Implementing Activity Structures Process Modeling On Top Of The MARVEL Environment Kernel

Final Report for Software Design & Analysis, Inc.

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

Columbia University
Department of Computer Science
New York, NY 10027

CUCS-027-91
13 September 1991

Abstract

Our goal was to implement the activity structures model defined by Software Design & Analysis on top of the MARVEL environment kernel. This involved further design of the activity structures process definition language and enaction model as well as translation and run-time support in terms of facilities provided by MARVEL. The result is an elegant declarative control language for multi-user software processes, with data and activities defined as classes and rules in the previously existing MARVEL Strategy Language. Semantics-based concurrency control is provided by a combination of the MARVEL kernel's lock and transaction managers and the send/receive synchronization primitives of the activity structures model.

This research was funded in part through a consulting contract with Software Design & Analysis. The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368, CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC 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. Popovich is supported in part by NASA grant NGT 50583.

1. Overview

1.1. Introduction

The basic goal is to implement activities structures [Riddle 91] on top of MARVEL [Kaiser 88, Kaiser 90], as one approach to supporting a high-level software process control layer for MARVEL. But what does it mean to ‘‘implement activities structures’’ and what does ‘‘on top of MARVEL’’ mean? There are many approaches, for example, that might take advantage of MARVEL’s object-oriented data modeling but effectively ignore the rule-based process modeling. We are primarily interested in approaches that exploit the rule-based process modeling and enaction. Since our rules are treated as multi-methods, this implies at least some use of the object-oriented data modeling as well.

Our high-level approach is then to translate activities structures into rules and enact activities structures using forward and backward chaining. Then, what are ‘‘activities structures’’, what does it mean to ‘‘translate’’, and what does it mean to ‘‘enact’’?

We represent activities structures in a language consisting of the five structuring primitives of constrained expressions and the send/receive synchronization primitives. Different portions of a software process are defined in this language, called the **Activity Structures Language** (ASL), augmented by additional constructs taken from the existing MARVEL Strategy Language (MSL).

A process is translated into MSL classes, rules and envelopes. In order to execute a process, there must be some instantiation whereby activities are mapped to the corresponding data on which they will be carried out and/or the user(s) who will be involved in carrying them out. This is done by creating first class objects that represent the activity structures, assigning userids to attributes of these objects, and linking data objects to attributes of these objects. A process is enacted by forward and backward chaining over the generated rules, where the status attributes manipulated by the rules are represented as attributes of activity structure objects. We thus map the three lifetime steps of (1) definition, (2) activation and (3) invocation into (1) definition of activities structures in ASL and translation into MSL, (2) instantiation and (3) enaction.

An important issue of *scale* arose during this investigation. MSL is intended to define a full software development process undertaken by teams of personnel, although in practice only the coding and testing portions of the lifecycle have been defined to date in C/Marvel, our main example. ASL strategies are similarly intended to define full processes, but individual activity structures define only a tiny portion of a process. Many activity structures instantiated by multiple users (or by the same user at different times or simultaneously in different windows) are required to perform a practical subset of a large scale process.

1.2. Activity Structures

Activity structures are defined by an *environment administrator* rather than an end-user of the environment (although it is not necessarily the case that these are always different people). The activity structures are completely specified in advance of generating an environment in which the corresponding process is enacted. This precludes the possibility of an administrator or user defining additional activity structures or modifying the current ones, after the environment has been in real use, and then evolving the environment on or off-line. (A small ‘‘test’’ objectbase is used in debugging the specifications, which are typically modified significantly during debugging.) We are studying the feasibility of off-line evolution for MSL rules, and have made some progress [Barghouti 91], but evolution for in-use objectbases is outside the scope of this project.

An activity structure might be defined by either a *single* string or by *multiple* strings. The latter might follow the approach described by Avrunin *et al.* [Avrunin 86], where there is a main system structure plus additional constraint structures, and the constraints refine the set of activities traces that are consistent with the intended process. We did not investigate the feasibility of translating and executing single activity structures specified by multiple strings. Instead, multiple strings define distinct activity structures that are each small portions of the process represented by the enclosing ASL strategy (i.e., file).

The administrator provides activity structures as strings in a textual format, for them to be parsed by a lex/yacc parser into MSL classes, rules and envelopes. These are further translated by the MARVEL loader into an internal representation that can be enacted through the MARVEL kernel.

Activities must be associated with corresponding data at some point in order to enact the process. This is the major problem posed for this investigation. We have identified four possibilities:

1. There is no association at all of activities with data by the administrator. The users of the environment are fully responsible for selecting the appropriate data at run-time without any assistance or restriction from the environment.

2. There is no association of activities with data *instances* by the administrator, but the activities are associated with formal parameters. These indicate the number and types of expected arguments, with types mapping to MSL classes. The users of the environment would be responsible for selecting appropriate data (MARVEL object instances) at run-time, but the environment could perform type checking and provide appropriate error messages regarding invalid choices of data.

3. There is no association of activities with data instances by the administrator, but the activities are associated with formal parameters with more information content than above. In particular, there would be some constraints among the multiple parameters of a single activity and/or the parameters of multiple activities participating in the same structure. For example, a C header file and the C source files including it (as opposed to unrelated C files) might by controlled by the same activity structure.

4. Activities would be associated with specific data instances at definition-time by the administrator, basically combining instantiation with definition. This implies a relatively rigid process, as well as a priori knowledge that might not normally be available.

We implemented the second, typing approach, but with limited support for the third approach allowing constraints across arguments of a single activity using the derived parameters already supported by MSL. This does not imply that data cannot be associated with activities in advance; this may be done statically during instantiation or dynamically during enaction, but not during definition.

1.3. Translation

We have identified four seemingly viable approaches to translating from ASL into MSL:

1. The *Generation* approach employs MARVEL's forward chaining capabilities to ‘‘generate’’ an actual process given a single activity structure. Thinking of an individual activity structure as defining a language, the translator would produce a generator of strings in the language (i.e., traces of activities). Choices such as among alternatives or number of repetitions could be chosen by the user in a ‘‘live’’ environment, or randomly in a simulation.

2. In the *Passive Recognition* approach, MARVEL's enforcement capabilities are used to ''recognize'' an actual process given a set of activity structures and strings of commands selected by the users. The translator would produce a set of parsers for strings in the language, for an a posteori analysis, or for string prefixes for parsing done on the fly as the process unfolds.
3. An extension of the above option, *Active Recognition*, exploits MARVEL's backward chaining capabilities to enhance the basic function of recognizing a process. The difference is that prefixes that would be disallowed by Passive Recognition might be allowed by Active Recognition, if MARVEL were able to augment the strings of user commands by inserting automatically triggered activities before the token (activity) where the recognizer entered an ''error state''. This necessarily requires on the fly parsing of prefixes to be meaningful.
4. In the *Hybrid* approach, Generation and Active Recognition are combined, approximating MARVEL's normal style of interaction. MARVEL waits for a user to enter a command before doing anything, and then does backward chaining to attempt to enable the command followed by forward chaining to attempt to carry out the implications of the command. It then stops and waits for the next user command.

The distinctions among these options can be clarified by considering some possible translations of the five operators, particularly the choices implied by alternation, repetition and concurrent repetition. For example, the administrator can state a process as "A | B", but it is necessary to determine whether to do A or instead do B; for "A*", it is necessary to determine how many times to actually do A. "A#" (concurrent repetition) is even more complex, because the number of users or windows must be determined.

Considering alternation "A | B" for the Generation approach, the environment might ask the environment user whether to do A or B, and then initiate the chosen activity (alternatively, for a process simulation, the environment might randomly choose one). In both Recognition approaches, the user simply does A or does B, that is, enters a command corresponding to selecting activity A or a command corresponding to B, and the environment allows either case. There is no real difference between these and the Hybrid approach for this simple example.

For repetition, e.g., "(A; B)*", a Generator environment would ask the user how many times to repeat, and then do A automatically followed by (i.e., forward chained to) B, followed by (forward chaining back to) A, and so on. A Passive Recognizer would allow the user to repeat "A; B" an arbitrary number of times by hand, but would not permit B unless preceded by A; an Active Recognizer would behave the same, except whenever the user attempted to do B without having already done A, the environment would automatically (backward chain to) do A first. A Hybrid system would add the functionality of automatically doing B whenever the user did A.

The Generation approach might handle "(A | B)#; C" by requesting of some distinguished user how many concurrent repetitions and also which specific end-users should control the corresponding MARVEL clients. (We discuss later on how multiple users are reflected in multiple MARVEL clients.) Then each of these users would be asked to choose between A and B. The environment would determine when all these A's and B's were done, and then do C in the client of the original user. (An alternative would be to require each of the concurrent repetitions to make the same choice, but this does not make much sense for practical processes.)

Passive Recognition might handle "(A | B)#; C" by allowing any number of end-users to do either A or B, but not both, until some user happens to do C. After that, it would prevent any user from initiating either

A or B. Active Recognition could be achieved for "(A | B)#; C" by following an approach similar to that suggested above for Generation: Say an user request to do C. Then the environment would backward chain to spawn clients, but would have to ask the user who requested C regarding how many users and which user in particular would take over control of each of the clients. There seem to be several possible Hybrid models, one directly combining the Generation and Active Recognition behaviors as described here. But another perhaps better approach might support a mixed mode, where a user could select either A or B, initiating the forward chaining, or C, triggering backward chaining.

We have implemented the Passive Recognition model, from the viewpoint of the control embodied in activity structure definitions, but also support the Hybrid model in the sense that standard MSL conditions and effects can be associated with activities to trigger both forward and backward chaining.

1.4. Execution

The ASL implementation is based on the preliminary (unreleased) form of multi-user MARVEL 3.0 rather than single-user MARVEL 2.65. Multiple clients for the same or different users are supported, for a single centralized server.

The choice of the Passive Recognition approach to translation favors interleaving instantiation and enaction. In particular, specific data selections might be made on the fly by the users participating in the process rather than fully binding all activities to corresponding data and users up front. The latter approach is not impossible in the context of Recognition, however, and the implementation supports both static and dynamic binding of data, but only static binding of users. In particular, a user (or actually a client) executes in the context of an explicitly activated activity structure, rather than being implicitly bound according to the activities selected by the user.

A problem arises with Recognition, however, when the same activity (symbol) appears multiple times in the same activity structure, as in "A; (B; (A | D))*", because the control is different for each case. In this example, the condition for the first A activity is that it (A) has not occurred yet, but the condition for the second A activity is that B has occurred and that no other A activity has occurred since the last B (as well as D has not occurred). There must be additional support at run-time so that a user can select an activity by name, for example, entering a command with the same name as the desired activity, rather than by indicating an explicit position in the activity structure. But then the activity should be mapped internally to the correct position in the current activity structure.

Our solution consists of two parts. First, the non-deterministic finite automata defined by the regular expressions is converted to deterministic finite automata for the purpose of enforcing control. Second, a run-time support package inserted in the MARVEL kernel augments the translation system, to track the current activity structure instantiation.

This package represents special support for the ASL implementation, and can be turned on and off with compiler directives. There is currently no general facility in MARVEL for an environment administrator to add code to the MARVEL system, except in the sense of the arbitrary tools that can be invoked from activity envelopes, and we do not anticipate adding such a facility.

The notion of activity *atomicity* when the activities execute on the same data is inherently supported by the MARVEL kernel, by the transaction manager, so the possibility of replacing activities like edit with begin-edit, end-edit, etc. does not come up. In particular, two shuffled activities edit(O) and proof(O) executing on the same object O would be automatically serialized by the MARVEL kernel, because its concurrency control protocol implicitly places write locks on all object arguments of activities after evaluating the read-only conditions but prior to initiating the actual activities. Read locks are obtained during the evaluation of the condition, as needed. (This behavior may in MARVEL version 3.1, which is planned to incorporate many of Naser Barghouti's thesis results [Barghouti 90, Barghouti 9x].)

In contrast, MARVEL permits any concurrency relationship among edit(X) and proof(Y), assuming there is no containment relation between X and Y (MARVEL automatically places intention locks on enclosing objects), so that they could in fact overlap in time and are thus non-atomic. Synchronization among shuffled activities is supported by the send/receive synchronization primitives. However, if actually atomic activities are desired, this is supported by an `atomic` directive (non-atomic is the default); this approach is a preferred alternative to introducing begin-edit, end-edit, etc. Any activity structures where such explicit synchronization is missing represents a ‘‘don't care’’ scenario where true parallelism should not be precluded.

A final execution issue involves the presentation view: How are in-progress activity structures represented to the user? The alternatives seem to be:

- No user-visible presentation at all of activity structures.
- Textual presentation in some form, for example, by printing out a summary in MARVEL's Text window. This could be done with or without describing the actual data on which the activities execute, and could support multiple modes, including a blow by blow listing of activities as they happen, a history of past activities, and a model of possible near and far future activities.
- Graphical presentation in the style of Steve Gaede's environment [Gaede 91], presenting activity structures only, with no conceptual or visible links to data.
- Graphical presentation of instantiated activity structures, including user-visible links to actual data instances employed in activities.

The fourth option was followed, as the most user friendly, and also because it turned out to be necessary for certain user interactions with the objects representing the activity structure instances. However, there is no cursor indicating the current position(s) in the activity structure definition, and the user interface leaves much to be desired.

1.5. Design and Implementation

The enaction requires some run-time representation of the activity structures, independent of visibility to the end-user. An activity is of course always reflected by the envelope that actually implements that activity, but additional information must be available during execution, such as the control state indicating position in the currently instantiated activity structure.

- One possibility would be to make control entirely implicit in the rules, those generated to encapsulate the activities and perform the control specified by the structure primitives, with all state somehow reflected in attributes of the underlying data. We already have facilities in MARVEL to instantiate multiple instances of the same rule with different data for the same or different clients, i.e., rules are re-entrant.
- Another approach would involve explicit representation of activities and/or structures as objects. These objects need not be visible to end-users, but they could maintain state. For example, an activity object might include links to the actual data used in that particular

instantiation of the activity — where particular instantiation might refer to position in space in the activity structure and/or to occurrence in time in a repetition or concurrent repetition structure.

The first approach would give us a minor problem in mapping activities to rules. A potential solution would be to have multiple rules with the same name, same activity and same formal parameters but different conditions, each corresponding to a different position of the activity symbol in the full activity structure. Then when an activity is selected, all the rules matching the name and formal parameters would be evaluated in turn until a satisfied condition was found. This approach requires an extension to the current MARVEL algorithm for mapping command names to rules, which was straightforward to implement.

A variant of the second approach, however, actually helps us solve this mapping problem. If we used a class (as opposed to an object) to distinguish each occurrence of an activity symbol, then we could use MARVEL's normal algorithm for mapping command names to rules, which supports overloading and subtyping. "A; A" would result in two distinct A classes as opposed to two distinct objects. But it is not clear how "(A; B)*" could be represented, or how to keep track of the current position in the overall activity structure.

If all activities executed on the same data object, then the first approach would be sufficient, but this is rarely the case in practical processes. Thus we introduced *structure classes*, whose objects represent an entire structure (or a shuffle operand substructure), effectively combining the two alternatives. Their state attributes represent the current position in the corresponding activity structure (or shuffle operand subexpression). The structure classes are generated during the translation from activity structures to MSL, with run-time instances representing the states of particular instantiations. The run-time support package provides the underlying mechanisms for manipulation and operation of these objects.

1.6. Schedule

- May 27: Contract begins.
- May 30-31: Kickoff meeting.
- June 7: Proposal.
- June 23: Progress report.
- July 7: First design document.
- July 8-9: Meeting.
- July 31: Second design document.
- August 1-2: Meeting.
- August 6: Preliminary version available by ftp.
- August 12: Third design document.
- August 23: Fourth design document. Contract ends.
- September 16-17: Final report, installation and presentation.

2. MARVEL Concepts

The multi-user MARVEL system is based on a client/server architecture [Ben-Shaul 91], where the clients communicate with the server via tcp/ip sockets. A MARVEL server can support zero or more clients sharing access to the same objectbase. When some user starts a client (using the MARVEL executable), a special daemon (marveld) installed in the operating system (/etc/inetd.conf on SunOS and Ultrix) checks whether or not there is already a server running for that objectbase. If so, it connects them up and if not it brings up a server on the same machine as the client. When all its clients have quit, the daemon shuts the server down. The daemon triggers the installed version of the MARVEL server.

To test new versions of the server, the user must explicitly start the server (using the MARVEL_server executable) or check that the appropriate server is already running before starting her client. Only one server can execute for a given objectbase at a time, and a special file (.server_port) in the objectbase indicates whether or not there is a server currently running and provides information that clients use internally to connect to this server.

A MARVEL objectbase is stored persistently in a particular file system directory, known as a "MARVEL environment". Invoking the MARVEL server or client executable while connected (cd) to such a directory automatically associates the server or client with that objectbase. Otherwise, the user is prompted for the file system pathname for the appropriate objectbase.

A MARVEL environment is a directory that contains a set of MARVEL Strategy Language (MSL) files, a set of envelopes, a binary objectbase, an internal representation of the contents of the MSL files, a ‘‘hidden’’ file system for binary and text attributes, and some additional subdirectories and files for things like failure recovery logs and maintaining a persistent counter for generating clientIDs. The binary representation of the in-memory objectbase is stored in a dbm file, while file attributes (text and binary) are stored in a ‘‘hidden’’ file system. This file system includes directories representing objects containing set attributes.

All MARVEL environments are entirely independent of each other, and there is no identifier resolution of any sort across environments. MARVEL environments are set up by ‘‘administrators’’, charged with defining appropriate data organization and behavior for a (class of) software project, and the typical end-user need know nothing about their contents. The administrator usually loads a set set of MSL files into a MARVEL environment, and thereafter end-users simply use that environment without further recourse to the `load` command.

MARVEL currently runs on Sun 3s (running SunOS 4.0.3), Sun 4s (SunOS 4.1.1) and DecStation 3100s (Ultrix 3.1; an old version, note the most recent version distributed by Digital is something like 4.2). There are no restrictions on mixing and matching clients, but an objectbase created by a server executing on a particular architecture can only be accessed later by servers executing on the same architecture because of binary incompatibility problems. However, a binary objectbase can be manually converted from one architecture to another using the bin2ascii and ascii2bin utilities.

A MARVEL client corresponds one-to-one with an operating system process. A client is not intended to be an abstraction of anything, but is instead a realization concept. A client is a very important concept with respect to MARVEL’s implementation. Every client executes as a ‘‘session’’ running from its invocation to its exit, where the session provides various client-specific information such as the controlling user’s Unix environment variables. A client, i.e., a session, is in the middle of zero or one rule chain at a time. A MARVEL server keeps a ‘‘context’’ for each session, and performs rule chaining with respect to a session.

When a client is not in the midst of chaining, this does not mean it is inactive; a client may also execute *built-in* commands. This includes `load` (the data and process models), `quit` (exit the client), `help` (obvious), a large number of commands for browsing the objectbase (including commands related to various display options), and another large number of commands for directly modifying the objectbase (`add`, `delete`, `move`, `copy`, etc.).

There is currently no ability for a client to run anything in the background except through envelopes (using the normal facilities of the operating system to fork new processes). It is possible, however, for an envelope to invoke a new MARVEL client in batch mode, feeding it a script to execute, and then terminate the client. It is not possible to create a client in this manner and hand it off to a human controller. Thus batch clients cannot become interactive.

A MARVEL end-user may have multiple clients for the same server running under her userid, either on the same or different machines. If the graphical front end is used, then multiple clients for the same server belonging to the same user would correspond to multiple windows. The MARVEL graphical user interface for a single client appears to have several subwindows, but from the X11 windows point of view, there is exactly one window. The internal windows are manipulated by MARVEL itself and cannot be manipulated using conventional X11 facilities. Even the top-level window does not respond normally to X11 user commands provided by the user’s choice of window manager (because the MARVEL graphical user interface is implemented directly in Xlib). An end-user can also interact with multiple clients through the command line interface, by using operating system commands to move them between the background and foreground, or separate user jobs controlled by multiple terminals (or X windows xterms). The graphical user interface may conform to some common X window manager in a later MARVEL release.

A MARVEL user corresponds one-to-one with an operating system userid. A user is not intended to be an abstraction of anything, but is instead a realization concept. If multiple humans are logged in under the same userid, then they appear to be the same user. The notion of a user is not an important concept from MARVEL’s viewpoint. Its only distinction is as a built-in type (and the corresponding CurrentUser variable) for use by environment administrators in writing classes and rules. MARVEL itself does not distinguish between clients controlled by the same versus different users.

The MARVEL server currently supports a fairly conventional scheduling mechanism among the clients. The scheduler does not round robin among the clients, but instead feeds off a queue of messages from clients. The messages might indicate either a new command (either built-in or a rule) or the completion of an activity. The server takes a message from the queue. If it is a built-in, it executes the command atomically — even if it is an extremely long duration command, notably Marvelizer [Sokolsky 91].

If the command is a rule, then any chaining among inference rules happens atomically in the server. An inference rule has a null activity, and is used to derive values of attributes from related attributes. In contrast, an activation rule as an activity, and is normally used to invoke a tool. The activity of an activation rule encountered during chaining is sent to the client for execution. The server stores the relevant chaining information and context switches to the client relevant for the next message.

If the top message in the queue is a return from an activity, the server restores that client’s chaining context and continues where it left off. It asserts the indicated effect of the activity, which may participate in an inprogress backward chain or initiate a new forward chain. In either case, any inferencing is done atomically, but an activation rule again results in sending the activity's arguments to the client for execution and a context switch.

MSL rules have two kinds of parameters, formal and derived. Formal parameters are just what one would expect, with their names and types given in the rule header (or signature). The corresponding actual parameters must be supplied explicitly by the user as arguments to a command, or determined by a complex parameter passing process during chaining (this is the subject of [Heineman 91]).

Derived parameters are determined in the optional bindings portion of each rule (also known as the "characteristic function", and not to be confused with data bindings for the Activity Structure Language (ASL), discussed in section 9). Such parameters may be derived from the formal parameters via structural relationships (ancestor/descendent, parent/child, linkto/linkfrom) represented explicitly in the objectbase, perhaps filtered by associative properties such as an attribute in some binary relation to another attribute or literal value.

Alternatively, derived parameters may be determined by associative queries, finding all instances of some class with specified properties. Deriving parameters via structural relationships is reasonably efficient, implemented using direct navigation. Associative access, however, is relatively expensive, involving (nearly) global search; a MARVEL objectbase maintains a primary index based on class, but no secondary indices based on attribute values.

In figure 2-1, f is a formal parameter and h is a derived parameter. The *bindings* (or characteristic function) consists of all the text between the first and second colon (":"). This example uses only structural relationships to determine the derived parameter, i.e., h is bound to all HFILEs linked to f through its ref attribute. The text after the second colon is the *property list*, which is the proper condition of the rule, one of whose two predicates may be satisfied by backward chaining. Bindings do not participate in chaining, except for determining formal parameters of later rules in the chain. The two mutually exclusive effects indicate the two possible results of the compilation activity.

```
compile [?f:CFILE]:
    (forall HFILE ?h suchthat (linkto [?f.ref ?h]))
    : (or (?f.compile_status = NotCompiled)
          (?f.compile_status = Error))

    { COMPILER compile ?f.contents ?f.object_code ?h.contents }

(?f.compile_status = Compiled);
(?f.compile_status = Error);
```

Figure 2-1: Compile Rule

The rule in figure 2-2 uses an associative query in the characteristic function. The condition states that the user can activate a designated activity structure (AS) instance if the user has been assigned as the owner of that instance, the client being controlled by the user is not the current client of any other instance, the instance does not have any other client assigned to it, and the state of the instance is that it has never been previously activated or it was activated and later terminated. The associative query is actually a hacking trick to consider all the activity structure instances in the objectbase, because it is guaranteed that none of them have their ‘‘as_string’’ set to ResetUser (which is the null string). The effect makes the user's client the current client of the instance and sets its state to ready.

```
    Activate [?SO:AS_1]:
      (and (exists test ?parent suchthat (member [?parent.children1 ?SO]))
           (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser)))
      : (and (or no_forward(?SO.state0 = Inactive)
                 no_forward(?SO.state0 = Terminated))
             no_forward(?SO.clientID = ResetClient)
             no_forward(?SO.owner = CurrentUser)
             no_forward(?parent.state0 = Active)
             no_forward(?s.clientID <> CurrentClient))
      { }
      (and no_backward(?SO.clientID = CurrentClient)
           no_backward(?SO.state0 = Ready));
```

Figure 2-2: Activate Rule

The no_forward and no_backward directives restrict chaining as indicated. No rule can forward chain to or from a no_forward predicate, or backward chain to or from a no_backward predicate. These directives are useful for making sure that rules intended to match user commands are invoked <u>only</u> due to user commands, not because their condition predicates are incidentally satisfied or their effect predicates are desired to satisfy the condition of some other rule. The no_chain directive, not shown in this example, covers both chaining directions.

The rule in figure 2-2 has no activity, and is thus an *inference* rule. The rule in the previous figure 2-1 has a non-null activity, so it is an *activation* rule. The purpose of inference rules is to infer new states of object attributes based on their relationships with the states of other attributes. Chaining to infer such states is always atomic. Activation rules, in contrast, invoke activities in order to *make* their effects true.

An activity consists of the name of a tool, the name of an envelope corresponding to a particular option or switch of that tool, and a collection of arguments to be provided to the envelope from among the attributes of formal and derived parameters. (The envelope language is the subject of [Gisi 91].) One important restriction is that the effects of a rule can only assign the attributes of formal parameters, not derived parameters. Therefore, forward chaining to another rule with the desired formal parameter(s) is often employed to undertake the desired side-effects of a command.

MARVEL 3.0's concurrency control mechanism supports relatively conventional serializability among multiple users. Rules are subtransactions and rule chains are nested transactions. Two-phase locking is used to enforce serializability. The main difference from standard mechanisms is that a rule or rule chain never blocks waiting for a lock, but instead is either terminated or aborted if it cannot acquire the necessary locks.

A *consistency* chain, only between consistency predicates — a directive normally attached only to predicates of inference rules — must be atomic by definition, and thus might be aborted (i.e., rolled back) in the case of a conflict with another consistency chain. A consistency chain has precedence over an automation chain. An automation chain would only be terminated with the most recent completed rule, since by definition automation is done only on ‘‘best effort’’ basis. The MSL rules generated from ASL input files use only automation predicates.

Thus if two activities attempt to acquire conflicting locks, one will be aborted or terminated, depending on which reflect consistency and/or automation. This is independent of the atomic/non-atomic activities issue discussed in a section 8, but this mechanism automatically enforces atomicity with respect to activities requiring conflicting (read/write or write/write) access to the same data.

MARVEL 3.1 will support relaxation of serializability as defined by the administrator in the *coordination model*, specified in terms of control rules, which describe how to handle conflicts among particular kinds of rules. This is basically Naser Barghouti's thesis work. Much of this code is actually already working with MARVEL 3.0, but will not appear in the release because it is not yet sufficiently robust for external distribution. MARVEL 3.1 is expected to be released around January 1992.

3. Executing Activity Structures Using MARVEL

An ASL file has the identical format as an MSL file, except for one addition to define the activity structures at the end, between the `activity structure` and `end activity structure` keywords. The format of an ASL file is shown in figure 3-1.

An ASL file can *import* arbitrarily many MSL files, by listing them after the `import` keyword. During translation, import is like an Ada with clause rather than a C include. That is, classes and tools defined in an imported MSL file may be used in the importing ASL file; however, rules defined in an imported MSL file are not considered in any way by the translator. During loading, import is treated like an include, in that all imported files (and the files they import, etc.) are automatically taken by the loader and converted to MARVEL's internal representation.

It is not meaningful to say anything except `all` following the `exports` keyword. (True export restrictions have not been and probably never will be implemented in MARVEL.) Classes for data objects are defined using the syntax for MSL classes, between the `objectbase` and `end_objectbase` keywords.

```
strategy <name>
imports <names of imported MSL files, without .load extension> ;
exports all;
objectbase
<class and tool definitions>
end_objectbase
rules
<activity definitions in the form of MSL rules>
activity structure
<name> : <activity structure definition>
...
end activity structure
```

Figure 3-1: Skeleton ASL File

The ASL implementation consists of a translator program and a run-time support module. The support module is linked into the MARVEL kernel. The translator executes entirely independently of MARVEL.

The ASL translator takes a single ASL file as input and generates a single MSL file as output. Both ASL and MSL files must have the ".load" filename extension, and are provided by the administrator on the command line invoking the translator. The input ASL file, the output MSL file, all imported MSL files and all envelopes named in their tool definitions must be located in the same directory, where this directory represents a ‘‘MARVEL/ASL environment’’.

When an ASL file with one or more named activity structures is input to the ASL translator, it produces several auxiliary files as well as an MSL file. There are two auxiliary files for each activity structure (AS) named in the ASL file: if the AS is named foo, then these files are foo.gen and foo.ctr.

There are also four other auxiliary files generated for each ASL input file: instantiate.env, get_user.env, send.env and receive.env. These files are the same for every input. When the instantiate activity is executed, the envelope prompts the user for the name of the activity structure class (i.e., the name given by the administrator in the ASL file). This is necessary because there may be arbitrarily many named activity structures defined in the same ASL input file. If the name entered is foo, the instantiate envelope then executes the foo.gen file with argument foo.

The foo.gen file assumes that the top-level SO is to be made a child of the root object in the objectbase, as a member of the ‘‘as’’ set attribute. It executes the internal get_as_obj_name program to read the foo.ctr file, to increment the persistent counter used in generating unique names for instances of the foo class — foo_1, foo_2, etc. It then outputs a tailored script for input to MARVEL, named $ASclass.marvelrc, where $ASclass is the class name of the top level SO.

This script uses built-in commands to add a new foo instance with the generated name, followed by construction of a tree of structure objects, with each descendent structure object representing an embedded shuffle subexpression (an operand of a shuffle operator). Instances of the class corresponding to the AS are named foo_1, foo_2, etc., while instances of the internal structure objects are named AS_1, AS_2, etc. Structure objects are explained in section 6. The foo.gen file returns to the instantiate envelope, which then invokes a MARVEL batch client with this script. The envelope deletes the script afterwards; it should not be reused since it employs the hardwired instance name.

The get_user.env envelope is explained in section 6 and the other two in section 7.

Additional information is provided in the instructions for running MARVEL/ASL and the release notes for the ASL translator provided with the implementation tape.

4. Regular Expressions

We initially consider activity structure definitions composed of activity symbols connected only via the regular expression operators: sequential (;), alternation (|) and repetition (*). (The shuffle and concurrent repetition operators are introduced in sections 6 and 12, respectively). Each activity symbol is followed by an ordered list of parameters, each of which indicates a parameter name (a symbol) and is typed with the name of a class. Activities may have different parameters types for different occurrences in the same activity structure definition, since overloading is supported, but then the parameter names must be different (e.g., edit[?f:DOCFILE] and edit[?c:CFILE]) because of how data bindings are done (section 9). There is a special epsilon (null) activity, with no arguments. Examples:

- (edit[?f:DOCFILE] ; (proof[?f:DOCFILE] | epsilon))*
- (edit[?c:CFILE] ; compile[?c:CFILE])* ; build[?m:MODULE]

Each activity is defined by an MSL rule. The header portion of the rule gives the activity name with named and typed parameters. The names, order and types of parameters must match those used in the activity structure definition. This apparently redundant information is necessary for matching overloaded activities. The activity body portion of the rule consists of a tool name, an operation, and zero or more arguments. The arguments must be references to attributes defined for the classes of the parameters. Examples:

```
    edit[?f:DOCFILE]:
      :
      { EDITOR editor ?f.contents }
      ;

    proof[?f:DOCFILE]:
      :
      { EDITOR proof ?f.contents }
      ;
```

Non-null conditions and effects in activity definitions are considered in section 10.

Tool definitions must be provided for each tool name used in an activity. These are similar in syntax to classes, and are all subclasses of the built-in TOOL superclass. Like classes, these are provided as part of the ASL input file, between the `objectbase` and `end_objectbase` keywords. Operations correspond to distinct envelopes, using the operation name given in the activity body. The envelopes must reside in the MARVEL/ASL environment directory, with the extension ".env"; it is not possible to use full or relative pathnames, but only the filename prefix. An example is shown in figure 4-1.

```
    objectbase

    EDITOR :: superclass TOOL;
      editor: string = editor;
      proof: string = proof;
    end

    end_objectbase

    rules

    edit[?f:DOCFILE]:
      :
      { EDITOR editor ?f.contents }
      ;

    proof[?f:DOCFILE]:
      :
      { EDITOR proof ?f.contents }
      ;
```

Figure 4-1: Tool and Activity Definitions

The translator implements control flow among ASL activities by generating appropriate conditions and effects for the corresponding MSL rules. The examples in figures 4-2, 4-3 and 4-4 each show an activity structure definition, followed by the rules similar to those that will be generated by the ASL translator. In these examples, state1 and state2 are internal status attribute names automatically generated for the translator; SO is always used as the name of the object representing the current activity structure. These examples have been simplified for presentation. The ASL translator actually generates additional rules where state0 indicates the status of the entire activity structure. It also includes no_forward and no_backward directives on the condition and effect predicates, to prevent automatic triggering of activities.

A structure object argument, SO, is always added to the rule's argument list by the translator. Its class name corresponds to the name of the activity structure. Each structure object class defines approximately

```
foo: A1[?a:C] ; A1[?a:C]

==>

A1[?a:C, ?SO:foo]:
    : (SO.state1 = Ready)
{ A1 activity body... }
(SO.state1 = Done);

A1[?a:C, ?SO:foo]:
    : (and (SO.state1 = Done) (SO.state2 = Ready))
{ A1 activity body... }
(SO.state2 = Done);
```

Figure 4-2: Sequencing Operator

```
bar: A1[?a:C] | A2[?b:D]

==>

A1[?a:C, ?SO:bar]:
    : (SO.state1 = Ready)
{ A1 activity body... }
(SO.state1 = Done);

A2[?b:D, ?SO:bar]:
    : (SO.state1 = Ready)
{ A2 activity body... }
(SO.state1 = Done);
```

Figure 4-3: Alternation Operator

```
mumble: A1[?a:C]*

==>

A1[?a:C, ?SO:mumble]:
    : (SO.state1 = Ready)
{}
(SO.state1 = Done)

A1[?a:C, ?SO:mumble]:
    : (SO.state1 = Done)
{ A1 activity body... }
(SO.state1 = Done);
```

Figure 4-4: Repetition Operator

one state attribute per regular expression operator, with the values of all such attributes initialized to Inactive. This is done with an enumerated type, consisting of the three values Inactive, Ready and Done. Thus in addition to copying the MSL classes provided as input, the ASL translator must also generate a structure object class specific to the activity structure, always a subclass of the ACTIVITY_STRUCTURE class generated by the translator (shown in figure 4-5. Specific structure object instance are created at run-time, as described in the section 6.

```
ACTIVITY_STRUCTURE :: superclass ENTITY;
  state0: (Ready,Done,Inactive,Active,Terminated) = Ready;
  as_string: string;
  owner: user;
  clientID: clientid;
  atomic_lock: (Shared,Exclusive,None) = None;
  lock_counter: integer = 0;
end
```

Figure 4-5: ACTIVITY_STRUCTURE Superclass

5. Disambiguating Nondeterministic Regular Expressions

Other than determining the intended semantics of activity structures and the recent confusion over what it means to integrate "MARVEL's data-centered view with activity structures", the disambiguation of nondeterministic regular expressions has been our single greatest challenge as well as time-sink. We have tried two different approaches. The first involved a non-deterministic parse, effectively maintaining multiple position pointers into the current AS instance representing all possibilities. This approach turned out to be extremely complex to implement, but enabled a fairly reasonable user view of the environment's behavior. This was abandoned after immense effort.

This problem is an artifact of the rule-based programming model. Once we set a state Ready, it will remain Ready until it is explicitly removed. Our problem was basically that we could not tell the difference between Ready states that were set as a result of the last command performed, and those that were set earlier, which should no longer be valid. More clearly, perhaps, there is no way in the rule-based programming model to detect cases where a rule is NOT matched. If we had the ability to set a state Inactive whenever it is Ready and some activity that is not a member of some particular set is run, or to have a Ready state revert to Inactive if it is not used by the next activity run from the activity structure, we could make this approach to disambiguation work.

The second involves converting the non-deterministic finite state automaton (NFSA) corresponding to the regular expression into the equivalent deterministic FSA (DFSA). But this approach requires extra machinery to represent a reasonable user view, in particular, reflecting the aggregate state of the DFSA back into the multiple positions in the underlying AS. This has not been implemented.

An AS is first translated to the corresponding NFSA (actually, it is not technically an NFSA due to the presence of shuffle expressions with non-atomic activities). This NFSA is converted to a DFSA using a standard algorithm. Then the DFSA is used to generate the control conditions and effects for the activities.

6. Shuffle Expressions

For clarity in presentation, we have made the shuffle operator an N-ary prefix operator rather than a binary infix operator. Thus the administrator must write "A&B&C" as "&(A,B,C)". The following terminology is used in this document: A ‘‘shuffle expression’’ consists of a shuffle operator and two or more operands. A ‘‘shuffle subexpression’’ or ‘‘shuffle child’’ is an operand of a shuffle expression.

The translator generates a new class for each distinct top-level activity structure and for each shuffle subexpression. These classes are all subclasses of the ACTIVITY_STRUCTURE class, which is built into the translator (but not into MARVEL). Top-level activity structures are explicitly named in the ASL input to the translator. The class name for top-level structure objects is the same as the top-level activity structure, while the class names of internal structure objects corresponding to shuffle children are generated as AS_0, AS_1, AS_2, etc.

A top-level activity structure is ‘‘instantiated’’ by creating a ‘‘structure object’’ instance of its class and recursively creating structure object instances for all the classes represented by its embedded shuffle subexpressions. The embedded shuffle expressions are also then said to be instantiated.

Structure objects are first class objects in a MARVEL objectbase. In the current implementation, structure objects corresponding to top-level activity structures are located in the objectbase as members of the ‘‘as’’ set attribute of the GROUP object at the root of the objectbase. The GROUP class is defined in the C/MARVEL data model. (This part of the implementation effectively ties activity structures to C/MARVEL, and at some point it will be necessary to parameterize the translator. We will also have to fix MARVEL itself, which has GROUP hard-wired as its root class.)

Structure objects corresponding to shuffle subexpressions are members of a ‘‘childrenXXX’’ set attribute of the structure object representing the immediately containing the shuffle expression. Since this structure object may represent a top-level activity structure or shuffle subexpression itself containing multiple shuffle operators, a unique ‘‘XXX’’ name is automatically generated for each one.

The structure object for a top-level activity structure is instantiated using the INSTANTIATE command, which corresponds to the INSTANTIATE rule built into the translator (but not into MARVEL). This INSTANTIATE rule takes no arguments, and has no explicit condition. The command could be selected only when the user is in the midst of enacting some other activity structure, it does not matter.

The INSTANTIATE rule operates through the instantiate.env envelope (generated by the translator). The envelope prompts the user for the name of the desired activity structure (i.e., the class name), and then dispatches to the appropriate script generator, which generates a tailored script. The envelope then executes a batch client on the current MARVEL objectbase, with commands determined by this script. The batch script selects the built-in (to MARVEL) ADD command when the focus of attention is the root GROUP object. Since it is currently not possible to pass a literal argument to a rule, the name of the desired activity structure (and hence its class) has to be provided through the prompt from the envelope. The prompt appears in the MARVEL ‘‘start-up’’ window (i.e., the X window — or the console — from which the user started up her MARVEL client).

If the ADD command is used directly by the user to create a structure object, the result is undefined. It is not immediately clear how to force MARVEL to prevent this. Overloading the ADD command seems dubious given that the object argument would be a GROUP object rather than a structure object, but we do not want to prevent using this command to add other children of GROUP. (In any case, overloading of built-in commands does not work in MARVEL 3.0, but probably will in a later version.)

The INSTANTIATE rule just creates the structure objects. Other commands are needed to assign all the shuffle subexpressions to specific users and to activate particular structure objects. It is not feasible to do this in the instantiation envelope, due to the problems of choice (alternation, repetition) and recursion (embedded shuffle expressions), since the user executing INSTANTIATE is often not the same user who should assign specific tasks.

A shuffle subexpression may be ‘‘assigned’’ to a particular user as her task. The ASSIGN command initiates a shuffle expression and associates one of the shuffle subexpressions with a particular user. An owner attribute is included in each structure object class, and set through the effect of the ASSIGN rule. The get_user.env envelope prompts for the userid. The parent structure object is automatically provided as the last argument by the run-time support.

Once an assignment has been done, it stays in effect until the termination of the structure object (or deactivation in the case of top-level). Absolutely everyone has the authority to make an assignment, assuming there is no current assignment. Thus it would be best for the manager to assign a structure object immediately after instantiating it (before anyone else is likely to know it exists). We made this design decision because a fancy authorization scheme would take a lot of time and is not related to the main goals of the project. Similarly, we do not allow assignment of a task to a group of users, any of whom might undertake any portion of the task. This is another bell or whistle.

A shuffle subexpression is basically a regular expression, although possibly containing embedded shuffle expressions. Ignoring embedded shuffle expressions for now, the translator generates a set of rules to enact a shuffle subexpression by simulating a finite state automaton. The special ‘‘state0’’ attribute of the corresponding structure object has the value Ready when the FSA is in its initial state and the value Done when it is in a final state. Epsilon transitions in the FSA are implemented by inferencing rules, and all other transitions are implemented by activation rules. When a structure object is created, i.e., instantiated, its state0 attributes are initialized to a third value, Inactive. The state0 attribute is defined in the ACTIVITY_STRUCTURE class, and is thus inherited by all structure object classes.

To put an instantiated and assigned structure object into its initial state, i.e., to set state0 to Ready, it is necessary to ‘‘activate’’ the structure object. Until a structure object has been activated, it is not possible to perform any of the activities in its corresponding activity structure or shuffle subexpression. The activation process must be performed by the client intended to enact the corresponding activity structure or shuffle subexpression. A top-level activity structure is always executed by one particular client, as is a shuffle subexpression. (It is possible to explicitly change the client binding in mid-stream, as discussed later on.)

The ACTIVATE command corresponds to a rule generated by the translator. The user must provide as an argument the relevant structure object in the objectbase. The rule’s condition checks that the CurrentUser is the user assigned to the task and the myclient attribute of the structure object is currently ResetClient (i.e., null). The effect sets myclient’s value to CurrentClient, as well as setting state0 to Ready. The condition should also check that the current client is not already enacting some other structure object. This might be done by checking that forall structure objects (in the entire objectbase, not just the top-level children of GROUP), the myclient attribute is not equal to CurrentClient. When activating a structure object corresponding to a shuffle subexpression, the rule’s condition should check that the stateXXX attribute for the enclosing shuffle expression is Active.

It is currently unclear what it would mean for a client to execute in the context of multiple activity structures. This is particularly problematical when the client that instantiated the enclosing shuffle expression is intended to enact one of its subexpressions. It is not the case, however, that the same user cannot enact both a shuffle expression and one of its subexpressions, since the same user may control multiple clients in different windows.

After a top-level activity structure or a shuffle subexpression has completed, the user should signal this by ‘‘terminating’’ the structure object. When all subexpressions of a shuffle expression have been terminated, the state attribute for the shuffle expression is automatically set to Done (with respect to the encompassing regular expression). Termination may be accomplished any time after the state attribute changes by explicitly selecting the TERMINATE command. This corresponds to a TERMINATE rule generated by the translator. Its only argument, the current structure object, is automatically provided by the run-time support. The rule’s condition checks that the state0 attribute is Done, meaning the structure object is in a final state (this check might trigger backward chaining through inferencing rules that set this attribute to Done). The effect sets the current structure object’s state0 attribute to Terminated, and reinitializes the owner and myclient attributes to ResetClient.

The INSTANTIATE, ASSIGN, ACTIVATE and TERMINATE rules intended to correspond to user commands should have their condition predicates indicated no_forward and their effect predicates indicated no_backward, so they are never automatically triggered without a user command. For example, the TERMINATE command must inherently be explicit, never automatically triggered, because of the ambiguity of final states for repetitions.

A previously instantiated, assigned, activated and terminated structure object can be ‘‘reused’’ by activating it again. The termination condition in the examples of the next section removes the user assignment, requiring an intermediate ASSIGN command, but this is not necessary. This approach supports repetitions, as well as simply reenacting an entire top-level activity structure.

6.1. Shuffle Expression Translation

Let’s consider some examples. The simplest case of

```
<name>: &( A[?a:X], B[?b:Y], C[?c:Z] )
```

is shown in figure 6-1. Notice that the particular activity structure to instantiate is embedded in the name of its INSTANTIATE command. An alternative approach would be to prompt the user for the class name (<name>) in the body of the envelope. The hide directive prevents a rule from being displayed in the user’s command menu. This is for internal rules intended only for chaining, as part of the ASL implementation, not for direct user invocation. This is also generally useful for MSL. The hide rules are available when one runs MARVEL as an administrator, for debugging purposes.

```
Instantiate[]:
  :
  { INSTANTIATE instantiate }
  ;

Activate [?SO:ACTIVITY_STRUCTURE]:
  (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))
  :
  (and no_backward(?SO.owner = CurrentUser)
       no_backward(?s.clientID <> CurrentClient))
  { }
  no_forward(?SO.clientID = CurrentClient);

Terminate [?SO:test]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state0 = Ready));

Terminate [?SO:AS_0]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive));

Terminate [?SO:AS_1]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive));

Terminate [?SO:AS_2]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive));

hide finish_7 [?SO:test]:
  :
  (?SO.state2 = Inactive)
  { }
  no_forward(?SO.active = Inactive);

hide finish_6 [?SO:test]:
  :
  (?SO.state2 = Ready)
  { }
  no_forward(?SO.active = Done);

Assign [?child:AS_2, ?SO:test]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

Assign [?child:AS_1, ?SO:test]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

Assign [?child:AS_0, ?SO:test]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

hide all_terminated [?SO:test]:
  (forall ACTIVITY_STRUCTURE ?child suchthat (member [?SO.children1 ?child]))
  :
```

```
  (and (?child.active = Terminated)
       (?SO.state1 = Ready))
  { }
  (and (?SO.active = Inactive)
       (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

hide all_assigned [?SO:test]:
  (forall ACTIVITY_STRUCTURE ?child suchthat (member [?SO.children1 ?child]))
  :
  (and (?child.owner <> ResetUser)
       (?SO.state0 = Ready))
  { }
  (and (?SO.active = Active)
       (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

hide finish_5 [?SO:AS_2]:
  :
  (?SO.state1 = Inactive)
  { }
  no_forward(?SO.active = Inactive);

hide finish_4 [?SO:AS_2]:
  :
  (?SO.state1 = Ready)
  { }
  no_forward(?SO.active = Done);

C [?set_c:Z, ?SO:AS_2]:
  :
  no_forward(?SO.state0 = Ready)
  { }
  (and (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

Activate [?SO:AS_2]:
  (and
  (exists test ?parent suchthat (member [?parent.children1 ?SO]))
  (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))

  )
  :
  (and (or no_forward(?SO.active = Inactive)
           no_forward(?SO.active = Terminated))
       no_forward(?SO.clientID = ResetClient)
       no_forward(?SO.owner = CurrentUser)
       no_forward(?parent.active = Active)
       no_forward(?s.clientID <> CurrentClient))
  { }
  (and no_backward(?SO.clientID = CurrentClient)
       no_backward(?SO.state0 = Ready));

hide finish_3 [?SO:AS_1]:
  :
  (?SO.state1 = Inactive)
  { }
  no_forward(?SO.active = Inactive);

hide finish_2 [?SO:AS_1]:
  :
  (?SO.state1 = Ready)
  { }
  no_forward(?SO.active = Done);

B [?set_b:Y, ?SO:AS_1]:
  :
  no_forward(?SO.state0 = Ready)
```

```
    { }
    (and (?SO.state0 = Inactive)
         (?SO.state1 = Ready));

  Activate [?SO:AS_1]:
    (and
    (exists test ?parent suchthat (member [?parent.children1 ?SO]))
    (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))

    )
    :
    (and (or no_forward(?SO.active = Inactive)
             no_forward(?SO.active = Terminated))
         no_forward(?SO.clientID = ResetClient)
         no_forward(?SO.owner = CurrentUser)
         no_forward(?parent.active = Active)
         no_forward(?s.clientID <> CurrentClient))
    { }
    (and no_backward(?SO.clientID = CurrentClient)
         no_backward(?SO.state0 = Ready));

  hide finish_1 [?SO:AS_0]:
    :
    (?SO.state1 = Inactive)
    { }
    no_forward(?SO.active = Inactive);

  hide finish_0 [?SO:AS_0]:
    :
    (?SO.state1 = Ready)
    { }
    no_forward(?SO.active = Done);

  A [?set_a:X, ?SO:AS_0]:
    :
    no_forward(?SO.state0 = Ready)
    { }
    (and (?SO.state0 = Inactive)
         (?SO.state1 = Ready));

  Activate [?SO:AS_0]:
    (and
    (exists test ?parent suchthat (member [?parent.children1 ?SO]))
    (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))

    )
    :
    (and (or no_forward(?SO.active = Inactive)
             no_forward(?SO.active = Terminated))
         no_forward(?SO.clientID = ResetClient)
         no_forward(?SO.owner = CurrentUser)
         no_forward(?parent.active = Active)
         no_forward(?s.clientID <> CurrentClient))
    { }
    (and no_backward(?SO.clientID = CurrentClient)
         no_backward(?SO.state0 = Ready));

  Assign [?SO:test]:
    :

    { ASSIGN get_user return ?uA }
    no_backward(?SO.owner = ?uA);
```

Figure 6-1: Simplest Shuffle Expression

Notice also that although the three structure object instantiations are grouped into a single envelope, through the script generation mechanism, we do not choose to group the three assignments. This is to permit separate task assignments, and reassignments (discussed later), but is not mandatory — all assignments could be done by a single rule with many arguments. The following examples, however, omit the assignment issue for brevity.

It is not appropriate, however, to similarly group the corresponding three activations, since these are done on behalf of three different clients. The user-controlled clients must execute the activations themselves, to associate the correct clientID, so it would not be possible to do this in one or even three batch clients. There is currently no way in MARVEL to change a batch client invoked on behalf of one user into a user-controlled client for another user.

The full text of the embedded case

```
<name>: D[?d:U]; &( A[?a:X], B[?b:Y], C[?c:Z] ); E[?e:V]
```

is shown in the appendix.

The instantiate.env envelope is shown in figure 6-2. Figure 6-3 shows the test1.gen envelope for a simple activity structure, named test1, without any shuffle operators.

```
#!/bin/sh

echo "Enter name of Activity Structure Class "
read ASclass

if [ "x$ASclass" = "x" ]
then
    echo "Must pick a class name"
    exit 1
fi

$ASclass.gen $ASclass

if [ $? -ne 0 ]
then
    echo "Script generator failed."
    exit 1
fi

MARVEL -b $ASclass.marvelrc

rm $ASclass.marvelrc

exit 0
```

Figure 6-2: Instantiation Envelope

```
#!/bin/sh
#
echo "#!MARVEL script" > test1.marvelrc
ObjName=‘get_as_obj_name test1‘
#
# Top Level Instance
#
echo "add -hi as $ObjName test1" >> test1.marvelrc
exit 0
```

Figure 6-3: Simple Generation Envelope

6.2. Shuffle Expression Runtime Support

The ASL run-time support is in the form of an activity structure manager. This module is linked into the MARVEL kernel. Its main job is to intercept every command corresponding to a rule and add the client's activity structure object, if any, at the end of the argument list. It finds the client's structure object by the rather inefficient mechanism of searching through the structure objects in the objectbase until it finds one whose myclient field matches CurrentClient. This approach works because there is no more than one structure object at a time per client.

This approach was taken to minimize changes to MARVEL. A more efficient approach might be to extend the internal context description with administrator-defined fields, which could be specified in MSL in the form of attributes declared for a new built-in CONTEXT class. If it were also possible to assign links in the effects of rules, then each client's context description could be linked directly to its corresponding activity structure, and thus search would be avoided.

In any case, if there is no structure object activated for the current client, the command will be passed through as is to the rule processor. This allows activity structure definitions to be mixed and matched with standard MARVEL rules, and in fact MARVEL rules might even have the same names and parameter signatures as the rules generated for the activity structure definitions. This works with the current MARVEL rule overloading support because of the implicit parameter.

There is one significant extension to MARVEL that we do plan for immediate implementation: Originally, the overloading process returned the single rule that best matches the actual parameters according to inheritance and subtyping. This has been changed to return all equally close matching rules in a list. In the main rule processing loop, the condition of the first rule on the list is evaluated, including backward chaining if one or more predicates are not marked no_chain (or no_backward). But if the condition cannot be satisfied, then instead of halting at this point the evaluation repeats with the next rule in the list, and so on.

This change allows multiple rules with the same name and arguments, but different conditions. This new facility provides a nice means for differentiating multiple occurrences of the identical activity in different places within an activity structure, where different condition and effects are needed to govern control flow. An example was shown in the previous section, with "A1; A1".

To complete this change, the MARVEL loader was modified to no longer disallow such rules, which were previously considered to be ‘‘conflicting’’. (An earlier facility to automatically ‘‘merge’’ the conditions and effects of multiple rules with the same name, parameters and activities had been removed long ago.)

The MARVEL kernel automatically associates a clientID with each client. This clientID is used within the server to determine the session context (e.g., its in-progress rule chain). Unfortunately, clientID's can currently be reused, through distinct server processes executing at different times for the same MARVEL objectbase.

This becomes a problem since the ACTIVATE rule involves persistent storage of clientID's in the myclient attribute of structure objects. An incidental reuse of a clientID could accidently associate the corresponding client within an activity structure previously activated by another client, now defunct, that happened to have the same clientID. The original client might have been explicitly exited, or might have crashed. Thus it is necessary to guarantee unique clientID's (for the same MARVEL objectbase) over all time. This can be done by making the clientID counter persistent, saved within the objectbase directory, rather than transient within the server.

There are several other problems with associating particular clientID's with structure objects. Most notably, when a user exits her client or a client crashes, the objectbase retains the previous clientID and imagines that it is still controlled by that client. To prevent the corresponding structure object from being left dangling forever, we need an ATTACH command/rule generated by the translator. This would set the myclient field of the selected structure object to the current client's ID.

This approach does not afford any protection, however, since an arbitrary client could suddenly take over from an in-progress client. (There can be only one client per activity structure instance and only one activity structure instance per client.) We could add a DETACH command, whose rule sets the myclient field to ResetClient (i.e., null), and then ATTACH would check this condition. But this works only for the explicit exit case, assuming the user remembers to give the DETACH command, and certainly not for failure recovery.

It does not seem possible to solve this problem without augmenting the run-time support. In particular, the MSL rule language could be extended to provide an operator that checks whether a given client ID (from a myclient attribute) is a member of the set of currently active clients (ActiveClients). Alternatively, the ATTACH (and perhaps also DETACH) commands could be explicitly included in the run-time support rather than generated by the translator, but this seems overly intrusive on the MARVEL kernel — and in any case the active clients check might be useful for other applications. Note that DETACH/ATTACH take the place of the previously discussed RESUME/SUSPEND commands. ATTACH and DETACH rules are shown in figure 6-4.

```
    Attach [?SO:ACTIVITY_STRUCTURE]:
      (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))
      :
      (and no_chain(?SO.owner = CurrentUser)
           no_chain(?s.clientID <> CurrentClient))
      { }
      no_forward(?SO.clientID = CurrentClient);

    Detach [?SO:ACTIVITY_STRUCTURE]:
      :
      no_chain(?SO.clientID = CurrentClient)
      { }
      no_forward(?SO.clientID = ResetClient);
```

Figure 6-4: Attach and Detach Rules

It might also be useful to have a DEACTIVATE command, as shown in figure 6-5, which resets a structure object to the same state as the ACTIVATE command — but without requiring previous termination. In a real rule within the context of an activity structure, there would be several predicates setting all the ‘‘stateXXX’’ attributes except state0 to Inactive. The DEACTIVATE command is very dangerous, since there are no conditions. It works as a catch-all to get out of any problem, but if used arbitrarily might create more problems than it solves. Realistic failure recovery and general robustness issues are outside the scope of this project.

Here is an activity structure describing the appropriate usage of these activity structure management commands with respect to a single activity structure instance. Each concurrent repetition would be over a

```
Deactivate [?SO:test3]:
  : no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.stateXXX = Inactive)
       no_chain(?SO.state0 = Ready));
```

Figure 6-5: Deactivate Rule

distinct instance.

```
(( instantiate; &((assign | epsilon), (bind)*); activate;
   ((detach | epsilon); attach)*; (terminate | deactivate) )*;
delete)#
```

7. Synchronization in Shuffle Expressions

Now let's consider the possibility of SEND and RECEIVE activities, where each SEND should be matched by a RECEIVE in some other descendent subexpression of their parent shuffle expression. It is important to point out that we have no intention of introducing actual interprocess communications among clients, since such an approach would be seriously at odds with the MARVEL client/server architecture. Instead, the SENDs and RECEIVEs must be virtual, through operations on the shared objectbase to which all clients are connected.

SEND and RECEIVE pairs are matched according to syntax in the original activity structure definition that indicates ''channels''. Each channel is represented in the objectbase by a distinct attribute of the root structure object (of the enclosing activity structure). Its value operates as a semaphore.

Both SENDs and RECEIVEs are treated as entirely asynchronous, since it is not reasonable to block the human user who selects either command. This is easiest for SEND, since the user can give this command at any time (permitted by the enclosing regular expression) and then go on to her next activity. But the RECEIVE user must implement her own busy-wait loop, to retry RECEIVE repeatedly until there is a matching SEND. Only after the match can she continue to the next activity in the sequence, although in the case of where the RECEIVE is one alternative of an alternation, the other alternative could be attempted instead. (Also, the user can carry out unrelated activities not mentioned in the top-level activity structure.)

To ameliorate the problem of receive busy-waiting, we added a notification scheme. This is done by generating a second rule for each occurrence of send and receive. In the receive case, the condition of the second rule checks that a matching send had not already been selected. In this case, the envelope adds the email address of the current user to the end of a file (this assumes an email attribute in the corresponding user object — or the owner field of the current structure object could be used). In the send case, the condition would check whether there had previously been a matching receive. If so, the envelope would send mail to the first email address in the file and delete this line from the file. Alternatively, the envelope could send mail to all the email addresses in the file, and clear the file. Then these users could race to execute RECEIVE, with one winner and the others added back to the file. The latter approach has been followed.

One approach to selecting the SEND and RECEIVE commands would require the users to find and select the correct channel object in the objectbase (using objects rather than attributes). Our alternative approach provides for a distinctly named SEND command for each channel, and likewise for RECEIVE. The generated rules find the appropriate channel attribute of the root structure object. The user does not need to give any argument, since the current structure object is provided automatically by the run-time support and the root is reached via forward chaining. The channels could be named or numbered by the administrator, and given in the ASL input via use of ‘‘SEND XXX’’ and ‘‘RECEIVE XXX’’ activities. We use numbers, as done in the original activity structure work.

7.1. Synchronization Translator

The example in figure 7-1 uses a counting semaphore, to allow multiple senders and receivers on the same channel.

The send.env and receive.env envelopes are used for the notification.

7.2. Synchronization Runtime Support

The MARVEL kernel supports rule ‘‘overloading’’, meaning that there may be any number of rules with the same name (e.g., edit an object of this type, edit an object of that type). But rules with the same name normally have different parameters and/or conditions. In the case of different parameters (i.e., number and types — types of rule parameters are always classes rather than built-in types or sets), the rule resolution mechanism normally returns the closest match using multi-method inheritance and subtyping.

However, there may be multiple rules that are equally ‘‘close’’ by the definition above, but with different conditions. The resolution mechanism returns all such rules. These rules are considered in some order, and the first one whose condition can be satisfied through backward chaining is executed.

8. Atomic and Non-Atomic Activities

The default is for an activity to be non-atomic, meaning that other activities can be performed concurrently by other clients. This reflects the practical reality of long-duration activities such as edit and compile.

But it is sometimes desirable for an activity to be atomic with respect to other activities within the same top-level activity structure. An alternative would be atomicity with respect to all other activities on the same objectbase (i.e., the same MARVEL environment), but this was not implemented. Note that atomicity is meaningful only in activity structures containing shuffles operators (or concurrent repetition operators).

Atomic activities are implemented by associating a two attributes with the structure object representing the top-level AS. One, the ‘‘atomic_lock’’ attribute, has three possible values: Exclusive, Shared and None. The other, ‘‘lock_counter’’, is an integer.

Before attempting to execute an activity marked as atomic in the activity structure definition, the run-time support attempts to obtain the lock in Exclusive mode. If this is not possible, because the lock is already held in either Exclusive or Shared mode, then the activity is dis-allowed. If it is possible to obtain the Exclusive lock, then the activity is performed and afterwards the run-time support restores the lock to None.

```
Receive_1 [?SO:AS_1]:
  (exists test3 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state0 = Ready)
       no_chain(?root.channel1_semaphore = 0))
  { }
  no_backward(?SO.channel1_receive = Waiting);

Receive_1 [?SO:AS_1]:
  (exists test3 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state0 = Ready)
       no_chain(?root.channel1_semaphore > 0))
  { }
  (and no_backward(?SO.channel1_receive = Ready)
       (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

hide do_receive_1 [?root:test3]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel1_receive = Ready)
  { }
  no_backward(?root.channel1_semaphore -= 1);

hide dont_receive_1 [?root:test3]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel1_receive = Waiting)
  { RECEIVE recv_wait "1" }
  no_chain(?root.channel1_waiting += 1);

hide after_receive_1 [?child:ACTIVITY_STRUCTURE]:
  (exists test3 ?root suchthat (ancestor [?root ?child]))
  :
  (and (or no_backward(?child.channel1_receive = Ready)
           no_backward(?child.channel1_receive = Waiting))
       no_chain(?root.channel1_semaphore >= 0))
  { }
  no_chain(?child.channel1_receive = Done);

Send_1 [?SO:AS_0]:
  :
  no_forward(?SO.state0 = Ready)
  { }
  (and (?SO.channel1_send = Ready)
       (?SO.state0 = Inactive)
       (?SO.state3 = Ready));

hide do_send_1 [?root:test3]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel1_send = Ready)
  { SEND send_waiting "1" }
  no_backward(?root.channel1_semaphore += 1);

hide after_send_1 [?child:ACTIVITY_STRUCTURE]:
  (exists test3 ?root suchthat (ancestor [?root ?child]))
  :
  (and no_backward(?child.channel1_send = Ready)
       no_backward(?root.channel1_semaphore > 0))
  { }
  no_chain(?child.channel1_send = Done);
```

Figure 7-1: Send and Receive Rules

In order to guarantee atomicity with respect to non-atomic activities as well as other atomic activities, it is necessary that all non-atomic activities obtain the lock in Shared mode. If the lock is in None mode, the run-time support sets the lock to Shared and increments the lock_counter from 0 to 1. When a non-atomic activity ends, the lock_counter is decremented. If it is now 0, the lock is restored to None. Thus an atomic activity cannot begin if one or more non-atomic activities are already in progress. Both Shared and Exclusive locks are obtained after the condition has been evaluated and any new bindings done, but before beginning execution of the activity envelope.

An activity is indicated as atomic by the `atomic` keyword, as follows:

```
    <name>: &( atomic A[?a:X], B[?b:Y])
```

In this example, A is atomic and B is non-atomic.

There is no notion of blocking in order to wait for a lock. Only the condition of the rule is executed, and failure to subsequently obtain the lock simply prevents the actual activity part of the rule from being initiated.

Different occurrences of the same activity A (with the same parameters) in an AS cannot be an arbitrary mixture of atomic and non-atomic. As explained in the section 5, disambiguation results in effectively merging multiple occurrences of the same activity when it is not possible to distinguish the states at this point in the recognition process. Thus it is necessary that all merged occurrences are atomic, or all are non-atomic. This is enforced by the translator.

9. Binding Data to Activity Structures

The basic issue is that an instance of an activity structure should be recognized with respect to a particular data item, or set of related data items, rather than with respect to an objectbase. The problem becomes clear by considering the example

```
    ( edit[CFILE]; compile[CFILE] )*
```

Say an edit is applied to CFILE X. Then its AS instance should be matched against a compile applied to CFILE X, not to say, CFILE Y.

This simplistic explanation is misleading, however, since it is rare that practical ASs would be applied only to a single data item. Consider

```
    ( edit[HFILE]; compile[CFILE] )*
```

where the intent is to compile a CFILE that includes the HFILE. The arguments of the two activities are not the same data item, and may not even be attributes of the same composite object since arbitrarily many HFILEs including standard libraries may be included by a CFILE.

Actually, it is probably desirable to recompile all the CFILEs that include this HFILE, not just one. The appropriate AS is

```
    ( edit[HFILE]; (compile[CFILE])* )*
```

But here the CFILE must be different on each iteration of the compile.

Our approach is to add parameter names as well as types to the activity structures, to indicate ‘‘bindings’’ of data items. The formal parameter names are hereafter called symbols. Bindings would be implemented by associating a symbol table with each root structure object, an instance of a named activity structure class. The symbol table is represented by a collection of link attributes, with the name of each attribute identifying the corresponding symbol. Those symbols that can take on multiple values, as in the

HFILE/CFILE example above, would be represented by set attributes linking to all objects permitted as actual parameters. (Bindings are handled in a slightly different way when we consider concurrent repetition in a later section, to permit different bindings in different repetitions.)

Consider the simplest example:

```
    foo: ( edit[?C:CFILE]; compile[?C:CFILE] )*
```

The symbol table for the AS instance would consist of exactly one entry, C. To apply this particular instance to a particular CFILE, O, the user would have to first instantiate the foo AS using the INSTANTIATE command described in the previous document, creating a structure object named something like foo1. Then the AS is assigned to a particular user with the ASSIGN command and activated with respect to a particular client controlled by that user with the ACTIVATE command. This process does not, however, undertake the binding. (Actually, the assignment could be optional. The condition of the activate rule would check whether or not the desired structure object had been assigned; if so, then the condition would require the current user to be the one assigned (or a member, if this is done as a set), but if not, then any user can activate.)

In order to execute edit or compile with respect to the client's current structure object, it is necessary to bind some object to C. The user would bind the O object to foo1 using a new BIND command to install a link from the C attribute of foo1 to the selected object O (it was necessary to add assignments to links in the effects of rules in order to support BIND). From this point on (until the link is changed) all edits and compiles are governed by this AS to be restricted to O. Edits and compiles on other objects remain unaffected. Additional structure objects for the same AS would have to be instantiated, assigned, bound and activated in order to constrain them. A client can have only one activated structure object at a time, but assigned and previously activated structure objects can be detached and attached to change between them as desired.

Now consider

```
    bar: ( edit[?H:HFILE]; (compile[multiset ?C:CFILE])* )*
```

The symbol table consists of two entries, H and C, with C represented as a set — indicated by the `multiset` option. Representing C as a single value would imply that only one CFILE could include the HFILE, which is not practical. The user would have to bind the H attribute of the client's current structure object before editing it, and would have to bind the C set attribute accordingly before compiling each CFILE, in order to associate these edits and compiles with that AS instance.

There is a difficulty with binding set attributes, because this cannot be done in a single command — MARVEL does not have any way of allowing a variable number of arguments. Instead, there must be multiple invocations of the BIND command for each member of the set, and it must be indicated which of possibly many set symbols (or other symbols) is intended.

The latter might be accomplished by prompting the user in the envelope, but it would be easier from an implementation standpoint to have multiple distinct rules for each symbol. Thus the BIND rules generated by the translator would be named distinctly, in the form BIND-classname-symbolname, where the classname is the name of the top-level activity structure (this will not work for concurrent repetitions, discussed in section 12, since separate bindings must be made for distinct repetitions). Alternatively, we could have names of the form BIND-symbolname, with the classname distinguished by the class of the actual argument (i.e., normal overloading resolution). We chose the latter, to avoid a user menu filled with an immense number of BIND-xxx commands.

The ASL translator would generate the following BIND and UNBIND commands from the bar input. An UNBIND must be done after a BIND in order to accomplish a later (re)BIND. This does not have to be enforced in the conditions of rules, because the underlying implementation gives an error message if one attempts to link a single-valued attribute that is already linked. (In the case of a multi-valued attribute, the link is added if it is not already there.) The relevant rules are shown in figure 9-1.

```
<name>: ( edit[?H:HFILE]; (compile[multiset ?C:CFILE])* )*

==>

Unbind_C [?SO:<name>]:
  (exists CFILE ?C suchthat (linkto [?SO.C ?C]))
  :

  { }
  (unlink [?SO.C ?C]);

Bind_C [?C:CFILE, ?SO:<name>]:
  :

  { }
  (linkto [?SO.C ?C]);

Unbind_H [?SO:<name>]:
  (exists HFILE ?H suchthat (linkto [?SO.H ?H]))
  :

  { }
  (unlink [?SO.H ?H]);

Bind_H [?H:HFILE, ?SO:<name>]:
  :

  { }
  (linkto [?SO.H ?H]);

Unbind [?SO:<name>]:
  (and
  (exists CFILE ?C suchthat (linkto [?SO.C ?C]))
  (exists HFILE ?H suchthat (linkto [?SO.H ?H]))

  )
  :

  { }
  (and (unlink [?SO.H ?H])
       (unlink [?SO.C ?C]));

Bind [?H:HFILE, ?C:CFILE, ?SO:<name>]:
  :

  { }
  (and (linkto [?SO.H ?H])
       (linkto [?SO.C ?C]));
```

Figure 9-1: Bind and Unbind Rules

It would of course be better to have only a single overloaded BIND name, with the symbolname provided as a literal (string) argument by the user. The problem is that MARVEL currently does not support any arguments to commands that are not objects. Adding literal arguments sounds easy, but would be amazingly complex because we would have to radically change our entire approach to preprocessing commands in the client and shipping them to the server. This may be done for MARVEL 3.1, since literal arguments would be nice to have (literal arguments are already permitted for activities themselves, but must be hardwired into the corresponding rules, since there is no way to pass them into rules). The single UNBIND command unbinds all the parameters.

9.1. Dynamic Binding

The above describes a static binding process, where all possible parameters must be determined in advance by the user prior to activation of an activity structure instance. Binding can be restricted to occur only before an activation and between a later deactivation and a future activation by adding yet another flag to the conditions and effects of BIND, ACTIVATE and DEACTIVATE rules. Note that this prevent intermediate BIND commands that might override the dynamic bindings explained below. (Actually, there is nothing preventing the user from changing the bindings at any time using the normal built-in link command, since MARVEL does not currently support overloading of built-in commands to add a condition to prevent this. Such a facility might be added in MARVEL 3.1.)

We believe static binding alone provides a poor user interface. We therefore augment this mechanism with a dynamic binding process, so that some symbols might be bound in advance but others could be determined on the fly according to the needs of the user.

We therefore add the set and use options to parameters. In

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

an edit command applied by the user to a particular CFILE would check whether the current SO already had its C binding set. If so, then the system enforces whether the current argument is the same as the bound one. An edit on another CFILE could not be applied in the context of this particular SO. If, however, the C binding were not already set, then the edit command would implicitly set the binding to the argument provided by the user. All future occurrences of edit with respect to this SO would find the C symbol already bound. If the edit is permitted to occur in the context of the SO, the SO's state is advanced to indicate a compile command is expected.

When the user enters the compile command for a particular argument, the use option indicates that the C binding must already have been set. If the argument is the same as the one bound, then the compile occurs in the context of the current SO, advancing its state to expect another edit. If the argument is not the same as the one bound, or if there is as yet no binding, then the compile can occur only outside the context of this SO.

The notion of ‘‘context of this SO’’ is subtle. A given client may or may not have a currently activated structure object (SO). If there is no current SO, then there are no restrictions whatsoever on control or data of activities beyond those defined in the conditions and effects of normal MSL rules. That is, ASL definitions are not considered in any way. Set, use, etc. options are meaningless when there is no current SO, and are totally disregarded by the system.

There is a current SO whenever there has been an ACTIVATE command that was not later followed by a DEACTIVATE, TERMINATE, DETACH (this SO) or ATTACH (to a different SO). Every activity attempted by the user is considered in the context of the current SO. If the arguments provided by the user are acceptable considering the bindings of the current SO and the set, use, etc. options employed in the activity structure definition, then and only then do control restrictions specified in the AS come into play. That is, data restrictions are considered first.

If the arguments provided by the user are not acceptable given the current SO, then the activity may still be performed (if there is a normal MSL rule defining the activity) but outside the context of any SO. It is not possible to automatically switch to the ‘‘correct’’ SO. First, such automatic switching among activity structures would be extremely complicated to present to the user, and in many cases would not be desired by the user (for the same reasons a lot of computer users do not like DWIM features that automatically do something apparently at random when a user mistypes a command). Second, there is unlikely to be a single correct SO since the same data may be bound in multiple SO’s (in fact this is necessary for the semantics of shuffle operators, since a distinct SO represents each shuffle operand). It would be necessary to apply the activity in the context of all the correct SO’s, in the sense of control restrictions, but actually carry out the activity only one time. This would require substantial changes to our current implementation.

Now consider

```
    bar: ( edit[set ?H:HFILE]; (compile[set ?C:CFILE])* )*
```

The H parameter is dynamically bound by the first edit with respect to a corresponding SO, and the C parameter is bound by the first compile. But this does not permit a collection of possible CFILEs, and we need some additional syntax to indicate that a set attribute is required. Thus we add the multiuse option, to support

```
    bar: ( edit[set ?H:HFILE]; (compile[multiuse ?C:CFILE])* )*
```

Note that this example requires the C symbol to have been bound using the static BIND command, since it is not set in the activity structure definition.

This approach does not permit the case where different bindings can be used on each iteration. Thus we add the reset option, which should be used instead of set when the administrator writes the activity structure, in order to indicate this is permitted.

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

Reset works as follows. The edit command dynamically rebinds C to its current argument, whether or not C was previously bound (by a previous edit or by BIND). Then the following compile is restricted to this argument.

We could get a multiset-like effect by employing reset.

```
    bar: ( edit[set ?H:HFILE]; (compile[reset ?C:CFILE])* )*
```

Each iteration of the compile resets the single C binding, not a set here, to the new CFILE argument. But this does not permit maintenance of a collection of CFILEs, as desired for the reasons explained above. Thus a real multiset is desirable, to dynamically add the current argument to the collection only if it is not already there.

We also add the multireset option, to replace the entire collection by the single new argument provided by the user. Additional items can still be added through later multiset options. Thus the "multi" component of the option keyword alerts the ASL translator and run-time support that a set attribute should be used to represent the binding, rather than a single-valued attribute.

To get the effect of a ‘‘don’t care’’ option throughout an activity structure, the reset option may be used with all symbols.

When ASL strategies import MSL strategies, a conceptual issue arises because the parameters passed in

MSL chains do not currently consider the ASL bindings. This issue does not cause an implementation problem, per se, because rules generated for ASL do not chain into normal MSL rules. This is because we generally translate into no_forward and no_backward predicates, and the exceptions are special cases intended for chaining into other generated rules. The exceptions can be distinguished from normal MSL rules by their extra parameter, which must be an instance of a subclass of ACTIVITY_STRUCTURE.

Actually, the above is true only when the administrator provides a bare MSL skeleton, with only header and activity, and no condition and effects. If the administrator includes condition and effects, then the output MSL rule (let's call it an ASL rule to lower confusion) will include both the control predicates added by the translator and it's original predicates. These predicates might cause it to backward and forward chain with respect to normal MSL rules imported in other files. There is a subtle problem with this, because it may turn out that an activity is permitted through an MSL rule when it was not permitted by the ASL rule. Or, alternatively, that an activity happens twice, once through forward chaining to the MSL rule and one due to forward chaining into the ASL rule (as described below). Another possibility is that the ASL rule forward chains into an MSL rule that changes the objectbase to a state such that the ASL rule's condition is not satisfied, when it was satisfied before the MSL rule was inadvertently considered first.

The last problem can be solved easily: when more than one rule can be fired, fire first those with the most parameters. All ASL rules have an extra parameter, the implicit activity structure instance, added by the translator. Making this ordering a general feature of the MARVEL kernel does not seem to break anything, and might even be desirable. But the other two problems can only be solved by disallowing import of MSL rules into ASL files. This seems problematic, we would lose all the C/MARVEL MSL rules needed to define a realistic C programming process. But there is a solution.

Let's consider how we might truly integrate activity structures and normal MSL rules, assuming these MSL rules are included directly in the ASL file rather than imported. Perhaps MSL's bindings obtained through dynamic parameter passing could be integrated by treating the ASL bindings as ‘‘constraints’’ on MSL chaining. Set could still be the default (although we might want to think about this some more). Then a potential MSL chain would be restricted to obeying the previous ASL bindings. If a particular parameter passing would violate the bindings, then it wouldn't happen. But if a parameter passing could set a previous unbound symbol, it would be a allowed, and this binding would hold for later rules executed in the context of the then-current SO. Thus only a subset of possible MSL chains would occur.

The above addresses only data restrictions. The current SO further implies control restrictions, based on the various state variables. In addition to the normal conditions of the MSL rule, the state conditions must be satisfied in order to execute the rule in the context of the current SO. So even though chaining is triggered by normal MSL predicates in effects and conditions, the chaining is constrained by the corresponding activity structure. Again, only a subset of possible MSL chains would occur.

This seems a reasonable interpretation of "integration" of ASL and MSL. And surprise, surprise, this is what we implemented.

9.2. Run-time Support for Data Bindings

The implementation is rather complicated, because of the way that commands are resolved to rules. Normally in MARVEL, rules may be overloaded provided there are different signatures and/or conditions (actually, the support for multiple conditions with the same signature was added due to issues that came up previously in the ASL project, but is also generally useful). The kernel finds the closest matches considering multiple inheritance and multiple arguments, and the collection of equally close rules is passed to the rule processor. It considers the conditions one by one, trying to satisfy them via backward chaining (this is complex due to the interactions with automation versus consistency predicates, and no_forward, no_backward and no_chain directives on predicates). The rule processor sends to the client for execution the activity for the first such rule whose condition can be satisfied.

For the ASL run-time support, we have added a capability to automatically add the current SO as the last parameter, in order to match against rules generated by the ASL translator to enforce the indicated control. The resolution process is now attempted twice, once with the current SO to find any matching rules when this argument is considered for the signature, and then if no matching rules are found this argument is dropped and only the regular MSL rules are considered.

There may be multiple equally close rules with different set/reset options (regular MSL rules are implicitly set option, since the same syntax is used as this default case, but they would never be executed in the context of a SO since this parameter does not appear in their signatures). We want to rebind (for reset) or disavow (for set when the argument does not match the binding) according to the options of a particular rule only if that rule's condition is actually satisfied, since otherwise it may actually be another rule in the equally close collection that will be matched and fired. Further, in the case of disavowal, we want to consider the equally close collection without the SO argument in the signature. But none of this can be known until after backward chaining is attempted by the rule processor. (This problem is quite subtle.)

When a command argument does not match the binding of the current SO, then there are two possible courses of action. The first is to not allow the command to be executed at all. This is what will happen if that activity is defined in the ASL input file but not in any of the imported MSL files. In this case, the only corresponding rules will be generated by the ASL translator, with conditions and effects to manipulate the SO state. Thus a SO argument must be supplied, and it is not possible to execute the activity on arguments not bound to the current SO.

The second option is to permit the command to be executed, but not affect the state of the SO in any way. This can be accomplished if an imported MSL file defines its own rules with the same name and signature as the activity definition. These rules do not require a SO argument. One implication of this option is that an activity disallowed at this point by the activity structure might actually be allowed because it matches a normal MSL rule. There is no way to prevent this behavior except to prevent importation of MSL rules with the same names and parameters as MSL rules contained in the ASL file (to which structure object arguments will be added). This is not enforced by the ASL transaction, so it is up to the environment administrator to avoid inclusion of conflicting rules in a MARVEL/ASL environment.

We change the resolution process for ASL to collect the equally close rules both with and without the SO argument, and concatenate the two lists for presentation to the rule processor. This permits the behavior where if a command should not be executed with respect to the current SO because its arguments do not match the bindings, it is still executed but outside this context if it matches a normal MSL rule.

This requires changes to the rule processor, to pass its N+1 arguments to any rules on the list taking N+1 parameters, and only pass the first N for the remaining rules. Another change is that the rule processor must carry out the bindings in those cases where there is a structure object argument. Set bindings are processed by first checking whether the symbol is already bound in the SO's symbol table (i.e., collection of link attributes). If so, then it checks whether the actual parameter to the command is the same as this bound object. If yes, it continues with the other parameters and then attempts to fire this rule. If not, then this rule fails and the rule processor goes on to the next possible rule in its list.

Set bindings on a set attribute are processed by checking whether the formal parameter's set attribute is already bound to one or more objects. If not, then this rule fails (another rule, with the same symbol and type as this one but without the special SO parameter, might succeed). If so, then the system checks whether the actual parameter is a member of this set. If it is, then the rule processor continues with the other parameters and attempts to fire this rule. If the actual argument is not a member of the bound set, then the rule fails. Reset (multireset) bindings are processed by replacing (adding) the new argument to the parameter attribute if not already there.

Once binding processing is completed with respect to a particular rule, the normal MARVEL backward chaining is initiated to attempt to satisfy the condition. If the condition can be satisfied, the activity is executed in the client and then the effects asserted in the server.

But if the condition cannot be satisfied, it is necessary to undo any new bindings that were made during the binding process, before considering the next rule. Otherwise, the chance ordering of rules in the ASL input file would result in different behaviors, since a failed rule could actually force a change in binding. Consider the example

```
    foo: ( edit[set ?C:CFILE]; compile[set ?C:CFILE] )*
```

If a failed rule could permanently set the binding, then executing the compile command on a CFILE O before doing the edit command would have the side-effect of binding C to O even though the compile command would not have been executed with respect to the structure object! Of course, the undo process must not affect any previous bindings, whether static or dynamic, only those considered specifically with respect to the failed rule.

10. Argument Constraints Across Multiple Activities

One problem with the HFILE/CFILE example above is it does not constrain the CFILEs to be related to the HFILE in any way. We would like to compile all the CFILEs whose ‘‘includes’’ set attribute links to the HFILE, but only those CFILEs. Some syntax like the following might be employed:

```
    ( edit[?H:HFILE];
      (compile[?C:CFILE suchthat (linkto [?C.includes ?H])])* )
```

However, any a priori binding of the set of possible ?Cs with respect to this activity structure would not permit co-existence of the following activity structure, with respect to the same or another user.

```
    edit[?D:CFILE]
```

The issue here is not the C versus D symbol. The problem is that the user can employ this rule to edit a CFILE in some way to change which particular HFILEs it includes. This should be reflected in the objectbase by changing the HFILEs linked via the CFILE's includes set attribute. (Either the end-user would have to update the links using BIND, or an appropriate set of rules would be needed to update the links.) But then the previously bound set C for the HFILE activity structure would be wrong. The only possible repairs are to rebind C, either implicitly or explicitly. This could be handled through either the static rebinding approach using a BIND command or dynamically using the multireset option on the C parameter — but now with the actual resets restricted to those CFILEs whose includes attribute already linked to the object bound to H.

We did not have time to design and implement this kind of argument constraint, using structural information, and limit constraints to employing only the parameter name.

There is also the issue of non-null conditions and effects in the original activity definitions. Non-null conditions permit expression of additional activity arguments obtained through binding queries from the command arguments, as well as additional constraints stated in the property list on both formal and derived parameters. Non-null effects express the multiple possible results of black-box external tools.

The translator supports these by ANDing together the generated bindings and property list of a rule with the bindings and property list provided by the administrator. The effect generated for a rule is ANDed with *each* of effects provided by the administrator.

Non-null conditions and effects in the MSL rules provided as part of the ASL file must be used with care, because it is easy to create a situation where a rule required to continue the activity structure cannot be executed because its condition requires a predicate that appears in an effect of a rule corresponding to another part of the activity structure. For example, in

```
((edit[reset ?h:HFILE] | edit[multiset ?c:CFILE]);
    (compile[multiuse ?c:CFILE]))*
```

it is possible that the recompile of a particular CFILE after a CFILE edit has set the status attributes of the CFILE in such a way that it cannot be recompiled again after editing an HFILE. The administrator must take care that the HFILE edit also effects the status attributes in such a way that the compile is re-enabled.

11. User View

All structure objects have a special string attribute, called ‘‘as_string’’, which gives in string form the corresponding activity (sub-)structure corresponding to that structure object. Printing the structure object to look at this attribute is the best way for a user to tell one un-activated structure object from another. Printing the structure object to look at its class name also indicates the name of the corresponding top-level activity structure, but does not provide useful discrimination among non-root SO’s since the class names are automatically generated using a counter. The print is displayed in the MARVEL text window, below the graphical display of the objectbase (i.e., print is a MARVEL built-in command, unfortunately named, that does not do anything remotely relevant to generating hardcopy).

There is no good way, however, to distinguish between two non-root structure objects at the same level in the hierarchy if they happen to have the identical activity substructure. Consider

```
&(A, B, C) | &(A, D, E)
```

In the internal implementation, the first A structure object is a member of the children1 set attribute and the second a member of children2. But the MARVEL objectbase displays only the six component objects in the order A, B, C, A, D, E without any indication of which are members of which attribute. This is because the current graphics support does not leave any room to display attribute names. (The MARVEL user interface has a lot of known problems, and this is one that probably will not be fixed even in MARVEL 3.1.)

For an activated structure object for the current client, the user can display the activity sub-structure as a string using the SHOW command, which is implemented by the following rule. The current SO argument is automatically added by the implementation. If there is no current SO, then no such argument will be provided, and the rule will not be matched. But if the user provides an SO argument explicitly, then there is no check that this is the current SO; its string is simply displayed.

```
SHOW[?SO:ACTIVITY_STRUCTURE>]
    :
{ envelope to display ?SO.as_string }
;
```

The display is necessarily in the MARVEL startup window (from which the MARVEL client was invoked) rather than the MARVEL text window. There is no way for an envelope or an external process implemented any other way to display in the internal (to MARVEL) text window due to the way X works.

It is not possible to show the user a cursor into the activity structure string showing her SO's current position, for two reasons. First, there may be multiple possible current positions because of the ambiguity problem. But we cannot even show all the possible current positions due to the second problem: the translation from non-deterministic to deterministic FSA inherently loses this information when it combines multiple states into one. It is possible only to distinguish a superset, i.e., a set of positions that do include the actually possible ones, but also other positions. This seems more likely to be confusing to the user than helpful, so we have not attempted to implement it.

12. Concurrent Repetition

The concurrent repetition operator, #, could be handled similarly to the shuffle operator. The top-level AS is represented by a concurrent repetition object (CR object) rather than a structure object (renamed shuffle object, still called SO for short). Each occurrence of a CR expression is translated to a new ‘‘CR_n’’ set attribute, where each member of the set is a child CR object representing one of the repetitions.

The main differences between CR objects and SO objects is regarding instantiation and symbol tables. By definition, when a structure operator is defined, there is a specific number of known operands. Not so with concurrent repetition operators. Arbitrarily many operands can be instantiated on the fly. Thus it is not appropriate to generate the full hierarchy when the top-level activity structure is instantiated. This recursive process must stop whenever a concurrent repetition operator is encountered, and then zero or more operands would be explicitly instantiated as needed.

Thus some mechanism is necessary to explicitly instantiate (add) a new CR object to the hierarchy. The user must be able to name the particular CR expression desired, and then assign and activate the new instance as previously described for SOs. Persistent counters would be needed for generating object names, or perhaps the user can be prompted for a human-meaningful name as part of the instantiation envelope. Different instantiation envelopes must be generated for each occurrence of a CR expression within a top-level AS.

There is a serious problem with knowing when a CR expression is ‘‘done’’. Consider the following simple example:

```
A; (B)#; C
```

We have already agreed that when a user attempts to initiate the C activity, she is forced to wait until after all currently activated B's are done. But the attempt to initiate the C activity should prevent any clients from activating any new B's (it is unclear what it means, actually, to deactivate and reactivate a previously instantiated CR object).

Thus there must be some means to record that some user attempted to start up C, even though that user is not blocked — and can undertake other activities outside this AS. This would have to be considered in the condition of the activate rule, to prevent further initiations of B's. This raises the possibility that a user might ‘‘change her mind’’ about the request for C, and in a good user view would have some way of removing this recording, so that additional B's can be permitted.

There is also a problem of assigning users to CR objects. It might be desirable for a managing user to somehow instantiate a CR expression, and give the set of users expected to instantiate individual CR subexpressions. Perhaps C could then be pending on all of these users activating and terminating (deactivating) their CR objects, without other users not in the assigned set being permitted to instantiate or activate new CR objects for this CR expression. If this approach is taken, there is no ambiguity as to when the CR expression has ended.

Returning to the problem of symbol tables, the other major distinction between SOs and CRs is that a distinct symbol table is needed for the subset of symbols used only in that CR (as opposed to in the surrounding AS). Recall that a single symbol table at the root is sufficient for handling an SO hierarchy. But now a new symbol table must be associated with each CR object, but with inheritance from any ancestor CR objects. Distinguishing which particular symbols are employed only within a single CR expression requires either additional syntax for ASL, or a second pass in the translator.

So binding, whether static or dynamic, must be done within the projected CR hierarchy (i.e., ignoring the SO hierarchy for the same AS instance). This requires a more involved user view for the static approach, so the end-user can accurately choose the right CR object for making links. Queries such as asking what is the current CR object seem warranted, but MARVEL does not currently support ad hoc queries (as opposed to browsing).

Dynamic binding, on the other hand, is relatively simple — the bindings (characteristic function) clause of a rule simply searches up the hierarchy for the containing CR object with the right set of link attributes (representing its symbol table). This means the translator must associate the appropriate link attributes only with the right CR classes (whose names must be automatically generated, or provided in the ASL syntax).

Note that although new symbols are generated for each CR object, the entire projected tree of CR objects shares the same channels. Thus channels are still implemented as attributes of the root, as described in the previously design document.

The concurrent repetition operator obviously opens many new questions, and we just did not have time to consider them.

Acknowledgments

Bill Riddle, Brian Nejmeh and Steve Gaede contributed substantially to the contents of this document. Naser Barghouti also made some helpful suggestions.

References

[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.

[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 91] Naser S. Barghouti and Gail E. Kaiser.
Scaling Up Rule-Based Development Environments.
In *3rd European Software Engineering Conference*. Milano, Italy, October, 1991.
In press. Available as Columbia University Department of Computer Science, CUCS-047-90, January 1991.

[Barghouti 9x] Naser S. Barghouti.
Concurrency Control in Rule-Based Software Development Environments.
PhD thesis, Columbia University, 199x.

[Ben-Shaul 91] Israel Z. Ben-Shaul.
An Object Management System for Multi-User Programming Environments.
Master's thesis, Columbia University, April, 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 *1st International Conference on the Software Process*. Los Angeles CA, October, 1991.
In press. Available as Columbia University Department of Computer Science CUCS-014-91, April 1991.

[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 Knowledge-Based Software Engineering Conference*. Syracuse NY, September, 1991.
In press. Available as Columbia University Department of Computer Science CUCS-022-91, May 1991.

[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.

[Kaiser 90] Gail E. Kaiser, Naser S. Barghouti and Michael H. Sokolsky.
Experience with Process Modeling in the Marvel Software Development Environment Kernel.
In Bruce Shriver (editor), *23rd Annual Hawaii International Conference on System Sciences*, pages 131-140. Kona HI, January, 1990.

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

[Sokolsky 91] Michael H. Sokolsky and Gail E. Kaiser.
A Framework for Immigrating Existing Software into New Software Development Environments.
Software Engineering Journal , 1991.
In press. Available as Columbia University Department of Computer Science CUCS-027-90, May 1990.

I. Full Example

ASL input file:

```
    strategy test

    imports data_model;
    exports all;

    objectbase

    # EDITOR, in this test example, represents a tool used both for editing
    # and for proofreading.  The two actions, however, are regarded as being
    # separate activities.

    FORMATTER :: superclass TOOL;
        format : string = format;
    end

    PRINTER :: superclass TOOL;
        spool : string = print;
    end

    DEBUGGER :: superclass TOOL;
        exec : string = execute;
    end

    BUILD :: superclass TOOL;
        build_program : string = build;
    end

    EDITOR :: superclass TOOL;
      editor: string = editor;
      proof: string = proof;
    end

    # COMPILER, in this test example, represents the C compiler.

    COMPILER :: superclass TOOL;
        compile : string = compile;
    end

    ASSEMBLE :: superclass TOOL;
        assemble: string = assemble;
    end

    end_objectbase

    rules

    format[?f:DOCUMENT]:
      :
      { FORMATTER format ?f.assembled ?f.formatted }
      ;

    printdoc[?f:DOCUMENT]:
      :
      { PRINTER spool ?f.formatted }
      ;

    edit[?f:DOCFILE]:
      :
      { EDITOR editor ?f.contents }
      ;

    edit[?h:HFILE]:
      :
      { EDITOR editor ?h.contents }
      ;

proof[?f:DOCFILE]:
  :
  { EDITOR proof ?f.contents ?f.spellfile }
  ;

# proof[?d:DOCUMENT]:
#  :
#  { EDITOR proof ?d.assembled }
#  ;

edit[?c:CFILE]:

    # if the file has been reserved, you can go ahead and edit it
    :
#    (?c.reservation_status = CheckedOut)

    { EDITOR editor ?c.contents }

    (and (?c.compile_status = NotCompiled)
         (?c.timestamp = CurrentTime));
#        (?c.analyze_status = NotAnalyzed)
#    no_chain (?c.reservation_status = CheckedOut) ;



compile [?f:CFILE]:

    (forall HFILE ?h suchthat (linkto [?f.ref ?h])):

    # if the C file has been analyzed successfuly but not yet compiled,
    # you can compile it.  The compilation changes the status of the C
    # file to either compiled or error.

         (or ( ?f.compile_status = NotCompiled)
             (  ?f.compile_status = Error ))

    { COMPILER compile ?f.contents ?f.object_code ?h.contents }

    (?f.compile_status = Compiled);
    [?f.compile_status = Error];


build[?p:PROGRAM]:
    (and
         (forall MODULE  ?m suchthat (member [?p.modules ?m]))
         (forall CFILE   ?c suchthat (member [?m.cfiles  ?c])))
    :
    (?c.compile_status = Compiled)
    { BUILD build_program ?c.object_code ?p.exec }

    (?p.build_status = Built);
    (?p.build_status = NotBuilt);


exec_prog [?p:PROGRAM]:
:
  (?p.build_status = Built)
    { DEBUGGER exec ?p.exec }
;

assemble [?d:DOCUMENT]:
 (and
  (forall DOCFILE ?Docf suchthat (member [?d.docfiles ?Docf]))
  (exists DOCFILE ?head suchthat (member [?d.header   ?head])))
  :
  { ASSEMBLE assemble ?head.contents ?Docf.contents ?d.assembled }
;


activity structure

# test1:
#       edit[?doc:DOCFILE];
#       (proof[?doc:DOCFILE] | edit[?doc:DOCFILE])*;
#       proof[?doc:DOCFILE]

# test3:  &((proof[?df1:DOCFILE] | atomic edit[?df1:DOCFILE] |
#           send 1; receive 2 |
#           receive 3; proof[?df2:DOCFILE]; send 4)*,
#          (proof[?df2:DOCFILE] | atomic edit[?df2:DOCFILE] |
#           send 3; receive 4 |
#           receive 1; proof[?df1:DOCFILE]; send 2)* );
#           assemble[?doc:DOCUMENT]

test2:  (((edit[multiset ?c:CFILE] | edit[multiset ?h:HFILE]);
         (compile[multiuse ?c:CFILE])*)* ; build[?p:PROGRAM] ;
         (exec_prog[?p:PROGRAM] | epsilon))*

test4: &(
          &(
             (edit[reset ?a:DOCFILE] ;
             ( send 1 ; receive 3 ; (edit [reset ?a:DOCFILE] | epsilon))*)
             ,
             (edit[reset ?b:DOCFILE] ;
             ( send 2 ; receive 4 ; (edit [reset ?b:DOCFILE] | epsilon))*)) ;
             assemble[?doc:DOCUMENT] ;
             format[?doc:DOCUMENT] ;
             printdoc[?doc:DOCUMENT]
        ,
         (( receive 1 ; proof[use ?a:DOCFILE] ; send 3) |
          ( receive 2 ; proof[use ?b:DOCFILE] ; send 4))*)


end activity structure
```

MSL file imported into the ASL strategy:

```
#
#                   Marvel Software Development Environment
#
#                                Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#

strategy data_model

# This strategy contains all the class definitions needed for a typical
# C environment.  The class definitions are imported by all other
# strategies that define various aspects of the process model for
# C/Marvel.

# Interface with other strategies.  Since this is a basic data model that
# all other strategies import, we don't specify anything.

imports none;
exports all;

# Class definitions
```

```
objectbase

# GROUP is the top-level class.  An instance of GROUP contains several
# projects.   The fact that it is top level is set in the user's
# environment as part of the startup of Marvel.  So a Marvel objectbase
# can contain several group objects.

GROUP :: superclass ENTITY;
    build_status : (Built,NotBuilt) = NotBuilt;
    projects : set_of PROJECT;
    as : set_of ACTIVITY_STRUCTURE;
end



# PROJECT is an entity that defines much of the structure of a typical
# software project.  PROJECTs can contain libraries, binaries, programs
# documents and includes in this example.

PROJECT :: superclass ENTITY;
    status : (Release,Maintenance,Development) = Development;
    archive_status : (Archived,NotArchived) = NotArchived;
    build_status : (Built,NotBuilt) = NotBuilt;
    libraries : set_of LIB;
    programs : set_of PROGRAM;
    doc : set_of DOC;
    incs : set_of INC;
end




# PROGRAM is important to distinguish from PROJECT.  A PROGRAM is a single
# executable unit, whereas a PROJECT is a collection of PROGRAMs, and other
# entities.  PROGRAMs thus contains things like documents, cfiles, modules,
# if it is large, and include files.

PROGRAM :: superclass ENTITY;
    build_status : (Built,NotBuilt,Error) = NotBuilt;
    debug_status : (OK,NeedsDebugging) = OK;
    docs : set_of DOC;
    modules: set_of MODULE;
    incs : set_of INC;
    exec : binary;
end



# LIB is a shared archive type library.  It consists of modules, which in
# turn contain c files.  The ultimate representation of a library is a
# .a file, that is, an archive format file.

LIB :: superclass ENTITY;
    archive_status : (Archived,NotArchived) = NotArchived;
    modules : set_of MODULE;
end




# Typically, Libraries contain several organizational MODULEs, each of which
# contain .c and possibly .h files.

MODULE :: superclass ENTITY;
    archive_status : (Archived,NotArchived) = NotArchived;
    afile : binary = ".a";
    cfiles : set_of CFILE;
```

```
end



# FILE is the generic class for anything that is represented as a unix
# file.  There are specializations (subtypes) for CFILE, HFILE and DOCFILE
# in this system.

FILE :: superclass ENTITY;
    name : string ;
    owner : user;
    timestamp : time;
    reservation_status : (CheckedOut,Available,Error) = Available;
    version : text;
    contents : text;
end


# Extra information is needed to record the state of compilation and
# analysis (lint, in our case) for CFILEs.

CFILE :: superclass FILE;
    compile_status : (Compiled,NotCompiled,Error) = Error;
    analyze_status : (Analyzed,NotAnalyzed,Error) = Error;

    contents : text = ".c";
    object_code : binary = ".o";
    ref : set_of link HFILE;
    uses : set_of link CFILE;
end



# For HFILEs, we only want to know if they have been modified recently,
# which will cause a global recompilation.

HFILE :: superclass FILE;
    recompile_mod : (Yes,No) = No;
    contents : text = ".h";
end



# For DOCFILEs, we only want to know if they have been reformatted recently,
# so we can reformat the document.

DOCFILE :: superclass FILE;
    reformat_doc : (Yes,No) = No;
    spellfile    : text = ".spl";
end


# DOC is a class that represents an entire set of documents, typically for
# a PROJECT or PROGRAM.  A DOC can contain individual documents, and files
# of it's own.

DOC :: superclass ENTITY;
    name : string;
    files : set_of DOCFILE;
    documents : set_of DOCUMENT;
end

# DOCUMENT represents a complete individual document, such as a user's manual
# or technical report.

DOCUMENT :: superclass ENTITY;
    name : string;
    type : (LaTex, Scribe, Troff, Text, ps, Unknown) = Unknown;
    header   : DOCFILE;
    docfiles : set_of DOCFILE;
    assembled: text;
    formatted: text = ".ps";
end


# INC represents a set of include (.h) files.

INC :: superclass ENTITY;
    name : string;
    archive_status : (Archived,NotArchived) = NotArchived;
    hfiles : set_of HFILE;
end



# BIN represents a place where binaries for PROGRAMs (parts of a PROJECT) are
# kept.

BIN :: superclass ENTITY;
    name : string;
    executable : binary;
end

end_objectbase
```

The following are all envelopes in the Shell Envelope Language:

```
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# assemble envelope
#

ENVELOPE assemble;

SHELL sh;

INPUT
   set_of text: header;
   set_of text: files;
   text: outfile;

OUTPUT
    none;

BEGIN

cat $header $files > $outfile

RETURN "0";

END
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# build envelope
#

ENVELOPE build;

SHELL sh;

INPUT
   set_of binary: ofiles;
   binary: execfile;

OUTPUT
    none;

BEGIN

cc $ofiles -o $execfile

if [ $? -ne 0 ]
then
    echo "Build failed."
    RETURN "1";
fi

RETURN "0";

END
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# compile envelope
#
# usage: compile [CFILE]
#
ENVELOPE

SHELL ksh;

INPUT
        text         : thefile;
        binary       : thebinary;
        set_of HFILE : hfiles;
OUTPUT
   none;

BEGIN

thedir=`dirname $thefile`

echo "$0 $thefile on `date`"
echo

echo "$0 $1 on `date`"
echo

# we need to make the -I list

tmp_dir=/tmp/compile$$
```

```
mkdir $tmp_dir

# Now the header files

for i in $hfiles
do
  ln -s $i $tmp_dir
done

echo "cc -g -c -ll -lc -lm -lX11 -I$tmp_dir -in: $thefile -out:$thebinary"
cc -g -c -I$tmp_dir $thefile -ll -lc -lm -lX11 -o $thebinary
CCSTATUS=$?

if [ "x$tmp_dir" != "x" ]
then
   rm -r $tmp_dir
fi

if [ $CCSTATUS -eq 0 ]
then
    echo compile successful
    RETURN "0";
else
    echo compile failed
    RETURN "1";
fi

END
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# editor envelope
#
# usage: edit [FILE]
#
# This edits the chosen file, and sends along the library which has
# power over it so that emacs will read in the TAGS file associated
# with it.  This also incorporates a simple locking mechanism by
# making the file writable when it edits it, and removing this capability
# when it leaves.

ENVELOPE

SHELL sh;

INPUT
    text : thefile;

OUTPUT

;

BEGIN

echo
echo Editing source file in $thefile ...

FILENAME=`basename $thefile`

Created="YES"
SaveReport=`ls -l $thefile`
if [ $? -eq 0 ]
then
```

```
   Created="NO"
fi


# Edit the file.  Check to make sure on an X Terminal.
#


if [ "x$EDITOR" = "x" ]
then
     vi $thefile
else
     emacs -fn 9x15 -geometry 80x55 $thefile
fi


# Check to make sure that the file really existed.
#
if [ $Created = "YES" ]
then
   echo "File $FILENAME Created."
   exit 0
fi

NewReport=`ls -l $thefile`

if [ "$SaveReport" = "$NewReport" ]
then
   echo "No Changes Made"
   exit 1
else
   echo "Changes Made and saved."
   exit 0
fi

END
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# execute envelope
#

ENVELOPE execute;

SHELL sh;

INPUT
   text: executable;

OUTPUT
    none;

BEGIN

echo "Executing $executable."

$executable

RETURN "0";

END
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# format envelope
#

ENVELOPE format;

SHELL sh;

INPUT
   text: input;
   text: output;

OUTPUT
    none;

BEGIN

echo "Formatting $input into $output."

scribe $input -remote
mv $input.ps $output

RETURN "0";

END
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# print envelope
#

ENVELOPE print;

SHELL sh;

INPUT
   text: file;

OUTPUT
    none;

BEGIN

echo "Printing $file."

if [ "x$PRINTER" = "x" ]
then
    PRINTER=copyrm
    export PRINTER
fi
lpr $file

RETURN "0";

END
#                    Marvel Software Development Environment
#
```

```
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# proof envelope
#
# usage: proof [FILE]
#
# This edits the chosen file, and sends along the library which has
# power over it so that emacs will read in the TAGS file associated
# with it.  This also incorporates a simple locking mechanism by
# making the file writable when it edits it, and removing this capability
# when it leaves.

ENVELOPE

SHELL sh;

INPUT
    text : thefile;
    text : spellfile;

OUTPUT
    none;

BEGIN

echo
echo Proofing file in $thefile ...

FILENAME=`basename $spellfile`

spell $thefile > $spellfile
if [ $? -ne 0 ]
then
  RETURN "1";
fi

if [ "x$EDITOR" = "x" ]
then
     vi $thefile $spellfile
else

  echo "(switch-to-buffer-other-window \"$FILENAME\")" > /tmp/proof.el
  echo "(other-window 1)" >> /tmp/proof.el
  emacs -fn 9x15  -geometry 80x55 $spellfile $thefile -l /tmp/proof.el
  rm /tmp/proof.el
fi

RETURN "0";

END
```

The following is the MSL output of the ASL translator:

```
strategy test

imports data_model;
exports all;

objectbase

ASSIGN :: superclass TOOL;
  get_user: string = get_user;
end

FORMATTER :: superclass TOOL;
  format: string = format;
end

PRINTER :: superclass TOOL;
  spool: string = print;
end

DEBUGGER :: superclass TOOL;
  exec: string = execute;
end

BUILD :: superclass TOOL;
  build_program: string = build;
end

EDITOR :: superclass TOOL;
  editor: string = editor;
  proof: string = proof;
end

COMPILER :: superclass TOOL;
  compile: string = compile;
end

ASSEMBLE :: superclass TOOL;
  assemble: string = assemble;
end

SEND :: superclass TOOL;
  send_waiting: string = send_waiting;
end

RECEIVE :: superclass TOOL;
  recv_wait: string = recv_wait;
end

ACTIVITY_STRUCTURE :: superclass ENTITY;
  channel4_send: (Ready,Waiting,Done) = Done;
  channel2_receive: (Ready,Waiting,Done) = Done;
  channel3_send: (Ready,Waiting,Done) = Done;
  channel1_receive: (Ready,Waiting,Done) = Done;
  channel4_receive: (Ready,Waiting,Done) = Done;
  channel2_send: (Ready,Waiting,Done) = Done;
  channel3_receive: (Ready,Waiting,Done) = Done;
  channel1_send: (Ready,Waiting,Done) = Done;
  active: (Inactive,Active,Done,Terminated) = Inactive;
  state0: (Ready,Done,Inactive) = Ready;
  as_string: string;
  owner: user;
  clientID: clientid;
  as_lock: (Shared,Exclusive,None) = None;
  lock_counter: integer = 0;
  channel1_semaphore: integer = 0;
  channel1_waiting: integer = 0;
  channel3_semaphore: integer = 0;
  channel3_waiting: integer = 0;
  channel2_semaphore: integer = 0;
  channel2_waiting: integer = 0;
  channel4_semaphore: integer = 0;
  channel4_waiting: integer = 0;
end

test2 :: superclass ACTIVITY_STRUCTURE;
  state1: (Ready,Done,Inactive) = Inactive;
  state2: (Ready,Done,Inactive) = Inactive;
  as_string: string =
 "(((edit[multiset ?c:CFILE] | edit[multiset ?h:HFILE]);
(compile[multiuse ?c:CFILE])*)*; build[set ?p:PROGRAM];
(exec_prog[set ?p:PROGRAM] | epsilon))*";
  p: link PROGRAM;
  h: set_of link HFILE;
  c: set_of link CFILE;
end

test4 :: superclass ACTIVITY_STRUCTURE;
  state1: (Ready,Done,Inactive) = Inactive;
  state2: (Ready,Done,Inactive) = Inactive;
  as_string: string =
"&(&((edit[reset ?a:DOCFILE]; (Send_1; Receive_3;
(edit[reset ?a:DOCFILE] | epsilon))*),
(edit[reset ?b:DOCFILE]; (Send_2; Receive_4;
(edit[reset ?b:DOCFILE] | epsilon))*));
assemble[set ?doc:DOCUMENT]; format[set ?doc:DOCUMENT];
printdoc[set ?doc:DOCUMENT], ((Receive_1;
proof[use ?a:DOCFILE]; Send_3) | (Receive_2;
proof[use ?b:DOCFILE]; Send_4))*)";
  children1: set_of ACTIVITY_STRUCTURE;
  a: link DOCFILE;
  b: link DOCFILE;
  doc: link DOCUMENT;
end

AS_0 :: superclass ACTIVITY_STRUCTURE;
  state0: (Ready,Done,Inactive) = Inactive;
  state1: (Ready,Done,Inactive) = Inactive;
  state2: (Ready,Done,Inactive) = Inactive;
  state3: (Ready,Done,Inactive) = Inactive;
  state4: (Ready,Done,Inactive) = Inactive;
  state5: (Ready,Done,Inactive) = Inactive;
  as_string: string =
"&((edit[reset ?a:DOCFILE]; (Send_1; Receive_3;
 (edit[reset ?a:DOCFILE] | epsilon))*),
(edit[reset ?b:DOCFILE]; (Send_2; Receive_4;
(edit[reset ?b:DOCFILE] | epsilon))*));
assemble[set ?doc:DOCUMENT]; format[set ?doc:DOCUMENT];
printdoc[set ?doc:DOCUMENT]";
  children1: set_of ACTIVITY_STRUCTURE;
end

AS_1 :: superclass ACTIVITY_STRUCTURE;
  state0: (Ready,Done,Inactive) = Inactive;
  state1: (Ready,Done,Inactive) = Inactive;
  state2: (Ready,Done,Inactive) = Inactive;
  state3: (Ready,Done,Inactive) = Inactive;
  as_string: string = "(edit[reset ?a:DOCFILE];
(Send_1; Receive_3; (edit[reset ?a:DOCFILE] | epsilon))*)";
end

AS_2 :: superclass ACTIVITY_STRUCTURE;
  state0: (Ready,Done,Inactive) = Inactive;
  state1: (Ready,Done,Inactive) = Inactive;
  state2: (Ready,Done,Inactive) = Inactive;
  state3: (Ready,Done,Inactive) = Inactive;
  as_string: string = "(edit[reset ?b:DOCFILE];
(Send_2; Receive_4; (edit[reset ?b:DOCFILE] | epsilon))*)";
end

AS_3 :: superclass ACTIVITY_STRUCTURE;
  state0: (Ready,Done,Inactive) = Inactive;
  state1: (Ready,Done,Inactive) = Inactive;
  state2: (Ready,Done,Inactive) = Inactive;
  state3: (Ready,Done,Inactive) = Inactive;
  state4: (Ready,Done,Inactive) = Inactive;
  as_string: string =
```

```
"((Receive_1; proof[use ?a:DOCFILE]; Send_3) |
(Receive_2; proof[use ?b:DOCFILE]; Send_4))*";
end

INSTANTIATE :: superclass TOOL;
  instantiate: string = instantiate;
end

end_objectbase

rules

Instantiate[]:
  :
  { INSTANTIATE instantiate }
  ;

Activate [?SO:ACTIVITY_STRUCTURE]:
  (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))
  :
  (and no_backward(?SO.owner = CurrentUser)
       no_backward(?s.clientID <> CurrentClient))
  { }
  (and no_forward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Inactive));

Deactivate [?SO:test2]:
  :
  no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state0 = Ready));

Terminate [?SO:test2]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state0 = Ready));

Deactivate [?SO:test4]:
  :
  no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state0 = Ready));

Terminate [?SO:test4]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state0 = Ready));
```

```
Deactivate [?SO:AS_0]:
  :
  no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive)
       no_chain(?SO.state4 = Inactive)
       no_chain(?SO.state5 = Inactive));

Terminate [?SO:AS_0]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive)
       no_chain(?SO.state4 = Inactive)
       no_chain(?SO.state5 = Inactive));

Deactivate [?SO:AS_1]:
  :
  no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive));

Terminate [?SO:AS_1]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive));

Deactivate [?SO:AS_2]:
  :
  no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive));

Terminate [?SO:AS_2]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive));

Deactivate [?SO:AS_3]:
  :
  no_backward(?SO.clientID = CurrentClient)
  { }
  (and no_backward(?SO.active = Inactive)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive)
       no_chain(?SO.state4 = Inactive));

Terminate [?SO:AS_3]:
  :
  (and no_backward(?SO.clientID = CurrentClient)
       no_forward(?SO.active = Done))
  { }
  (and no_backward(?SO.active = Terminated)
       no_forward(?SO.clientID = ResetClient)
       no_chain(?SO.state0 = Inactive)
       no_chain(?SO.state1 = Inactive)
       no_chain(?SO.state2 = Inactive)
       no_chain(?SO.state3 = Inactive)
       no_chain(?SO.state4 = Inactive));

Attach [?SO:ACTIVITY_STRUCTURE]:
  (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser))
  :
  (and no_chain(?SO.owner = CurrentUser)
       no_chain(?s.clientID <> CurrentClient))
  { }
  no_forward(?SO.clientID = CurrentClient);

Detach [?SO:ACTIVITY_STRUCTURE]:
  :
  no_chain(?SO.clientID = CurrentClient)
  { }
  no_forward(?SO.clientID = ResetClient);

Unbind_doc [?SO:AS_3]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.doc ?doc]);

Bind_doc [?doc:DOCUMENT, ?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.doc ?doc]);

Unbind_b [?SO:AS_3]:
  (and (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.b ?b]);

Bind_b [?b:DOCFILE, ?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.b ?b]);

Unbind_a [?SO:AS_3]:
  (and (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.a ?a]);

Bind_a [?a:DOCFILE, ?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.a ?a]);

Unbind [?SO:AS_3]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (and (unlink [?root.a ?a])
       (unlink [?root.b ?b])
       (unlink [?root.doc ?doc]));

Bind [?a:DOCFILE, ?b:DOCFILE, ?doc:DOCUMENT, ?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (and (linkto [?root.a ?a])
       (linkto [?root.b ?b])
       (linkto [?root.doc ?doc]));

Unbind_doc [?SO:AS_2]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.doc ?doc]);

Bind_doc [?doc:DOCUMENT, ?SO:AS_2]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.doc ?doc]);

Unbind_b [?SO:AS_2]:
  (and (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.b ?b]);

Bind_b [?b:DOCFILE, ?SO:AS_2]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.b ?b]);

Unbind_a [?SO:AS_2]:
  (and (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.a ?a]);

Bind_a [?a:DOCFILE, ?SO:AS_2]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.a ?a]);

Unbind [?SO:AS_2]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (and (unlink [?root.a ?a])
       (unlink [?root.b ?b])
       (unlink [?root.doc ?doc]));

Bind [?a:DOCFILE, ?b:DOCFILE, ?doc:DOCUMENT, ?SO:AS_2]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (and (linkto [?root.a ?a])
       (linkto [?root.b ?b])
       (linkto [?root.doc ?doc]));

Unbind_doc [?SO:AS_1]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.doc ?doc]);

Bind_doc [?doc:DOCUMENT, ?SO:AS_1]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.doc ?doc]);

Unbind_b [?SO:AS_1]:
  (and (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.b ?b]);

Bind_b [?b:DOCFILE, ?SO:AS_1]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.b ?b]);

Unbind_a [?SO:AS_1]:
  (and (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.a ?a]);

Bind_a [?a:DOCFILE, ?SO:AS_1]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.a ?a]);

Unbind [?SO:AS_1]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (and (unlink [?root.a ?a])
       (unlink [?root.b ?b])
       (unlink [?root.doc ?doc]));

Bind [?a:DOCFILE, ?b:DOCFILE, ?doc:DOCUMENT, ?SO:AS_1]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (and (linkto [?root.a ?a])
       (linkto [?root.b ?b])
       (linkto [?root.doc ?doc]));

Unbind_doc [?SO:AS_0]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.doc ?doc]);

Bind_doc [?doc:DOCUMENT, ?SO:AS_0]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.doc ?doc]);

Unbind_b [?SO:AS_0]:
  (and (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.b ?b]);

Bind_b [?b:DOCFILE, ?SO:AS_0]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.b ?b]);

Unbind_a [?SO:AS_0]:
  (and (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (unlink [?root.a ?a]);

Bind_a [?a:DOCFILE, ?SO:AS_0]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (linkto [?root.a ?a]);

Unbind [?SO:AS_0]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists test4 ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (and (unlink [?root.a ?a])
       (unlink [?root.b ?b])
       (unlink [?root.doc ?doc]));
```

```
Bind [?a:DOCFILE, ?b:DOCFILE, ?doc:DOCUMENT, ?SO:AS_0]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (and (linkto [?root.a ?a])
       (linkto [?root.b ?b])
       (linkto [?root.doc ?doc]));

Unbind_doc [?SO:test4]:
  (exists DOCUMENT ?doc suchthat (linkto [?SO.doc ?doc]))
  :
  { }
  (unlink [?SO.doc ?doc]);

Bind_doc [?doc:DOCUMENT, ?SO:test4]:
  :
  { }
  (linkto [?SO.doc ?doc]);

Unbind_b [?SO:test4]:
  (exists DOCFILE ?b suchthat (linkto [?SO.b ?b]))
  :
  { }
  (unlink [?SO.b ?b]);

Bind_b [?b:DOCFILE, ?SO:test4]:
  :
  { }
  (linkto [?SO.b ?b]);

Unbind_a [?SO:test4]:
  (exists DOCFILE ?a suchthat (linkto [?SO.a ?a]))
  :
  { }
  (unlink [?SO.a ?a]);

Bind_a [?a:DOCFILE, ?SO:test4]:
  :
  { }
  (linkto [?SO.a ?a]);

Unbind [?SO:test4]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?SO.doc ?doc]))
       (exists DOCFILE ?b suchthat (linkto [?SO.b ?b]))
       (exists DOCFILE ?a suchthat (linkto [?SO.a ?a])))
  :
  { }
  (and (unlink [?SO.a ?a])
       (unlink [?SO.b ?b])
       (unlink [?SO.doc ?doc]));

Bind [?a:DOCFILE, ?b:DOCFILE, ?doc:DOCUMENT, ?SO:test4]:
  :
  { }
  (and (linkto [?SO.a ?a])
       (linkto [?SO.b ?b])
       (linkto [?SO.doc ?doc]));

Unbind_c [?SO:test2]:
  (exists CFILE ?c suchthat (linkto [?SO.c ?c]))
  :
  { }
  (unlink [?SO.c ?c]);

Bind_c [?c:CFILE, ?SO:test2]:
  :
  { }
  (linkto [?SO.c ?c]);

Unbind_h [?SO:test2]:
  (exists HFILE ?h suchthat (linkto [?SO.h ?h]))
  :
  { }
  (unlink [?SO.h ?h]);

Bind_h [?h:HFILE, ?SO:test2]:
  :
  { }
  (linkto [?SO.h ?h]);

Unbind_p [?SO:test2]:
  (exists PROGRAM ?p suchthat (linkto [?SO.p ?p]))
  :
  { }
  (unlink [?SO.p ?p]);

Bind_p [?p:PROGRAM, ?SO:test2]:
  :
  { }
  (linkto [?SO.p ?p]);

Unbind [?SO:test2]:
  (and (exists CFILE ?c suchthat (linkto [?SO.c ?c]))
       (exists HFILE ?h suchthat (linkto [?SO.h ?h]))
       (exists PROGRAM ?p suchthat (linkto [?SO.p ?p])))
  :
  { }
  (and (unlink [?SO.p ?p])
       (unlink [?SO.h ?h])
       (unlink [?SO.c ?c]));

Bind [?p:PROGRAM, ?h:HFILE, ?c:CFILE, ?SO:test2]:
  :
  { }
  (and (linkto [?SO.p ?p])
       (linkto [?SO.h ?h])
       (linkto [?SO.c ?c]));

hide after_receive_4 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and (or no_backward(?child.channel4_receive = Ready)
           no_backward(?child.channel4_receive = Waiting))
       no_chain(?root.channel4_semaphore >= 0))
  { }
  no_chain(?child.channel4_receive = Done);

hide dont_receive_4 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel4_receive = Waiting)
  { RECEIVE recv_wait "4" }
  no_chain(?root.channel4_waiting += 1);

hide do_receive_4 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel4_receive = Ready)
  { }
  no_backward(?root.channel4_semaphore -= 1);

hide after_send_4 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and no_backward(?child.channel4_send = Ready)
       no_backward(?root.channel4_semaphore > 0))
  { }
  no_chain(?child.channel4_send = Done);

hide do_send_4 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel4_send = Ready)
  { SEND send_waiting "4" }
  (and no_backward(?root.channel4_semaphore += 1)
       no_chain(?root.channel4_waiting = 0));

hide after_receive_3 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and (or no_backward(?child.channel3_receive = Ready)
           no_backward(?child.channel3_receive = Waiting))
       no_chain(?root.channel3_semaphore >= 0))
  { }
  no_chain(?child.channel3_receive = Done);

hide dont_receive_3 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel3_receive = Waiting)
  { RECEIVE recv_wait "3" }
  no_chain(?root.channel3_waiting += 1);

hide do_receive_3 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel3_receive = Ready)
  { }
  no_backward(?root.channel3_semaphore -= 1);

hide after_send_3 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and no_backward(?child.channel3_send = Ready)
       no_backward(?root.channel3_semaphore > 0))
  { }
  no_chain(?child.channel3_send = Done);

hide do_send_3 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel3_send = Ready)
  { SEND send_waiting "3" }
  (and no_backward(?root.channel3_semaphore += 1)
       no_chain(?root.channel3_waiting = 0));

hide after_receive_2 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and (or no_backward(?child.channel2_receive = Ready)
           no_backward(?child.channel2_receive = Waiting))
       no_chain(?root.channel2_semaphore >= 0))
  { }
  no_chain(?child.channel2_receive = Done);

hide dont_receive_2 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel2_receive = Waiting)
  { RECEIVE recv_wait "2" }
  no_chain(?root.channel2_waiting += 1);
```

```
hide do_receive_2 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel2_receive = Ready)
  { }
  no_backward(?root.channel2_semaphore -= 1);

hide after_send_2 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and no_backward(?child.channel2_send = Ready)
       no_backward(?root.channel2_semaphore > 0))
  { }
  no_chain(?child.channel2_send = Done);

hide do_send_2 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel2_send = Ready)
  { SEND send_waiting "2" }
  (and no_backward(?root.channel2_semaphore += 1)
       no_chain(?root.channel2_waiting = 0));

hide after_receive_1 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and (or no_backward(?child.channel1_receive = Ready)
           no_backward(?child.channel1_receive = Waiting))
       no_chain(?root.channel1_semaphore >= 0))
  { }
  no_chain(?child.channel1_receive = Done);

hide dont_receive_1 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel1_receive = Waiting)
  { RECEIVE recv_wait "1" }
  no_chain(?root.channel1_waiting += 1);

hide do_receive_1 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel1_receive = Ready)
  { }
  no_backward(?root.channel1_semaphore -= 1);

hide after_send_1 [?child:ACTIVITY_STRUCTURE]:
  (exists test4 ?root suchthat (ancestor [?root ?child]))
  :
  (and no_backward(?child.channel1_send = Ready)
       no_backward(?root.channel1_semaphore > 0))
  { }
  no_chain(?child.channel1_send = Done);

hide do_send_1 [?root:test4]:
  (exists ACTIVITY_STRUCTURE ?child suchthat (ancestor [?root ?child]))
  :
  no_backward(?child.channel1_send = Ready)
  { SEND send_waiting "1" }
  (and no_backward(?root.channel1_semaphore += 1)
       no_chain(?root.channel1_waiting = 0));

hide finish_11 [?SO:test4]:
  :
  (?SO.state2 = Inactive)
  { }
  no_forward(?SO.active = Inactive);

hide finish_10 [?SO:test4]:
  :
  (?SO.state2 = Ready)
  { }
  no_forward(?SO.active = Done);

Assign [?child:AS_3, ?SO:test4]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

Assign [?child:AS_0, ?SO:test4]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

hide all_terminated [?SO:test4]:
  (forall ACTIVITY_STRUCTURE ?child suchthat (member [?SO.children1 ?child]))
  :
  (and (?child.active = Terminated)
       (?SO.state1 = Ready))
  { }
  (and (?SO.active = Inactive)
       (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

hide all_assigned [?SO:test4]:
  (forall ACTIVITY_STRUCTURE ?child suchthat (member [?SO.children1 ?child]))
  :
  (and (?child.owner <> ResetUser)
       (?SO.state0 = Ready))
  { }
  (and (?SO.active = Active)
       (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

hide finish_9 [?SO:AS_3]:
  :
  (?SO.state0 = Inactive)
  { }
  no_forward(?SO.active = Inactive);

hide finish_8 [?SO:AS_3]:
  :
  (?SO.state0 = Ready)
  { }
  no_forward(?SO.active = Done);

Send_3 [?SO:AS_3]:
  :
  no_forward(?SO.state4 = Ready)
  { }
  (and (?SO.channel3_send = Ready)
       (?SO.state4 = Inactive)
       (?SO.state0 = Ready));

proof [?use_a:DOCFILE, ?SO:AS_3]:
  :
  no_forward(?SO.state3 = Ready)
  { EDITOR proof ?use_a.contents ?use_a.spellfile }
  (and (?SO.state3 = Inactive)
       (?SO.state4 = Ready));

Receive_1 [?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state0 = Ready)
       no_chain(?root.channel1_semaphore = 0))
  { }
  no_backward(?SO.channel1_receive = Waiting);

Receive_1 [?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state0 = Ready)
       no_chain(?root.channel1_semaphore > 0))
  { }
  (and no_backward(?SO.channel1_receive = Ready)
       (?SO.state0 = Inactive)
       no_backward(?SO.state3 = Ready));

Send_4 [?SO:AS_3]:
  :
  no_forward(?SO.state2 = Ready)
  { }
  (and (?SO.channel4_send = Ready)
       (?SO.state2 = Inactive)
       (?SO.state0 = Ready));

proof [?use_b:DOCFILE, ?SO:AS_3]:
  :
  no_forward(?SO.state1 = Ready)
  { EDITOR proof ?use_b.contents ?use_b.spellfile }
  (and (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

Receive_2 [?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state0 = Ready)
       no_chain(?root.channel2_semaphore = 0))
  { }
  no_backward(?SO.channel2_receive = Waiting);

Receive_2 [?SO:AS_3]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state0 = Ready)
       no_chain(?root.channel2_semaphore > 0))
  { }
  (and no_backward(?SO.channel2_receive = Ready)
       (?SO.state0 = Inactive)
       no_backward(?SO.state1 = Ready));

Activate [?SO:AS_3]:
  (and (exists test4 ?parent suchthat (member [?parent.children1 ?SO]))
       (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser)))
  :
  (and (or no_forward(?SO.active = Inactive)
           no_forward(?SO.active = Terminated))
       no_forward(?SO.clientID = ResetClient)
       no_forward(?SO.owner = CurrentUser)
       no_forward(?parent.active = Active)
       no_forward(?s.clientID <> CurrentClient))
  { }
  (and no_backward(?SO.clientID = CurrentClient)
       no_backward(?SO.state0 = Ready));

hide finish_7 [?SO:AS_0]:
  :
  (?SO.state5 = Inactive)
  { }
  no_forward(?SO.active = Inactive);

hide finish_6 [?SO:AS_0]:
  :
  (?SO.state5 = Ready)
  { }
  no_forward(?SO.active = Done);

Assign [?child:AS_2, ?SO:AS_0]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

Assign [?child:AS_1, ?SO:AS_0]:
  :
  no_forward(?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  no_backward(?child.owner = ?uA);

printdoc [?set_doc:DOCUMENT, ?SO:AS_0]:
  :
  no_forward(?SO.state4 = Ready)
  { PRINTER spool ?set_doc.formatted }
  (and (?SO.state4 = Inactive)
       (?SO.state5 = Ready));

format [?set_doc:DOCUMENT, ?SO:AS_0]:
  :
  no_forward(?SO.state3 = Ready)
  { FORMATTER format ?set_doc.assembled ?set_doc.formatted }
  (and (?SO.state3 = Inactive)
       (?SO.state4 = Ready));

assemble [?set_doc:DOCUMENT, ?SO:AS_0]:
  (and (forall DOCFILE ?Docf suchthat (member [?set_doc.docfiles ?Docf]))
       (exists DOCFILE ?head suchthat (member [?set_doc.header ?head])))
  :
  no_forward(?SO.state2 = Ready)
  { ASSEMBLE assemble ?head.contents ?Docf.contents ?set_doc.assembled }
  (and (?SO.state2 = Inactive)
       (?SO.state3 = Ready));

hide all_terminated [?SO:AS_0]:
  (forall ACTIVITY_STRUCTURE ?child suchthat (member [?SO.children1 ?child]))
  :
  (and (?child.active = Terminated)
       (?SO.state1 = Ready))
  { }
  (and (?SO.active = Inactive)
       (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

hide all_assigned [?SO:AS_0]:
  (forall ACTIVITY_STRUCTURE ?child suchthat (member [?SO.children1 ?child]))
  :
  (and (?child.owner <> ResetUser)
       (?SO.state0 = Ready))
  { }
  (and (?SO.active = Active)
       (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

Activate [?SO:AS_0]:
  (and (exists test4 ?parent suchthat (member [?parent.children1 ?SO]))
       (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser)))
  :
```

```
  (and (or no_forward(?SO.active = Inactive)
           no_forward(?SO.active = Terminated))
       no_forward(?SO.clientID = ResetClient)
       no_forward(?SO.owner = CurrentUser)
       no_forward(?parent.active = Active)
       no_forward(?s.clientID <> CurrentClient))
  { }
  (and no_backward(?SO.clientID = CurrentClient)
       no_backward(?SO.state0 = Ready));

hide finish_5 [?SO:AS_2]:
  :
  (and (?SO.state1 = Inactive)
       (?SO.state3 = Inactive))
  { }
  no_forward(?SO.active = Inactive);

hide finish_4 [?SO:AS_2]:
  :
  (or (?SO.state1 = Ready)
      (?SO.state3 = Ready))
  { }
  no_forward(?SO.active = Done);

Send_2 [?SO:AS_2]:
  :
  no_forward(?SO.state3 = Ready)
  { }
  (and (?SO.channel2_send = Ready)
       (?SO.state3 = Inactive)
       (?SO.state2 = Ready));

edit [?reset_b:DOCFILE, ?SO:AS_2]:
  :
  no_forward(?SO.state3 = Ready)
  { EDITOR editor ?reset_b.contents }
  (and (?SO.state3 = Inactive)
       (?SO.state1 = Ready));

Receive_4 [?SO:AS_2]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state2 = Ready)
       no_chain(?root.channel4_semaphore = 0))
  { }
  no_backward(?SO.channel4_receive = Waiting);

Receive_4 [?SO:AS_2]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state2 = Ready)
       no_chain(?root.channel4_semaphore > 0))
  { }
  (and no_backward(?SO.channel4_receive = Ready)
       (?SO.state2 = Inactive)
       no_backward(?SO.state3 = Ready));

Send_2 [?SO:AS_2]:
  :
  no_forward(?SO.state1 = Ready)
  { }
  (and (?SO.channel2_send = Ready)
       (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

edit [?reset_b:DOCFILE, ?SO:AS_2]:
  :
```

```
  no_forward(?SO.state0 = Ready)
  { EDITOR editor ?reset_b.contents }
  (and (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

Activate [?SO:AS_2]:
  (and (exists AS_0 ?parent suchthat (member [?parent.children1 ?SO]))
       (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser)))
  :
  (and (or no_forward(?SO.active = Inactive)
           no_forward(?SO.active = Terminated))
       no_forward(?SO.clientID = ResetClient)
       no_forward(?SO.owner = CurrentUser)
       no_forward(?parent.active = Active)
       no_forward(?s.clientID <> CurrentClient))
  { }
  (and no_backward(?SO.clientID = CurrentClient)
       no_backward(?SO.state0 = Ready));

hide finish_3 [?SO:AS_1]:
  :
  (and (?SO.state1 = Inactive)
       (?SO.state3 = Inactive))
  { }
  no_forward(?SO.active = Inactive);

hide finish_2 [?SO:AS_1]:
  :
  (or (?SO.state1 = Ready)
      (?SO.state3 = Ready))
  { }
  no_forward(?SO.active = Done);

Send_1 [?SO:AS_1]:
  :
  no_forward(?SO.state3 = Ready)
  { }
  (and (?SO.channel1_send = Ready)
       (?SO.state3 = Inactive)
       (?SO.state2 = Ready));

edit [?reset_a:DOCFILE, ?SO:AS_1]:
  :
  no_forward(?SO.state3 = Ready)
  { EDITOR editor ?reset_a.contents }
  (and (?SO.state3 = Inactive)
       (?SO.state1 = Ready));

Receive_3 [?SO:AS_1]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state2 = Ready)
       no_chain(?root.channel3_semaphore = 0))
  { }
  no_backward(?SO.channel3_receive = Waiting);

Receive_3 [?SO:AS_1]:
  (exists test4 ?root suchthat (ancestor [?root ?SO]))
  :
  (and no_chain(?SO.state2 = Ready)
       no_chain(?root.channel3_semaphore > 0))
  { }
  (and no_backward(?SO.channel3_receive = Ready)
       (?SO.state2 = Inactive)
       no_backward(?SO.state3 = Ready));

Send_1 [?SO:AS_1]:
  :
  no_forward(?SO.state1 = Ready)
  { }
  (and (?SO.channel1_send = Ready)
       (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

edit [?reset_a:DOCFILE, ?SO:AS_1]:
  :
  no_forward(?SO.state0 = Ready)
  { EDITOR editor ?reset_a.contents }
  (and (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

Activate [?SO:AS_1]:
  (and (exists AS_0 ?parent suchthat (member [?parent.children1 ?SO]))
       (forall ACTIVITY_STRUCTURE ?s suchthat (?s.as_string <> ResetUser)))
  :
  (and (or no_forward(?SO.active = Inactive)
           no_forward(?SO.active = Terminated))
       no_forward(?SO.clientID = ResetClient)
       no_forward(?SO.owner = CurrentUser)
       no_forward(?parent.active = Active)
       no_forward(?s.clientID <> CurrentClient))
  { }
  (and no_backward(?SO.clientID = CurrentClient)
       no_backward(?SO.state0 = Ready));

Assign [?SO:test4]:
  :
  { ASSIGN get_user return ?uA }
  no_backward(?SO.owner = ?uA);

hide finish_1 [?SO:test2]:
  :
  (and (?SO.state0 = Inactive)
       (?SO.state1 = Inactive))
  { }
  no_forward(?SO.active = Inactive);

hide finish_0 [?SO:test2]:
  :
  (or (?SO.state0 = Ready)
      (?SO.state1 = Ready))
  { }
  no_forward(?SO.active = Done);

edit [?multiset_c:CFILE, ?SO:test2]:
  :
  no_forward(?SO.state0 = Ready)
  { EDITOR editor ?multiset_c.contents }
  (and (?multiset_c.compile_status = NotCompiled)
       (?multiset_c.timestamp = CurrentTime)
       (?SO.state0 = Inactive)
       (?SO.state2 = Ready));

edit [?multiset_h:HFILE, ?SO:test2]:
  :
  no_forward(?SO.state0 = Ready)
  { EDITOR editor ?multiset_h.contents }
  (and (?SO.state0 = Inactive)
       (?SO.state2 = Ready));

edit [?multiset_c:CFILE, ?SO:test2]:
  :
  no_forward(?SO.state1 = Ready)
  { EDITOR editor ?multiset_c.contents }
  (and (?multiset_c.compile_status = NotCompiled)
       (?multiset_c.timestamp = CurrentTime)
       (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

edit [?multiset_c:CFILE, ?SO:test2]:
  :
  no_forward(?SO.state2 = Ready)
  { EDITOR editor ?multiset_c.contents }
  (and (?multiset_c.compile_status = NotCompiled)
       (?multiset_c.timestamp = CurrentTime));

edit [?multiset_h:HFILE, ?SO:test2]:
  :
  no_forward(?SO.state2 = Ready)
  { EDITOR editor ?multiset_h.contents }
   ;

compile [?multiuse_c:CFILE, ?SO:test2]:
  (forall HFILE ?h suchthat (linkto [?multiuse_c.ref ?h]))
  :
  (and (or (?multiuse_c.compile_status = NotCompiled)
           (?multiuse_c.compile_status = Error))
       no_forward(?SO.state2 = Ready))
  { COMPILER compile ?multiuse_c.contents ?multiuse_c.object_code ?h.contents }
  (?multiuse_c.compile_status = Compiled);
  [?multiuse_c.compile_status = Error];

build [?set_p:PROGRAM, ?SO:test2]:
  (and (forall MODULE ?m suchthat (member [?set_p.modules ?m]))
       (forall CFILE ?c suchthat (member [?m.cfiles ?c])))
  :
  (and (?c.compile_status = Compiled)
       no_forward(?SO.state2 = Ready))
  { BUILD build_program ?c.object_code ?set_p.exec }
  (and (?set_p.build_status = Built)
       (?SO.state2 = Inactive)
       (?SO.state1 = Ready));
  (and (?set_p.build_status = NotBuilt)
       (?SO.state2 = Inactive)
       (?SO.state1 = Ready));

edit [?multiset_h:HFILE, ?SO:test2]:
  :
  no_forward(?SO.state1 = Ready)
  { EDITOR editor ?multiset_h.contents }
  (and (?SO.state1 = Inactive)
       (?SO.state2 = Ready));

build [?set_p:PROGRAM, ?SO:test2]:
  (and (forall MODULE ?m suchthat (member [?set_p.modules ?m]))
       (forall CFILE ?c suchthat (member [?m.cfiles ?c])))
  :
  (and (?c.compile_status = Compiled)
       no_forward(?SO.state1 = Ready))
  { BUILD build_program ?c.object_code ?set_p.exec }
  (?set_p.build_status = Built);
  (?set_p.build_status = NotBuilt);

exec_prog [?set_p:PROGRAM, ?SO:test2]:
  :
  (and (?set_p.build_status = Built)
       no_forward(?SO.state1 = Ready))
  { DEBUGGER exec ?set_p.exec }
  (and (?SO.state1 = Inactive)
       (?SO.state0 = Ready));
```

```
build [?set_p:PROGRAM, ?SO:test2]:
  (and (forall MODULE ?m suchthat (member [?set_p.modules ?m]))
       (forall CFILE ?c suchthat (member [?m.cfiles ?c])))
  :
  (and (?c.compile_status = Compiled)
       no_forward(?SO.state0 = Ready))
  { BUILD build_program ?c.object_code ?set_p.exec }
  (and (?set_p.build_status = Built)
       (?SO.state0 = Inactive)
       (?SO.state1 = Ready));
  (and (?set_p.build_status = NotBuilt)
       (?SO.state0 = Inactive)
       (?SO.state1 = Ready));

Assign [?SO:test2]:
  :
  { ASSIGN get_user return ?uA }
  no_backward(?SO.owner = ?uA);
```

The translator also generates two .gen files:

```
#!/bin/sh
#
echo "#!marvel script" > test2.marvelrc
ObjName=`get_as_obj_name test2`
#
# Top Level Instance
#
echo "add -hi as $ObjName test2" >> test2.marvelrc
exit 0
```

```
#!/bin/sh
#
echo "#!marvel script" > test4.marvelrc
ObjName=`get_as_obj_name test4`
#
# Top Level Instance
#
echo "add -hi as $ObjName test4" >> test4.marvelrc
ObjName=`get_as_obj_name AS_0`
#
# -- begin set
#
echo " add -hi children1 $ObjName AS_0" >> test4.marvelrc
ObjName=`get_as_obj_name AS_1`
#
# -- begin set
#
echo " add -hi children1 $ObjName AS_1" >> test4.marvelrc
ObjName=`get_as_obj_name AS_2`
echo "change -up" >> test4.marvelrc
echo "add -hi children1 $ObjName AS_2" >> test4.marvelrc
#
# -- end set
#
echo "change -up" >> test4.marvelrc
#
# Changing Back to $ObjName
#
echo "change -up" >> test4.marvelrc
ObjName=`get_as_obj_name AS_3`
#
# -- begin set
#
echo " add -hi children1 $ObjName AS_3" >> test4.marvelrc
#
# -- end set
#
echo "change -up" >> test4.marvelrc
exit 0
```

This is the instantiate shell script:

```
#!/bin/sh

echo "Enter name of Activity Structure Class "
read ASclass

if [ "x$ASclass" = "x" ]
then
    echo "Must pick a class name"
    exit 1
fi

$ASclass.gen $ASclass

if [ $? -ne 0 ]
then
    echo "Script generator failed."
    exit 1
fi

marvel -b $ASclass.marvelrc

rm $ASclass.marvelrc

exit 0
```

Finally, these three Shell Envelope Language envelopes are used in every ASL environment:

```
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# assign envelope
#

ENVELOPE pu;

SHELL sh;

INPUT
   none;

OUTPUT
    string : ret_string;

BEGIN

echo "Enter userid to be assigned:"
read ret_string

if [ "x$ret_string" = "x" ]
then
    echo "Must specify a user id"
    RETURN "1" : "" ;
fi

RETURN "0" : $ret_string;

END
#                    Marvel Software Development Environment
```

```
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# recv_wait envelope
#

ENVELOPE pu;

SHELL sh;

INPUT
   int: channelNum;

OUTPUT
    none;

BEGIN

if [ ! -f channel$channelNum.wait ]
then
    touch channel$channelNum.wait
fi

grep -s "^$LOGNAME\$" channel$channelNum.wait

if [ $? != 0 ]
then
    echo $LOGNAME >> channel$channelNum.wait
fi

RETURN "0";

END
```

```
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# send_waiting envelope
#

ENVELOPE pu;

SHELL sh;

INPUT
   int: channelNum;

OUTPUT
    none;

BEGIN

if [ ! -r channel$channelNum.wait ]
then
    RETURN "0";
fi

echo -n "There is a message available on " > /tmp/asl-channel$channelNum.tmp
echo "channel $channelNum in the" >> /tmp/asl-channel$channelNum.tmp
echo "$PWD Marvel/ASL database." >> /tmp/asl-channel$channelNum.tmp
echo -n "You may receive this message "  >> /tmp/asl-channel$channelNum.tmp
echo -n "now, unless another user receives " >> /tmp/asl-channel$channelNum.tmp
echo "it first." >> /tmp/asl-channel$channelNum.tmp
/usr/ucb/mail -s "Message Available" ‘cat channel$channelNum.wait‘ \
< /tmp/asl-channel$channelNum.tmp

rm -f channel$channelNum.wait /tmp/asl-channel$channelNum.tmp

RETURN "0";

END
```