Rule-Based Process Servers for Software Development Environments (Thesis Proposal)

Steven S. Popovich

Columbia University
Department of Computer Science
New York, NY 10027

CUCS-013-92
14 April 1992

Advisor: Prof. Gail E. Kaiser

The author is supported in part by NASA grant NGT 50583 and in part by IBM Canada Ltd. 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, IBM and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

ABSTRACT

Rule-Based Process Servers for Software Development Environments

Steven S. Popovich

Current software development environments generally take one of two approaches. The integrated toolset approach provides an extensible set of utilities to the developer, and a common inter-tool communication and integration facility. However, it provides no guidance as to how the tools should be used. The process-based approach integrates a toolset with a process control engine so as to provide an environment that enacts a particular development process. Unfortunately, it is not always feasible to go the process-based route; in the case where the existing tools are integrated with one another, the process engine integration mechanism may be in conflict with that of the integrated toolset. Process servers are an alternative means of integrating a process control engine with a toolset, which avoids this conflict by adapting the process control engine to use the toolset's existing integration mechanism.

Many process-based environments are rule-based environments: Their process specification language is a rule language. Rules deal well with local constraints, but often interact counter-intuitively and make it difficult to express the global nature of a process. Higher-level process languages are needed to describe global constraints effectively, but without losing the advantages of rules.

We propose to integrate the two approaches in an environment centered around a *rule-based process server*, which is a process server with a rule-based control engine. We propose also to provide a framework for implementing declarative process modeling languages and for interfacing the interpreter to the target toolset. Although our process server is to be rule-based, we plan to support a broader, multi-paradigm notion of a process modeling language. These process modeling languages will include declarative global process specifications in addition to local rule-based constraints, but translated into a rule-based form, plus appropriate run-time support wrapping around the rule interpreter, for execution.

1. Introduction

Software development environments began as programming environments, collections of tools designed to speed the process of developing the code for a system. Many environments still share a similar basic model; of a set of tools, communicating via some sort of inter-tool framework, each handling a particular facet of the development process. What such environments lack is a model of the development process itself. Although their tools may assist the developer, the environment as a whole is unable to enforce a particular set of practices or development methodology, and is unable to assist the developers in deciding what tool should be applied at any given point in the process, and how it should be used.

In the last few years, the notion of process-centered environments has become popular in the environment community [Perry 89a, Katayama 90]. A process-centered environment is one that has at its core a specification, in some formalism, for a particular development process that it implements. Some process-centered environments, such as Inscape [Perry 89b], are hard-coded to support a single development process, while others have a process language in which development processes to be supported may be described. These languages are either procedural or declarative. The idea of specifying a process in a procedural language is known as process programming [Osterweil 87]; this was the earliest form of process specification, and is one of two main alternatives being researched today. It exists today, however, only as a pure specification language, rather than as an executable specification; no environments based on process programming have actually been implemented, although Arcadia is proposed for implementation.

The other main alternative is the rule-based development environment (RBDE). In an RBDE, the process is specified not in the form of a conventional program, but in the form of a set of rules that govern the application of the various tools. RBDE's have been created with several purposes in mind; some, exemplified by CLF [CLF 88], are intended mainly to support automation of certain tasks and use a forward chaining model. Some, such as Darwin [Minsky 88], are meant to enforce a development process on their users, and so use a backward chaining model, "proving" to themselves at each development step that the proposed step is legal within the supported process. Finally, there are environments, such as MARVEL [Kaiser 88, Barghouti 88, Barghouti 90a, Barghouti 90b], that use what has been termed a "maximalist" model [Barghouti 91], involving both forward and backward chaining, and may provide both automation and enforcement.

One major problem with RBDE's is the intrusiveness of the rule model; the developer using an RBDE is forced to think in terms of rules, as in MARVEL, where the developer's commands are not given in terms of tools to run, but in terms of rules to fire. This problem is not unique to RBDE's; for example, an environment centered around a procedurally programmed process would presumably present the difficulty of forcing the developer to think in terms of the constructs of the process programming language. In order to address this difficulty, and allow the user to interact primarily with the tools of an RBDE rather than with its rule processor, Kaiser has put forth the idea of moving the process description into a rule-based process server

(RBPS) [Kaiser 91a], which is consulted by the tools whenever the developer attempts to perform a significant development step. The RBPS extends Schaefer *et al.*'s notion of a process control server [Peuschel 90] to RBDE's. The server is a natural choice for a reusable component; however, no standard is emerging for either process specification languages or for tool communication and interfacing with the rest of the environment. Therefore, we propose to develop a framework for constructing rule-based process servers for a wide range of development environments and process specification languages. We see this work as applying to a subset of environments following the integrated toolset model and having some sort of shared communication mechanism to facilitate tool integration.

2. Motivation

In order to clarify the reason why this problem is important to solve, we need to understand two points. First, we need to make clear why an SDE needs a process server[1]. Second, we need to understand why translating declarative specification languages into an underlying rule-based execution model is a good idea.

Many SDE's make tools and facilities available to the programmer, but fail to guide their application. They may facilitate using a particular methodology, but they cannot enforce it. One example is the ISTAR environment [Dowson 87]. ISTAR is based on a *contractual* model of software development, where the software to be developed is specified in terms of a set of deliverables. Its basic development transaction is the contract, which is an agreement for one developer (or group of developers) to provide specific deliverables to another. Work may be *subcontracted*; that is, a developer (or group) who has a contract may contract with yet another developer (or group), who may produce a subset of the original deliverables and/or additional intermediate deliverables for the primary contractor. ISTAR's "contractual" approach provides enforcement in the sense that certain specified deliverables must be produced before the contract can be completed, but it says nothing about the process to be used in development. In fact, Dowson states (in the referenced paper) that:

> The essence of a contract is that it is the *contractor's* responsibility to decide how to fulfill it, subject to any constraints in the contract specification.

It is later made clear that the constraints that may be specified are constraints on what, if any, work may in turn be subcontracted; they do not constrain the development process, in the methodological sense. In contrast, process-based environments constrain the process at some level, often the coarse level of tool invocations, although finer granularities may be available. The contractual model is orthogonal to the enforcement of a development process; actually, the sort of specifications that we propose here should be able to coexist with ISTAR's notion of contracts.

---

[1] The terms "process server" and "policy server" are equivalent for all practical purposes. The term "policy tool" is broader, since it covers implementations of policy other than servers. In this proposal, however, all three terms will be used interchangeably.

What ISTAR does provide, besides contracts, is a framework for tool integration. In fact, its idea for encapsulating tools in an "envelope" was the inspiration for MARVEL's black-box tool integration facility [Gisi 91], although the enveloping mechanisms in the two environments are completely dissimilar. It also provides a well-defined pathway for communication between tools, in the form of the contract database. So ISTAR can be summarized as a contract mechanism, along with a set of tools in an integration framework for specifying and executing the contracts.

Many other environments, such as NSE [Honda 88], DSEE [Leblang 84], and Field [Reiss 90], have been structured along similar lines — as an "integrated tool set" and a mechanism to allow integration of additional tools. Sometimes the integration mechanism is a shared objectbase, as in ISTAR; sometimes it is a shared communication channel, as in Field. In either case, these environments provide no control over the development process other than that which is implicit within the tools (*e.g.*, needing, as in NSE, to check out sources using a source control system, such as SCCS or RCS, before they can be changed). Furthermore, there is no guarantee that even implicit constraints will be honored; the implicit requirement to check out sources before changing them exists only if they have first been checked into the source control system.

These environments would need policy tools to control the application of the other tools they provide in order to enact a software development process. Examples of policy tools are the Forest [Garlan 90] policy tool developed by Garlan and Ilias for the Field environment, and the Field/MARVEL experiment, which we will describe later in this paper. The policy tool approach also has other advantages, *e.g.*, making dependencies between tools explicit rather than hiding them in the tools and separating policy decisions from the other aspects of the environment. Actually, since each environment's tool interface and invocation protocols differ, what is needed is not a single policy tool, but instead a "toolkit" consisting of a policy interpretation "kernel" and tools for building the individual environment interfaces.

Why do we need a policy tool, rather than a policy-based environment such as any of the numerous rule-based development environments? One reason a development project might have for incorporating a policy tool into its environment, rather than replacing its environment with a policy-based one, is that by doing so they will be able to continue using their existing toolset. This is often a powerful real-world consideration.

On a more abstract plane, rule-based SDE's like MARVEL and CLF can enforce a methodology by requiring developers to work within the prescribed rules, but they force developers to use a non-intuitive model of tool invocation as explicit rule invocation. The problem is one of intrusion of the rule-centered nature of the environment into the development process. Rather than *governing* the process, the rules *become* it.

For example, to edit a file in MARVEL, the developer must go to the MARVEL control window and call the "Edit" rule on an object in the objectbase that represents that file. We would prefer that the developer would not have to deal with this additional level of control within the environment, yet would still have the benefit of process enactment. Ideally, the developer should simply visit a file in the editor, with the process component being informed of this action both before and after the fact. Before, so that it can take any appropriate action, including disallowing the edit, and after, so that it can propagate the changes made to other tools via chaining.

Integrated toolsets, such as Field, use a different model, where independent tools communicate with one another through a common server to share information. In such an environment, a policy tool can monitor the communication channel and track the developer's actions using its model of the development process. The rules in the process server can stay more in the background, while still being enforced. In the case of Field, as we will show later, enforcement is not actually possible without modifying the Field tools; however, this does not invalidate the general model. While this situation (enforcing policy from the background) is not handled well by existing RBDE's, a policy server can be built with, essentially, the rule engine of an RBDE as its "kernel".

To explain why we plan to use a rule-based execution model for our process servers, it is necessary to present some background information. Most of the work done by the software engineering community on process specification languages has used one of two very different but highly intuitive alternatives. The older of the two alternatives, often referred to as "process programming" [Osterweil 87], expresses the process procedurally, in a form highly reminiscent of conventional programming languages. The other, more recently developed, alternative is the rule-based paradigm, where the rules governing the development process may be expressed directly in a rule-based language. We refer to process programming as the older alternative, even though at least one rule-based environment, CLF, predated process programming. This is because CLF's rules were not described as a process model until after the notion of process programming had become popular.

Process programming represents the software development process as a global flow of control. It has the advantage of using familiar programming constructs to describe the process. It also, however, has the disadvantage of expressing the process as a global control flow — it can be difficult to express local constraints on the application of a particular tool. Also, process programs for any real-world development process are necessarily parallel programs, so process programming naturally inherits all the difficulties of parallel programming. One example of a process programming language is APPL/A [Sutton 90].

The rule-based model provides a natural form for expressing local constraints, but it has problems describing the process at a global level. It has, in fact, been described as a "data-centered" view of the process [Riddle 91a], seriously deficient with regard to expressing the flow of control within a software process.

A third alternative, which we consider as part of the same branch as the rule-based languages, is to use another form of declarative specification, such as an augmented form of Petri nets [Deiters 90, Gruhn 91], or regular expression-based activity structures [Riddle 91b]. We consider this alternative as part of the same branch as the rule-based languages, since the specifications are non-procedural and readily translated into a rule-based form (*e.g.*, the transitions in a Petri net or in the finite state machine implicit in a regular expression map to rules). These languages have a major advantage over both process programming languages and rule-based languages: They are able to express both local constraints and global control flow within a software process. So in supporting declarative process specifications and translating them into an underlying rule-based model, we are integrating the major advantages of both high-level specifications and rule-based environments, while attempting to avoid their disadvantages.

3. Proposal

The problem we propose to address is to develop an implementable language formalism for defining software development processes in a sufficiently generic (environment-independent) form. This problem is faced by the designers of software processes and process description languages. We intend to solve this problem in a general sense, across a range of specification languages, rather than for a single, specific language. As stated, there are two natural aspects of the problem, namely the language (formalism) aspect and the environment interface aspect.

Our position on the language aspect is that the rule-based formalism is suitable for expressing local data constraints in a process modeling language, but is too low-level to be acceptable as the sole paradigm in such a language. Also, a rule notation that is sufficient for describing a particular software process may not be sufficiently expressive to succinctly describe all such processes. Therefore, we propose to use a rule-based execution model for our process engine, but intend this as an "assembly language" into which high-level process languages will be compiled. In some process languages, such as ASL, a notion of rules may survive into the high-level language, but only as a means of specifying local data constraints.

Our position on the environment interface aspect is that the description of a software development process should be independent of the environment. A process should be able to execute in any environment without substantial modification, providing that the necessary tools are available. Therefore, we propose to separate both the specific tool interfaces and the generic tool integration interface from the environment specification.

The rule-based process server (RBPS) system that we propose is to be divided into three main parts:

- A rule engine kernel, with a configurable execution model.
- A language definition toolkit, for defining declarative languages that will be translated into a rule-based model to be run by the kernel.
- An environment interface library. The main thrust here is to separate out, as far as possible, the mechanics of interfacing to a particular environment from the rule-based process model itself.

We now discuss each of the proposed components. To avoid confusion between the various classes of users of the RBPS system, we will use the term "RBPS instance implementor", or "implementor" for short, to refer to a user who builds a process server for a particular language using the RBPS facilities. We will use the term "administrator", or more fully, "RBPS environment administrator", to refer to a user who performs privileged administration functions in an RBPS instance environment, such as defining the process to be enacted. Finally, we will use the term "user" to refer to an end user of an RBPS environment instance, without any special privileges.

3.1. Rule Engine Kernel

The rule engine kernel is to be an execution engine for a general form of rules, consisting of a condition, an optional development activity, and a set of mutually exclusive effects, one of which is chosen to be asserted based on the results of the activity. The condition itself may consist of two parts — one to establish variable bindings, and the second using those bindings to produce values using those bindings. The proposed model is a generalization of the rule model used in the MARVEL environment.

3.1.1. Rule Representation

Several features of the MARVEL rule model have proven to be helpful in describing software development processes, and will be retained in the proposed rule engine kernel as fixed features of the interpreted rule language. They relate to the structure of the rules themselves.

Rules are to consist of four parts:

- An optional set of variable bindings, the structure and interpretation of which are specific to a particular instance of the rule language.
- A single condition, whose structure and interpretation are likewise specific to a particular rule language instance.
- A single activity, the structure and interpretation of which are specific to the particular rule language. When a rule is invoked while its conditions are satisfied, or if its conditions become satisfied as a result of rule execution and forward chaining from the satisfying to the satisfied rule is allowed by the rule language's chaining model, the rule will have its activity executed. The activity may provide multiple return values.
- Multiple mutually exclusive effects, the structure and interpretation of which are specific to the particular rule language. After running the activity, its return values are used to determine which effect is to be asserted, and may also be used in determining the values to be asserted in the selected effect.

Aside from these basic points in the design of rule languages, the content of the rules is not the concern of the kernel; it is instead the concern of a particular instance implementation. We propose also to provide a toolkit for constructing declarative language instances that will translate into the rule model to be handled by the kernel. The content of this toolkit will be discussed below; here, we attempt to characterize the range of rule models that will be implementable in the kernel and that should be supported by the toolkit.

The kernel's rule model is intended to be configurable in several ways. We have already mentioned two of them, condition expressions and activities, in passing. Condition expressions will be configurable in that their makeup and evaluation function will not be fixed. Thus, it will be possible to consider anything from a MARVEL-like logical expression to an opaque piece of C code, used as a database trigger, as a rule condition. Conditions are to be represented as a tree structure, with a function attached at each node, which is responsible for evaluating that node. There should be a function for each node type. Thus, an interpreter instance for a MARVEL-like rule language would have node types representing logical AND and OR operators and various relational operators, while an instance for opaque triggers might have a node type for each distinct trigger, where the attached function simply performs the trigger test. Effect expressions will likewise be configurable, but will not be constrained by the condition expressions. While there must be a connection between the two languages for chaining to occur, other non-chaining uses of rules should also be allowed, *e.g.* to run a particular C function based on the satisfaction of a logical condition or on accessing a particular variable in the specification language ("active values").

The MARVEL-style chaining model could be described as a hybrid of the "data driven" and "event driven" chaining model. Although the initial invocation of a rule is done at the user's request (the event driven model), all chaining, both forward and backward, is driven by the satisfaction of, or the need to satisfy, particular predicates on the data (the data driven model). It should be possible, in the proposed system, to implement a purely data-driven chaining model by having the request filter make use of a RETE network or similar structure generated from the conditions in the rule specification. The data structures and functions of the chaining model specification would also reference the RETE network. Since the RETE network implements only forward chaining, a separate backward chaining network would also be needed to implement both chaining directions. Such a purely data-driven chaining model is most attractive in environments where tool communication is in terms of data (*e.g.*, a shared objectbase, where transaction monitoring or a trigger mechanism can be used by the request filter to keep track of changes to the data). Adding a match network to MARVEL would be considerably more complicated, since MARVEL uses separate mechanisms for invocation of rules and "built-in" operations such as object creation and deletion, and both mechanisms would need to interface to the network. RBPS, as we shall soon see, will use a single mechanism for both rules and "built-in" object management operations.

Chaining support in RBPS should be sufficiently general to allow, alternatively, a purely event driven interpretation of chaining. In this model, exemplified by the Postgres database rule manager [Stonebraker 89], the condition of a rule does not consist entirely of predicates. It consists of an event specification on a named class of objects, along with a predicate-like qualification, which restricts the rule's applicability to the objects within the class that meet the condition. The effect of a rule is a set of database update commands. Chaining links in this model would be from an update command to all of the event specifications that match the update command, and vice versa. Links would not exist between the update commands and the qualifications, since merely satisfying a qualification cannot cause chaining in Postgres. This is analogous to the way that predicates in the binding section of a MARVEL rule do not affect chaining. Incidentally, and unlike MARVEL, predicates in RBPS rule bindings will be able to affect chaining. The MARVEL rule interpreter cannot detect the case where a rule is satisfied by adding to or deleting from an attribute that is used in its bindings, and we have at times found this deficiency to be limiting.

3.1.2. Activity Representation

Activities will be configurable to aid in porting a process between environments. To the rule interpreter, an activity will consist of a tool object that is applied to a set of parameters. The parameters are objects of various types (possibly primitive) within the object base. Tool objects provide a level of encapsulation that will hide the implementation of the tool in a particular environment. Unlike other objects, which consist of a set of attribute definitions (data), tool objects consist of a set of method definitions (program). Tool objects need not be stored in the objectbase; since tool invocation lifetimes are shorter than that of the process server, tool-specific data do not need to be stored persistently. Their definitions should be stored, but most likely will be stored separately from the objectbase.

MARVEL's current tool objects are much weaker than those we describe here; they are only a set of name definitions, which translate from the names that rule activities use for the tool's operations to the names of the underlying SEL scripts that implement the tools [Gisi 91]. A RBPS tool object will be responsible not only for translating operation names, as in MARVEL, but also for translating from its argument objects to whatever back-end constructs are applicable in the target environment. For example, tool objects for a MARVEL-like system would translate objects to their pathnames within the MARVEL objectbase directory and activities to shell envelope calls. On the other hand, tool objects for a Field-like system would translate objects to filenames within the Field environment directory and activities to message sending and receiving calls. It is likely that each tool object will need to be implemented as a separate executable program, to enable tool definitions to change without having to re-link the environment executable. Where dynamic linking facilities are available, *e.g.*, under SunOS, these may be used instead.

The activity interface is also, perhaps, a likely place to integrate objectbase operations such as object creation and deletion into the chaining mechanism. MARVEL suffered from an inability to specify chaining either to or from these "built-in" operations (as opposed to those implemented as rules). We do not see RBPS environments as having "built-in" commands at all, in the MARVEL sense; instead, objectbase commands will be "built-in" tools, represented by a subclass of tool objects. The rules controlling execution of "built-in" activities would simply be like any other rules in the system; it is only in their activities that they would differ.

3.1.3. Objectbase Support

The foregoing paragraph made the assumption that RBPS had a built-in objectbase, rather than sharing the environment's objectbase. This is not a necessary assumption, although it is a possibility. The alternative would be to include facilities in the environment interface toolkit to facilitate integration of RBPS to the target environment's objectbase, if it provides one, or if not, to a built-in default objectbase. In this case, the environment interface implementor would need to specify the tool objects that we have referred to as "built-in". The implementor would also need to specify the objectbase query and update interfaces as part of the environment interface specification, since RBPS would no longer be using its own objectbase. In this case, the objectbase interface would be part of the environment interface.

If we intend to model the environment in RBPS rather than integrating RBPS with the environment through the objectbase, however, we should build a separate objectbase into RBPS. As we will state in more detail in Section 3.3, its communication with the environment is intended to be through well-defined request and reply interfaces. Integrating RBPS with the environment's objectbase makes those interfaces difficult, if not impossible, to enforce, since it introduces a new communication channel (the objectbase) between RBPS and the environment. In the interest of enforcing these interfaces it might be advisable to keep the RBPS objectbase separate from any environment objectbase that might exist. If the built-in objectbase is sufficiently general to model the necessary range of target environments, then, it is the only objectbase that RBPS should use.

3.1.4. Chaining Model

We have now discussed each of the parts that make up an individual rule, so we now continue to consider the interactions between rules. The kernel's chaining model will also be configurable. We will preserve the MARVEL idea of having multiple types of chains that chain to one another under specified conditions [Barghouti 91]. Each condition and effect node will have a set of chaining types attached to it, and each element of this set will consist of a set of rules that may possibly be chained to. This is the case in the current MARVEL design, where there are four chaining types: Consistency forward, consistency "backward" (bringing currently inconsistent objects into a consistent state) [Barghouti 92], automation forward, and automation backward. Consistency "backward" chaining is not currently implemented in MARVEL; instead, no inconsistency is tolerated. If a consistency implication of a user action cannot be performed, then the entire chain is aborted atomically.

Restrictions on chaining can be stated in terms of these chaining types. For example, in the MARVEL Strategy Language (MSL), there are `no_forward`, `no_backward`, and `no_chain` directives that may be specified to restrict automation chaining; consistency chaining, on the other hand, cannot be restricted. Multiple chaining types are also useful for interfacing to a multi-user concurrency control algorithm.

In MARVEL 3.0, the chain that attempts to acquire a conflicting lock is always the one aborted. However, there is a design for semantics-based concurrency control, which should be implemented in an upcoming version of MARVEL. This design, called SCCP in Barghouti's thesis [Barghouti 92] uses the type of a chain, along with other information, to make decisions as to which chain to abort whenever two chains conflict. Chaining types, however, can also be used to restrict which rules may chain to which other rules. For example, an administrator might state as a condition that a consistency chain may not chain into an automation chain. This would mean that only a user's direct action, rather than some implication of the action, could cause tools to be run automatically. Automation would be somewhat restricted, but might be less surprising to the user. It could also be possible to selectively activate and deactivate particular chaining types; for example, to suppress automation while still observing consistency rules.

It should be possible, using chaining type specifications, to specify substantially different chaining models from MARVEL's current one. For example, MARVEL's backward chaining is non-backtracking, but we expect to be able to implement Prolog-style backtracking [Clocksin 81] as one alternative backward chaining type. This might be done by keeping the full list of alternative rule invocation possibilities for each backward chaining attempt even after backward chaining has "succeeded", and trying later alternatives on these lists if a later backward chaining attempt (in the same transaction) fails. Backtracking is complicated considerably by the fact that it is not generally possible to roll back activities after they complete. Therefore, backtracking might work only until the first activity in the transaction is performed.

3.1.5. Conflict resolution

Conflict resolution (in the OPS5 sense) is another aspect of the chaining model that should be configurable. During each cycle, the rule engine will build up a list of possible chains resulting from the cycle's actions, and then will call a function, defined by the RBPS instance implementor, to eliminate any conflicting elements from the list. After the conflict resolution function runs, the rules corresponding to the elements remaining in the list will be fired. If no conflict resolution function is specified, all rules that apply will be fired; this is the same behavior as the Soar production system [Laird 86]. Thus, the default result will be the firing of all rules that apply in (simulated) "parallel". This differs from the current MARVEL model, where all rules that apply are fired in (essentially random) depth-first order. That is, each rule that applies is fired, with all possible chaining being done from it, before the next rule is fired. Furthermore, in MARVEL, earlier rules may invalidate the conditions of later rules, causing them to be skipped rather than fired, since the conditions of each rule are re-evaluated before it is actually run.

In a single-threaded system, simulating parallelism reduces to firing all applicable rules as in MARVEL, but without re-evaluating the conditions; thus, a rule that happens to fire early cannot invalidate the conditions of one that happens to fire later. It will be necessary, however, to handle such an invalidation as a concurrency control conflict if and only if the branches lead to the assertion of contradictory effects. It will also be possible, in RBPS, to implement an arbitrarily strict conflict resolution strategy, which might pare down the list to a single element as in OPS5 [Brownston 85], or even eliminate all elements to end chaining.

This notion of conflict resolution will need to be reconciled with the existing MARVEL concept of consistency chaining before Oz can be built using RBPS; Oz is described in Section 6. Conflict resolution should not be able to prevent consistency implications of a user action from occurring, but it should be able to affect the order in which they are asserted. This is not a problem with the RBPS design, since RBPS itself should know nothing about consistency chaining, but it is something that should be specifiable in RBPS.

3.1.6. Synchronization

Finally, although this has already been covered implicitly, we want to make it explicit that synchronization mechanisms between rule chains should be another configurable aspect of the kernel. The ASL implementation was forced to synchronize through the object base by implementing semaphores using two integer variables and six rules for each semaphore. The kernel should not force such an inefficiency. It should be possible to include synchronization actions as conditions and/or effects of a rule if a particular process language should happen to require them. This has implications for many other aspects of the kernel, as well as on the concurrency control mechanism in a multi-user system.

For example, there is the problem of how separate invocations of an RBPS instance environment may communicate for synchronization purposes, and where synchronization data such as semaphores are to be kept. In the MARVEL environment, there is a single *server*, which maintains all of the objects and does all of the rule processing. Each invocation of the MARVEL environment starts a *client*, which is a user interface to the server. Each client has a context defined in the server for it, in which rules and chains initiated by that client are run. In RBPS, we see the kernel rule processor functionality as being provided by a server, with environment interfacing being handled by a client process, as was done in the Field/MARVEL prototype.

Given this client/server model, it seems clear that the data needs to reside in the server, and that the communication channels should be kept separate from those used by the kernel to avoid confusion between user and kernel messages. However, most synchronization questions, such as the choice of primitives (send/receive *vs.* signal/wait *vs.* semaphores, etc.) and whether, or how long, a primitive can be allowed to block (*vs.* failing), are specific to a particular system and cannot be answered in advance.

Interactions of explicit synchronization operations with implicit synchronizations will also be dealt with. For example, consider a process language based on Petri nets. The transitions of the Petri net map into rules. However, unlike the finite state machine model, where each transition simply moves a single token representing the current state, in a Petri net, each transition may consume (and produce) multiple tokens. A transition that needs more than one token (which translates simply into a rule with a conjunction of the same number of state conditions) represents an implicit synchronization. If RBPS as a whole is to be kept free of deadlocks, these synchronizations, which are implicit in the rule network, should be detected and considered by a deadlock avoidance or detection algorithm.

A question also arises as to how much synchronization information needs to be persistent, and whether it would be advisable to make chaining contexts persistent in RBPS. Chaining contexts might be made persistent in order to avoid the need to roll back already completed activities in the event of a failure during later processing of the same chain. It seems clear that, in any case, synchronization state information, such as semaphore counters, needs to be persistent. The pattern of implicit synchronizations is already persistent, through the rule network, so all that remains in question is the pattern of explicit synchronizations at any given moment; *i.e.*, who is waiting for whom. If this information is to be persistent, that would imply that chaining contexts were also persistent, and furthermore that rules and chains were not aborted during failure recovery, but rather continued. It seems, however, that synchronization semantics can equally well be preserved using transient explicit synchronizations, transient contexts, and rollback of running rules and chains upon recovery. Thus, RBPS will most likely not have persistent chaining contexts.

3.2. Language Definition Toolkit

The Language Definition Toolkit will allow tailoring the rule interpreter provided by the kernel to execute a broad class of rule-based languages, as described above, and will facilitate building translators from declarative process modeling languages (such as ASL) to the rule language structures supported by the interpreter.

The Unix Lex and Yacc tools provide adequate support for the lexical analysis and parsing phases of language translators; thus, the Language Definition Toolkit will include these tools, and the research work in developing it will center on providing facilities for specifying the rule model for the Rule Engine Kernel; we also plan to provide libraries of semantic and back end support functions for translators whose output is the rule structures of the Rule Engine Kernel, as well as a library supporting extensible notion of context that will allow making language-specific additions to the rule interpreter’s client description information, for use by the evaluation and chaining functions for that particular language. Section 4.2.3 shows why an extensible context is necessary in policy servers, and Section 4.1.2 gives an example of an extended context from our previous experience.

A translator, then, will use a front end based on Lex and Yacc to build up, using functions provided by the Toolkit, structures in memory that represent rules in the "assembly language" to be emitted as its output. Some of these functions will be called explicitly from semantic action routines, while others will implicitly elaborate the administrator’s rule descriptions with the information provided by the rule model specification.

This specification information will include the representation and evaluation functions for condition and effect expressions, the chaining model as discussed above, and the query interface between expressions and the objectbase. We include assignments to variables and object attributes in the term "expressions". This is all of the information needed by the kernel to run the rule model, with the exception of information about the representation and execution of activities. The activity interface is intentionally kept separate, and is one subject of the Environment Interface Toolkit.

3.3. Environment Interface Toolkit

The Kernel Rule Engine, tailored using the Rule Language Toolkit, will provide a process server for processes implemented in a specified language. Once we have decided upon the particular environment where we want to incorporate the process, we must use the Environment Interface Toolkit to integrate the process server into the target environment. We have already interfaced the MARVEL process server to a single target environment, namely Field. This has provided us with insights as to the requirements of this toolkit (see Section 4.2).

The purpose of the environment interface is to integrate the policy server into the target environment using the target environment's tool integration facilities, *e.g.*, the message server in Field. If the environment has no tool integration facilities, either the process server must be integrated with every tool in the environment, or it must control the environment, as it does in MARVEL. The environment interface must monitor the activities being performed in the environment, and decide when it is appropriate to make a request to the rule engine. This request filtering process is a subject for our investigation; however, we do not expect to be able to build a high-level specification language for request filters, from which the code for such filters could be generated. This is because the process of request filtering is intimately connected with the target environment, particularly with its inter-tool communication mechanism. The request filter will be a program, *in the target environment*, that can monitor the communication mechanism and communicate with the process server. We do not believe that it is feasible to generate these programs automatically, since at present many environments have highly idiosyncratic communication mechanisms.

After the environment interface makes a request, it will wait for a reply from the policy server. Once the request has been performed, the environment interface will be responsible for returning the results of the request to the environment.

In addition to filtering requests from the target environment and translating the requests into the proper form for RBPS, the environment interface will also encapsulate the idea of an activity in the environment and provide translations between activities in RBPS and in the environment, and between the environment's entities and the RBPS objectbase. Using the Environment Interface Toolkit, a process administrator will be able to specify the process for performing all of the necessary translations, as well as that to be used for performing activities in the target environment.

3.4. Summary

We propose to investigate the area of rule-based process servers for software development environments. Toward this goal, we plan first to characterize the languages and environments that our work will be applicable to. At the current time, we intend for it to apply to declarative process specification languages that specify either rule-like descriptions of individual development activities or some sort of pattern of activities to be matched by the development process. We also intend for it to apply to development environments where tool communication and integration is provided by the use of a shared communication mechanism or channel (perhaps including a shared database), as opposed to environments where each individual tool must be tightly integrated with the others. The work will precisely define these classes of applicability, and perhaps also expand them.

With these areas in mind, we then propose to define the interfaces between the kernel, the process specification language, and the target development environment. We also propose to develop the kernel and the two interface toolkits, and to show the applicability of our solution by implementing process servers for various languages and environments using our toolkits. These implementations should include at least two languages other than the rule-based "assembly language", and one should be in substantial depth. The implementations should also target at least two environments other than MARVEL; again, one should be in substantial depth.

4. Evidence

4.1. ASL Translator (Language Aspect)

4.1.1. Problem Overview

In the summer of 1991, we worked on integrating global control structures with MARVEL rule specifications. The project was to design and implement, to the extent feasible, a relatively high-level language for describing activity-centered control structures in an SDE to augment the data-oriented rule-based paradigm, producing a MARVEL strategy (MSL rules) as output. Named activity structures were specified as regular expressions with the addition of a shuffle operator, which expressed concurrency among its operands, and a synchronization mechanism based on sending and receiving messages. Each activity within an expression was specified as a name and a set of parameter variables to be bound in certain ways.

4.1.2. Experience

A translator from the activity structure language (ASL) to MARVEL rules was built [Kaiser 91b]. ASL specifications are translated internally into non-deterministic finite state machines, which are reduced to deterministic machines and then translated into rules. Each transition in the deterministic finite state machine produces a rule, since each stands for the execution of a single activity in the software process.

Only one major change to MARVEL was needed in order to run the translator's output. The MARVEL strategy loader and the run-time rule selection component (the *overloader*) needed to be changed to allow the existence of multiple rules with the same signature. These are generated by the translator whenever the deterministic state machine for an AS contains multiple transitions with the same name and parameters, which most commonly occurs because an activity name is used more than once in an AS. The loader had to be changed to make such rules legal, and the assumption that the rule name and parameter types uniquely specified a rule had to be removed from the code. The overloader was the piece of code that was responsible for resolving which of several rules with the same name (but different parameters) was to be invoked, given the actual parameter types. It had to be changed to send a set of rules, rather than only a single rule, to the rule interpreter, and to stop evaluating rules from the set as soon as one of them fired successfully.

The reason that these changes were necessary was that a single activity might appear multiple times in an activity structure expression, and a single appearance might generate multiple rules. Each of these rule instances would have an identical signature, but their conditions would be mutually exclusive, so that only one of them would be applicable at any given time. Thus, the overloader could evaluate the rules in any order, and could stop after the first applicable rule was found, without violating the semantics of ASL.

For example, the ASL expression `edit[?c:CFILE]; (compile[?c:CFILE]; edit[?c:CFILE])*; compile[?c:CFILE]` specifies a simple process for developing a C program in a single file. The file must first be edited, then compiled and edited repeatedly as long as it contains errors. The process ends with a compilation that produces no errors. This ASL expression produces three edit rules with identical signatures and, similarly, two compile rules. These rules implement the three-state finite state machine shown in Figure 4-1. The rules all contain state conditions, so that each rule corresponds exactly to one transition, and so the conditions of two `edit` or two `compile` rules will never be true simultaneously.

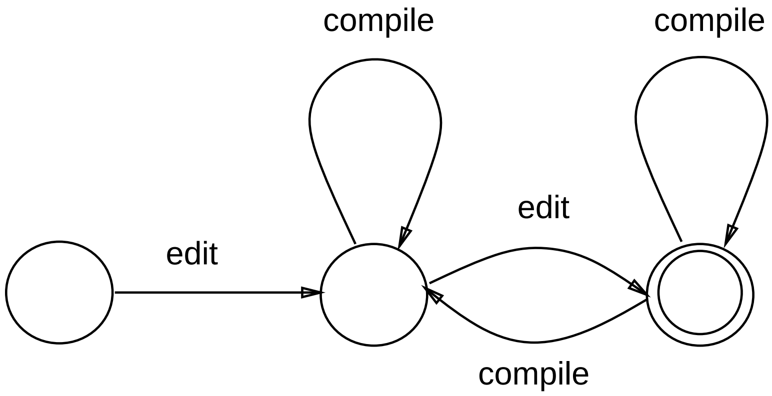

Figure 4-1: FSM for `edit; (compile; edit)*; compile`

Only one other small change to MARVEL's rule language, the addition of operators allowing the creation and deletion of links as an effect of a rule, was required in order to implement static binding and unbinding of parameter symbols. Furthermore, it can be argued that the operators actually belong in MSL, since they are also useful in other contexts. So in actuality, only an extended context structure and its support code (called the Activity Structure Manager) needed to be added to MARVEL to implement ASL. The Activity Structure Manager was responsible not only for maintaining the context, but for maintaining bindings (links) to the context to be used during rule evaluation. It was implemented as a layer around the MARVEL rule processor, as shown in Figure 4-2. The extended context was necessary in order to be able to implement the notion of a "current activity structure" being run by a particular client. We use the term *client* here in the MARVEL sense of a particular invocation of the user interface. The current activity structure is an implicit parameter to the rules for running the activities of the AS. We will show later, in Section 4.2.3, that the extended context mechanism would be generally useful. Thus, with only a few minor exceptions (discussed below), the changes were not specific to the ASL environment. This experience showed that MSL could be used as an "assembly language" for SDE policy description languages, which is how we intend to use a generalized version of it in this thesis.

For each activity structure definition, the translator produced an *activity structure object* class, which could be instantiated by the use of an `Instantiate`. These classes were known as *root* classes, since they were not contained within any other AS classes. We will see shortly that shuffles produced nested AS classes. An instance could then be given as a parameter to an `Activate` rule in order to begin its execution. `Instantiate`, `Activate` and other rules for controlling AS execution were generated automatically by the translator.

Since MARVEL already supported concurrency among clients, shuffle was modeled as a nesting of smaller activity structures within a larger one. Separate activity structure objects were created for each "leg" of the shuffle, in addition to the one for the parent; any or all of these could be active at any one time. There is a restriction that only one activity structure may be active in a particular MARVEL client at any time. This is not an onerous restriction, since there is no limit on the number of clients that a single user may have active (on his screen) simultaneously. So, the concurrency between "legs" of a shuffle is modeled as the concurrency between clients in MARVEL.

In MARVEL, there are no mechanisms for communication between clients other than through the objectbase, so the send and receive operations between branches of a shuffle were implemented as rules that manipulate attributes of the root activity structure object, which is a common ancestor of the sending and receiving activity structures. Since RBPS environments may include inter-client communication facilities, this sort of workaround will not always be necessary in RBPS.

The root activity structure is used instead of the closest common ancestor, since shuffles may be nested, and messages may be sent among any of the legs of arbitrarily nested shuffles, not only among the legs of a single shuffle. Messages are queued, so it is not necessary for the receiving leg to be active at the time that a message is sent. It is possible to cause a deadlock with careless use of send and receive, but no part of the ASL system checks this.

Actually, send and receive were each implemented as three rules. This was due to restrictions in the MARVEL chaining model which, at the time, allowed changing only attributes in a rule's parameters (rather than in its bound variables) and allowed chaining through only a single parameter of a rule. The first rule, executed in the context of the inner activity structure, asserts an attribute that causes chaining to the root activity structure. The second rule performs the work in that context, and the third removes the attribute that the first rule asserted. In RBPS, we hope to relax such restrictions, and in any event we intend to make it possible to add synchronization operations as primitives, so that it will not be necessary to communicate through the objectbase in cases like this.

Binding of parameters was a big problem area for the ASL project. By parameters, we mean, *e.g.*, the symbol `?c` in the AS `(edit[?c:CFILE]; compile[?c:CFILE])*`. Parameters are, as the example shows, named and typed. The default behavior is that the first time a parameter name is used in an AS, any value of the appropriate type is acceptable, and the

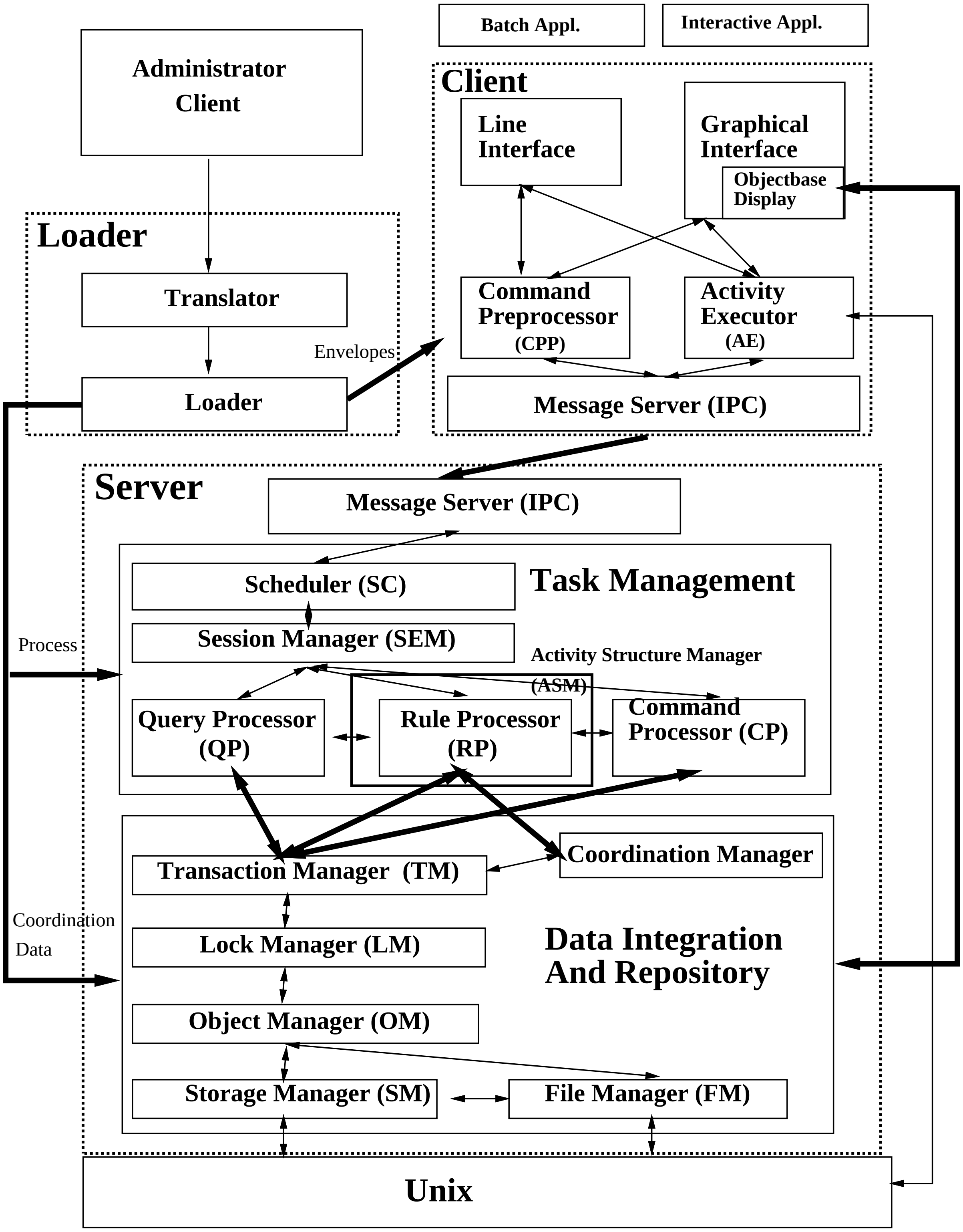

Figure 4-2: MARVEL 3.0 Architecture with Activity Structure Manager

parameter symbol will be "set" to that value. Later uses of the same name must use the same value in order to be accepted. For example, `(edit[?c:CFILE]; compile[?c:CFILE])*` specifies that the same CFILE that was edited must be compiled. However, it also implies that the same CFILE must be edited the second and any subsequent times through the loop. In order to override this behavior, an appropriate keyword must be included in the parameter specification. The AS `(edit[reset ?c:CFILE]; compile[?c:CFILE])*` specifies the intuitively "correct" behavior, where only the CFILE most recently edited may be compiled, but any desired CFILE may be edited. There is also a `use` keyword, which specifies that the parameter symbol must already have been set to an appropriate value in order for the activity invocation to be allowed.

This binding behavior is implemented by a piece of code in the activity structure manager called the binder, which is called automatically at certain points during rule evaluation. The binder first checks that each parameter of the rule is legal with respect to the existing bindings, then sets the new bindings and fires the rule. The binder is called whenever a rule is evaluated as a candidate for firing, either in response to a user command or as part of a chain. So the binder needs to have hooks into both the condition evaluation mechanism (for checking bindings) and into the rule execution function (for asserting bindings). It also needed to be connected to the extended context mechanism, since the bindings were kept the root AS object. This was one feature that it would have been possible to model in the generated rules, adding conditions to each rule to check the bindings and effects to assert them, but was much simpler to implement as part of the MARVEL rule interpreter. Implementing dynamic binding via rules would have entailed adding conditions and effects to every *activation rule* generated by the ASL translator to check and assert, respectively, the bindings whenever the rule was to be fired. An activation rule is one that contains an activity, as opposed to an *inference rule*, which has no activity.

These binding keywords in activity structures specify the *dynamic* binding behavior of the AS. The ASL translator also generates rules to allow the user to *statically* bind symbols in an AS to particular objects or, through multiple applications of a rule, to sets of objects. A `Bind` rule is generated for each symbol used within the AS, as well as one for the AS as a whole that will bind all symbols simultaneously. Similar `Unbind` rules are generated; each of those for a particular symbol removes the binding to that symbol, and one for the AS as a whole removes all bindings from that AS. At any time after creation of an AS, these rules can be invoked by a developer to override the dynamic binding behavior. We have not, in our limited experience, had occasion to use these rules, except to repair inconsistencies left by bugs in MARVEL or in the ASL translator.

Static binding is intended to allow a manager who assigns work to a developer through an AS to bound the subset of the objectbase that the developer, using the AS, will be authorized to change. The above examples allowed changing any CFILE in the objectbase; static binding, along with another set of binding keywords, enables restrictions to be put into effect. These keywords are `multiset`, `multireset` and `multiuse`; they are analogous to `set`, `reset` and `use` respectively, with one important difference. They handle bindings of multiple objects to a single symbol. To continue our simple edit/compile example, the AS (edit[multiuse ?c:CFILE]; compile[multiuse ?c:CFILE])* describes a restricted compile/edit loop. If the symbol ?C is bound to the objects A, B and C, those are the only objects that may be edited or compiled using this AS. multiset allows the use of any object of the proper class and adds it to the binding set; multireset also allows any object with the proper class, but resets the binding to the singleton set containing the chosen object.

The other major feature that was implemented in the kernel was the notion of atomic activities *vs.* non-atomic activities. An atomic activity is one that, while executing, locks out all other activities from running in the context of its activity structure object, rather than only those which would cause conflicting data access. This was translated into a special notation in the parameter list of the atomic rule, which the modified kernel interpreted as a request to lock the activity structure, using a different mechanism than usual.

4.2. Field/MARVEL Environment (interface aspect)

The MARVEL policy environment is set up to model the Field environment. Objects are created for Field "objects" such as C files and programs. Only status information is kept in the MARVEL objectbase; the actual contents, *e.g.*, of the C files, must be in the Field directory, so as not to interfere with the operation of the Field tools. This sort of setup will generally be required for a process server, unless the objectbase can be shared between the process server and the environment being modeled. The modeling approach runs contrary to MARVEL's implicit assumption that the rule interpreter is the top level in control of the environment. What we want is for the rule interpreter to be in control, but also to be in the background.

4.2.1. Implementation Overview

The policy tool is a client of the MARVEL server, and communicates with it using the same protocol as the normal MARVEL user interface. Although some changes to this protocol may be desired, this has not been necessary thus far. Individual Field tools also retain their interface to the message server. There is now, however, a distinguished connection to the message server for a single policy server. This situation is illustrated in Figure 4-3.

The policy server accepts messages from the Field server, monitoring all messages. It was necessary to monitor all messages, rather than using the pattern matching facilities of the Field message server. This is accomplished simply in Field, by registering for the pattern %1r, which matches the entire text of any message in a single string parameter. The more usual use for the %<number>r pattern is in matching variable-length options, as in the Field configuration management message pattern FORM CONFIG %1s %2s %3r. Here, the two fixed parameters are a project (directory) name and a filename within the project, while the third parameter matches the rest of the message. It begins with a command, such as CHECKIN or CHECKOUT, which may be followed by keyword options such as LOCK or UNLOCK. It may also include options with parameters, such as LOG, which is followed by an arbitrary string containing the log message to be passed to the configuration management program (SCCS or RCS).

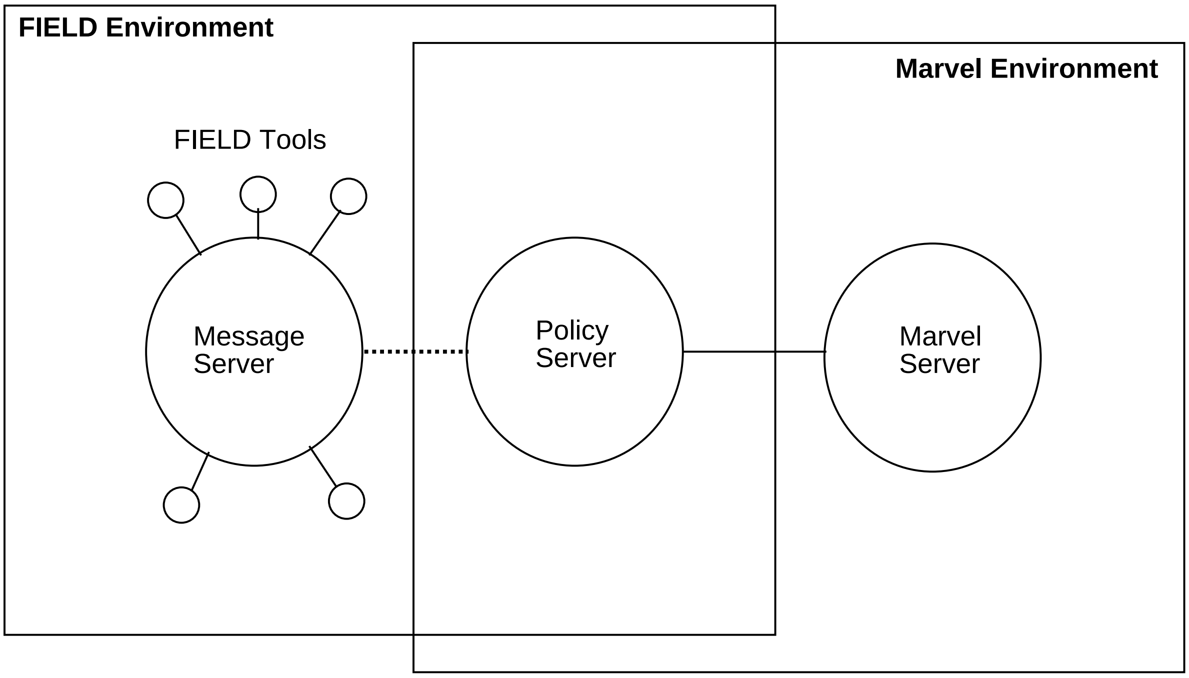

Figure 4-3: Top-level Field/MARVEL Architecture

The reason we used the `%1r` pattern was that we needed to be able to handle all messages, regardless of the number of parameters, using a single routine in the policy server. This was the only means available to accept all messages using a constant number of parameters. Using separate patterns for each message would have meant that each might have a different number of parameters, so they could not all be handled by the same routine.

Field provides two kinds of messages, synchronous and asynchronous. Synchronous messages are sent using the `CALL` primitive, and suspend the sender until replies are received from all recipients. They return a value to the sender, which is taken from the first non-null reply message received. Asynchronous messages are sent using the `SEND` primitive, and neither wait for a reply nor, understandably, return a value to the sender.

In Field/MARVEL, synchronous messages have an extra step taken for them by the message server; for a message `X Y Z`, a `POLICY REQUEST X Y Z` message is sent to the policy server beforehand, and the actual message is sent only if the return is successful. Asynchronous messages, since they are regarded as notifications in the Field model rather than as requests, are forwarded as is. They could be treated similarly to synchronous messages, but if they were, there would be no way to notify the sender of any policy server denials; the message would simply vanish.

`POLICY REQUEST` messages are compared against a set of patterns, and if one matches, it specifies a rule to be invoked with a set of parameters derived from the message. All of our patterns have only a single parameter, but this is incidental — Field tools tend to work on single objects, and only objects can (currently) be parameters to MARVEL rules. Chaining proceeds normally as in MARVEL, with activities replaced by messages. Instead of specifying a command line to a shell envelope, the activity is treated as a message and sent to the Field message server, where it does not generate `POLICY REQUEST` messages. It is possible to specify multiple messages in an activity. If this is done, those after the first are treated as patterns to be matched by incoming messages; the parameters from the last pattern matched are used as the return values from the activity. In the case where there is only a single message in an activity, its return value from the message server is used as the return value.

In the process of completing the implementation sketched above, several problems became evident. Some of them will be discussed here as a means of motivating various features of this proposal.

4.2.2. Denying requests

Even though Field includes a policy tool, it is not set up for the enforcement of policy on the user. Field's policy tool, like Forest, handles only automation. This seems to have been done because messages are often sent as notifications, rather than requests, from the tool that performs an action. Even if the policy tool were to decide that the action violated a policy, it cannot affect the behavior of the sending tool. An example is the message sent when a file is saved. The editor does not ask permission to save the file, but simply notifies other tools of the fact. If our policy tool is to strictly enforce a policy, the Field tools must be changed to allow it.

Messages that are sent as requests, such as the `BUILD COMPILE` message which requests compilation of a named source file, do not cause this problem. They can, however, cause problems of their own. For example, in the reply to a `BUILD COMPILE` message, Field returns only the result code and a filename containing the captured standard output and standard error output from the commands invoked by the message. During the execution of the commands, this transcript output is also sent as `BUILD TRANSCRIPT` messages, and the Field user interface receives these messages, displaying them as they are produced. On the other hand, some other requests, assumed to take less time than a compilation, do display their return values rather than requiring special notification messages. This interface inconsistency in Field should be removed at the same time as permission requests are to be added to fix the "requests *vs.* notifications" problem described above. At the very least, policy denial messages should be recognized as what they are.

MARVEL has its own technical problems with denying requests. The client protocol provides no easy way to tell whether an attempted invocation of a rule succeeded or failed, since no meaningful result is returned to the client. This can be worked around by checking whether a particular message ("Execution cycle complete") is sent by the server while invoking the rule, as opposed to another one ("Execution cycle terminated") that signifies failure. This is an example of one (rather trivial) change that should be made to the MARVEL server to make it friendlier to programs that want to use it. Currently, the MARVEL client does no interpretation of the results returned from the client; it only displays them to the user. So no return result has been needed, until now. This is a minor, low-level problem, and serves here only as an example of how the current MARVEL server does not consider the fact that a client program might act as more than simply a user interface. RBPS clients will not interact with the user, but rather with a programmed request filter. Thus, the RBPS client/server protocols will be designed with the assumption that the client is under program control; *e.g.*, distinct result values will be returned to the client for success and failure of a rule invocation.

4.2.3. Policy variables

A policy tool should have a set of options that are settable by the user to allow him a limited facility to adapt the environment to his actions. In Forest, these are called policy variables. MARVEL makes it difficult to implement this, since there is no such thing as a variable in MARVEL, standing alone and separate from any object. There are pseudo-variables, such as CurrentUser and CurrentTime, but these are read-only, and an administrator cannot create new pseudo-variables without modifying the MARVEL kernel. Also, the MARVEL kernel uses standard UNIX environment variables for some of its parameters, but these variables are not available for either reading or writing in the rule language. The only way to provide variables would be to use one or more slots in an object with a well-known name to hold the values; even then, it would be necessary either to extend MSL with a primitive for accessing objects by name, or to relate the special object to all existing and future MARVEL objects by using some existing means such as the link mechanism.

One solution to the problem of providing variables is a mechanism like the one in ASL, which allowed designation of a particular object (the activity structure for a session) to be treated specially by the MARVEL server's rule processor and added to the parameter list of each rule invoked. A cleaner way to provide such a mechanism in RBPS would be by extending MARVEL's notion of a client context structure to include a context object, which would be extensible in MSL. This context object would hold the policy variables, along with any other client-specific variables the administrator wishes to create.

4.2.4. Translations between MARVEL and Field

It is necessary to make several translations between the MARVEL and Field environments. In a policy server, each of these should have a well-defined interface. Part of the problem of this thesis is to define such interfaces. As implemented, there were three interfaces:

Request Translation

It is necessary to translate incoming messages into requests. This was done in the implementation by using a simple pattern-matcher from Field and considering all of the string-valued parameters of the message to be object names that should be passed through to MARVEL. There is obviously much more that needs to be done here. However, there is a problem with attempting to generalize request translation. The problem is basically that environments are highly idiosyncratic in their communication mechanisms and in the uses that are made of them. In fact, they are so idiosyncratic that we do not yet see any general framework that can be used to describe them. Although we will investigate whether request translation can be generalized, we suspect that it cannot, and that the request translation component must therefore be essentially a program written for the target environment that filters requests and sends them to the rule interpreter through a standard interface.

Activity Translation

In the implementation, MSL-style activities had to be translated into messages to be sent and/or waited for. This had the disadvantages that every message had to be expressed in the syntax of MSL activities, which was frequently clumsy, and that only a certain pattern of messages (one synchronous request followed by a number of asynchronous responses in a linear order) could be accepted as an activity. In RBPS, this should not be a run-time translation, but should instead be part of the rule language. We plan to support encapsulation of activities and their invocation mechanisms as part of the environment interface toolkit. Activities will still be called from the rule language using a single standard calling sequence for all environments, but the translation process should be moved from run-time to compile-time. It will be the responsibility of the RBPS instance implementor to provide the translation functions. We also plan to use a more expressive calling sequence than the current MSL sequence, which forces the entire call into a single ASCII string. This is to avoid parsing an ASCII string into a message pattern, as Field/MARVEL currently does.

Object Translation

It is necessary to translate the incoming names from Field to MARVEL object ID's, and the outgoing object ID's in activities (tool calls) back to their Field names. A simple translation table can take care of this, but the job is complicated by MARVEL's assumption that the on-disk objectbase is structured in a particular way. The rule interpreter should be decoupled from assumptions about how attributes are stored on the disk.

Another problem is that MARVEL must track object creation and deletion in Field. Field does not have object creation and deletion operations; instead, object creation must be inferred from the messages sent (such as an editor save message on a filename that is not already in the translation table). Object deletion cannot be inferred in this manner for obvious reasons; if an object is not accessed for a long period, it will not appear in messages sent during that time, but should not be considered to have been deleted. Deletion can be inferred by comparing the MARVEL and Field object bases, but this is a very expensive operation, and highly impractical if the objectbases are large. One way of avoiding this problem in Field would be to require all Unix utilities that affect the filesystem to send messages, as was done in Yeast [Rosenblum 91].

The implementation does not currently infer object creation from the message traffic, but keeps (approximately) up to date by watching for messages noting that the Makefile has changed and then using an existing tool, part of Field's configuration management subsystem, to parse it into an object representation of the new system structure. This structure is compared with the MARVEL objectbase structure to determine what objects have been created and deleted in Field. There is, however, a technical problem with links — the structure created by the Field tool does not contain dependencies between .h files and .c files, and so the implementation does not currently track those dependencies. This is a deficiency of the Field environment itself, which depends on Make to interpret these dependencies properly and does not display or allow manipulation of these dependencies in its configuration management tool. This will most likely be addressed by extending the Field configuration tool to emit the proper dependencies, since otherwise we would need to reimplement Field's makefile parser in order to construct a tool for determining these dependencies.

4.2.5. Tracking Rules *vs.* Activity Rules

Another problem that comes up when trying to use MARVEL as a policy tool occurs because Field/MARVEL is both tracking and automating activities in the external environment (Field). For example, there's a problem with the "reserve" and "deposit" messages. In Field, there is a single message for "reserve", and it is handled as a request to reserve a file. MARVEL must treat this message, when sent by other tools, as a request to reserve a file, and also must send this message itself in order to reserve the file, *e.g.*, by backward chaining from reserve. This is not in itself a problem, since messages sent by the policy server are, naturally, not referred to the server for policy enforcement. The problem appears because the reserve rule that may be backward-chained to by the edit rule must actually reserve the file, that is, it must send the "reserve" message in its activity. The alternative tracking rule, however, should be fired by performing a "reserve" (actually, "Check Out") command in Field, and since Field will perform the actual (RCS) reservation, should not send the message in its activity.

Figure 4-4 shows some edit and reserve rules from Field/MARVEL. It includes rules to be fired at both the beginning and the end of an edit. `start_edit` simply checks that the file is reserved by the current user, and will cause backward chaining to reserve the file if it has not yet been reserved. `end_edit` incorporates the same condition, to guard against the possibility that the reservation may have been removed during the edit, and asserts as its effects that the file's object code is invalid and its source code was changed at the current time. The two reserve rules are an example of a tracking/activity rule pair. `reserve` is the tracking rule, and `auto_reserve` is the activity rule. The only difference between them, aside from their names, is that only `auto_reserve` contains an activity and has multiple possible effects. Since `reserve` has no activity, it can have only a single effect.

Figure 4-5 shows timelines of important events during an explicit reservation and during an edit operation that requires an implicit reservation. Notice that, if the activity rule were fired instead of the tracking rule during the explicit reserve, two reserve messages would be sent to, and executed by, the environment. Notice also that if the tracking rule were fired instead of the activity rule during the implicit reserve, no reserve messages would be sent to the environment, so the file would not actually be reserved before the user edited it.

How can we make certain that we get the right one of these two rules in each situation? We must give them different names, and have request translation generate only the tracking rule name. In Field/MARVEL, this was done by listing only the tracking rule name in the request translation pattern file, as shown in Figure 4-6. The rule name to be invoked is listed first on each line; the rest of the line is the Field message pattern that, when matched, will result in

```
start_edit[?c:CFILE]:
    :
    (and (?c.owner = CurrentUser)
         (?c.reservation_status = CheckedOut))
    { }
    ;

end_edit[?c:CFILE]:
    :
    (and (?c.owner = CurrentUser)
         (?c.reservation_status = CheckedOut))
    { }

    (and (?c.compile_status = NotCompiled)
         (?c.timestamp = CurrentTime));

reserve[?f:FILE]:
    :
    (or (?f.reservation_status = Available)
        (?f.reservation_status = Initialized))

    { }

    (and no_chain (?f.reservation_status = CheckedOut)
         no_chain (?f.owner = CurrentUser));

auto_reserve[?f:FILE]:
    :
    (or (?f.reservation_status = Available)
        (?f.reservation_status = Initialized))

    { RCS reserve "CONFIG *" ?f.contents "CHECKOUT" }

    (and no_forward (?f.reservation_status = CheckedOut)
         no_forward (?f.owner = CurrentUser));
    no_chain (?f.reservation_status = Available);
```

Figure 4-4: Example Edit and Reserve Rules

invoking the named rule.

We expect that it will be necesssary to maintain this distinction between tracking and activity rules in RBPS, particularly since we plan further RBPS work using the Field environment. It will continue to be the case that whenever one rule is applicable, the other rule will also apply. Thus, to avoid inadvertently firing the activity rule when the tracking rule is desired, their names must be distinct. Also, the conditions and effects of the tracking rule must be `no_chain`, so that chaining will always use the other rule. We have brought up this problem in order to show that MSL is more like an assembly language than a good high-level specification language. The issue of tracking rules *vs.* activity rules is an example of one that the process language compiler, rather than the administrator, should handle.

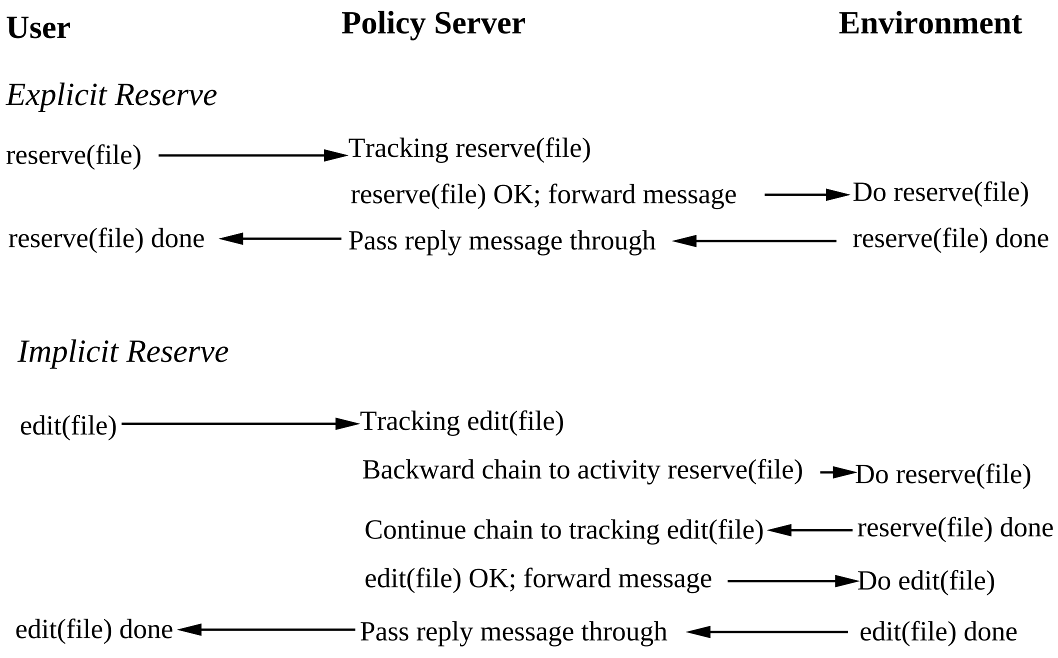

Figure 4-5: Timelines for Explicit and Implicit Reserve

```
start_edit ANNOT EDIT %1s
end_edit ANNOT SAVE %1s
compile BUILD COMPILE %1s
build BUILD MAKE %1s
quit MARVEL QUIT
reserve FORM CONFIG %s %1s CHECKOUT %r
deposit FORM CONFIG %s %1s CHECKIN %r
build BUILD COMMAND %s %1s
```

Figure 4-6: Request Translation Patterns for Field/MARVEL

4.3. Experience with MARVEL

In the process of the two experiments that we have presented here, and other uses of MARVEL by our project members, our experience has been mainly good. We now consider the C/MARVEL environment that the project has developed to be sufficiently capable that we are using it in our own further development. In addition to the general idea of rules, several specific features of the MARVEL 3.0 rule model have been beneficial to us, and we intend to keep these in the RBPS "assembly language". Some examples are:

- Both forward and backward chaining will be available (although some RBPS instance environments may wish to disable one or the other).
- Multiple types of chaining (consistency and automation) will be available to support concurrency control in a multi-user system, and to allow an extra level of control over chaining in the single-user system. For example, automation could be turned off if desired.
- Rules are to be expressed as not only conditions and actions, but as a single condition with a tool-based action and multiple mutually exclusive effects,

depending on the result of the tool. The method of tool invocation, however, may differ between environments, as well as the matter of whether the rule interpreter is controlling or is simply tracking and perhaps constraining the tool invocations.

- Tools will be encapsulated in activities (*e.g.*, via envelopes). The method of encapsulation may differ — e.g., messages in Field *vs.* envelopes in MARVEL.
- It will be possible to explicitly restrict chaining by specifying, at the level of individual rules, that a given expression within the rule's condition is not to be used for, *e.g.*, forward or backward chaining, or is not to be used as a basis for chaining at all.

In this thesis, we will assume a single-user system, except that we need to explore where hooks must be placed for extension to a multi-user model. We should, however, not make any design choices that would foreclose the possibility of a multi-user system; for example, we will assume a client/server model even though the single-user system does not require this.

In order to provide some degree of motivation for these decisions, we will compare these design features of MARVEL 3.0 with the corresponding ones in the original MARVEL prototype, which dates from 1986. The MARVEL "version 0" prototype was built on an existing programming environment, Smile [Kaiser 87]. Although the terminology did not exist at the time, today we would call Smile a process-based environment; its policy, however, was hardwired in the system's code. The prototype was intended to test the concept of an environment as a "moderately stupid assistant" (as opposed to an "intelligent assistant") for the programmer, with automation of menial programming tasks through a modifiable policy, which was expressed using a rule notation.

Rules had, then as now, a precondition, multiple mutually exclusive postconditions, and an activity; conditions were expressed in first-order logic. Effects were also written in first-order logic, with the exception that EXISTS quantifiers and OR were not valid. Thus, the rule language was very similar to MSL in many ways. However, the notion of enveloping did not exist in the prototype, and the activity consisted of the name of a C function and a set of parameters; in order to add a tool to the system, it was necessary to add code to MARVEL. Although tool integration had been one of the goals of the Smile environment, the MARVEL prototype's intent was to investigate the idea of *modifiable* policy; consideration of tool integration was deferred until the "second prototype", known as Version 1.0 of MARVEL.

Forward and backward chaining information, then as now, was attached to the rule network. It was attached at the rule level, rather than at the predicate level, but this fact makes very little difference. There was only one type of chaining, which today we call automation chaining; the prototype made no distinction between consistency and automation. Today, we would consider this fact a drawback; Barghouti and Kaiser [Barghouti 91] have shown why such a distinction is desirable.

There was also no support for undoing the effects of a rule or chain on the objectbase. Indeed, there was no support for crash recovery in the objectbase, other than that which was already provided by Smile. Smile saved its entire objectbase to disk as a backup periodically, and after commands, such as adding or deleting an object, which changed the objectbase's structure.

Since there was no concept of consistency chaining in the prototype, no chains were required to happen; all were "best effort". This meant that it was possible for an activity to run without all of its effects (in the sense of consistency implications) being asserted on the objectbase. This was not a practical problem, however, in the prototype, since it, like Smile, was essentially a single-user, single-threaded system. Although Smile allowed multiple users to be active in its main database, only one at a time could make changes such as checking modules in or out. Modules were checked out into single-user experimental databases, where the actual program changes were made. Except for module check-in and check-out commands, there was no concurrent access to objects. Thus, no concurrency control was needed, aside from a simple locking scheme to assure that the module check-in and check-out operations were performed atomically. Locking for check-in and check-out was implemented in those operations, rather than in the Smile database. So, in practice, nothing would interrupt a forward chain in progress.

Not only was there only one chaining type, but there was also no way to add chaining conditions such as `no_forward`, `no_backward`, and `no_chain` to MARVEL Version 0 rules. Instead, the prototype supported a special type of rule, called a "hint", that consisted of a condition and an activity alone, and which added its condition to any rules with a matching activity. Hints were observed in forward chaining, but were ignored in backward chaining; they served to avoid infinite loops in automation. Such loops could occur, for example, with the complementary Reserve and Deposit rules. Figure 4-7 shows, in the MARVEL 3.0 syntax, a simplified form of Reserve and Deposit rules for modules made up of a single source file. In MARVEL 3.0, this loop would be avoided by adding a chaining condition to either or both of the rules — most likely both, since it would generally be considered undesirable to chain forward into either of these operations. In MARVEL Version 0, it would have been necessary to add hints with always-false conditions to both of the rules to prevent the loop from occurring.

```
reserve[?m:MODULE]:
    :
    (?m.reserve_status = Available)
    { RCS reserve ?m.source_file }
    (and
        (?m.reserve_status = CheckedOut)
        (?m.owner = CurrentUser));

deposit[?m:MODULE]:
    :
    (and
        (?m.reserve_status = CheckedOut)
        (?m.owner = CurrentUser))
    { RCS deposit ?m.source_file }
    (?m.reserve_status = Available);
```

Figure 4-7: Loop in Reserve and Deposit Rules

After the initial prototype, the next version of MARVEL (version 1.0) was also based on Smile. It was a completely new version, however, using none of the initial prototype's code. Version

1.0 provided slightly less functionality, in some ways, than the Version 0 prototype; hints, for example, were omitted. Version 1.0 was intended as a base for further development rather than as a simple proof of concept, and was more cleanly written. It also contained an enveloping facility for tool integration. Further development of MARVEL continued from this "second prototype", Version 1.0, to the final single-user version, 2.6, and eventually to the current multi-user version, 3.0.

In summary, the Version 0 prototype confirmed our intuition that our rule-based model with forward and backward chaining was viable as a choice for a policy language in a software development environment. This was to be expected, since CLF and Refine [Reasoning 86] already existed at the time. CLF was a rule-based environment for Common Lisp development, which used rules both to maintain the consistency of its database and for partial automation of some development steps, but did not consider its rules as a process model. Refine was a transformation-based environment, in which the rules defined alternatives for the translation of relatively high-level constructs to the next lower level. Development in Refine consisted of writing a high-level specification and repeatedly applying transformations to produce executable code. However, the MARVEL prototype was, to our knowledge, the first system to take the rule-based environment concept out of the Lisp-based AI domain and apply it in a more conventional programming context. The Version 0 prototype also served to point out several areas in the rule language notation that needed work, and thus began the development path that led to today's MARVEL environment.

5. Scope

The problems that we are attacking are those of providing a broadly applicable process engine for a range of languages and environments. The scope of the thesis is to be limited so as to exclude cases falling outside of the range and issues which, while they may be important in real-world software development environments, are not essential to the basic problem of specifying a development process.

5.1. Declarative Language Restriction

This work is intended to model only declarative process modeling languages; procedural languages such as APPL/A [Sutton 90] are explicitly omitted from consideration as basically incompatible with the rule-based execution model of our process interpreter.

5.2. Not a Generator System

This work will not be carried through to the level of refinement and integration necessary to build a complete generator system for rule-based process servers. Such a generator system would allow the entire system instance to be specified declaratively, or would provide specific problem-oriented languages for each aspect of the system that could not be specified declaratively. In contrast, it will be necessary in our framework to write significant amounts of code in a conventional language (C or C++) as part of the specification. This restriction will be made to keep the size and length of the project within reasonable bounds for a Ph.D. thesis.

5.3. Multi-User Issues

Multi-user concurrency control issues are explicitly outside the scope of this thesis; these were the topic of Barghouti's thesis [Barghouti 92]. Also, another student on the MARVEL project is working in this area and is likely to choose as his thesis topic a general concurrency control tool for software development environments. I do not intend to duplicate his work.

Multi-user synchronization issues, on the other hand, are explicitly inside the scope of this thesis. The line is to be drawn between synchronization constructs such as messages, semaphores and rendezvous, and concurrency control constructs such as object locking and transactions.

5.4. No Schema Evolution, Process Evolution or Views

This work will consider the enactment of policies in environments; in order to cover the core issues in the necessary depth, we must consider some deep but peripheral (to the proposed work) issues outside of our scope. These issues specifically include schema evolution, both of the data in the environment and of its description in the policy language understood by the kernel. Barghouti and Kaiser have designed a process evolution facility for MARVEL [Barghouti 91], much of which should remain applicable to RBPS environments. Similarly, we must also consider the issues of schema evolution to be outside of our scope.

6. Plan

Our plan for this thesis was summarized in Section 3.4. This section will provide additional detail on the steps in the plan, as well as discussing the means by which we may evaluate the results.

Once we have characterized our areas of applicability, the first step in the research will be to define the language and environment interfaces that are to allow the proposed kernel rule engine to handle a generalized notion of a rule. This notion should be sufficiently general to include, in addition to conventional condition-action rules, MARVEL-like rules with activities, Darwin-like laws (which resemble Prolog in syntax), and other rule-like constructs such as Postgres database triggers. In Postgres, these triggers are termed "rules"; we use a different name to call attention to their event-driven, rather than data-driven, nature.

At that point, we can implement the kernel, making such changes to the interfaces as may be proved necessary in the process. In order to test the kernel, we will also need to implement enough of the language and environment interface facilities to specify an environment similar to MARVEL. Since MARVEL was originally named after Professor Marvel, the "man behind the curtain" in *The Wizard of Oz*, this generated environment is to be called Oz.

Naturally, this implementation effort will feed back into the kernel and its interfaces; we expect the actual development to be an iterative, "spiral" process rather than the linear, "waterfall" process that we present here as a model. This portion of the implementation is conceptually equivalent to restructuring our existing system to use our new model; due to the generalization of the rule model, however, we do not foresee much of the existing code surviving into the new RBPS. Furthermore, we are likely to do this reimplementation in an object-oriented language, C++, rather than, as MARVEL is coded, in C. The Oz implementation will demonstrate our process server's applicability to an environment that we already understand well.

Following this initial implementation, our goal will be to show the broad applicability of our solution by interfacing the kernel to at least two other languages and two other environments, with at least one language and one environment being treated in depth to yield a "usable" environment capable of implementing a solution to the ISPW7 benchmark problem [Thomas 91], and the others more superficially as prototypes meant simply to discover flaws in the design. Since the ISPW7 problem is regarded by the software process modeling community as the only existing benchmark for the capabilities of process languages, solving it will show, in some sense, the "completeness" of the languages that can be implemented using the RBPS. We will most likely take advantage of our existing experience by using Field as one of the environments and an extended ASL as one of the languages.

One problem we will encounter at this stage is that very few environments are actually available for our use in testing. We have installed and used Field at Columbia, and of course MARVEL and ASL were developed here, so there will be no problem with using them. But there are essentially no other systems to which we are certain that we will have access. The problem is twofold. First, many of the existing process languages have never been implemented, having been developed as theoretical exercises. Therefore, it may conceivably turn out that some of them are unimplementable. Second, there is more work being done on SDE's in the industrial research community than in the academic community. Many of the target environments we might consider are unavailable to us because they are proprietary to their developers.

Nevertheless, we do have some plans for making use of what is available. We have already mentioned our plans to implement translators for the MSL and ASL languages; another possibility is to implement the Prolog-based specification language used in Darwin, although the actual Darwin system is unavailable to us. This would raise questions regarding the generality of our "assembly language", since Prolog's chaining model involves backtracking and the MSL chaining model, although it includes backward chaining, has no backtracking. It is more likely, though, that we will implement the Oikos [Ambriola 90] specification language, since we should be able to obtain that system. Other possibilities are the Merlin [Schaefer 92] rule-based environment and the Melmac Petri net-based environment.

As for target environments, we have already mentioned Field (or FUSE, which is a commercially available version of Field) and MARVEL; other possibilities are CLF and Oikos. Such process-based environments, however, may have notions of process that conflict with ours and would complicate our work. Also, our main objective in this thesis is to add process capabilities to non-process-based environments, so we would prefer to use only such environments as our targets.

Once we have shown that our solution does indeed solve the intended problem, we will also have to show that it is a reasonably efficient solution. There are several aspects to efficiency in policy tools, dealing with the process of writing in the specification languages and compiling them into rules, and with the actual execution of the rules.

The main performance-related topic that we will have to address with regard to specification languages is the trade-off between the ease of writing policies and the amount of overhead in the generated rule set. The question here is, how well does our rule-based "assembly language" work as a target language for specification language compilers, and how much slop, *e.g.*, additional rules as compared to the same process coded by hand in the rule-based "assembly language", are compilers forced to generate? It could also be productive to consider the same policy written in multiple higher-level specification languages and compiled into the rule-based language. This sort of comparison could give valuable insights into how much overhead is inherent in translation to the rule-based language, and how much might be due to a particular specification language being a bad match for the target language.

In addition to the static overhead of additional rules generated by the compiler, we should also consider the dynamic overhead of executing the rules. Some criteria that might be used here are the number of additional rules actually fired in an application, versus the number that would be run if the results of tool execution were to be known in advance, and the number of conditions evaluated under the same circumstances. Note that the number of conditions evaluated is not necessarily the same as the number of rules fired, because conditions that are not satisfied will not result in the firing of a rule. Another aspect of this measure could also prove useful: The number of conditions evaluated in an application, versus the number that must be evaluated (assuming again the existence of an oracle for determining the results of evaluation).

Another important issue is how well a rule-based policy tool will scale up to larger environments containing, *e.g.*, large numbers of tools and rules (and more complex rules), as well as large software databases. We should also investigate the effect on performance of adding policies, tools, rules, and conditions to a process specification, to determine which aspects of RBPS might have unacceptable asymptotic performance if scaled up to a large environment.

The preceding paragraphs dealt with the static and dynamic overhead of having a rule-based policy tool; we should also consider the overhead due to the policy tool itself, as opposed to the rule model. We expect this overhead, basically the cost of passing messages between the "ordinary" tools and the policy tool, to be relatively small. However, it is important to be able to factor this effect out of the evaluations we have discussed above.

7. Contributions

The main contribution of the proposed research will be to provide a framework for specifying both a development process (without unduly restricting the form of the specification) and its integration with an existing SDE, in such a way that process concerns are kept separate from environment concerns. In order to demonstrate that the research actually solves the proposed problem, we plan to implement the framework and make use of it in test cases involving multiple policy languages and multiple environments. Although we will restrict our application domain to software development environments, the process server approach should be useful in other sorts of environments where enactment of a process or enforcement of a policy is desired. For example, the rule-based framework for tool application could be useful in computer-aided design environments for other sorts of engineering. Such a framework might also be able to guide the application of tools in a network management environment.

Also, as part of our evaluation of the research, we plan to develop means and measures for evaluation of environments and of large-grain rule-based systems such as the proposed process server. These are expected to differ significantly from the criteria for evaluating smaller-grain rule-based systems.

References

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

[Barghouti 88] Naser S. Barghouti and Gail E. Kaiser.
Implementation of a Knowledge-Based Programming Environment.
In *21st Annual Hawaii International Conference on System Sciences*, pages 54-63. IEEE Computer Society, Kona HI, January, 1988.

[Barghouti 90a] Naser S. Barghouti and Gail E. Kaiser.
Multi-Agent Rule-Based Software Development Environments.
In *5th Annual Knowledge-Based Software Assistant Conference*, pages 375-387. Syracuse NY, September, 1990.

[Barghouti 90b] 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 A. van Lamsweerde and A. Fugetta (editor), *3rd European Software Engineering Conference*, pages 380-395. Springer-Verlag, Milano, Italy, October, 1991.

[Barghouti 92] Naser S. Barghouti.
Concurrency Control in Rule-Based Software Development Environments.
PhD thesis, Columbia University, February, 1992.
CUCS-001-92.

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

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

[Clocksin 81] W.F. Clocksin and C.S. Mellish.
Programming in Prolog.
Springer-Verlag, Berlin, 1981.

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

[Dowson 87] Mark Dowson.
Integrated Project Support with IStar.
IEEE Software 4(6):6-15, November, 1987.

[Garlan 90] David Garlan and Ehsan Ilias.
Low-cost, Adaptable Tool Integration Policies for Integrated Environments.
In Richard N. Taylor (editor), *SIGSOFT '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 1-10. ACM Press, Irvine CA, December, 1990.
Special issue of *Software Engineering Notes*, 15(6), December 1990.

[Gisi 91] Mark A. Gisi and Gail E. Kaiser.
Extending A Tool Integration Language.
In *1st International Conference on the Software Process*, pages 218-227. Redondo Beach CA, October, 1991.

[Gruhn 91] Volker Gruhn.
Validation and Verification of Software Process Models.
PhD thesis, Universitat Dortmund, June, 1991.

[Honda 88] M. Honda.
Support for Parallel Development in the Sun Network Software Environment.
In *2nd International Workshop on Computer-Aided Software Engineering*, pages 5-5 - 5-7. 1988.

[Kaiser 87] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *32nd IEEE Computer Society International Conference*, pages 236-241. IEEE Computer Society Press, San Francisco CA, February, 1987.

[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 91a] Gail E. Kaiser.
A Rule-based Process Server Component for Constructing Rule-based Development Environments.
In Iam Thomas (editor), *7th International Software Process Workshop: Support for the Software Process*. IEEE Computer Society Press, Yountville, California, October, 1991.
Position paper. In press.

[Kaiser 91b] Gail E. Kaiser, Israel Z. Ben-Shaul and Steven S. Popovich.
Implementing Activity Structures Process Modeling On Top Of The MARVEL Environment Kernel.
Technical Report CUCS-027-91, Columbia University, September, 1991.

[Katayama 90] Katayama, T. (editor).
Sixth International Software Process Workshop: Support for the Software Process.
IEEE Computer Society Press, Hakodate, Hokkaido, Japan, 1990.

[Laird 86] J. E. Laird.
Soar User's Manual
Fourth edition, Xerox PARC, 1986.

[Leblang 84] David B. Leblang and Robert P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. ACM Press, April, 1984.
Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[Minsky 88] Naftaly H. Minsky and David Rozenshtein.
A Software Development Environment for Law-Governed Systems.
In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65-75. ACM Press, Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[Osterweil 87] Leon Osterweil.
Software Processes are Software Too.
In *9th International Conference on Software Engineering*, pages 1-13. IEEE Computer Society, Monterey CA, March, 1987.

[Perry 89a] Perry, D. (editor).
Fifth International Software Process Workshop.
ACM Press, Kennebunkport, ME, 1989.

[Perry 89b] Dewayne E. Perry.
The Inscape Environment.
In *11th International Conference on Software Engineering*, pages 2-9. IEEE Computer Society, Pittsburgh PA, May, 1989.

[Peuschel 90] B. Peuschel, W. Schafer, S. Wolf.
A Flexible Environment Architecture as a Basis for Distributed Software Development.
In Takuya Katayama (editor), *6th International Software Process Workshop: Support for the Software Process*. IEEE Computer Society Press, Hakodate, Hokkaido, Japan, October, 1990.
Position paper.

[Reasoning 86] *Refine Software Development Tool*
Reasoning Systems, Palo Alto CA, 1986.

[Reiss 90] Steven P. Reiss.
Connecting Tools Using Message Passing in the Field Environment.
IEEE Software 7(4):57-66, July, 1990.

[Riddle 91a] William E. Riddle.
Private Communication.
April, 1991

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

[Rosenblum 91] David S. Rosenblum and Balachander Krishnamurthy.
An Event-Based Model of Software Configuration Management.
In Peter H. Feiler (editor), *3rd International Workshop on Software Configuration Management*, pages 94-97. ACM Press, June, 1991.

[Schaefer 92] Wilhelm Schaefer and Burkhard Peuschel and Stefan Wolf.
A Knowledge-based Software Development Environment Supporting Cooperative Work.
International Journal on Software Engineering & Knowledge Engineering , March, 1992.
In press.

[Stonebraker 89] Michael Stonebraker, Eric N. Hanson and Spyros Potamianos.
The POSTGRES Rule Manager.
IEEE Transactions on Software Engineering 14(7):897-907, July, 1989.

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

[Thomas 91] Thomas, I. (editor).
Seventh International Software Process Workshop: Support for the Software Process.
IEEE Computer Society Press, Yountville, California, 1991.
In press.