Design Rationale for TELOS, A PASCAL-based AI Language

Larry Travis, Masahiro Honda,
Richard LeBlanc and Stephen Zeigler

Computer Sciences Department
University of Wisconsin - Madison

TELOS is a PASCAL-based AI language intended to facilitate efficient development of efficient, well-structured programs. The design emphasizes powerful data abstraction and control abstraction mechanisms rather than the provision of particular high-level constructs. Among the many capabilities of TELOS are those intended to make it especially suitable for systematic AI model building, for example, in the areas of knowledge representation, planning, and reasoning. An event facility is provided which unifies the handling of conditional interrupts (demons), process suspension, process communication and execution faults. The context-dependent TELOS data base is referenceable either associatively or directly.

Key words and phrases: AI language, abstraction mechanisms, structured programming, extensible language, associative referencing, knowledge representation, problem-solving strategies, PASCAL

CR Categories: 3.6.0, 4.2.2

TELOS is an artificial intelligence language based on PASCAL in two ways: it includes PASCAL as a subset and it is implemented in PASCAL. This paper identifies and explains the main requirements which have motivated the development of TELOS, and then describes and illustrates aspects of the design intended specifically to respond to these requirements.

We can convey the motivation of TELOS by answering two simple questions:

(a) Why develop yet another AI language when there are available several obviously good ones like QLISP and SAIL?

(b) Why does AI research need a special programming language at all? That is, what capabilities are lacking in the more general languages, especially the relatively young ones like PASCAL, SETL or SIMULA 67?

1. Why another AI language?

There are a number of characteristics and capabilities which distinguish TELOS from other AI languages and which constitute its reason for being. Lesser though important ones derive from requirements of portability, program efficiency, and programmer productivity. The features more central for the TELOS design, however, derive from the dominating aim of incorporating within it recent insights about the centrality to effective programming of the process of abstraction.

1.1 The lesser reasons

1.1.1 Portability

With a few notable exceptions, existing AI languages have been highly dependent on particular kinds of machines or, even worse, on non-standard base languages or operating systems available only at single sites. This has made their dissemination and availability beyond home base expensive and haphazard.

The effect is serious for the progress of AI research: science cannot proceed effectively if scientists are prevented from easily testing and critically evaluating the results of their predecessors and their peers, and are prevented from building on top of earlier results. The progress of AI research and model building has been significantly slowed because the expense and difficulty of transporting programs has frustrated this process of mutual criticism and cumulative model building.

Implementing TELOS in PASCAL should contribute significantly to the portability of TELOS, and consequently of programs written in TELOS. PASCAL is rigidly standardized and is becoming widely available.

1.1.2 Program efficiency

The programmer must be provided a means of precisely managing the various expensive operations required for AI computation. Toward this end TELOS has the following characteristics:

(a) Features which entail expensive overhead, for example, content addressing of memory or data base context saving and restoring, have been isolated so that only he who uses a particular feature incurs any cost for it, and that in proportion to the extent of use.

(b) Further, the operation of such features is never automatic. When something expensive is done, for example, the exclusion of duplicates from the data base, it is because the user explicitly programs it.

(c) Many of the search operations in AI which are open-ended, for example, general backtracking pattern recognition operations used to detect patterns in objects of variable structure, are deliberately not built in. Instead of being provided with a general-purpose operation whose cost often increases exponentially with operand or problem complexity, the TELOS user is provided with tools for building special purpose data structures and operations which can lower computational cost by taking advantage of known problem structure.

1.1.3 Programmer productivity

Because PASCAL program structuring features are designed to facilitate the writing of well-structured programs, making PASCAL a subset of TELOS started us on the road toward a language suitable for system building by stepwise refinement. But TELOS is an AI language, and many capabilities required for AI directly conflict with principles of good program structuring. Resolution of this conflict has presented a major design challenge. The resulting language is more conducive to good structuring than previous AI languages, and consequently has greater potential for programmer productivity, in the following ways:

(a) It has PASCAL's program structuring dictions which, if used correctly, can result in modularized programs with comprehensibly simple and explicitly visible control flow within modules.

(b) The abstraction mechanisms provided facilitate the hierarchical construction of program systems so that at a given level the system can be understood without concern for supporting representation and implementation details of lower levels. (More about abstraction in Section 1.2.)

(c) The non-hierarchical control regimes required for AI programming can obscure flow of control among modules. TELOS centralizes all control transfers among non-hierarchically related coroutine modules within overseers, and requires that all invocations of coroutines be from an overseer. (For example, there are thus no suspends which result in transfer back to some unknown invoker; and there are no invokes, that is gotos, out of one coroutine module into another.) This requirement can be imposed without any loss of power with respect to what kinds of control regimes can be programmed, though the programmer may have to think harder (prior to debugging!), anticipating and allowing for all the complexities of coroutine interaction within a particular regime being designed, rather than discovering them by accident. But that's the point.

(d) A central AI language requirement which violates the structuring principle that module communication interfaces should be explicit and visible, is the need for an associatively referenceable, globally available data base. The TELOS design mediates this conflict by requiring that all data base changes be explicitly programmed (i.e., they never occur without use of an explicitly designated store or update function); by requiring all data base context changes to be explicit; and by limiting access to data base contexts. In particular, two modules can affect each other through operations on the data base only with respect to those data base contexts which they both know about (and no module is given the capability to clamber freely around the context tree).

(e) Conditional interrupts (demons) used widely in AI programming may result in things happening that are not apparent from program text. TELOS mediates this conflict by forcing clear demarcation of the textual scope within which demons are operative.

(f) Finally, there is the problem of pointers, which enable multiple access paths to data, making data flow hard to follow. These pointers are integral to TELOS implementation of the highly variable data structures needed for AI work. To facilitate judicious and structured pointer use, we provide in TELOS a capability for centralizing reference to structure-defining pointers within capsules, with benefits analogous to those indicated above for centralizing coroutine control transfers within overseers.

1.2 The main reason: abstraction

The occurrence and usefulness of abstraction in human intellectual activity has long been observed. (For example, recall Descartes' Discourse on Method.) It can be seen most consciously practiced in mathematics where concepts are defined sequentially and cumulatively, with the needed properties of a concept at a given level summed up in certain theorems. Once these theorems have been established from the definition of the concept, the mathematician need no longer be concerned with the numerous and intricate details of the underlying sequence of definitions. This abstraction from definitional details enables him to keep problem complexity within humanly manageable bounds.

The central insight of recent programming research has been that an analogous approach can make a major contribution toward containment of program complexity. The tools for building definitional sequences of procedures and functions have been available in languages like LISP and ALGOL for many years, though they usually have not been systematically applied to achieve abstraction as such. Proceeding beyond simple procedural abstraction, new languages are now appearing with data and control abstraction capabilities.

Data abstraction capabilities enable the user to define problem-specific data structures with details of representation and implementation encapsulated within a single definition. The procedures and functions which realize possible operations on these structures are an integral part of the definition, and the structures are characterized and used in terms of these defining operations. Data type definitions of this kind within TELOS are called capsules.

Control abstraction capabilities enable an analogous user definition of control regimes, e.g., the backtracking regime built into several previous AI languages. Just as data abstractions enable the localizing of data representation details, control abstractions localize the process control-transfer and communication details needed to realize a desired control regime. Control regime definitions of this kind within TELOS are called overseers.

There is an added reason why an AI language needs powerful abstraction facilities, beyond the obvious one that AI programs are usually big and complicated. One of the points of work on AI is development of a general theory of intelligence. Such a theory will need to reference programs. Its development will depend on the programs being reasonably comprehensible and precisely characterizable. It will be necessary that connection between high-level theoretical concepts and program abstractions be simple and direct.

Thus, a major thrust in the TELOS design is provision of efficient and effective abstraction facilities within an AI language. A primary difference between TELOS and other AI languages is that where their design has focused on building in certain powerful high-level constructs, the TELOS design has focused on building in powerful abstraction mechanisms with which those particular high-level constructs, as well as numerous others, can be defined and implemented with reasonable ease.

A simple example will suffice at this point. An AI language must provide the ability to easily describe, manipulate, and access data objects whose structure varies dynamically and unpredictably. This requirement was the primary motivation underlying the first generation of AI languages, that is, the "list processing" languages like IPL and LISP, and it is no less central to AI today. However, TELOS is an AI language without lists! Rather than building in list structures of some particular kind as a basic data type, TELOS makes available mechanisms by which different, problem-adapted kinds of list structures can be easily implemented as required. And these abstraction mechanisms usable for list defining are of course usable for building variable-structure data objects perhaps more suitable for particular applications than lists, e.g., classes; bags; collections with attached, multiple, alternative orderings; trees; graphs; etc.

But powerful abstraction mechanisms are useful everywhere computers can be used and programs must be programmed. As a language with powerful abstraction mechanisms, in what sense is TELOS specifically an AI language?

2. Why any AI Language?

Indeed, we do expect that languages akin to TELOS will be widely useful outside AI. But an AI language must include abstraction mechanisms specially tailored to AI requirements, as well as including certain other special features not user-implementable by abstraction mechanisms, at least not in a natural way.

2.1 A large store of general knowledge

The language must be suitable for building and experimenting with the knowledge-based systems which have become the focus of much recent AI research; that is, systems which integrally involve large stores of general knowledge. This means that certain basic operations for creating, accessing, managing, and navigating a large data base must be built into the language; and it also means that data abstraction mechanisms must include features for specifying and controlling how a data type being defined is to be connected into and retrieved from the data base.

2.2 Reversibility of knowledge store modifications

There are other fields which utilize large data bases, even those oriented toward storing general knowledge. But AI in addition requires that, for a planning or problem-solving system treating the data base as a cognitive model, it be possible to make successive, tentative modifications of the data base and then later recover a previous state of the data base at any desired point in the succession. Further, it must be possible for alternative tentative data base modifications to co-exist.

2.3 Content addressing of data, procedures and processes

Yet another special AI requirement involves the data base. It must be possible to associatively reference and retrieve data and "stored" procedures and processes. Among other things, this enables the construction of goal-oriented systems where procedures to be tried for reaching a goal are identified by description rather than by name; and it enables the cumulative addition of procedural or declarative fragments to a data store, thus increasing the knowledge-based "intelligence" of a system, where the organized use of these fragments is determined by some control abstraction whose specification may well have preceded the fragments in the system.

This associative referencing requirement has dictated that data abstraction mechanisms in TELOS include pattern-function defining and pattern-function calling mechanisms by means of which associative referencing mechanisms for user-defined data types can be specified and implemented; and it has dictated that control abstraction mechanisms include the procedure-variable, process-variable and parameter-assignment mechanisms required to specify control regimes which identify the procedures and processes they control by description rather than by direct reference.

2.4 Experimental control regimes

AI research is specifically motivated to experiment with novel kinds of control regimes, that is, with complex problem-solving strategies for initiating solution attempts, allocating resources to them and focusing attention among them. Objects of interest range all the way from strategies which depend on blind, depth-first backtracking to sophisticated strategies which construct and articulate global, non-linear plans. Thus, for the present at least, AI languages must include control abstraction capabilities more versatile and powerful than those needed in languages intended for use in other fields.

We now turn to explaining and exemplifying how the TELOS design incorporates these various desiderata.

3. Data type definability and data abstraction

3.1 Capsules

TELOS extends the already powerful data type definition mechanisms of PASCAL by enabling type definitions to be realized as capsules. As with simpler kinds of data type definition, a capsule includes specification of the new type's structuring method (e.g., Record, Array, or interconnection of a "graph" of records with pointers). However, it must include in addition a procedural specification of the primitive operations with which objects of the type being defined can be manipulated, accessed or altered. The only representation details of the type which are available to routines not defined within the capsule are those which the definer chooses explicitly to make available. Although in defining a capsule type the programmer may well have to worry a great deal about low-level representation details, e.g., pointer manipulation, such details are localized to the capsule. Once "encapsulation" has been achieved, he may think about and use objects of this type in terms of intended use and external properties, abstracting from the internal details of representation.

A capsule consists essentially of a data structure specification and a set of routines that determine the primitive operations. Constants and subordinate types, including other capsules, may also be defined within a capsule. All identifiers declared within the capsule (including routine names) are accessible outside of the capsule scope only if they are explicitly exported.

Since capsules may be parameterized by types (see the list capsule example in Appendix A), a capsule defines a type generator rather than a data type as such. To create a type from a capsule, it is necessary to provide specific instances of the formal type parameters as actual parameters (see the use of the list capsule in the control abstraction example in Appendix B).

In specifying that a capsule may take a type as a parameter, the capsule definition states what operations must be defined for a type provided as an actual parameter. For instance, a type parameter might be specified as:

```
t [ =, Function merge (t, t) : t ]
```

This specification restricts actual parameters for t to types s for which an equality function and a merge function (with the indicated argument and value types) are available. If such a type s were itself defined with a capsule, the equivalence of particular functions defined within the capsule to these "standard" operations must be explicitly stipulated (as illustrated below).

The exports clause includes routine names, possibly equivalencing them to "standard" operations as with:

```
Exports   eq Is =,
          absorb Is merge
```

... and possibly stipulating a calling syntax alternative to standard procedure or function calls as with:

```
Exports   inject (c, a, v) Is c(a) <- v,
          picture Is [?   ?]
```

The third example shows how the user can avail himself of assignment statement syntax (using the special capsule assignment operator "<-") for insertion of content into capsule components, thus achieving improved data flow visibility. (Compare use in PASCAL of forms like "a[i]" and "r.f" on the lefthand side of the assignment operator ":=".)

The fourth example illustrates how the TELOS programmer connects a procedure he defines within the capsule with TELOS pattern-applying syntax involving the use of the brackets "[?" and "?]". In the example, the user function picture can be a string interpreter which defines the language of patterns to be used, among other things, for associative retrieval of objects of the capsule type from the data base.

Also included in the exports list may be identifiers from the structure part of the capsule, names of internally defined types, and details of internally defined types (e.g., names of record type fields or value names of enumerated types). A variable from the structure part whose identifier is included on the exports list may be externally accessed, but may not be assigned to externally unless the identifier is prefixed with Var on the list.

The structure of a capsule type object is considered to be the named variables of the structure part (treated as fields of a record) plus some or all objects reachable through pointers from this "header". A crucial condition for understanding the properties of TELOS capsules is understanding how this "some or all" is decided. Typically pointers intended to be structure defining will be protected from external alteration, reserving capsule structure alteration to operations whose defining routines are included within the capsule. Thus for the list capsule example in Appendix A, the structure-defining list links are not externally alterable because the cell record field names pred and succ are not exported. How pointers within a capsule type instance are to be handled when the capsule is copied or stored in the data base (e.g., in the latter case, reflected in the data base version of the capsule with Nil, or with a component

pointer, or with a pointer to an independent data base object) may be explicitly specified within the capsule. A standard Store function is provided which in effect assumes that all pointers are to capsule components, but of course the user need not use this.

A final useful feature: a capsule type instance may have attached to it a generator in a suspended state. The method and point of this is discussed below in Section 4.

3.2 A data abstraction example: simple lists

In Appendix A, we illustrate some of these data abstraction facilities with a capsule which defines doubly linked lists, lists identified with pointers into their headers (cf. SLIP). The control abstraction example in Appendix B illustrates use of lists so defined.

We include within the illustration only some of the primitive operations which a user might include in the real case. Also, to keep things manageable within the space available, we do not illustrate some of the more powerful TELOS data abstraction features. For example, a real list-defining capsule might well include specification of a backtracking pattern interpreter for lists. Or it might include specification of a subordinate SLIP reader (cursor) capsule in a way enabling use of the form "m(c)" to reference the content of the component of list m pointed to by cursor c.

4. Control regime definabliity and control abstractions

4.1) Overseers

Manipulation of coroutine activations (called Processes), as may be necessary to implement a particular control regime, is localized in TELOS within a control abstraction called an overseer. All transfer of control from process to process is channeled through an overseer (that is, one process may not directly invoke another). Further, the intention is that overseers manage process-to-process messages to the maximum extent possible. (The greater the extent, the better structured the program.)

A number of special features of the language may only be used in conjunction with overseers, thus enforcing their use as control abstractions. These features include the use of process and routine reference variables; the dictions for associatively referencing routines; the dictions for creating, associatively referencing and managing processes; and the dictions for referencing suspended-process parameters.

Process creation is accomplished using the function:

NewProcess(<coroutine>,<actual parameter list>)

An instance of the coroutine (either named explicitly or by a coroutine reference variable) is created as a process with the specified actual parameters. The data context in which the new process begins execution (when it is explicitly invoked) is that which is operative when NewProcess is executed. NewProcess returns as its value a reference to the newly created process, which will be assigned to a ProcessRef variable for later use by the overseer.

The process reference (e.g., p) is then used by the overseer to manage the process by applying to it operations like the system-provided Invoke, Clone, and Terminate. Reference to a process parameter is possible within the overseer with a form like "p.a" where a is a formal parameter identifier in p's coroutine archetype. All processes created by an overseer are automatically terminated if the overseer itself terminates.

The full power of the overseer concept is apparent when it is recognized that a control abstraction may utilize control abstractions. A coroutine being executed as a process, or any procedure it calls, may in turn call overseers, enabling the construction of multi-level control regimes.

Another kind of routine available in TELOS is the Genroutine. Genroutines supply sequences of values or states usable, for example, to drive a For statement or to set up control structures that generate parallel streams of values.

Genroutines are executed as processes (referred to as Generators), but their use is not limited to overseers. A generator's existence is tied to the existence of the routine instance that creates it, with the following important exception: generators created within capsule operations from genroutines defined within the same capsule and, for a particular capsule instance, assigned to a generator reference variable within the structure part of the capsule, will exist as long as that capsule instance exists. This enables use of the generator as an implicit data representation or to generate successive states of the capsule instance.

4.2 Event handling: demons, suspensions, faults and messages

TELOS incorporates event handling mechanisms designed to unify several related control capabilities including conditional interrupts (demons), handling execution faults, process suspension, and communication among procedures and processes. The word event is used to mean the occurrence of a fault, the use of a feature designated as a system event, or the explicit instigation of a user-defined event. Three categories of events are available:

(a) Escape events require the termination of the computation which caused the event. These events include run-time errors detected by the system and user-defined events meant to signal abandonment or completion of an activity.

(b) Signal events are informational in nature and need not be trapped by any event handler. These include system-defined events such as procedure entries and exits. A handler for such an event may return control to the interrupted computation by use of Resume or it may cause termination by completing its execution without a Resume (indicating what value is to be returned if the interrupted computation is a function or generator).

(c) Suspend events are used for communication and control transfer between processes and their controlling overseers. Events of this type may only be raised in process archetypes (coroutines) and procedures defined within them, or in overseers. Handlers for suspend events may only appear in overseers (with the exception of those attached to a top-level overseer call). An important instance of localizing control details within overseers: a suspend event must be handled in the overseer directly controlling the process in which the event is raised.

Event declarations have the form:

<event category> Event <event id> <parameter spec>

Event parameters are like those for a procedure; they may be Const, Var or value. These parameters are the means by which information, available within the immediate environment of an event occurrence, is communicated to an event handler.

A user-defined event is caused to occur by executing a statement of the form:

<event category> <event id> <actual parameters>

... where the parameters provided must correspond in type and number to the formal parameters in the event declaration.

A set of event handlers may be appended to any statement or function call. An event handler set is of the form:

```
[* <event id> : <statement>;
   <event id> : <statement>;
        .
        .
        .
   <event id> : <statement> *]
```

Within the statement corresponding to any <event id>, a parameter for that event is referenced by <event id>.<parameter id>. Examples of event definitions and handlers for suspend and escape events are provided in the overseer example in Appendix B.

A set of related or frequently used event handlers may be grouped together as a Team. A team identifier can take the place of one of the name-statement pairs in a handler set. This activates all the handlers that are part of the team for the statement or function call with which the set is associated. Teams or single events can be explicitly Enabled and Disabled (for example, conditionally upon the occurrence of other events) within handlers.

A set of important system-defined signal events are those associated with entry and exit for any procedure, for example, the system-provided data base management procedures. For type-generic system procedures a type identifier prefix is used, for example, t$Update, in naming the event within an event handler. An event handler for the system signal event Enter$t$Update will have access (through a parameter of Update) to the value of the data base object about to be changed. Similarly Exit$t$Update provides access to the final value of the object. Handlers for such events provide one means within TELOS (though not the only one) for implementing PLANNER-like erase and antecedent data base managing demons.

The TELOS event mechanism provides a well-structured way to monitor execution with a demon-like capability at a reasonable cost (depending, of course, on the cost of the tests and actions specified within the handlers). The same mechanism allows the programmer to detect and trap error conditions and limit the scope of their effects, if desired. The parameterization of events turns out to be very useful in establishing straightforward communication from processes to their controlling overseer.

4.3 A control abstraction example: a downhilling strategy

The overseer attempt specified in Appendix B is given a task description and a set of routines (for example, a set determined by some associative reference) that might be able to perform the task. These routines will use the event step done to inform the overseer of their estimated progress (unless they report win or lose). The overseer will win (report success to its caller) as soon as any of the routines have done so. It will lose (report failure to its caller) if all of the routines lose. If neither of these cases have occurred, it will invoke the routine that last returned the highest progress estimate.

5. Data base structuring, referencing, and saving

5.1 Data base objects and data base object pointers

To enable TELOS to be used for building knowledge-based models, any system built with TELOS contains as a part a global data base potentially referenceable from all routines which are part of the system. Data base objects may be associatively referenced and retrieved. However, a number of features are included in TELOS to make it possible to reduce the high cost of data base associative referencing (which has made some of the earlier high-level AI languages prohibitively expensive to use for many applications). Associative referencing is not the only means of accessing data base objects as it was for some of the earlier languages; the objects may also be directly referenced.

The TELOS data base has as its basic units any objects which a TELOS working space pointer may reference. In addition, the language makes explicit provision for dealing with record structures. A record structure determined by a given pointer (working space or data base) is the immediately referenced record (called the "top record" of the structure) plus all objects reachable from any object in the structure by following pointers. These pointer closures are quite general. For example, note the significance and possible usefulness of record structures whose node records are of the type:

```
Type   lisp_node = Record
          car: ->(atom, lisp_node);
          cdr: ->lisp_node
       End   (*lisp_node*)
```

(The form of the car field exemplifies the TELOS syntax for specifying a type-variant pointer. atom is intended to be the name of some user-defined type. "->" is here used for the standard PASCAL vertical arrow.)

Each data base object (DBO) has a referenceable header. To allow context dependence within the data base, each header is associated in each context with precisely one data item, called the value of the DBO in that context. A user references a DBO (actually, the DBO's value in a given context) by using a special kind of pointer called a DBO pointer, or DBOP.

With the definition of DBOs and DBOPs, the form of the TELOS data base emerges. The DBOs may be regarded as constituting nodes of the data base, with values dependent on context and composed of any referenceable type, most typically record structures. Since fields of records may be declared of DBOP types, these nodes may be linked together in arbitrary ways via DBOPs (forming DBO cross-references).

The data base is accessible and modifiable only through TELOS system primitives which guarantee the integrity of the data base structure and which maintain validity of the inverted index lists used for associative referencing. In read-only situations DBOPs can essentially be used just as working space pointers. Data base component pointers, DBCPs, with even more circumscribed powers (in particular, they are context-specific), allow pointer access within specific values of DBOs. Both DBOPs and DBCPs, while not allowing write-access for normal assignment, may be used in the Update system procedure to surgically alter parts of DBO values. This procedure maintains DBO header and index list validity.

The data base is segmented by data object type and, if finer resolution is required, by user-specified data-object patterns. (Since all TELOS patterns are prefixed by a type identifier, any pattern automatically determines a "subtype".) The resulting segments may be the units of transfer to and from secondary storage, as separate associative-referencing index lists are maintained for each segment.

5.2 Patterns and associative referencing

TELOS patterns have the form:

<type name> [? <data description> ?]

Most use of patterns in other AI languages has been for describing variable-structure, sequential objects, in particular, hierarchical list structures. We indeed can use patterns in TELOS to describe such structures, but a backtracking pattern interpreter is not built into the language. Defining such an interpreter for pattern "pictures" involving user-decided conventions (or the simpler task of just specifying pattern functions for detecting variable-structure patterns, if picturing them is not important) can be a central part of a particular data abstraction definition. Tools for defining such pattern interpreters and pattern functions are an important part of the TELOS-provided data abstraction mechanisms.

TELOS does have built in a pattern interpreter for types defined using the PASCAL record and array structuring methods. Given the following declarations:

```
Type person =  Record
        name:  string;
        age:  Integer
     End (*person*);
 Var years: Integer;
```

... the pattern:

```
person [? (years:=) age: ?, name: 'SMITH' ?]
```

describes records containing the specified name, with the match assignment setting years to have the value of the age field when the pattern is used in a successful match operation. Such a match assignment variable can be used to interconnect different parts of a pattern or to export a value (possibly a pointer) outside the pattern.

Non-procedural patterns of the kind just illustrated are of two essential types, point-of-use patterns and pattern objects. The former are constructed at compile time and may contain expressions involving variables with values set or accessed in the surrounding program text. Pattern objects, on the other hand, are interpreted rather than compiled. They may be assigned to variables (of appropriate pattern type); and they may be built, altered, and examined during program execution. The same pattern constructor form (i.e, <type name> [?<subpattern list>?]) is used for specifying both point-of-use patterns and pattern objects.

The user may supplement the non-procedural pattern language by procedurally defining any pattern function he wishes. Thus he might write a pattern function palindrome satisfied by any string which is a palindrome. Then:

```
person [? name:  palindrome ?]
```

... characterizes the class of person records which have a palindrome within the name field.

Among the many uses of pattern functions: wrapping a non-procedural point-of-use pattern within a function definition in order to parameterize the variables with which the point-of-use pattern connects with surrounding program text. Another use: intermixing of procedural and non-procedural pattern specifications to define complex, conditional data descriptions.

Pattern objects may consist of explicit values, or of subpatterns. As illustrated above, they, or any of their subparts, may contain calls to pattern functions. If the function specified in such a call has parameters other than the match candidate, they are set at specification time and preserved in a user-accessible data structure, to be examined and altered by the user at will. The actual call to the pattern function is initiated by the system during match operations which use the pattern object.

A primary use of patterns is for associative referencing, say of the member entities of some capsule instance representing a set of some kind; or of the data base, as with the instruction:

```
Get (person [? age: greater (20) ?])
```

... where greater is an appropriately defined pattern function, or with the generator controlled loop:

```
For psn := Each (person [? name: 'SMITH' ?]) Do ...
```

Associative referencing of the data base proceeds in two stages. First, a candidate set of DBOP's is determined by intersection of inverted index lists. The index lists used in the intersection are first selected for the types involved and then, for each type, for the index terms extractable from indexed fields of the pattern (fields declared indexable when the types are defined) and from the pattern functions used (from values of index expressions specified as such when the functions are defined). Second, the pattern is matched against some or all of the candidate objects pointed to from the set.

The order of the candidate set is determined by the ordering of the intersected lists which in turn have a primary ordering determined by the user's having declared, if he desires (again, at type definition time), certain DBO top record or capsule header fields as Sequencer fields; and a secondary ordering determined by recency of DBO storage in the data base. The user can without difficulty specify an operation which provides him with all DBOP's in a candidate set, which he may then wish to examine and possibly re-order before performing expensive pattern match operations against any of its members.

5.3 Tentative modification of the data base

The data base context tree used in TELOS allows many different data base versions to exist simultaneously, whether intended to be successive or intended to be alternative -- or simply being saved for some historical purpose, e.g., failure analysis. All context creation and abandonment must be done explicitly with system-provided operations like Sprout, Release, and Finalize, which manipulate the context tree. References to contexts are with variables of type ContextRef.

Just as altering the context tree must always be completely explicit, so must be the switching of the current execution context. Thus, data base contexts and control structures are independent. And any of the context tree manipulating operations, like establishing a new context (with Sprout), never change the current context. To specify a change of current context, TELOS requires:

```
InContext c Do <statement>
```

... where c is a variable of type ContextRef.

The entire program is implicitly an InContext statement for the initial root context. The InContext statement structures the use of contexts by clearly delineating their scope, and does so without any loss of power compared to unstructured context resetting mechanisms of the sort that occur in other AI languages. Furthermore, it restricts the scope of a context to be completely contained within structured statements such as while loops, and within procedure bodies.

The TELOS context mechanism is implemented using a tree of context descriptors. This tree corresponds identically to the context tree the programmer has created. Each context descriptor contains relevant information concerning that context. A corresponding but simpler skeleton tree is associated with each DBO; it contains nodes only for contexts within which the DBO has been altered in value. Each node P of a skeleton tree has a pointer, P.v, to a value and another pointer, P.c, to the node in the context tree representing the context where P.v was entered into the data base.

A context tree encoding has been developed which enables efficient discovery, given a pointer to a DBO Q in context C, of which node P of Q's skeleton tree has the desired P.v for C; and which enables efficient maintenance of the context descriptor tree. (See Honda et al., 1977, for details.) Priority is given to fast look-up, and a good deal of preprocessing is done at the time of context creation.

A final feature of TELOS data base contexts: relative variables may be declared at the global level (i.e., the outermost lexical level of a TELOS program). Such variables may take on different values for different contexts. These values, in contrast to data base object values, may be referenced and used exactly as any working space value. The value of a relative variable v for some context c other than the current one is referenced with c.v. Relative variables provide the TELOS programmer with yet another control over efficiency of his program, in particular, one he can use for those parts of his problem which require context saving and restoring of values which he can name directly and never needs to reference associatively.

6. Concluding comments

We hope that TELOS survives the harsh natural selection process that applies to all newly born computing languages. The issue will be decided mainly by whether we are right in believing that powerful abstraction mechanisms are more important for an AI language than the particular high-level constructs it contains; and by whether we have been successful in articulating and implementing a set of such abstraction mechanisms sufficient for AI requirements. But even if TELOS per se doesn't survive, the design may serve as a provocative source of further ideas.

Acknowledgements and references

We have been much influenced by:

(a) Ideas about the role of abstraction in the programming process as reflected in Dijkstra [3] and Wirth [13].
(b) Work on languages implementing data abstraction mechanisms, in particular, SIMULA 67 [2], CLU [9] and ALPHARD [14].
(c) Previous AI language design work, especially the seminal work on PLANNER [6], QA4 [10] and SAIL [4].
(d) Goodenough [5] on exception handling.
(e) PASCAL!

[1] Bobrow, D. and B. Wegbreit. 1973. A model and stack implementation of multiple environments. ACM Communications, 16.
[2] Dahl, O., B. Myhrhaug and K. Nygaard. 1970. SIMULA common base language. Norwegian Computing Center Publication S-22.
[3] Dijkstra, E. 1972. Notes on structured programming. In Structured programming, by O. Dahl, E. Dijkstra and C. Hoare. Academic Press.
[4] Feldman, J., J. Low, D. Swinehart and R. Taylor. 1972. Recent developments in SAIL, an ALGOL-based language for artificial intelligence. FJCC.
[5] Goodenough, J. 1975. Exception handling: Issues and a proposed notation. ACM Communications, 18.
[6] Hewitt, C. 1972. Description and theoretical analysis (using schemata) of PLANNER, a language for proving theorems and manipulating models in a robot. MIT, AI Memo 251.
[7] Honda, M., R. LeBlanc, L. Travis and S. Zeigler. 1977. An improved data context mechanism. UW-MSN, Academic Computing Center Technical Report.
[8] Jensen, K., and N. Wirth. 1974. PASCAL, User manual and report. Springer-Verlag.
[9] Liskov, B., A. Snyder, R. Atkinson and C. Schoffert. 1977. Abstraction mechanisms in CLU. MIT, Laboratory for Computer Science CSG Memo 144-1.
[10] Rulifson, J., J. Derksen and R. Waldinger. 1973. QA4: A procedural calculus for intuitive reasoning. SRI, AI Note 73.
[11] Travis, L., R. LeBlanc, M. Honda and S. Zeigler. 1977. TELOS design specifications. UW-MSN, Academic Computing Center Technical Report.
[12] Wegbreit, B. 1976. Faster retrieval from context trees. ACM Communications, 19.
[13] Wirth, N. 1971. Program development by stepwise refinement. ACM Communications, 14.
[14] Wulf, W., R. London and M. Shaw. 1976. Abstraction and verification in ALPHARD, Introduction to language and methodology. CMU, Computer Science Department.

Appendix A: Capsule example

```
list = Capsule (comp_type [ := ])
  Exports
    cell (item), head, create, insert_after,
    delete, succ_element, pred_element, elements,
    header_delete;

  Type
    list_ptr = ->list;  cell_ptr = ->cell;
    cell_kinds = (header, component);
    cell = Record
      pred, succ : cell_ptr;
      Case kind : cell_kinds of
        (*header : empty*)
        component : (item : comp_type)
      End (*cell*);

  Structure
    head : cell_ptr
  End (*Structure*);

  Escape Event header_delete (c : cell_ptr);

  Function create : list_ptr;
    Var lp : list_ptr;
  Begin
    New (lp);
    With lp-> Do Begin
      New (head, header);
      With head-> Do Begin
        pred := head;
        succ := head
      End (*With head->*)
    End (*With lp->*);
    create := lp
  End (*create*);

  Procedure insert_after (c : cell_ptr;
                          val : comp_type);
  Var
    new_cell : cell_ptr;
  Begin
    New (new_cell, component);
    new_cell->.item := val;
    With c-> Do Begin
      new_cell->.succ := succ;
      new_cell->.pred := c;
      succ->.pred := new_cell;
      succ := new_cell
    End (*With c->*)
  End (*insert_after*);

  Procedure delete (Var c : cell_ptr);
  Begin
    If c->.kind = header Then
      Escape header_delete (c)
      (*This illustrates use of a user
       defined error event; see Sec. 4.2. *)
    Else
      With c-> Do Begin
        pred->.succ := succ;
        succ->.pred := pred
      End (*With c->*);
    c := Nil
  End (*delete*);

  Function succ_element (c : cell_ptr) : cell_ptr;
  Begin
    succ_element := c->.succ
  End (*succ_element*);

  Function pred_element (c : cell_ptr) : cell_ptr;
  Begin
    pred_element := c->.pred
  End (*pred_element*);

  Genroutine elements (l : list_ptr) : cell_ptr;
  Var
    c : cell_ptr;
  Begin
    c := l->.head->.succ;
    While c->.kind <> header Do Begin
      Yield (c);
      c := c->.succ
    End (*While*)
  End (*elements*)
End (*List*)
```

Appendix B: Overseer example

```
Type
  routine_list = list (RoutineRef);
  task_desc = Record
      (* description of the job to be performed by
        the routines provided to the overseer *)
     End;
  routine_list_ptr = -> routine_list;

Suspend Event step_done (eval : Integer);
Escape Event win;
Escape Event lose;

Overseer attempt (task : task_desc;
                  try_list : routine_list_ptr);
  Type
    process_rec = Record
        last_eval : Integer;
        ctxt : ContextRef;
        p : ProcessRef
      End (*process_rec*);
    process_list = list (process_rec);
  Var
    try : -> routine_list.cell;
    process_cell, best_process
      : -> process_list.cell;
    plist : -> process_list;
    procrec : process_rec;
    max_eval : Integer;

  Procedure find_max_eval;
    Begin
    max_eval := 0;
    For process_cell :=
        process_list$elements (plist) Do
      With process_cell->.item Do
        If last_eval > max_eval Then Begin
          max_eval := last_eval;
          best_process := process_cell
          End
    End (*find_max_eval*);

Begin
max_eval := 0;   best_process := Nil;
plist := process_list$create;
process_cell := plist.head;

(* First, create a process corresponding to each
  coroutine on the try list and allow it to run to
  its first evaluation point. *)

For try := routine_list$elements (try_list) Do
 With proc_rec Do Begin
  last_eval := 0;
  ctxt := Sprout (CurrentContext);
  Incontext ctxt Do  p := NewProcess (try, task);

  Invoke (p) [* (*event handlers follow*)
    step_done : Begin
      last_eval := step_done.eval;
      process_list$insert_after
          (process_cell,proc_rec);
      process_cell :=
        process_list$succ_element (process_cell);
      If last_eval > max_eval Then Begin
        best_process := process_cell;
        max_eval := last_eval   End (*If*);
      End (*step_done handler*);
    lose : Begin
      Terminate(p);  Release(ctxt);
      End (*lose handler*);
    win : Begin
      Finalize (CurrentContext,ctxt);
      Escape win  (*exit indicating win*)
      End (*win handler*)  *]
  End (*With and For*);

(* If execution reaches this point, all routines
  on the try list have been executed to their first
  suspension and none have  indicated a win. *)

While max_eval > 0 Do Begin

  (* Indicate current state to a higher level
    instance of the same overseer.*)
   Suspend step_done (max_eval);

  (* Execution continues here when restarted. *)
  Invoke (best_process)  [*
    win : Begin
      Finalize (CurrentContext,ctxt);
      Escape win  (*exit indicating win*)
      End (*win handler*);
    lose : Begin
      process_list$delete (best_process);
      find_max_eval
      End (*lose handler*);
    step_done : Begin
      best_process->.item.last_eval :=
        step_done.eval;
      If step_done.eval >= max_eval
        Then max_eval := step_done.eval
        Else find_max_eval
      End (*step_done handler*)  *]
    End (* While *);

(* If all of the processes indicate "lose",
  find_max_eval will find the list empty
  and set max_eval to 0.  Then the While
  loop will terminate and the following
  statement will be executed. *)

Escape lose

End (*attempt*);
```