Meta State Machine (MSM)." Internet: http://www.boost.org/doc/libs/1_49_0/libs/msm/doc/HTML/index.html, 2010 (Cited on page 197.)

[181] J. McCutchan, R. Love, and A. Griffis. "inotify - monitoring file system events." Internet: http://linux.die.net/man/7/inotify (Cited on page 234.)

[182] Microsoft. "Microsoft Robotics Developer Studio." Internet: http://www.microsoft.com/robotics, 2012 (Cited on page 4.)

[183] C. B. Ries. "UML 2 Statemachine for C++." Internet: https://sourceforge.net/projects/uml2stm4cpp/ [Online. Last accessed: 2013] (Cited on pages 197, 198, 225, 230 and 249.)

[184] C. B. Ries. "Visu@lGrid – Integrated Development Environment for BOINC." Internet: http://visualgrid.sourceforge.net [Online. Last accessed: 2013] (Cited on pages 169, 189, 211, 214, 223, 234, 241, 245, 249, 269, 300 and 303.)

[185] C. B. Ries. "libries – Research library for Visu@lGrid." Internet: http://libries.sourceforge.net [Online. Last accessed: 2012] (Cited on pages 190, 200, 223, 227, 229 and 249.)

[186] C. B. Ries. "LMBoinc – BOINC 'Hello World!' Example." Internet: http://lmboinc.sourceforge.net [Online. Last accessed: 23rd April 2012] (Cited on pages 6, 70, 223, 234, 245 and 293.)

[187] The Apache Software Foundation. “Apache HTTP Server.” Internet: http://projects.apache.org/projects/http_server.html (Cited on pages 15 and 59.)

[188] The PHP Group. “PHP: Hypertext Preprocessor.” Internet: http://www.php.net (Cited on page 59.)

[189] “AndroMDA.” Internet: http://www.andromda.org [Online. Last accessed: 13th April 2012] (Cited on page 24.)

[190] “Boincoid.” Internet: http://boincoid.sourceforge.net [Online. Last accessed: 17th April 2012] (Cited on page 53.)

[191] “Eclipse Modeling Framework Project (EMF).” Internet: http://www.eclipse.org/modeling/emf/ [Online. Last accessed: 12th April 2012] (Cited on page 21.)

[192] “Eclipse – The Eclipse Foundation open source community website.” Internet: http://www.eclipse.org [Online. Last accessed: 16th April 2012] (Cited on page 21.)

[193] “Globus.” Internet: http://www.globus.org/ [Online. Last accessed: 17th April 2012] (Cited on page 25.)

[194] “GEF (Graphical Editing Framework).” Internet: http://www.eclipse.org/gef/ [Online. Last accessed: 12th April 2012] (Cited on page 21.)

[195] “Graphical Modeling Project (GMP).” Internet: http://www.eclipse.org/modeling/gmp/ [Online. Last accessed: 12th April 2012] (Cited on page 21.)

[196] “Linux.org.” Internet: http://www.linux.org [Online. Last accessed: 13th April 2012] (Cited on page 8.)

[197] “Magick++ C++ API.” Internet: http://www.imagemagick.org/script/magick++.php, 2012 (Cited on page 241.)

[198] “Microsoft Excel 2010 - Office.com.” Internet: http://office.microsoft.com/de-de/excel/ (Cited on page 34.)

[199] “Multiphysics Modeling and Simulation Software – COMSOL.” Internet: http://www.comsol.com [Online. Last accessed: 29th October 2011] (Cited on pages 15 and 81.)

[200] "MySQL – The world's most popular open source database." Internet: http://www.mysql.com (Cited on page 59.)

[201] National Instruments. "NI LabVIEW." Internet: http://www.ni.com/labview/ (Cited on page 4.)

[202] "Oracle VM VirtualBox." Internet: https://www.virtualbox.org [Online. Last accessed: 17th April 2012] (Cited on page 59.)

[203] "Puppet Labs: IT Automation Software for System Administrators." Internet: http://www.puppetlabs.com (Cited on page 236.)

[204] "SAMBA – opening windows to a wider world." Internet: http://www.samba.org/ [Online. Last accessed: 30th May 2012] (Cited on page 84.)

[205] "The UNIX system, UNIX system." Internet: http://www.unix.org/ [Online. Last accessed: 13th April 2012] (Cited on page 7.)

[206] "Ubuntu." Internet: http://www.ubuntu.com/ [Online. Last accessed: 13th April 2012] (Cited on page 8.)

[207] "UML, BPMN and Database Tool for Software Development." Internet: http://www.visual-paradigm.com (Cited on pages 8, 23 and 169.)

[208] "Xtext – language development made easy." Internet: http://www.eclipse.org/Xtext/ [Online. Last accessed: 12th April 2012] (Cited on page 21.)

[209] "VMware Virtualization Software for Desktops, Servers & Virtual Machines for Public and Private Cloud Solutions." Internet: http://www.vmware.com [Online. Last accessed: 17th April 2012] (Cited on page 59.)

[210] Y. Lamo. "Model Driven Engineering (MDE)." Presented at Faculty of Engineering, Bergen University College (Norway), 26th April 2011. (Cited on page 3.)

Index