A Bi-Level Language for Software Process Modeling

Gail E. Kaiser, Steven S. Popovich and Israel Z. Ben-Shaul

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027

212-939-7000/fax: 212-666-0140
{kaiser,popovich,israel}@cs.columbia.edu

CUCS-016-92
14 September 1992

Abstract

Most software process modeling formalisms are well-suited to one of two levels of specification, but not both. Some process languages concentrate on *global* control flow and synchronization; these languages make it easy to define the broad outline of a process, but harder (or even impossible) to refine the process by expressing constraints and policies on individual tools and data. Other process formalisms are inherently *local*: it is easy to define constraints, but far from straightforward to express control flow. We propose combining global and local formalisms to produce single but bi-level process modeling languages. We present our bi-level Activity Structures Language, which combines constrained expressions with rules, and its implementation on top of the MARVEL 3.0 multi-user environment.

The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants and fellowships from AT&T, BNR, DEC, IBM and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

keywords: multi-user environments, process-centered environments, process programming languages

1. Introduction

The concept of process modeling has increasingly attracted attention in the software engineering community, as evidenced by the upcoming 8th annual International Software Process Workshop and 2nd annual International Conference on the Software Process. A dominant theme is the notion of a process-centered environment, which operates as an ‘‘enaction’’ engine for a specific process modeling language (PML). Enaction enforces and/or automates the desired process (see, *e.g.*, [Thomas 91, Dowson 91]).

Many PMLs are based to some extent on notational paradigms originally devised for other purposes. For example, APPL/A [Sutton 90a] extends Ada, Funsoft nets [Gruhn 91] are based on Petri nets, AP5 [Cohen 86] uses OPS5-like production rules, Darwin [Minsky 91a] employs Prolog-like logic rules, Adele [Belkhatir 91] follows database triggers, HFSP [Katayama 89] is similar to attribute grammars, etc. Many PMLs have successfully been used to model part or all of the ISPW6 [Kellner 91] or ISPW7 [Heimbigner 91] ‘‘example problem’’, but it is generally agreed that some language paradigms are better than others at expressing different aspects of software processes [Kellner 90].

One of the most important distinctions among different styles of PMLs is their affinities towards global control flow *vs.* local constraints. *Control flow* directly addresses the required, permitted and/or prohibited (partial) orderings among process steps, generally but not always at a relatively global level of abstraction, whereas *constraints* specifically restrict the applications of individual tools and the modifications to product data in process steps, usually at a local level. Nearly all proposed PMLs are reasonably well-suited to one or the other aspect, and most are sufficient for both in the same sense that a Turing machine can express any computation, but few handle both kinds of modeling issues equally well. We believe that PMLs would best be constructed by selecting a *pair* of formalisms, one oriented towards control flow and the other towards constraints, and then combining these formalisms in a fashion that facilitates enforcement of both kinds of requirements. We use the terms "global" to refer to control flow and "local" to designate constraints, although these generalizations are not strictly accurate.

Our notions of global and local process formalisms are both distinguished from the *action languages* that appear in many process modeling systems. For example, AP5 specifies the process in a rule-based form whose actions are implemented in Common Lisp. The action portion of an Adele trigger has a distinctly different syntax from the condition part. Darwin’s object scripts can be implemented in any programming language. Additional formalisms might also be adopted to address data modeling, resource management, human-computer interaction, etc. for process-centered environments, but we are concerned in this paper primarily with the global and local issues of process modeling.

There are then two basic approaches to developing a process-centered environment based on a bi-level PML: a common enaction engine with an internal process database, or separate enaction engines interoperating on a common but external process database. The latter has been conjectured by Heimbigner [Heimbigner 92], but we know of no existing systems that support such interoperability among multiple process modeling formalisms. We present a working multi-user implementation of the former approach in this paper.

We selected the MARVEL Strategy Language (MSL) [Kaiser 88] for local constraints, and a variant of constrained expressions [Avrunin 86] described by Riddle [Riddle 91] for the global control flow component. MSL provides planning system-style rules for specifying the local conditions and effects on the data parameters of each individual process step, as well as the arguments to be supplied to tools used in the step. Constrained expressions are based on regular expressions, with sequencing, alternation, iteration and concurrent shuffle of process steps, augmented by send and receive primitives for synchronization among concurrent subexpressions. Constrained expressions were originally developed for specifying event patterns in concurrent systems, and similar formalisms have been widely used in both sequential [Bruegge 83] and concurrent [Bates 88, Ponamgi 91] debuggers.

We defined a new PML called the Activity Structures Language (ASL). MSL is a proper subset of ASL. The rest of ASL consists of Riddle's constrained expressions extended with data parameters, whose types are defined by data modeling facilities already provided by MSL. MARVEL's envelope language [Gisi 91] was used ''as is'' for defining tool interfaces. ASL was implemented by translation of the extended constrained expressions into MSL, followed by normal enaction of MSL using the multi-user MARVEL 3.0 kernel [Ben-Shaul 92]. The only modification to this 85,000 line system was the addition of a 750 line module to the process engine. The idea of implementing constrained expressions in terms of MARVEL was first suggested by Williams [Williams 88], but he did not mention the global *vs.* local dichotomy, which was the main purpose for our work.

We motivate our approach by presenting a small process segment — taken from the ISPW7 example problem — that requires modeling of both control flow and constraints. We briefly review some existing process modeling languages based on a single level (global or local) of specification, and introduce our designs for the ASL language and its enaction model. We show a portion of our solution to the example problem using ASL, and briefly compare to related work on bi-level process modeling languages. We finish with a description of the implementation, and summarize our contributions.

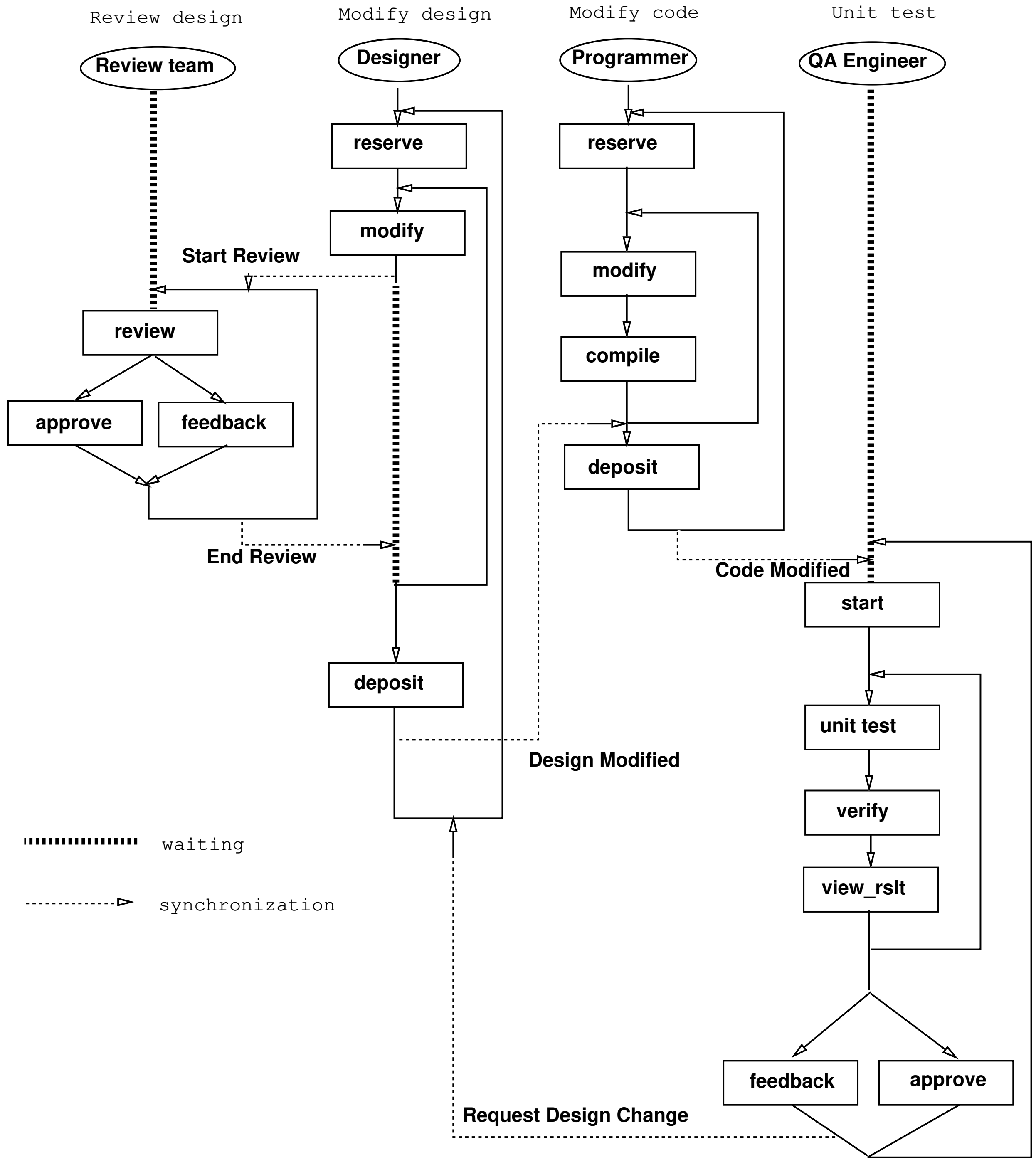
2. Motivation

The ISPW7 example problem, which consists of specifying a process for making several common types of modifications to a large software system, is the only known ''benchmark'' in the software process modeling community. We consider one representative subprocess, that of making the design and code changes necessitated by a requirements change. This subprocess begins after the proposed change has been approved and the requirements have been modified. This process segment, illustrated in Figure 2-1, includes four distinct user roles:

1. The designers responsible for maintaining the design specification.
2. The review team, which must approve the modified design before the code can be updated.
3. The programmers who make all changes to the actual system code.
4. The QA engineers who perform unit tests on the changed system components and ensure that they meet the specification as revised.

Design and code changes may begin in parallel with the reservation of the appropriate documents and source code modules. Once the design has been modified, the review team must approve it before it can be deposited into the public repository. This completes the modification of the design, but the code modifications are still pending. In the meantime, the programmers have been performing an edit-compile loop to change the code. They may have already completed their modifications, but are not allowed to deposit their code until the design modification has been finished. Note that the the "Design Modified" synchronization may arrive any time during the programmers' subprocess before the deposit, but we only illustrate the last possible time in the figure. Once the programmers have deposited, the QA engineers can begin their work. These testers may either approve the code, in which case the entire subprocess is complete, or may give feedback to the design team as to which tests were failed. The designers in turn may modify the design and/or require the programmers to fix the bugs found by the unit testers. This repeats until the code passes its unit tests.

Figure 2-1 shows an abstract control flow that hides the *criteria* for deciding among alternative paths, *i.e.*,



Boxes represent high-level process steps. Solid arrows represent control flow of the process within each role, and dashed arrows the synchronization between roles. An arrow pointing from one process step to another means that the first step must precede the second.

Figure 2-1: ISPW7 Example Problem: Teamwork Subset

whether or not to iterate. For example, the control flow specifies only that the designers must wait until the review team has completed either an approval or a feedback step before continuing their modification process segment. Thus it would be valid for the designers to deposit their changes immediately after receiving feedback from the reviewers stating that more changes are needed! Instead, the designers' choice of which branch to take should consider the actions of the review team. In particular, they should not have the option of exiting the loop and depositing the design unless the reviewers have approved it. Another example is the programmer's edit-compile loop, where the criteria for whether or not to iterate may be specific to the tools used, *i.e.*, whether or not the compiler detects syntax errors. Further, the programmers should not be permitted to deposit code that has not compiled successfully.

Some but not all global PMLs omit constructs for expressing such decision criteria while all local PMLs (by definition) provide facilities for indicating some conditions under which an individual process step should or should not be executed. On the other hand, global PMLs (by definition) always state control flow paths directly, but the constraints of local PMLs usually only imply potential precedence relationships among process steps, not all of which are appropriate.

3. Global PMLs

APPL/A [Sutton 90a] is an Ada-like process programming language. Process roles are represented as tasks and communicate *via* rendezvous. Within a task, conventional programming constructs specify the control flow among process steps. Decision criteria are expressed as conditionals at branch points, and user choice can be incorporated through I/O statements. Subroutines minimize duplication of process segments and can encapsulate conditions together with tools and data operations. Consistency can be enforced or relaxed by transaction-like constructs [Sutton 90b]. APPL/A arguably includes both global and local facilities, but is discussed here rather than in Section 8 because it does not seem ''bi-level''.

The Articulator environment [Mi 92] models processes as flow graphs in which each node represents a process step, and input/output dependencies can refine graph edges. A node passes through a sequence of states, beginning with None and normally proceeding through Allocated, Ready and Active before finishing in the Done state. Other states may be reached under exceptional conditions, which require external repair to continue the process [Mi 91]. Once a node enters the Done state, its successor nodes are enabled. A user has the freedom to choose among multiple enabled nodes and to switch back and forth among in-progress subgraphs. Multiple users may operate simultaneously, synchronizing implicitly through the precedence relations among subprocess nodes. Melmac [Deiters 90] operates similarly to the Articulator, with processes specified using a Petri net-like formalism called Funsoft nets [Gruhn 91]. Multiple transitions can be fired simultaneously to reflect simultaneous participants in the process. Funsoft nets attach conditions to transitions, restricting their application. There is no notion in either system, however, of attaching conditions to individual tools or data.

Activity structures [Riddle 91] are an abstract process modeling formalism based on constrained expressions [Avrunin 86]. Each event models a process step. A shuffle operator specifies that its operand subexpressions are concurrent and their execution may be interleaved arbitrarily, except as explicitly restricted by `send` and `receive` primitives in the subexpressions. Concurrent repetition is a shuffle of an indeterminate number (zero or more) copies of a single operand subexpression. Activity structures are a pure control specification, and cannot express decision criteria or other conditions, not even the number or types of data parameters to activities.

Global PMLs are naturally strong in modeling the control flow aspects of both single and multiple user processes. They vary in the degree to which decision criteria, including local data- and tool-specific constraints and policies, can be specified. There seems to be a tradeoff between user freedom to select among valid activities and capabilities to specify which activities are valid. APPL/A has the most developed mechanism for stating and enforcing conditions, but prescribes exactly the limited cases in which user choice is possible. At the other end of the spectrum, the activity structures formalism burdens the users with essentially all decisions, although it prohibits invalid process sequences.

4. Local PMLs

MARVEL [Kaiser 88] encapsulates each process step in a rule that defines its command name and typed data parameters. The condition's predicate clause must be satisfied prior to invoking the tool envelope, and each effect specifies a set of subsequent assignments. MARVEL backward chains in an attempt to satisfy the condition of the selected rule, whereas forward chains attempt to trigger the implications of the rule's effect. Mandatory chaining to maintain consistency is distinguished from automation chaining, which can be restricted by `no_backward`, `no_forward` or `no_chain` directives [Barghouti 92a]. Synchronization is implicit in data access, since MARVEL 3.0 enforces serializability among concurrent consistency chains and automation rules [Barghouti 92b]. Merlin [Schafer 92] is similar to MARVEL. Backward chaining on logic programming rules establish which process steps are currently valid in the user's current role(s), and then forward chaining on planning system rules guide the user through an automated process sequence.

The Common Lisp Framework (CLF) [CLF 88] implements its process in AP5 rules with data-driven forward chaining [Cohen 86]. Separate rule bases specify consistency preservation and automation. Process steps that imply violation of consistency constraints are either repaired or aborted, while automation rules are executed after commit. Concurrency control operates through a ''checkout'' mechanism with change notification [Narayanaswamy 91], and data rendered inconsistent by importing from another user can be marked for later restoration of consistency [Balzer 91]. Adele 2's triggers [Belkhatir 91] are similar to AP5. `PRE` and `POST` triggers correspond to consistency rules, `FAIL` to repairs, and `AFTER` triggers to automation rules. Adele 2 employs hierarchical checkout [Belkhatir 92].

Darwin/1 *laws* [Minsky 91a] govern their own evolution as well as the process. Laws are syntactically and semantically identical to Prolog rules with unification and backtracking. Message passes among objects model process steps, with tools and users represented as active objects. Consistency is enforced by blocking offending messages: message sends are evaluated as goals with respect to the law, and either succeed by deriving a delivery or fail. The law can also replace the message contents and/or intended recipient of any message. Law-governed systems have been shown applicable to distributed protocols [Minsky 91b] and multi-user configuration management [Minsky 90].

Local PMLs are oriented towards modeling local constraints and policies. They all can state a condition that must be true (or derivable) before initiating a process step. Forward chaining PMLs, such as MSL and AP5, also require that consistency implications of a tool invocation or data modification be carried out atomically. Local PMLs are generally reactive, commencing relatively short enaction segments in direct or indirect response to user commands. It is theoretically possible, however, to enact an entire process role as a single rule chain. The Brownston *et al.* OPS5 book [Brownston 85] dedicates many pages to ''hacking'' control flow, by introducing state variables to force rules to chain in the desired order. Chaining directives and distinctions between consistency and automation chaining aid in such hacking.

5. ASL Syntax and Semantics

Our Activity Structures Language combines activity structure definitions (AS-defs) with MSL in a single but bi-level PML. ASL's notation for activity structures differs from Riddle's both cosmetically, in the concrete syntax of operators, and in the abstract syntax and semantics of the shuffle operator. Ampersand (&) is used to denote shuffle, and unlike Riddle's infix shuffle, is a prefix operator with a parenthesized, comma-separated list of an arbitrary number of operand subexpressions; this more clearly expresses multi-participant processes and better fits our representation of concurrent roles, as explained in Section 6.2. The concurrent repetition operator is omitted from the current version of ASL. The basic syntax of the activity structures subset of ASL is illustrated in Figure 5-1.

```
task4:
 &(
    &(
       ( Edit ; ( send 1 ; receive 3 ; ( Edit | epsilon ) )* )
     ,
       ( Edit ; ( send 2 ; receive 4 ; ( Edit | epsilon ) )* )
      ) ;
     Assemble ; Distribute
   ,
    ( ( receive 1 ; Proof ; send 3 )
      |
      ( receive 2 ; Proof ; send 4 )
    )*
   )
```

Activities are identified by name, with `epsilon` representing the null activity (usually as an alternative for an optional activity). The `send` or `receive` primitives are associated with a number specifying a send or receive synchronization operation, respectively, on the numbered channel. Semicolon (`;`) is the sequencing operator; vertical bar (`|`) the alternation operator; and star (`*`) is the repetition operator. Repetition has the highest precedence, and alternation the lowest among the regular expression operators.

Figure 5-1: Simplified ASL Example

5.1. Data Constraints

Riddle's activity structures cannot specify any data parameters on which the activities operate. In the small process shown in Figure 5-1, for example, `Edit` and `Proof` might each apply to a single document file, and `Assemble` and `Distribute` to a manuscript consisting of multiple document files. Signatures similar to those used in programming languages, which specify the number, order and types of parameters, are not sufficient. It must also be possible to specify *semantic* constraints such as the first proofreader `Proof`s the <u>same</u> document file(s) that the first editor `Edit`s.

Therefore we introduced in ASL a notion of formal parameter *bindings*. The general idea is that each formal parameter symbol is bound to an object, and each occurrence of the same symbol within a single AS-def refers to the same object. A symbol may be designated as *multiple*, meaning that it can be bound to multiple objects and any one of these objects may serve as the actual parameter. Finally, to allow flexibility in the process, a symbol may be designated as *resetable*, meaning it can be rebound to different objects while a process is in progress. The desired option is represented in ASL by prefacing each formal parameter symbol with a keyword indicating how that symbol becomes bound. The keywords are `set`, `use`, `reset`, `multiset`, `multiuse` and `multireset`; `set` is the default and may be omitted.

```
( edit[set ?c:CFILE] ; compile[use ?c:CFILE] )*
```

indicates that the actual parameter provided to both `edit` and `compile` must be an instance of `CFILE` or one of its subclasses. Further, the correctly typed actual parameter provided by the user on the first invocation of `edit` is bound to the "?c" symbol (all parameters start with "?") and the current object bound to "?c" is required as the actual parameter to `compile`. This process fragment is very restrictive, in that the second and any subsequent iteration must `edit` and `compile` the same `CFILE` as the original (*i.e.*, "?c" is bound only the first time "set ?c" it is encountered). However,

```
( edit[reset? c:CFILE] ; compile[use ?c:CFILE] )*
```

allows "?c" to be rebound by every `edit` command, while the `compile` is still limited to the same file that was just `edit`ed. Finally,

```
( edit[multiset ?c:CFILE] ; compile[multiuse ?c:CFILE] )*
```

adds every actual parameter selected by the user when invoking the `edit` command to the group of objects bound to "?c". It allows the user to select any object currently bound to this group as the actual parameter to `compile`. The `multireset` option replaces the entire group of objects bound to a symbol with the newly selected object.

In addition to this *implicit* binding mechanism as a side-effect of a user's activity invocation, the ASL implementation also supports *explicit* binding commands `Bind` and `Unbind`. These allow a user to establish bindings *a priori*, before a (sub)process begins, and/or to change and remove bindings established by the implicit mechanism. To effectively disable implicit binding and thus require explicit binding, all parameters can be annotated as `use` or `multiuse`. `Bind` complements the `Assign`, `Activate`, etc. commands of the ASL enaction model, presented in Section 6.2.

Parameter types and the composition of the objectbase are defined by object-oriented classes, *e.g.*, `CFILE`, written in the data modeling subset of MSL. Classes declare *state* attributes of primitive or enumerated types; *file* attributes, either text or binary; *component* attributes, an instance or set of instances of a given class; and *link* attributes, also an instance or set of instances of a given class. Initial values may be specified for state attributes and filename extensions for file attributes. Component attributes allow construction of composite objects, *i.e.*, when a parent object is deleted all of its components are likewise deleted, while link attributes support an arbitrary graph of relationships among objects. Multiple inheritance is supported, with conflicts treated as errors.

5.2. Tool Constraints

Activity structures cannot specify any constraints on tool invocations other than global control flow and synchronization. For example, a process might require that a file be `reserve`d before `edit`ing and that it be `edit`ed only by the same user who `reserve`d the file. A process might further require that a file's source timestamp be more recent than its object timestamp to be `compile`d, but that the timestamp not be updated if there were no changes made during the `edit`. An access control mechanism could be used to enforce the owner restriction (although this was not implemented until MARVEL 3.0.1), ASL sequencing could specify the ordering restrictions (although a context-sensitive rather than regular grammar would be needed to allow `reserve` arbitrarily preceding `edit` and `edit` arbitrarily preceding `compile`), and separate `edit` and `view` activities could distinguish between update and read-only operations (but would require an *a priori* decision not to modify). These more flexible constraints can be specified locally in the process modeling subset of MSL, using *rule skeletons*. An example is shown in Figure 5-2.

The MSL `no_chain` directives on every predicate and assignment are needed to ‘‘turn off’’ the MARVEL kernel's default behavior of opportunistically backward and forward chaining whenever possible. If the `no_chain` directives were not present in Figure 5-2, MARVEL would automatically invoke `edit` on a

```
edit[?c:CFILE]:
    :
    (and no_chain(?c.locker = CurrentUser)
         no_chain(?c.reserve_status = CheckedOut))
    { EDITOR edit ?c.contents }
    no_chain(?c.time_stamp = CurrentTime);
    no_assertion;
```

The first line gives the rule signature, which must match an activity. The ":" in the second line separates between an optional binding clause, not needed here, and the predicate clause. The predicate clause can consist of arbitrarily nested `and`s and `or`s. "?c.<attribute_name>" specifies an attribute that must be defined in `CFILE` or one of its superclasses. `CurrentUser` and `CurrentTime` are built-in operations that return the current userid and timestamp, respectively, while `CheckedOut` is one of the enumerated values for the `reserved_status` attribute. The tool envelope invocation are given between braces "{...}"; the details of envelopes are outside the scope of this paper, see [Gisi 91]. This is followed by one or more ";"-separated effects corresponding to the multiple possible results of the tool invocation, each an assertion clause — possibly an `and` of multiple assignments to attributes — or `no_assertion` — if the tool result indicates ‘‘no change’’.

Figure 5-2: Simple Rule Skeleton

file as soon as the file was `reserve`d and automatically invoke `reserve` on a file whenever a user attempts to `edit` it; the former case is more serious than the latter, since backward chaining occurs only if the condition is not already satisfied. Chaining directives are discussed further in Section 6.

A more complex local constraint might restrict the set of `HFILE`s that can possibly be included in a `CFILE`, *e.g.*, those that are components of the same `MODULE`. This would be enforced by automatically gathering these `HFILE`s as essentially local variables to the `compile` activity, accessed through a query on the objectbase. Note that the same goal could be achieved in ASL employing the `Bind` command and the `multiuse` formal parameter binding, but our alternative lessens the burden on the user in selecting parameter objects when the invoking an activity. The original purpose of the bindings clause in MSL was to support chaining from one activity to another, to automatically determine parameters through a structural inversion algorithm when the chained activities operated on different but related parameters [Heineman 91]. An example is shown in Figure 5-3.

For ASL we use the term "rule skeleton", rather than simply "rule", since the ASL implementation inserts additional predicates and assignments to enforce the control flow specified by activity structures. When the same activity occurs multiple times in an AS-def, its rule skeleton is cloned and then different such predicates and assignments are added separately for each occurrence. Williams proposed an alternative approach [Williams 88], where conditions and effects as well as control flow are effectively specified separately for each occurrence of an activity; he did not address the problem of local variables. These implementation issues are explained in Section 9.1.

6. ASL Enaction Model

The ASL enaction model is intended to be user-driven and proscriptive. The environment does nothing until the user enters a command. If the command corresponds to an activity (as opposed to one of the built-in commands such as `help`), the environment accepts the command only if it is valid according to the current (sub)process as specified by the current activity structure, considering <u>both</u> the local constraints and the state of the process with respect to the specified global control flow.

If the user enters the `compile` command without first executing the preceding `edit` command in

```
( edit[set ?c:CFILE] ; compile[use ?c:CFILE] )*
```

```
compile [?c:CFILE]:
    (and (exists MODULE ?m suchthat (member [?m.files ?c]))
         (forall HFILE ?h suchthat (member [?m.includes ?h])))
    :
    (or no_chain(?c.time_stamp > ?c.object_timestamp)
        no_chain(?h.time_stamp > ?c.object_timestamp))

    { COMPILER compile ?c.contents ?c.object_code ?h.contents "-g" }

    (and no_chain(?c.status = Compiled)
         no_chain(?c.object_time_stamp = CurrentTime));
    no_chain(?c.status = ErrorCompile);
```

The `exists` part of the bindings clause finds the single `MODULE` containing the supplied `CFILE` and binds it to the "?m" variable. Then the `forall` part collects the set of `HFILE`s contained in its `includes` component attribute, binding all of them to the "?h" variable. A bindings clause may contain arbitrarily many `exist` and `forall` parts, as needed. Since "?h" represents a set of objects, a predicate referencing "?h.<attribute_name>" is repeatedly applied to each member of "?h".

Figure 5-3: Rule Skeleton with Local Variables

the `compile` would be disallowed (even if the data parameter had been specified as `set` or `reset` and the condition stated in the rule skeleton were satisfied). If the user enters the `edit` command at a valid point, after the beginning of the process fragment or after a `compile`, that command is accepted and executed. However, the `compile` command is not then automatically initiated afterwards even though it is now valid (and presumably invokes a batch processing tool that does not require user interaction); instead, the environment waits for the user to select it. Automation is possible only if the explicit chaining directives mentioned in Section 5.2 are omitted from the rule skeletons. In that case, local constraints would be enacted by automation, but global control flow still only by enforcement.

This approach contrasts with the conventional MARVEL enaction model sketched in Section 4, where as much as possible of the process is enacted by opportunistic forward and backward chaining, although restricted by any explicit automation chaining directives included in the process model. One alternative would have been to match the MARVEL enaction model. In this case, selection of the `compile` command by the user would have triggered the `edit` activity on the same `CFILE`, and selection of the `edit` command would have resulted in initiation of the `compile` activity. A compromise would be to support automation of prerequisite activities, since they could be thought of as implicitly requested by the user's command to fulfill the goal represented by the selected activity, but not subsequent activities.

In a purely prescriptive model, the environment would start at the beginning of the process and enact the entire process step by step *via* forward chaining. Decisions at branch points might be made randomly, which might be appropriate for a process simulation, or by prompting the user, which might be useful for training. This style of enaction was in fact followed by Gaede [Gaede 91] in an earlier attempt to implement activity structures on top of the single-user MARVEL 2.6. The non-conventional (for us) ASL enaction model demonstrates the versatility of the MARVEL kernel.

6.1. State Representation: Activity Structure Objects

Activity structure expressions are presented to the user(s) of the environment as *activity structure objects*, or *AS objects*. The attributes of an AS object represent the current state during the enaction of the corresponding expression, and the value of one distinguished attribute is the expression represented by the object. We chose to represent AS objects as first-class objects, but this is not necessary for our enaction model. An activity graph could have instead been stored and displayed separately.

There is always at least one AS object for each AS-def, and this is a composite object if the AS-def contains any shuffles. This *root AS object* has children representing each of the operands of shuffle operators at the top level of the activity structure; its children may in turn contain other AS objects representing shuffle operand subexpressions of nested structures, to an arbitrary depth. The AS object hierarchy thus reflects the nesting of shuffles in the AS-def; individual activities are not explicitly represented. The hierarchy is displayed to the users, showing this structure and allowing browsing. We chose this granularity of representation to materialize the work breakdown in the AS object hierarchy, with each user's role in the process represented as an AS object. However, smaller granularities would have been feasible. For example, Gaede's activity structure implementation and Heimbigner's ProcessEngine both represent individual activities as first-class objects.

Figure 6-1 shows a MARVEL display window including the AS objects — the `task5.1` object and its children — for our example process segment (from Figure 2-1). This environment, and particularly the objectbase, is a very sparse, simple one that we have used only for testing our ISPW7 process model. For example, the `proj` project contains only one program, `pgm`, which in turn has its entire source represented by a single object, `src`. The objectbase is not intended to be realistic.

6.2. ASL Process Management

A process is generally specified in ASL as a single top-level activity structure, broken down into individual roles by the use of the shuffle operator (actually, a process can consist of several AS-defs, but each one is enacted separately as described here). To begin the process, a user (usually a manager at this point) will `Instantiate` the top-level AS object and `Activate` it. The `Instantiate` command executes a *static* instantiation: it creates not only the top-level object, but all of its descendants, even if some of the nested shuffles may never be reached due to choices among alternatives. This decision was made for implementation reasons explained in Section 9.1. The alternative, which we consider preferable, would be to allow *dynamic* instantiation, where child AS objects might not be created until just before they are to be `Activate`d, as with tasks in Heimbigner's ProcessEngine. With dynamic instantiation, it would be relatively simple to implement concurrent repetition as a shuffle for which any number of identical children can be instantiated, either before or during its execution.

The manager who instantiated the top-level AS object will then `Assign` each of the roles within the top-level shuffle to a particular user (perhaps himself), who can then `Activate` their subsidiary AS objects to perform their roles. Each role must execute in a separate user interface client of the MARVEL server (the client/server architecture is presented in [Ben-Shaul 92]); that is, a single client may not perform multiple roles simultaneously. These roles may in turn contain nested shuffle operators. When such a subsidiary shuffle is reached during process enaction, it is handled in the same way, with all of its child AS objects first being `Assign`-ed to particular users, who can then `Activate` them.

A user's current role can be suspended at any point using the `Detach` command, and resumed later with the `Attach` command. In between, the user can `Activate` (or `Attach` if previously `Activated`) to other AS objects that have been `Assign`ed to him. When the single-user process within a role is

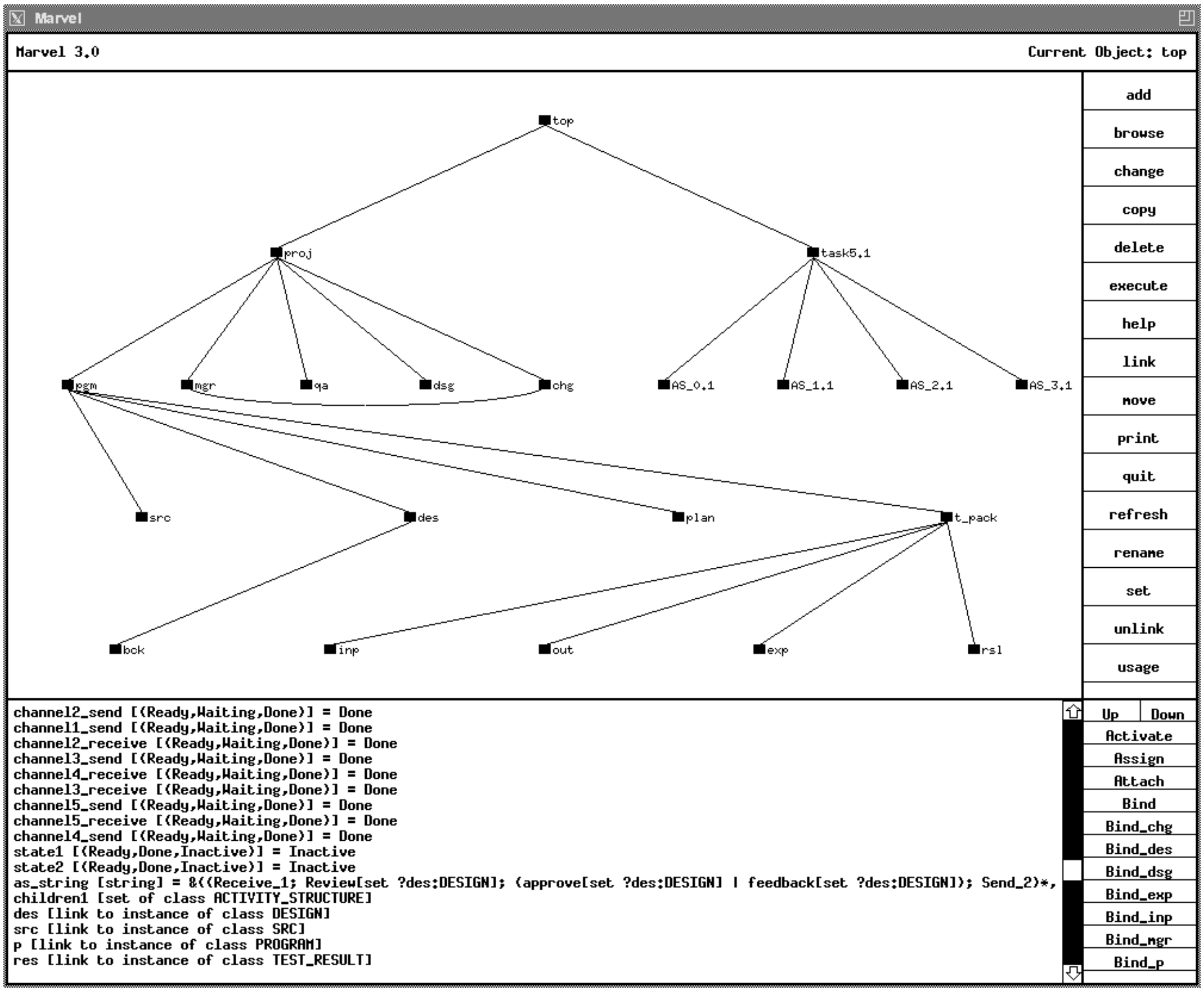


Figure 6-1: ISPW7 Example Problem: Activity Structure Object Hierarchy

finished, the user reports this successful completion by selecting the `Terminate` command. A role may also be exited unsuccessfully at any point by using the `Deactivate` commands, but later on it must again be `Activate`d and eventually `Terminate`d before its parent activity structure will be permitted to continue. In other words, all child AS objects of a shuffle operator must terminate successfully before the parent will be allowed to continue past the shuffle.

6.3. Role Synchronization

As in Riddle's activity structures, the `send` and `receive` primitives take a *channel* number as their argument. When a `send` is executed on a channel, it may be matched by any `receive` on the same channel number, provided both are embedded (at any arbitrary and perhaps different depths) within the same top-level activity structure. Thus in Figure 5-1, only the first editor may synchronize with the proofreader using channels 1 and 3, and the second editor can synchronize only using channels 2 and 4; pairs of channels prevent roles from intercepting their own messages (and thus it is possible for a role to retrieve the same messages it sent, if desired for some reason). In a variation with a single pair of channels, either editor could `send` on channel 1 and either editor (not necessarily the same one that requested the proofreading) could `receive` the reply on channel 2. In any case, however, a `send` is always matched with exactly one `receive` on the same channel and *vice versa*.

These primitives are available to the user as explicit `Send_x` and `Receive_x` commands, for each channel number `x`. Neither command ever blocks. There is no explicit notion of a ‘‘full’’ channel, so a `Send` always succeeds immediately (channels are implemented as general semaphores represented internally as integers). A `Receive` on an ‘‘empty’’ channel fails, and must be repeated later on by the user (or the user may select an alternative activity, if provided for in the process). It was important to avoid blocking, because that would have prevented the MARVEL 3.0 client from doing anything else in the meantime (although a user can execute an arbitrary number of clients in different windows). However, to avoid busy-waiting on an ‘‘empty’’ channel, the user requesting the failed `Receive` is automatically added to a list of users, who are notified by electronic mail when the next `Send` is executed on that channel. When the mail notification arrives, the affected users can retry the `Receive`, at which point one will be successful and the others will fail again. A simple alternative would have been to notify only the first user on the list, to effect a queuing policy, with automatic removal from the queue if a user goes on to select an alternative activity (*e.g.*, a `Receive` on another channel).

7. ASL Example

```
task5:
&(
    #
    # review design
    #

  (receive 1; review[?des:DESIGN];
   (approve[?des:DESIGN] | feedback[?des:DESIGN]); send 2)*
 ,

    #
    # modify design
    #

  (reserve[reset ?des:DESIGN]; modify[?des:DESIGN];
   (send 1; receive 2; (modify[?des:DESIGN] | epsilon))*;
   deposit[?des:DESIGN];
   send 3; receive 4)*
 ,
    #
    # modify code
    #

  reserve[reset ?src:SRC]; (modify[?src:SRC]; compile[?src:SRC])*;
  receive 3; deposit[?src:SRC]; send 5
 ,
    #
    # unit test
    #

  receive 5; start[?p:PROGRAM];
  (unittest[?p:PROGRAM]; verify[?p:PROGRAM];
   view_rslt[?res:TEST_RESULT])*;
  (approve[?p:PROGRAM] | (feedback[?des:DESIGN]; send 4))

 )
```

Figure 7-1: ISPW7 Example Problem: Teamwork in ASL

The complete ASL specification for the ISPW7 example contains 36 rule skeletons consisting of 509 lines, including comments, and 5 activity structure definitions consisting of 113 lines, exclusive of the data model (classes and tool definitions, the latter not addressed in this paper). Figure 7-1 shows the AS-def specifying the global control flow of the subset of the ISPW7 problem discussed in Section 2. As written, this allows only a single instance of each of the four roles in the problem; if multiple instances of a role were desired, a concurrent repetition (not implemented) could be nested around the subexpression for that role in the top-level shuffle. The other four AS-defs deal with the initial approval of the change request, the manager’s scheduling of the work to be done, the creation of the testing plan, and the building of the test package to be used by the unit testers.

Three of the rule skeletons, concerned with modifying, compiling and depositing a source file, are shown in Figure 7-2. The rule skeletons are textually the longest part of an ASL specification, but are simpler to write than a rule-based specification of the entire process, since they contain no embedded global control information. For example, the only condition of the `modify` rule skeleton is that the source file in question must first have been reserved. This check is actually redundant in the process specification as written, since the AS-def in Figure 7-1 requires the developer to modify only the last source file reserved. If, however, it was desired to remove this part of the global control flow as unnecessarily restrictive, the activity structure would be modified to use `multiset` instead of `reset`:

```
reserve[multiset ?src:SRC]; (modify[?src:SRC]; compile[?src:SRC])*;
```

Then the local constraint in the rule template would again be necessary. Leaving it in the rule template as a "sanity check" costs very little, while avoiding introduction of errors as the process evolves.

8. Related Work on Bi-Level PMLs

Heimbigner’s ProcessEngine [Heimbigner 92] is not a complete environment, but an environment component — a process state server with a predefined representation for steps, called *tasks*, hierarchical breakdown of tasks and control flow among tasks. The types of task input and output parameters can be specified, and the server supports a mechanism for binding parameters across multiple tasks. This representation is not intended to replace a PML. Instead, Heimbigner describes an intriguing notion of *process constructors*, clients that construct process specifications using the ProcessEngine primitives, and *process constrainers*, other clients that check additional constraints on the process. Process constructors and process constrainers correspond roughly to our notions of global and local specification, respectively, and this approach should permit interoperability among multiple global and local PMLs.

EPOS [Liu 91] is closer to bi-level than most rule-based environments. Its primitive *tasks* are essentially planning system rules, but with additional input/output *constraints* that achieve parameter passing among tasks. A sophisticated planner refines high-level tasks into partially ordered sequences of subtasks, ultimately rules. The resulting *hierarchical plan* specifies the global control flow, which the EPOS execution manager can then enact. This approach breaks down when a (sub)task fails to achieve its goal, but the authors propose an incremental replanning mechanism. Process enaction currently operates only in isolated single-user workspaces, but work is underway on facilities to support process-directed notifications and version merging across workspaces [Conradi 91].

Oikos [Ambriola 90] is implemented in Extended Shared Prolog, which combines Linda-like tuple spaces with a multi-process variant of Prolog. The top layer consists of a hierarchical structure of *blackboards*, which represent the work breakdown as mutually recursive *process* and *environment* blackboards. Process blackboards generally represent predefined processes while environments offer freedom to choose among available processes. The bottom layer is an individual blackboard, containing facts and local constraints expressed by Prolog rules. These are organized into *theories* that represent process roles. Theories, data (documents) and tools are accessed through predefined *services*, with a checkout model for

```
#
# modification of source
# reset compile status
#
modify[?src:SRC]:
  :
  (?src.reserve_status = CheckedOut)

  { MODIFY modify ?src.contents }

  (and no_forward(?src.modify_status = Modified)
       no_chain(?src.compile_status = NotCompiled)
       (?src.last_modified = CurrentTime));

#
# compile source
#
compile[?src:SRC]:
  (exists PROGRAM ?p suchthat (member[?p.sources ?src]))
  :

  { COMPILE compile ?src.contents ?p.exec }

  (and no_chain(?p.build_status = Built)
       no_chain(?src.compile_status = Compiled));
  (?src.compile_status = NotCompiled);

#
# this rule will be fired when the modification of code has been
# finished completely based on the corresponding modified design.
#
deposit[?src:SRC]:
  (and (exists PROGRAM ?p suchthat (ancestor[?p ?src]))
       (exists DESIGN ?des suchthat (member[?p.designs ?des])))
  :
  (and no_chain(?des.modify_status = Approved)
       no_chain(?src.compile_status = Compiled))

  { RCS checkin ?src.contents }

  (and no_chain(?src.reserve_status = CheckedIn)
       (?src.end_code = CurrentTime));
```

The `no_forward` directive prevents forward chaining out of an effect assignment or into a condition predicate while permitting backward chaining. `no_backward` has an analogous effect.

Figure 7-2: ISPW7 Teamwork Example: Sample Rule Skeletons

concurrency control. Tools are executed by a workspace service, with escapes to the operating system. Theories may specify synchronization through insertion and removal of facts on shared blackboards.

HFSP [Katayama 89] is a truly bi-level PML based on attribute grammars. The global aspect of HFSP appears in its context-free grammar: Productions represent sequencing, alternation, iteration and hierarchical structuring (recursion) of process steps. Terminal symbols represent invocations of individual tools, which are accessed through interfaces defined in a *process enaction script*. HFSP's local formalism consists of *decomposition conditions*, which are placed on productions to restrict their application, and attribute definitions. Their semantic functions specify the derivation of input parameters from the parent subprocess and output parameters from the children. Process steps can be executed concurrently if there are no data dependencies among their parameters. However, the purely functional nature of attribute grammars leads to convoluted mechanisms for expressing user interaction and requires expensive re-evaluation of all affected attributes throughout the tree after a local update. A related effort by the same group, Objective Attribute Grammars [Shinoda 89], supports message passing and imperative computations on attributes treated as objects, with a distinction between propagated *static* attributes and non-propagated *dynamic* attributes.

The PDL project implemented a bi-level PML by translation into another PML [Iida 91], although the latter does not constitute one of the levels as does MSL in ASL. Processes are defined by context-free grammars extended with *restriction conditions* on data parameters of process steps; it is not clear whether these restrictions are defined once for each step, as what we call a local constraint, or separately for each occurrence in some production. The enaction model is a menu-based navigation system, as in Merlin, although the authors postulate an monitoring model analogous to our ASL environment.

A process is translated into the underlying Process Description Language (PDL) [Inoue 89], a functional process programming language. The PDL process program, called a *script*, is then refined manually from an abstract to a concrete form that can be directly enacted by the PDL interpreter. It seems unlikely that the concrete form is quite as concrete as implied by the authors, referencing actual data parameters (*i.e.*, specific file names), as opposed to construction of arbitrary products whose composition is determined while the process is in-progress. In any case, the interpreter supports a general escape mechanism, whereby undefined functions in the process program can be defined interactively during enaction, which would presumably permit the user to supply new data parameters on the fly. Both PDL and the bi-level PML are currently single-user, with no support for multi-participant processes or concurrent process roles, although multiple tools can be executed in parallel.

9. Implementation

9.1. ASL Translator

An ASL process specification consists of one ASL file and arbitrarily many imported MSL files. Tool envelope files are also required. The ASL file contains classes, tool definitions and rule skeletons written in MSL, plus one or more named activity structure definitions. The AS-defs are translated to a pure MSL specification and loaded into the MARVEL kernel for enaction.

Each AS-def translates to an MSL class whose attributes include state variables that track the progress of process enaction. This class, whose name is taken from the AS-def, is its *root class* in the composite object hierarchy. Each shuffle operator in an AS-def is similarly translated to a set of MSL classes, one for each operand subexpression, whose instances are AS objects. Each AS-def is also broken down into a hierarchy of regular expressions. The top-level AS-def is transformed to a single regular expression, where each shuffle operator together with all its operands is effectively replaced by a single symbol. Then each operand subexpression is transformed, with each of its own nested shuffle operators replaced, and so on recursively. A designated string attribute of each AS object class is initialized to text of the corresponding regular expression, for display during enaction. Each regular expression is translated to a set of rules representing the corresponding deterministic finite state automaton. Each transition in this DFSA then produces a single rule representing a single process step.

The generated rules are derived from the supplied MSL rule skeleton for the corresponding activity, and contain its signature, binding clause, predicate clause, tool invocation, and effects. The translator adds predicates on state attributes to constrain each rule to apply only when the DFSA is in the appropriate state, and assignments to update the state attributes following a transition. An additional parameter is added to each signature, whose type is the AS object class for the corresponding top-level AS-def or shuffle operand. Independent of the rule skeletons, rules are generated for each `Send` or `Receive` transition. Other rules correspond to the `Instantiate`, `Attach` and `Detach` commands, the `Assign`, `Activate`, `Deactivate` and `Terminate` commands for each AS object class, and the `Bind` and `Unbind` commands for each formal parameter of each activity that occurs in an AS-def.

The translator also outputs a tool envelope and several Unix shell files. The envelope for the `Instantiate` command is always the same, but a different shell file is generated for each AS-def. The envelope prompts the user for the name of the desired AS-def and runs the corresponding shell file, which constructs a MARVEL command script. The envelope then forks a MARVEL command line client (an alternative to the graphical user interface), giving the option to treat the script as its input. The user-level commands in the script construct the AS object hierarchy. This elaborate mechanism is necessitated by some limitations of MARVEL 3.0's MSL. The user cannot invoke the `Instantiate` rule with the (string) class name, because MSL rules accept only objects as parameters. The AS object instances are created from the ‘‘user interface’’ of a batch client because the only built-in operations available for rule effects are assignment of literals, `link` and `unlink`; is not possible to add or delete object instances from MSL.

9.2. ASL Run-Time Support

We added the Activity Structures Manager (ASM) module to the MARVEL kernel to support (1) an extended context mechanism to associate a current AS object with each client, (2) the implicit parameter binding facility, and (3) an option for ‘‘atomic’’ activities (not discussed in this paper). ASM interfaces to the process engine at three roughly corresponding points. When a user command is resolved to a list of possible rules according to the actual parameter types and MSL's overloading policy [Barghouti 90], ASM automatically supplies the current AS object as an implicit parameter. A rather inefficient search finds the AS object whose clientID attribute, set by the `Attach` and `Detach` rules, matches the current client. ASM is also called after a rule's condition has been satisfied, to enforce the binding requirements and establish any new bindings. It parses formal parameters names to obtain the binding directives, avoiding syntactic changes to MSL; thus we can add ASL specifications to any existing MARVEL rule sets. At each point, only a single line of code and two or three lines of compilation conditionals were added.

9.3. Translator Output for ASL Example

The ASL translator generates 134 rules, exclusive of the numerous binding rules, for the five activity structure definitions in the complete example specification. These rules consist of 1255 lines, with no comments. The generated state machines, five for the AS-defs and four for shuffle operands, contain a total of 40 states. Of the generated rules, 35 handle process management (instantiating, assigning, etc.), 41 synchronization, 20 internal state bookkeeping, and the remaining 38 invoke tools.

Figure 9-1 shows the tool invocation rules generated from the three rule skeletons of Figure 7-2. Each generated rule has one added parameter, for the current AS object, and the binding annotations have been concatenated with the formal parameters. From 23 lines of rule skeletons, the translator generated only 33 lines for these three rules, by adding one state predicate to each skeleton's predicate clause and two state assignments (leaving one state and entering the next) to each skeleton's effect. An alternative state representation, with one multi-valued state variable rather than many binary variables, would reduce the latter to a single assignment. Thus most of the expansion from translating ASL into MSL is due to other aspects of the ASL enaction model, not enforcing global control flow.

```
modify [?set_src:SRC, ?SO:AS_2]:
  :
  (and (?set_src.reserve_status = CheckedOut)
       no_forward(?SO.state1 = Ready))
  { MODIFY modify ?set_src.contents }
  (and no_forward(?set_src.modify_status = Modified)
       no_chain(?set_src.compile_status = NotCompiled)
       (?set_src.last_modified = CurrentTime)
       (?SO.state1 = Inactive)
       (?SO.state5 = Ready));

compile [?set_src:SRC, ?SO:AS_2]:
  (exists PROGRAM ?p suchthat (member [?p.sources ?set_src]))
  :
  no_forward(?SO.state5 = Ready)
  { COMPILE compile ?set_src.contents ?p.exec }
  (and no_chain(?p.build_status = Built)
       no_chain(?set_src.compile_status = Compiled)
       (?SO.state5 = Inactive)
       (?SO.state1 = Ready));
  (and (?set_src.compile_status = NotCompiled)
       (?SO.state5 = Inactive)
       (?SO.state1 = Ready));

deposit [?set_src:SRC, ?SO:AS_2]:
  (and (exists PROGRAM ?p suchthat (ancestor [?p ?set_src]))
       (exists DESIGN ?des suchthat (member [?p.designs ?des])))
  :
  (and no_chain(?des.modify_status = Approved)
       no_chain(?set_src.compile_status = Compiled)
       no_forward(?SO.state2 = Ready))
  { RCS checkin ?set_src.contents }
  (and no_chain(?set_src.reserve_status = CheckedIn)
       (?set_src.end_code = CurrentTime)
       (?SO.state2 = Inactive)
       (?SO.state3 = Ready));
```

Figure 9-1: ISPW7 Teamwork Example: Generated Rules

9.4. Status

The translator consists of about 6300 lines of C code (including 150 lines of headers), 2900 lines of YACC and 750 lines of Lex. Eight files were reused essentially unchanged from MARVEL's loader, which parses MSL files into the kernel's internal form. The seven added files include approximately 3500 lines of C, 800 lines of YACC and 80 lines of Lex. ASM is a single file of about 750 lines of C. The translator and MARVEL system run on SPARCstations (SunOS 4.1.1) and DECstations (Ultrix 4.2), with X11R4 windows.

The ASL translator and ASM have recently been upgraded to compatibility with MARVEL 3.0.1. This involved about 30 lines of code spread over three files: An attribute that conflicted with the new access control scheme was renamed, and an initialized state attribute was declared in every AS object class to work around a bug in MSL's inheritance semantics. MARVEL version 3.0.2 fixes this bug and removes all the implementation limitations mentioned in this paper, but ASL has not yet been ported to 3.0.2.

10. Contributions

We explained why complex software processes require two levels of specification, and that some process modeling languages are more suitable for describing global control flow and synchronization while others are better suited for expressing local constraints on data and tools. We demonstrated through the Activity Structures Language and its enaction model that process-centered environments could profitably incorporate both global and local formalisms in a bi-level PML. We described a relatively straight-forward implementation of this PML based on the existing MARVEL environment. Our system has been used to implement a multi-participant solution to the ISPW7 example problem.

Acknowledgments

ASL was developed in consultation with Bill Riddle, Steve Gaede and Brian Nejmeh. Most of our ISPW7 solution is due to Hideyuki Miki, a resident visitor at Columbia from Software Research Associates. MARVEL 3.0 or 3.0.1 has been licensed to 25 educational institutions and industrial sponsors. MARVEL 3.0.2 is in daily use internally, and will be demonstrated at the KBSE '92 and SIGSOFT '92 conferences.

References

[Ambriola 90] V. Ambriola, P. Ciancarini and C. Montangero. Software Process Enactment in Oikos. In Richard N. Taylor (editor), *SIGSOFT '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 183-192. Irvine CA, December, 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[Avrunin 86] George S. Avrunin, Laura K. Dillon, Jack C. Wileden and William E. Riddle. Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems. *IEEE Transactions on Software Engineering* SE-12(2):278-292, February, 1986.

[Balzer 91] Robert Balzer. Tolerating Inconsistency. In *13th International Conference on Software Engineering*, pages 158-165. Austin TX, May, 1991.

[Barghouti 90] Naser S. Barghouti and Gail E. Kaiser. Modeling Concurrency in Rule-Based Development Environments. *IEEE Expert* 5(6):15-27, December, 1990.

[Barghouti 92a] Naser S. Barghouti and Gail E. Kaiser. Scaling Up Rule-Based Development Environments. *International Journal on Software Engineering and Knowledge Engineering* 2(1):59-78, March, 1992.

[Barghouti 92b] Naser S. Barghouti. Supporting Cooperation in the MARVEL Process-Centered SDE. In *Fifth ACM SIGSOFT Symposium on Software Development Environments*. Washington DC, December, 1992. In press.

[Bates 88] Peter Bates. Distributed Debugging Tools for Heterogeneous Distributed Systems. In *8th International Conference on Distributed Computing Systems*, pages 308-315. IEEE Computer Society, San Jose CA, June, 1988.

[Belkhatir 91] Noureddine Belkhatir, Jacky Estublier and Walcelio L. Melo. Adele 2: A Support to Large Software Development Process. In Mark Dowson (editor), *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 159-170. IEEE Computer Society Press, Redondo Beach CA, October, 1991.

[Belkhatir 92] Noureddine Belkhatir, Jacky Estublier and Walcelio L. Melo. Cooperative Work in Large-Scale Software Systems. *Journal of Software Maintenance: Research and Practice* , 1992. In press.

[Ben-Shaul 92] Israel Z. Ben-Shaul, Gail E. Kaiser and George T. Heineman. An Architecture for Multi-User Software Development Environments. In *5th ACM SIGSOFT Symposium on Software Development Environments*. Washington DC, December, 1992. In press.

[Brownston 85] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin. *Programming Expert Systems in OPS5*. Addison-Wesley, Reading MA, 1985.

[Bruegge 83] Bernd Bruegge and Peter Hibbard. Generalized Path Expressions: A High-Level Debugging Mechanism. *The Journal of Systems and Software* 3(4):265-276, December, 1983.

[CLF 88] CLF Project. *CLF Manual* USC Information Sciences Institute, 1988.

[Cohen 86] Donald Cohen. Automatic Compilation of Logical Specifications into Efficient Programs. In *5th National Conference on Artificial Intelligence*, pages 20-25. AAAI, Philadelphia, PA, August, 1986.

[Conradi 91] Reidar Conradi and Carl Chr. Malm. Cooperating Transactions and Workspaces in EPOS: Design and Preliminary Implementation. In R. Andersen, J.A. Bubenko, Jr., and A. Solvberg (editors), *Lecture Notes in Computer Science*. Number 498: *Advanced Information Systems Engineering: 3rd International Conference CAiSE '91*, pages 375-392. Springer-Verlag, Trondheim, Norway, 1991.

[Deiters 90] Wolfgang Deiters and Volker Gruhn. Managing Software Processes in the Environment MELMAC. In Richard N. Taylor (editor), *SIGSOFT '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 193-205. Irvine CA, December, 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[Dowson 91] Mark Dowson (editor). *1st International Conference on the Software Process: Manufacturing Complex Systems*. IEEE Computer Society Press, Redondo Beach CA, 1991.

[Gaede 91] Steven L. Gaede, Brian Nejmeh and William E. Riddle. *Interim Report Process Management: Infrastructure Exploration Project*. Technical Report 7-48-5, Software Design & Analysis, March, 1991.

[Gisi 91] Mark A. Gisi and Gail E. Kaiser. Extending A Tool Integration Language. In Mark Dowson (editor), *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218-227. IEEE Computer Society Press, Redondo Beach CA, October, 1991.

[Gruhn 91] Volker Gruhn. *Validatiuon and Verification of Software Process Models*. PhD thesis, Forschungsberichte des Fachbereichs Informatik der Universitat Dortmund, 1991. Bericht Nr. 394/91.

[Heimbigner 91] Dennis Heimbigner and Marc Kellner. Software Process Example for ISPW-7. August, 1991 Available by anonymous ftp from ftp.cs.colorado.edu.

[Heimbigner 92] Dennis Heimbigner. The ProcessEngine: A Process State Server Approach to Process Programming. In *5th ACM SIGSOFT Symposium on Software Development Environments*. Washington DC, December, 1992. In press.

[Heineman 91] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti and Israel Z. Ben-Shaul. Rule Chaining in MARVEL: Dynamic Binding of Parameters. In *6th Annual Knowledge-Based Software Engineering Conference*, pages 215-222. IEEE Computer Society Press, Syracuse NY, September, 1991.

[Iida 91] Hajimu Iida, Takeshi Ogihara, Katsuro Inoue and Koji Torii. Generating a Menu-Oriented Navigation System from Formal Description of Software Development Activity Sequence. In Mark Dowson (editor), *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 45-57. IEEE Computer Society Press, Redondo Beach CA, October, 1991.

[Inoue 89] Katsuro Inoue, Takeshi Pgihara, Tohru Kikuno and Koji Torii. A Formal Adaptation Method for Process Descriptions. In *11th International Conference on Software Engineering*, pages 145-153. IEEE Computer Science Press, Pittsburgh PA, May, 1989.

[Kaiser 88] Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software* 5(3):40-49, May, 1988.

[Katayama 89] Takuya Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *11th International Conference on Software Engineering*, pages 343-352. IEEE Computer Science Press, Pittsburgh PA, May, 1989.

[Kellner 90] Marc I. Kellner and H. Dieter Rombach. Session Summary: Comparisons of Software Process Descriptions. In Takuya Katayama (editor), *6th International Software Process Workshop: Support for the Software Process*, pages 7-18. IEEE Computer Society Press, Hakodate, Japan, October, 1990.

[Kellner 91] Marc I. Kellner, Peter H. Feiler, Anthony Finkelstein, Takuya Katayama, Leon J. Osterweil, Maria H. Penedo and H. Dieter Rombach. Software Process Modeling Example Problem. In Mark Dowson (editor), *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 176-186. IEEE Computer Society Press, Redondo Beach CA, October, 1991.

[Liu 91] Chunnian Liu. Software Process Planning and Execution: Coupling vs. Integration. In R. Andersen, J.A. Bubenko, Jr., and A. Solvberg (editors), *Lecture Notes in Computer Science*. Number 498: *Advanced Information Systems Engineering: 3rd International Conference CAiSE '91*, pages 356-374. Springer-Verlag, Trondheim, Norway, 1991.

[Mi 91] Peiwei Mi and Walt Scacchi. Articulation: Supporting Dynamic Evolution of Software Engineering Processes. In Ian Thomas (editor), *7th International Software Process Workshop*. IEEE Computer Society Press, Yountville CA, October, 1991. Preprints.

[Mi 92] Peiwei Mi and Walt Scacchi. Process Integration in CASE Environments. *IEEE Software* 9(2):45-53, March, 1992.

[Minsky 90] Naftaly H. Minsky and David Rozenshtein. Configuration Management by Consensus: An Application of Law-Governed Systems. In Richard N. Taylor (editor), *SIGSOFT '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 44-55. ACM Press, Irvine CA, December, 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[Minsky 91a] Naftaly H. Minsky. Law-Governed Systems. *Software Engineering Journal* 6(5):285-302, September, 1991.

[Minsky 91b] Naftaly H. Minsky. The Imposition of Protocols Over Open Distributed Systems. *IEEE Transactions on Software Engineering* 17(2):183-195, February, 1991.

[Narayanaswamy 91] K. Narayanaswamy and Neil M. Goldman. Team Coordination: Information Sharing + Policies. In Ian Thomas (editor), *7th International Software Process Workshop*. IEEE Computer Society Press, Yountville CA, October, 1991. Preprints.

[Ponamgi 91] M. Krish Ponamgi, Wenwey Hseush and Gail E. Kaiser. Debugging Multi-Threaded Programs with MPD. *IEEE Software* 8(3):37-43, May, 1991.

[Riddle 91] William E. Riddle. *Activity Structure Definitions*. Technical Report 7-52-3, Software Design & Analysis, March, 1991.

[Schafer 92] Wilhelm Schafer, Burkhard Peuschel and Stefan Wolf. A Knowledge-based Software Development Environment Supporting Cooperative Work. *International Journal on Software Engineering & Knowledge Engineering* 2(1):79-106, March, 1992.

[Shinoda 89] Yoichi Shinoda and Takuya Katayama. Towards Formal Description and Automatic Generation of Programming Environments. In Fred Long (editor), *Lecture Notes in Computer Science*. Number 467: *Software Engineering Environments International Workshop on Environments*, pages 261-270. Springer-Verlag, Chinon, France, 1989.

[Sutton 90a] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado, 1990.

[Sutton 90b] Stanley M. Sutton, Jr. A Flexible Consistency Model for Persistent Data in Software-Process Programming Languages. In Alan Dearle, Gail Shaw and Stan Zdonik (editor), *Implementing Persistent Object Bases Principles and Practice: The 4th International Workshop on Persistent Object Systems*, pages 297-310. Morgan Kaufmann, Martha's Vineyard MA, September, 1990.

[Thomas 91] Ian Thomas (editor). *7th International Software Process Workshop*. IEEE Computer Society Press, Yountville CA, 1991. Preprints.

[Williams 88] Lloyd G. Williams. Software Process Modeling: A Behavioral Approach. In *10th International Conference on Software Engineering*, pages 174-186. Raffles City, Singapore, April, 1988.