Experience with Process Modeling in the Marvel Software Development Environment Kernel

Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027
212-854-3856
Kaiser@cs.columbia.edu

CUCS-446-89

10 July 1989

Abstract

We have been working for several years on rule-based process modeling and the implementation of such models as part of the foundation for software development environments. We have defined a kernel, called MARVEL, for such an architecture and implemented several successive versions of the kernel and several small environments using the kernel. We have evaluated our results to date, and discovered several significant flaws and delineated several important open problems. Although the details are specific to rule-based process modeling, we believe that our insights will be valuable to other researchers and developers contemplating process modeling mechanisms.

Copyright © 1989 Gail E. Kaiser

Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

keywords: controlled automation, rule-based systems, software development environments

1. Introduction

The long-term goal of the MARVEL project is to investigate *rule-based process modeling* as the basis for an architecture for design and engineering environments, particularly software development environments. This is an instance of the "process programming" paradigm proposed by Osterweil [Osterweil 87], although we began working on this problem (June 1986) about nine months before learning of his work (March 1987, at the ICSE conference). The gist of "process programming" is (1) to define software processes (the design process, coding, testing, and so on) in a formalism well-understood by software engineers — a programming language and (2) take advantage of well-understood programming language implementation techniques to "enact" (carry out, automate, or otherwise implement) as much as possible of the software process.

We prefer to refer to our work as process modeling, although we do interpret the process model for what we call *controlled automation* of the software process. The distinction is that process programming (unfortunately) implies to many researchers use of an imperative, sequential programming language, which seems obviously unsuitable for complex, many-activities-in-parallel, changing-in-mid-stream software processes. We use instead a rule-based language, where multiple rule sets can be employed separately or together and the currently in force rules can be changed dynamically by the software team or by the rules themselves. Solving the concurrency control problem is one of our major concerns for future work.

The gist of our rule-based process model is that each software activity is defined by a rule, consisting of three parts. The first part, the *precondition*, is a logical expression on the state of the project including the contents and attributes of any software objects. The precondition must be true in order to carry out the activity. The second part is the activity itself, which may be the invocation of a tool, or a description of some real-world activity to be performed off-line, or the selection of another rule set defining the activities for the next phase of the process or for a hierarchical subtask in the process.

The third part is one or more *postconditions*, each a logical assertion on the objectbase representing the software effort. When the activity is completed, exactly one of the postconditions becomes true, which one determined by the actual results of the activity. Multiple postconditions are needed to reflect the different possible results of software activities in the many cases where the correct direction can only be determined while the activity is in

progress, and thus cannot be pushed into the precondition. Three parts rather than the classical two (condition/action) are necessary to treat the activity part as a "black box" that has the effects given by one of the postconditions on the working memory used for matching the preconditions. This black box nature is necessary for activities involving commercial off-the-shelf (COTS) tools, where modification of the tool is impossible (or at least avoided), as well as for off-line activities.

Controlled automation is achieved by what we call *opportunistic processing* on the rules, employing backward chaining and forward chaining as the opportunity arises to automatically initiate activities. Other forms of controlled automation include monitoring user activities — keeping a record and determining whether or not they conform to the rules, or using the preconditions as constraints that the software development team must fulfill before moving on, or as a way of leading the team by the hand through the various stages of the particular process envisioned (or required) for that software project.

We have defined a kernel for software development environments, called MARVEL, based on our design for rule-based process modeling and controlled automation. Our work on MARVEL has been published widely [Kaiser 87a, Feiler 87, Barghouti 88a, Kaiser 88a, Kaiser 88b, Kaiser 88c, Kaiser 89]. Several versions of MARVEL have been implemented, and a small environment for C programming (C/Marvel [Barghouti 88b]) and a small environment for documentation production (see appendix) have been developed. Version 2.01 and the C/Marvel environment were demonstrated at the ACM SIGSOFT Practical Software Development Environments conference in November 1988. The latest version, 2.10, consists of approximately 35,000 lines of C code, 1000 lines of Yacc rules and subroutines and 300 lines of Lex rules. MARVEL runs on 4.2 and 4.3 Berkeley Unix™, provides an X11 windows graphical user interface and uses any Unix shell language as its "envelope language" for interfacing between external tools and MARVEL's objectbase.

MARVEL's object management system is "quick-and-dirty": The object-oriented data model is rather simple and does not support methods except for the special case of tools, all relationships among objects are maintained in memory (but this data structure is checkpointed before each substantial change), objects are mapped directly to the Unix file system in the obvious manner, and object clustering is by alphabetical order. The Marvelizer and Organ tools have been developed to immigrate existing software systems from the file system into the

objectbase and to reorganize the components of software systems within an objectbase, respectively [Sokolsky 89]. MARVEL itself has been Marvelized into a C/Marvel environment; this takes about ten minutes. The primary missing facility is schema evolution, a very hard problem.

So far we have investigated only a small fraction of the potential for rule-based process modeling and controlled automation, and their application to MARVEL. There is much more work to do to fully understand how rule-based process modeling can and should work, how controlled automation can and should work, how it can be applied to multiple users sharing an objectbase with perhaps different rule sets, and whether the combination is indeed a good foundation for a software development environment architecture. In the following sections, we describe and evaluate our rule-based process modeling language, describe its current implementation for controlled automation and open problems, and briefly discuss related work.

2. Rule-Based Process Modeling

```
name [?id:type, parameters...] :
    precondition
    { activity }
    postcondition;
    postconditions...
```

Figure 2-1: Generic Rule

We have defined a rule language, the Marvel Strategy Language (MSL). A rule consists of a name, typed parameters and a body, as shown in Figure 2-1. The body consists of a precondition, an activity, and one or more postconditions.

```
tool operation argument, arguments...
```

Figure 2-2: Generic Activity

The activity sends a message to a tool object to execute one of its operations, as depicted in Figure 2-2. The tool object must be declared either in the same strategy (module) as the rule, or exported by one of the strategies imported by the containing strategy. The tool object must have an operation (method) that corresponds to the operation named in the message from the activity. Thus activities are currently restricted to simple tool invocations. It is not yet possible to invoke

arbitrary MARVEL commands, most notably load or unload a strategy, for example, to automatically change the set of currently active rules under certain conditions. Although this would be relatively easy to add to the implementation, the implications are unclear. One major problem would be if there was a conflict, such as two exported rules with the same name but different activities, between the strategy to be loaded and an already loaded strategy. It would be easy enough to have the production of error messages as one of the alternative postconditions, but it is unlikely that these messages would mean anything to a typical user. (We assume one or more ''superusers'' who develop a library of strategies, which can be made available to users via an information retrieval facility [Maarek 87].) A general exception handling facility will be necessary to deal with opportunistic processing results that typical users are not prepared to handle.

It will be much more difficult to add support for hierarchical and/or off-line activities, since we do not yet fully understand what is needed or how to represent such activities within MSL. Yet both hierarchical breakdown of software development tasks and representation of off-line activities such as design meetings are necessary to extend MARVEL beyond our current experimentation with edit/compile/debug and document production activities.

2.1. Preconditions and Postconditions

The precondition of a rule specifies the logical condition that has to be satisfied before the activity can be initiated. This logical condition can either be a simple predicate or it could be a complex clause (either disjunctive or conjunctive). A precondition clause consists of three parts. The first lists existential and universal quantifiers, together with a *characteristic function* that characterizes any or all of the quantified variables. The third part is a property list that has to be satisfied for the precondition to be true. The easiest way to think about what this means is in terms of sets. The quantified variables together with the characteristic function selects a set of objects that meet this characteristic; this is done for efficiency purposes, since the characteristic set is typically represented explicitly in the objectbase as an aggregate attribute of some object.

Then, all the predicates in the property list are matched against each element of the set. Each predicate is an expression that tests the value of an attribute of an object with respect to a literal value, another attribute or a special value such as the current system clock (NOW) or the userid of the current user (USER). Set expressions either test for membership of a particular element in a set, or add or delete an element from a set. The member operation is usually used in the

preconditions of rules while the add and remove operations are used in the postconditions of rules. A relation expression works on user-defined binary relations between objects.

Each rule has mutually exclusive postconditions, one of which is asserted after the activity part of the rule is executed. Postconditions modify the attributes of objects. The choice of which postcondition to assert depends on the result of the activity, and is selected by the envelope that interfaces the external tool to MARVEL.

```
rules

    edit [?p: PROCEDURE]:
        suchthat
        :

        { EDITOR edit ?p }

        (?p.edited = Edited);

    compile [?m: MODULE]:
         forall PROCEDURE ?p
         suchthat
         (member [?m.procs ?p])
         :
             and((?p.analyzed = Analyzed)
                 (?m.status = ModNoComp))

         { COMPILER compile ?m }

         (?m.status = ModIsComp);
         (?m.status = ModNoComp);
```

Figure 2-3: Two Example Rules from C/Marvel

Probably the simplest examples are the edit and compile rules shown in Figure 2-3. Additional rules are given in the appendix. The first rule states that editing a component results in updating its timestamp. There is no precondition for editing, since if the component does not exist the editor creates it. The second rule states that the COMPILER can/should be applied to a MODULE object if all its PROCEDURE objects have been analyzed and the module has not yet been compiled. The result of the COMPILER activity could be either error messages or successful compilation, reflected by the status variable; which one can be determined only by running the compiler.

The language facilities available for use in preconditions and postconditions are currently rather trivial: all preconditions and postconditions that can currently be expressed are analogous to those in this example. We would like to express more general queries in preconditions, and be able to express quantifiers and more complicated assignments in postconditions.

2.2. Strategies

The complete description of a target project is captured in a collection of interacting units in a manner similar to modules in a conventional programming language. The unit of modularity of MSL is called a strategy. A single strategy might provide only a partial view of the target project. Several strategies are merged to provide a complete description of a project. The intent is that different collections of strategies will support different phases of the software lifecycle, different user roles, different tool sets, but that some subset of the strategies are likely to shared among these. For example, early development versus maintenance might have different rules that specify under what circumstances source code can be modified and what happens when it is, but the same editor and compiler are likely to be used even though maintenance might require additional tools such as a bug tracking system. Alternative strategies that share imported strategies are illustrated in the appendix.

Each strategy has a name and consists of five parts.

- Interface in terms of exports and imports,
- Classes the describe the structure of the components of a software project,
- Tool objects in the form of subclasses of the built-in TOOL class,
- Relations between classes, and
- Rules the describe the desired behavior of the environment during the process of developing the target project.

A complete template for a strategy is given in Figure 2-4, and there are some small but complete examples in the appendix.

Multiple strategies loaded at the same time are *merged*. For example, classes with the same name are combined in the style of multiple inheritance [Cardelli 84], so the resulting class used by MARVEL defines the union of attributes from all the contributing classes from different strategies. Conflicts such as separately inherited attributes with the same name but different types are detected and the merge is disallowed. Conflict detection could be done in advance by precomputing which sets of strategies in a common library are compatible with each other, but

```
STRATEGY: name

Interface:
   Imports: list of imported strategies;
   Exports: list of exported classes,
            tools, relations and rules;

Objectbase:

      class_name ::= superclasses: list of superclasses:
          Attributes:
            attribute_name: type = default value if any;
            ...
      END

      ...

      tool_name:: superclasses: TOOL:
          operations:
                operation_name : string = envelope_file_name;
                ...
      END

      ...

Relations:

     relation_name : domain_class range_class;
     ...

Rules

     rule_name [ list of parameters ]:
           precondition
           { activity }
           list of postconditions

     ...

End name
```

Figure 2-4: Generic Strategy

that would prevent users from developing and adding their own strategies on the fly — this is currently supported but is probably not a good idea for anyone except a "wizard" user, because rule sets are difficult to debug as discussed later on.

Multiple rules with the same name are combined by ANDing the preconditions, which makes sense and usually works; originally MARVEL also XORed the postconditions, which may make sense but does not really work. The problem is the envelope checks the results of external tools to select among the postconditions, so the envelope must know about all the postconditions. Since envelopes are currently imperative (shell scripts), it is not possible for MARVEL to ''merge'' them. One longer-term goal is to develop a real envelope language, originally imperative but hiding the details of MARVEL's objectbase implementation, and eventually declarative.

As yet we have done relatively little investigation of strategies. As mentioned previously, we would like to permit loading and unloading of strategies as activities that could be defined by rules. Loading a strategy (which can currently be initiated only by the human user invoking the load command) operates by first merging all the strategies reachable by import chains, and then merging the result with the already loaded set of strategies (perhaps none). MARVEL detects minor naming conflicts, but does not carry out any inferencing to determine non-trivial logical conflicts between rules. We would like to be able to mix and match strategies, including support for different tools operating on different objects but triggering each others' activation and bridging between different tools that should be operating on the same objects. To achieve this, we must develop some way to express logical relationships such as implication (one property implies another or the negation of another) and equivalence (two properties defined using different terminology in different strategies really mean the same thing) and be able to do inferencing on these relationships.

3. Controlled Automation

We have designed and implemented a kernel for software development environments that interprets software processes defined by rules. The basic mode of operation is as follows.

When the user enters a command corresponding to the name of a rule, a component of MARVEL called the Opportunist checks the truth of the precondition against working memory (the objectbase representing the software system under development). If the precondition is satisfied, then the Opportunist sends the message defined in the activity part of the rule. If the precondition is not currently satisfied, the Opportunist performs backward chaining to attempt to satisfy the precondition. Once the precondition is satisfied, the activity is executed, and one of its postconditions asserted. The Opportunist then performs forward chaining to fire any rules

whose preconditions have now been satisfied, under the assumption that their results will be required later on.

The Opportunist follows the AND-OR tree mechanism of other backward chaining systems, with one significant exception. Since rules may have multiple alternative postconditions, it is possible for the precondition of the rule to be true but firing the rule does not produce the necessary postcondition. One way to deal with this would be to simulate firing such rules, and commit the results with respect to working memory only if the desired postcondition is achieved. But this is unrealistic in a context where executing the action (activity) component of the rule may take an arbitrary period of time — a few minutes for most processing tools such as compilers, and perhaps hours for interactive tools such as editors. Furthermore, access to the undesired results of firing the rule is often necessary to eventually produce the desired result; consider the case where the desired result of a compile rule is correct object code but the undesired result produced was error messages. Therefore, all rules applied during backward chaining are not simulated, but instead have ''permanent'' effects.

This leads to another problem: When there are alternatives, it is conceivable that firing one alternate first, and getting an undesired result, could negate the precondition of another alternative, but if this second alternative had been fired first it would have resulted in the desired result. The answer may be using planning to order consideration of alternatives.

The Opportunist also carries out forward chaining in a slightly different manner than most forward chaining systems, in that the conflict resolution strategy is to fire <u>all</u> the rules whose preconditions are satisfied. This makes sense given our assumption that all the results will eventually be needed. There is again the question of ordering, since the postcondition of one rule may negate the precondition of another rule that otherwise would have fired.

Another concern with both forward and backward chaining involves the different degrees of control appropriate for opportunistic processing and for the human user. If the human user wants to compile a module, or edit a file, under most circumstances the Opportunist should let him. However, factors such as load average on the machine and whether the user is likely to make further changes that will require recompilation of the same module may make it undesirable for MARVEL to automatically compile the module.

Part of the control problem can be solved by what we call *hints*, which are additional preconditions that apply only during forward chaining and thus do not affect activities initiated

by the human user. Hints as such have already been designed, and could easily be added to the implementation. However, we need more experience using MARVEL to determine what kinds of hints are most useful in practice, and what kinds of language facilities are needed to express them.

Another aspect of our solution is *implicit queries*, essentially a capability for self-reflection. The idea is that at each step in forward or backward chaining, the Opportunist would ask itself questions such as "how long is executing this activity likely to take?", "how many objects is it likely to read or write?", "does this activity require human intervention?", and so forth. When an activity is too expensive or otherwise questionable, the Opportunist could request confirmation from the human user or apply some meta-rule to decide what to do. It would be easy to hardcode special cases, such as the first two questions, user-specified threshold values and user confirmation, but we would prefer to allow such implicit queries to be associated separately with each rule and/or each tool, and meta-rules to be associated with strategies to account for the specific semantics of the modeled software process. We will need to add suitable primitives to MSL, both to extend the query language (used in the preconditions of rules and in the future to become available for ad hoc queries from the user interface) and to express meta-rules.

There are certain difficulties with the Opportunist automatically initiating an interactive activity such as editing. It is not clear when it is appropriate to automatically activate an interactive tool, and determining this would involve human-computer interactions research outside the scope of this project. Assuming we allow the strategy implementor to decide this, there is the very hard problem of capturing the full postconditions of an interactive tool. For example, a user can do almost anything from within the Emacs text editor [Stallman 81] and nothing short of kernel modifications (which we do NOT plan to undertake!) seems capable of detecting (or preventing) all unanticipated side-effects of black box tools.

3.1. Multiagent Rule Systems

There is another problem with interactions among rules more significant than incompatible attribute naming, ordering alternatives and controlling runaway opportunism. We would like forward chaining to go on in one or more background processes, permitting the human user of MARVEL to continue carrying out software development activities in the foreground. This was implemented in the first version of the MARVEL kernel. Unfortunately, it did not work — because we did not then have a sufficient understanding of what we now call the "multiagent

problem''.

Classical rule systems are designed for one agent to be modifying working memory at a time: either the human user adds and deletes objects, or the forward and/or backward chaining system fires rules that add and delete objects. We know of no rule system that permits both to proceed in parallel. Even parallel rule systems, such as Stolfo's [Stolfo 84] and Gupta's [Gupta 86] work on parallelizing OPS5, assume and in fact require independence of the parallel rules. Multiple rules may modify working memory concurrently, but the writesets are guaranteed to be disjoint from each other and from the readsets of parallel chains.

In addition to supporting multiple agents, where only one is the human user and the rest are rule chains, it will be necessary to extend MARVEL to support multiple human users cooperating to develop/maintain the same software system. This compounds the problem, since in the first case at least the background chains are operating on behalf of the human user and we might pragmatically leave sorting out any inconsistencies that might arise to this human, but this option is not acceptable when there are multiple human users whose activities affect the same objects. Partitioning the objectbase among the humans might work for those objects representing source code, object code, documentation, and so on, but cannot for those objects reflecting the global status of the project and other shared information.

3.2. Match Algorithms for Controlled Automation

In the current implementation, the performance of both backward and forward chaining is improved by building two tables of potential chaining ''links'' when loading (unloading) strategies. The two tables are the activity table, which has an entry for each rule keyed by activity, and the predicate table, which has an entry for each predicate or assertion that appears in a precondition or a postcondition respectively. Each entry of the activity table contains the bindings of quantifiers in the corresponding rule, and pointers to the precondition and postconditions in the predicate table. The bindings associate a type with each variable used in the activity of the rule. Each entry of the predicate table stores a list of pointers to all activities whose postconditions might satisfy it (backward chaining) and another list of pointers to all rules whose preconditions might be satisfied if this predicate becomes true (forward chaining). In other words, MARVEL detects all potential chaining when it first loads a strategy and merges it with active strategies. This enables the Opportunist to quickly scan the rule base during operation to decide which rule(s) can be fired. The figure below shows how the rules given

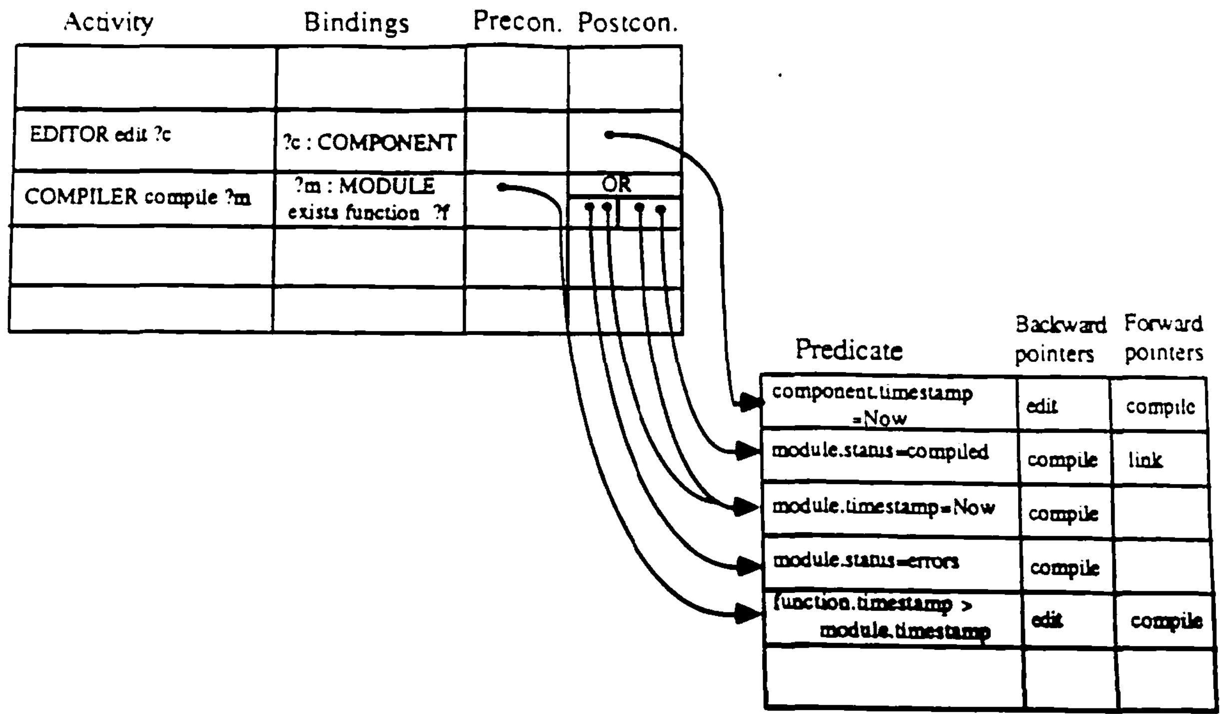

Figure 3-1: Example Chaining Tables

above are stored in the two tables. Example tables are shown in Figure 3-1.

The obvious question is why do we not use the Rete algorithm [Forgy 82] or some parallel variant such as Treat [Miranker 87]. The reason is that existing match algorithms assume a relatively large number of rules, a relatively small number of objects, certain restrictions on quantifiers, and that it is feasible to treat the modification of an object as a delete followed by an add. In contrast, for rule-based process modeling we anticipate a relatively small number of rules (most likely no more than $O(N^2)$ for N tools) and a relatively large number of objects (e.g., each procedure in a software system might be represented as a distinct object). Many rules are of the form "forall members of the characteristic set, such-and-such properties must be true (or false)". Objects are too large and complex to delete and recreate, so the match algorithm has to permit them to be modified in place; this is relatively minor difficulty compared to the rule/object ratio, since Rete could easily be extending to keep two way links between objects and to permit modified objects to be unlinked.

We are not satisfied, however, with our (admittedly trivial) match network. The first is it does not yet provide any special support for quantifiers. We have designed a simple extension to handle this for the special case where the characteristic set is an aggregate attribute of an object, which has been typical for the rules we have developed so far. For each precondition that could be applied to the aggregate, a precondition/count pair is maintained with the object to keep track of how many members of the aggregate currently satisfy the precondition's clauses. This will make precondition queries involving quantifiers much faster than currently, since keeping the count up to date will be amortized over all modifications to objects in the aggregate.

The second problem is really part of solving the multiagent problem. Our current match network does not contain pointers to objects, so there would be no problem with each user keeping his own copy reflecting the rules defined by his currently loaded strategies, even though multiple users (and their rules) may operate on the same objectbase concurrently. Keeping the counts in the objects also works fine for multiple users, although some form of concurrency control will be needed for accessing the counts — but as discussed above concurrency control will be needed in any case for the shared objectbase. But a more efficient match algorithm tuned to the problems of controlled automation must be developed in concert with our solution to the multiagent problem, for example, we anticipate that a concurrent data structure will be necessary.

4. Related Work

The first version of MARVEL was implemented by modifying the Smile programming environment [Kaiser 87b] developed as part of the Gandalf project [Habermann 86] at Carnegie Mellon University starting in 1979 and continuing through the early 1980's (the current MARVEL implementation is entirely independent of Smile). Smile supports multiple users programming in C and runs on Unix. Smile has been relied on by the Gandalf and Gnome [Garlan 84] projects at CMU and by the Inscape project [Perry 89] at AT&T Bell Labs, and has been distributed to at least forty sites. Smile passes the crucial test of supporting its own maintenance. It has supported the simultaneous activities of at least seven programmers, and the largest software system developed and maintained in Smile has approximately 61,000 lines of source code.

Smile provides a "fileless environment" to its users, answers simple queries, coordinates the activities of multiple programmers, and automatically invokes various tools under certain conditions; it hides the particularities of the Unix file system and utilities and presents its own

model of the programming world. Smile's objectbase is implemented through a combination of the file system and an in-core object structure that is kept persistent in a file. Smile's knowledge of software objects and the programming process is hardcoded into the environment, and cannot be changed except by modifying the code. From experience with Smile and other environments we gained insights into the development of practical environments and became convinced that a generalization of Smile's internal architecture would be a good basis for a software development environment architecture. Unfortunately, Smile's support for multiple programmers is overly restrictive — for example, a programmer has to exclusively lock all modules that import his module in order to modify the interface of his module — and thus was not adapted for MARVEL.

The ongoing research projects closest to MARVEL include the Formalized System Development project [Balzer 85] being developed by Balzer's group at ISI (the distributed version of the system is called the CommonLisp Framework (CLF) [CLF 88]), Refine [Smith 85] developed by Kestrel Institute and marketed by Reasoning Systems, Darwin [Minsky 88] being developed by Minsky and Rozenshtein at Rutgers University, Grapple [Huff 88] developed by Huff and Lesser at University of Massachusetts, and Arcadia [Taylor 88] being developed by a consortium including the University of Massachusetts, University of Colorado, University of California at Irvine, TRW and other institutions.

CLF is the direct intellectual ancestor of MARVEL, dating back to our original work in 1986 at the CMU Software Engineering Institute. The motivation for initiating the Marvel project was to develop a rule-based programming environment kernel similar to CLF that (1) operated in the Unix rather than the Lisp world, (2) provided controlled automation of existing stand-alone tools and (3) supported multiple users cooperating on a software project. We have achieved the first two goals, although improvements are needed as discussed in this paper. CLF is limited by its CommonLisp implementation, use of the AP5 specification language [Cohen 86] for rules, forward chaining architecture and single-user emphasis. Furthermore, the FSD project is more concerned with executable specification languages and rapid prototyping than with controlled automation. Refine is primarily an automatic transformation system, for the purpose of program synthesis, although it also provides a limited form of controlled automation in the style of CLF. In contrast, we are not interested in trying to automate the creative aspects of software development, since that is the proper realm of AI research, but instead offload menial chores onto the software development environment.

Darwin is a rule-based system closer to Prolog, while MARVEL is closer to OPS5. Darwin restricts what programmers can do by treating rules as constraints, which is also a reasonable extension of MARVEL as mentioned in the introduction, but does not automate activities. It is also limited by its Prolog implementation, and cannot handle "black box" activities. Grapple's rules are closest in form to MARVEL's, but interpreted solely for planning and plan recognition, with the goal of new insights into AI. Arcadia focuses on process programming as the basis for a software development architecture, but as yet is considering neither rule-based process modeling nor concurrency control for multiple users. Their focus is on object management systems and user interfaces suitable for software development activities, as well as investigating the process programming paradigm in general.

5. Discussion

We have discovered several difficulties with the MARVEL paradigm that seem likely to apply at least partially to implementation of other process modeling notations. First, constructing and debugging a process model is hard. We have run into many unanticipated interactions among rules, particularly among those defined in different strategies that happen to be loaded into the MARVEL kernel at the same time. We are working on debugging facilities that will uncover all direct and transitive dependencies by constructing a graphical dataflow graph of the currently active rules. We would like to permit the user to enable and disable individual rules, both during debugging to see how that affects the dataflow among rules and also during operation because certain behavior is not currently desired. This would be dangerous, however, if done by a naive user: if rules are turned off and on at arbitrary points, it is possible for MARVEL to get into what is an inconsistent state with respect to the full set of rules in such a way that automatic recovery is impossible.

Similar difficulties seem likely to arise for any executable process modeling paradigm. The following questions need to be answered: How does an implementor go about writing, testing and debugging a process model? For non-trivial processes, the model is likely to be very large and hence should be modularized. What is the appropriate style of interface among process model modules and how is it enforced? Is the process model visible to the users and, if so, how is it presented? If the users can modify the process model during execution, how does this affect continuing the process? Should (can) consistency be enforced or should (can) inconsistencies be detected and repaired?

Second, constructing envelopes is even harder. We currently use the various Unix shell languages as notations for defining envelopes. Writing the scripts requires knowledge of both the tool interface and the MARVEL objectbase implementation. Although there is no way to avoid the former, we hope to minimize the latter by developing a higher level envelope language that includes facilities to query the the MARVEL objectbase to obtain objects in file system form.

Again, similar difficulties seem inevitable for other process modeling mechanisms intended to work with external tools. The questions: Is it possible (or desirable) to support existing tools, particularly tools where retrofitting is impractical or impossible? If only new tools are supported, can a relatively simple standard interface be developed and published so these tools can be developed separately by vendors? Would such an interface pre-empt too many design decisions, or would it be feasible for vendors to upgrade their current products?

Third, supporting multiple users is hardest of all. This is a well-known general problem of current software development environments [Rowe 89]. Developing a suitable extended transaction model is the major focus of our current research effort.

In summary, MARVEL is implemented and in limited use, it works in some cases, it doesn't work in others, it raises a lot of questions, and both we the MARVEL group and we the software engineering community have a lot more work to do to deliver the promise of process modeling.

Acknowledgments

Nasser Barghouti and Mike Sokolsky are primarily responsible for developing and testing MARVEL. David Jamroga, Ari Shamash, Miriam Sporn, Mike Tannenblatt and Kok-Yong Tan are also working on the current version of MARVEL. Laura Johnson and Victor Kan developed the DocPrep document production environment given in the appendix and Shyhtsun Felix Wu contributed to the C/Marvel programming environment. Peter Feiler and Steve Popovich collaborated with Kaiser on the initial conception of MARVEL, Bob Schwanke contributed to an early design, and Wendy Dilliard, Russel Goldberg, Christine Hong, Wai Keung Hui, Qifan Ju, Christine Lombardi, Alexander Mogieleff, Joe Milligan, Michael Sacks, Tam Tran, and Timothy Yuan participated in the implementation of earlier versions of MARVEL.

References

[Balzer 85] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[Barghouti 88a] 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. Kona HI, January, 1988.

[Barghouti 88b] Naser S. Barghouti and Michael H. Sokolsky.
MARVEL User's Manual Version 2.01.
Technical Report CUCS-371-88, Columbia University Department of Computer Science, November, 1988.

[Cardelli 84] Luca Cardelli.
A Semantics of Multiple Inheritance.
In G. Kahn, D.B. MacQueen and G. Plotkin (editor), *Semantics of Data Types International Symposium*, pages 51-67. Springer-Verlag, New York, June, 1984.

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

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

[Feiler 87] Peter H. Feiler and Gail E. Kaiser.
Granularity issues in a knowledge-based programming environment.
Information and Software Technology 29(10):531-539, December, 1987.

[Forgy 82] Charles L. Forgy.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence 19:17-37, 1982.

[Garlan 84] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In Peter Henderson (editor), *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 65-72. Pittsburgh, April, 1984.
Special issue of *SIGPlan Notices*, 19(5), May 1984.

[Gupta 86] Anoop Gupta.
Parallelism in Production Systems.
PhD thesis, Carnegie Mellon University, March, 1986.
CMU-CS-86-122.

[Habermann 86] A.N. Habermann and D. Notkin.
Gandalf: Software Development Environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[Huff 88] Karen E. Huff and Victor R. Lesser.
A Plan-based Intelligent Assistant that Supports the Software Development Process.
In Peter Henderson (editor), *ACM SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 97-106. ACM Press, Boston MA, November, 1988.
Special issue of *SIGPlan Notices*, 24(2), February 1989.

[Kaiser 87a] Gail E. Kaiser and Peter H. Feiler.
An Architecture for Intelligent Assistance in Software Development.
In *9th International Conference on Software Engineering*, pages 180-188. Monterey CA, March, 1987.

[Kaiser 87b] 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 88a] Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich.
Intelligent Assistance for Software Development and Maintenance.
IEEE Software :40-49, May, 1988.

[Kaiser 88b] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler and Robert W. Schwanke.
Database Support for Knowledge-Based Engineering Environments.
IEEE Expert 3(2):18-32, Summer, 1988.

[Kaiser 88c] Gail E. Kaiser and Naser S. Barghouti.
An Expert System for Software Design and Development.
In *Joint Statistical Meetings*, pages 10-19. New Orleans LA, August, 1988.
Invited paper.

[Kaiser 89] Gail E. Kaiser.
A Marvelous Extended Transaction Processing Model.
In Gerhard Ritter (editor), *11th World Computer Conference IFIP Congress '89*. Elsevier Science Publishers B.V., San Francisco CA, August, 1989.
In press.

[Maarek 87] Yoelle S. Maarek and Gail E. Kaiser.
Using Conceptual Clustering for Classifying Reusable Ada Code.
In *Using Ada: ACM SIGAda International Conference*, pages 208-215. ACM Press, Boston MA, December, 1987.
Special issue of *Ada LETTERS*, December 1987.

[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.

[Miranker 87] Daniel P. Miranker.
TREAT: A Better Match Algorithm for AI Production Systems.
In *AAAI 87 6th National Conference on Artificial Intelligence*, pages 42-47. AAAI, Seattle WA, July, 1987.

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

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

[Rowe 89] Lawrence A. Rowe and Sharon Wensel (editors).
1989 ACM SIGMOD Workshop on Software CAD Databases.
, Napa CA, 1989.

[Smith 85] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[Sokolsky 89] Michael H. Sokolsky.
Data Migration in an Object-Oriented Software Development Environment.
Master's thesis, Columbia University Department of Computer Science, April, 1989.
CUCS-424-89.

[Stallman 81] Richard M. Stallman.
Emacs The Extensible, Customizable, Self-Documenting Display Editor.
In *SIGPlan SIGOA Symposium on Text Manipulation*, pages 147-156. ACM, June, 1981.
Special issue of *SIGPlan Notices*, 16(6), June 1981.

[Stolfo 84] Salvatore J. Stolfo.
Five Parallel Algorithms for Production System Execution on the DADO Machine.
In *AAAI 84 National Conference on Artificial Intelligence*. AAAI, August, 1984.

[Taylor 88] Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil and Alex L. Wolf.
Foundations for the Arcadia Environment Architecture.
In Peter Henderson (editor), *ACM SIGSoft/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1-13. ACM Press, Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989.

I. Example MARVEL Strategies

The following DocPrep strategies were developed by two students, Laura Johnson (MS student) and Victor Kan (undergraduate), as their term project in the E6123y Programming Environments and Software Tools course in Spring 1989. Neither student had any prior knowledge of or experience with MARVEL, and were not familiar with its internal implementation. They successfully used the DocPrep environment to produce their final project report. The twelve envelopes, bind, delete-sect-order, format, printdoc, printsec, review-format-err, review-spell-err, runeditor, specify-header, specify-printer, specify-sect-order and spell-check, are omitted.

```
STRATEGY: docprep_ob_defs;
Imports: doc_tools_rules;
Exports: all;

ObjectBase:

DOC_PROJ :: superclass: ENTITY:
   status : (Complete,Modify) = "Modify";
   types : set_of DOC_TYPES;
END

DOC_TYPES :: superclass: ENTITY:
   printname : (Resume,Thesis,Hmwk,User_Manual,Book) = "User_Manual";
   timestamp : real = "0.0";
   bind_all_stat : (YesBindAll,NoBindAll) = "NoBindAll";
   print : (YesPrint,NoPrint) = "No_Print";
   head_created : (HeadFileExists,HeadFileNonExists) = "HeadFileNonExists";
   printer_created : (PRFileExists,PrFileNonExists) = "PrFileNonExists";
   sects_ordered : (YesSecOrder,NoSecOrder) = "NoSecOrder";
   sects : set_of SECTIONS;
   header : set_of HEADER_FILE;
   printer : set_of PRINTER_TYPE;
END

SECTIONS :: superclass: ENTITY:
   printname : (Intro,Chaps,Body,Biblio,Index,Prob_Secs) = "Intro";
   timestamp : real = "0.0";
   edited : (SecIsEd,SecNoEd) = "SecNoEd";
   spellcheck : (SecIsSpell,SecNoSpell) = "SecNoSpell";
   format : (SecIsFormat,SecNoFormat) = "SecNoFormat";
   spell_err_chk : (NoErr,YesErr) = "NoErr";
   format_err_chk : (NoErr,YesErr,NoHeadErr) = "NoErr";
   printer_created : (PrFileExists,PrFileNonExists) = "PrFileNonExists";
   preview : (SecYesView,SecNoView) = "SecNoView";
END

END ObjectBase
```

```
STRATEGY: doc_tools_rules;
Imports: none;
Exports: all;

ObjectBase:

DOC_HEAD :: superclass: TOOL:
doc_head : string = "specify-header";
END

ORD_SECTS :: superclass: TOOL:
ord_sects : string = "specify-sect-order";
END

BIND :: superclass: TOOL:
bind : string = "bind";
END

DOC_DEV :: superclass: TOOL:
doc_dev : string = "specify-printer";
END

PRN_DOC :: superclass: TOOL:
prn_doc : string = "printdoc";
END

END ObjectBase

Rules:

doc_head[?d:DOC_TYPES]:
suchthat
:
{ DOC_HEAD doc_head ?d }
(?d.head_created = HeadFileExists);
(?d.head_created = HeadFileNonExists);

ord_sects[?d:DOC_TYPES]:
suchthat
:
{ ORD_SECTS ord_sects ?d }

bind[?d:DOC_TYPES]:
suchthat
:
{ BIND bind ?d }
(?d.bind_all_stat = YesBindAll)
(?d.bind_all_stat = NoBindAll)

doc_dev[?d:DOC_TYPES]:
suchthat
:
{ DOC_DEV doc_dev ?d }
(?d.printer_created = PrFileExists);
(?d.printer_created = PrFileNonExists);

prn_doc[?d:DOC_TYPES]:
```

```
suchthat
:
{ PRN_DOC prn_doc ?d }
```

```
STRATEGY: sect_tools;
Imports: none;
Exports: all;

ObjectBase:

EDIT :: superclass: TOOL:
edit : string = "runeditor";
END

FORMAT :: superclass: TOOL:
format : string = "format";
END

FMT_ERR :: superclass: TOOL:
fmt_err : string = "review-format-err";
END

SPELL_CHK :: superclass: TOOL:
spell_chk : string = "spell-check";
END

SPELL_ERR :: superclass: TOOL:
spell_err : string = "review-spell-err";
END

SECT_DEV :: superclass: TOOL:
sect_dev : string = "specify-printer";
END

PRN_SECT :: superclass: TOOL:
prn_sect : string = "printsect";
END

END ObjectBase
```

```
STRATEGY: auto_sect_rules;
Imports: docprep_ob_defs, sect_tools;
Exports: all;

Rules:

edit[?s:SECTIONS]:
suchthat
:
{ EDIT edit ?s }
(?s.edited = SecIsEd)(?s.format = SecNoFormat);
(?s.edited = SecNoEd)(?s.format = SecIsFormat);

format[?s:SECTIONS]:
suchthat
:
{ FORMAT format ?s }
(?s.format_err_chk = NoErr)(?s.format = SecIsFormat);
(?s.format_err_chk = YesErr);

fmt_err[?s:SECTIONS]:
suchthat

  s.format_err_chk = YesErr)
. FMT_ERR fmt_err ?s }
(?s.format = SecNoFormat)(?s.edited = SecIsEd);

spell_chk[?s:SECTIONS]:
suchthat
:
and((?s.format_err_chk = NoErr)(?s.format = SecIsFormat))
{ SPELL_CHK spell_chk ?s }
(?s.spell_err_chk = NoErr)(?s.spellcheck = SecIsSpell);
(?s.spell_err_chk = YesErr);

spell_err[?s:SECTIONS]:
suchthat

 ?s.spell_err_chk = YesErr)
. SPELL_ERR spell_err ?s }
(?s.spell_err_chk = NoErr)

sect_dev[?s:SECTIONS]:
suchthat
:
(?s.spell_err_chk = NoErr)
{ SECT_DEV sect_dev ?s }
(?s.printer_created = PrFileExists);
(?s.printer_created = PrFileNonExists);

prn_sect[?s:SECTIONS]:
suchthat
:
(?s.printer_created = PrFileExists)
{ PRN_SECT prn_sect ?s }
```

```
STRATEGY: man_sect_rules;
Imports: docprep_ob_defs, sect_tools;
Exports: all;

Rules:

edit[?s:SECTIONS]:
suchthat
:
{ EDIT edit ?s }
(?s.edited = SecIsEd)(?s.format = SecNoFormat);
(?s.edited = SecNoEd)(?s.format = SecIsFormat);

format[?s:SECTIONS]:
suchthat
:
{ FORMAT format ?s }
(?s.format_err_chk = NoErr)(?s.format = SecIsFormat);
(?s.format_err_chk = YesErr);

fmt_err[?s:SECTIONS]:
suchthat
:
(?s.format_err_chk = YesErr)
{ FMT_ERR fmt_err ?s }
(?s.format = SecNoFormat)(?s.edited = SecIsEd);

spell_chk[?s:SECTIONS]:
suchthat
:
{ SPELL_CHK spell_chk ?s }
(?s.spell_err_chk = NoErr)(?s.spellcheck = SecIsSpell);
(?s.spell_err_chk = YesErr);

spell_err[?s:SECTIONS]:
suchthat
:
(?s.spell_err_chk = YesErr)
{ SPELL_ERR spell_err ?s }

sect_dev[?s:SECTIONS]:
suchthat
:
{ SECT_DEV sect_dev ?s }
(?s.printer_created = PrFileExists);
(?s.printer_created = PrFileNonExists);

prn_sect[?s:SECTIONS]:
suchthat
:
{ PRN_SECT prn_sect ?s }
```