# Process Evolution for the MARVEL Environment

Gail E. Kaiser
Israel Z. Ben-Shaul
George T. Heineman
John K. Hinsdale*
Wilfredo Marrero+

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
212-939-7000/fax:212-666-0140
kaiser@cs.columbia.edu

CUCS-047-92
8 November 1992

## Abstract

We present a schema and process evolution tool, called the Evolver, for the MARVEL process-centered environment. The Evolver analyzes the differences between the new and installed process models of an existing environment, detecting each case where the notion of consistency defined by the process model has been strengthened or weakened. The Evolver then automatically updates the environment's objectbase to guarantee that the objects are consistent according to the new specifications. The Evolver can be applied while the installed process is in progress, temporarily halting normal operation while it updates the objectbase, after which development continues using the new process. We have had several months of experience using the Evolver to make repeated changes in the process that supports our own further development of MARVEL, and include in this paper one small but practical example of a recent change made to a real MARVEL process.



*Hinsdale is now employed by JYACC, New York, NY. +Marrero is now a PhD candidate in Computer Science at Carnegie Mellon University, Pittsburgh, PA.

**keywords:** automation, consistency, object-oriented database, process-centered environment, rules, schema evolution

## 1. Introduction

MARVEL [17] is a process-centered environment where the process model consists of a set of planning system-style rules and process enaction is implemented by forward and backward chaining over the rule base. The process model is augmented by an object-oriented data model (schema), which defines the composition of both product data and process data in an objectbase. In January 1992, we began using the first multi-user version of MARVEL, 3.0 [9], in our own further development of the MARVEL system. We constructed data and process models and a set of tool envelopes, collectively called C/Marvel, based primarily on the previous organization of the MARVEL code in the file system, our manual development process and the corresponding Unix utilities, respectively. We used an immigration tool, called Marvelizer [29], to construct a C/Marvel objectbase containing our source, headers, libraries, executables, test cases, documentation, etc.

Not surprisingly, we soon discovered numerous reasons to evolve the C/Marvel process model, to improve our own process and to take advantage of new MARVEL capabilities as they were developed. Most changes to the process required corresponding changes to the data model to modify classes and attributes employed in the process. But we now had a rather large objectbase, containing about 100,000 lines of source code, that had to be evolved whenever the data model changed in a way that affected existing objects (i.e., schema evolution). Further, many changes to the process model also directly affected existing objects. In particular, the values of status attributes (product data) must be updated to be consistent with the new process model.

Initially all objectbase evolution was done by hand, but this approach was tedious and error-prone, and we could not afford to introduce any errors! We were (and are) totally dependent on the integrity of this C/Marvel objectbase to continue our work. It quickly became apparent that we needed tools and a process to support evolution, not just specifically for C/Marvel but for any MARVEL environment.

We developed the first version of our Evolver tool, which supports certain kinds of changes to the data model. There are based on the schema evolution facilities supported by the Orion object-oriented database management system [3]; MARVEL's object management system is very similar to Orion's except in MARVEL's support for file attributes. Classes, their superclasses and their attributes can be added, deleted, modified and renamed, except changes that would delete file attributes or restructure the composite object hierarchy are not automated (they must be done manually, using other facilities of MARVEL that are outside the scope of this paper). We intentionally prohibit automatic deletion of file attributes for safety reasons, since the bulk of the product data resides in file attributes. Schema restructuring is an open problem for object-oriented database research [28, 21].

The Evolver works in the context of an existing MARVEL objectbase, with an existing data and process model. It accepts a new data model as input, and compares it to the existing class hierarchy; any renaming of classes or attributes is indicated on the command line. The tool presents to the MARVEL environment administrator a detailed analysis of the differences between the two data models, reporting any modifications it can not support. If the new data model is acceptable, it also presents a complete list of required changes to existing objects. On confirmation by the administrator, the Evolver automatically updates the objectbase accordingly (after saving the original) and installs the new data model in the environment. This Evolver tool was included in the MARVEL 3.0.1 release in spring 1992.

The next step was a major revision of the Evolver tool, to also support certain changes to the process model (which had previously been done manually) in addition to schema evolution. The new Evolver depends on MARVEL's distinction between two purposes for process enaction, *consistency* and *automation* [5], which are determined by special annotations on rules in the process model. In essence, when one rule would forward chain to another rule to maintain consistency, the second rule is considered an implication of the first, and by definition must be fired whenever the first rule is fired. If it is not possible to execute an entire chain defined recursively by such implications, that chain must be rolled back (i.e., the entire consistency chain executes as a transaction), and thus the opportunity for backward consistency chaining can never arise. In contrast, if one rule would forward or backward chain to another rule solely for automation purposes, the chaining is considered optional. Automation chaining may be explicitly turned off, if desired, or can be terminated at any rule boundary without rollback of the entire automation chain.

The second version of the Evolver tool accepts a new process model as input, and generates a graph reflecting only the consistency implications among rules. It compares this to the consistency rule graph representing the existing process. The tool analyzes the differences between these two graphs, to detect cases where consistency is either strengthened (adding an edge) or weakened (deleting an edge). This may require interaction with the administrator to match a rule with its replacement, since MARVEL allows multiple rules with the same name but different conditions. The Evolver automatically generates a batch script of MARVEL commands to fire any consistency (sub)chains necessary to make the objectbase consistent with respect to the new consistency graph. The script is executed using an option in the MARVEL command line client (an alternative to the graphical user interface). Rule changes that are concerned only with automation, and do not affect consistency, are also installed but do not require any updates to the objectbase.

This Evolver tool was briefly used in a stand-alone fashion, but was soon incorporated into P/Marvel, a MARVEL environment for developing and evolving data models, process models and tool envelopes. A P/Marvel objectbase contains the source and internal representation of a data model and a process model, and includes references to both testing and "real" objectbases whose environments are instantiated by these models. Like any MARVEL environment, P/Marvel maintains consistency among its target objects, in this case components of data and process models, and automates aspects of the process, in this case installation and testing of new or evolved data and process models. P/Marvel and the new Evolver tool are now part of the internal version MARVEL 3.0.2, which we use in our own continuing development, and are planned for release as part of MARVEL 3.1 in early 1993.

All the evolution facilities described above operate in an ‘‘off-line’’ fashion, meaning that the administrator takes over the particular MARVEL objectbase and all other users are temporarily prevented from using that environment during evolution. However, the main goal of our schema and process evolution facilities is to be able to continue existing long-lived processes after changes. Thus the process itself can be in progress when evolution occurs but must be quiescent, i.e., all chains triggered by previous user commands have terminated and the server is waiting for the next request to continue the process. Changes to previously developed data and process models for use in brand new environments are also assisted by P/Marvel, but do not require the sophisticated support of the Evolver tool. The functionality of P/Marvel, other than the Evolver, is outside the scope of this paper.

## 2. Contributions

The primary goal of a process evolution tool is to guarantee that the objectbase is not inconsistent with respect to the new process. In a previous paper [5], we proposed an approach that rejected changes to the process that might potentially result in inconsistency. The analysis we described was excessively conservative, in essence prohibiting nearly all changes involving consistency chaining (although permitting any changes concerned solely with automation). In this paper we introduce a more powerful approach that accepts any new process model (which is syntactically correct and matches the data model), and automatically updates the objectbase according to the new process. This second approach was implemented in our Evolver tool.

The key insight that makes our new approach tractable is that it is unnecessary for the Evolver to analyze the current contents of the objectbase to determine whether or not it is consistent. Instead, the Evolver compares the old and new process models, and determines the set of rules affected by changes related to consistency. It then generates a list of all possible instantiations of these rules, considering only the types but not the contents of the objects in the objectbase. Then these rules are supplied as if they were user commands to MARVEL's normal process engine.[1] Not all such rules will actually result in in updates to the objectbase, though, since their conditions are not necessary satisfied. In fact, there may not be any updates at all, but all affected rules must still be attempted on all possible parameters, since this is not known a priori.

In addition to MARVEL, our approach should be immediately applicable to the CLF process-centered environment [11], which supports consistency forward chaining conceptually similar to our own. It should also be easily extended to other environments whose process enaction is based primarily on forward chaining or triggers, including Merlin [27], EPOS [12], ALF [22] and Adele2 [8]. As far as we know, none of these systems currently provides an automated process evolution tool.

The Darwin/1 backward chaining environment includes a sophisticated mechanism for accepting or rejecting process changes [24], where the process (or ''law'') itself specifies the permissible range of future changes. We do not know how Darwin handles update of existing objectbases to conform to changed processes, if it in fact does so, but whatever works for Darwin would probably also work for other backward chaining environments such as Oikos [1]. We expect that our approach could be modified for backward chaining, basically by reversing the direction of source versus destination rules as described in Sections 5 and 6. We also think that the consideration of consistency that underlies our automated approach might be adaptable to other declarative process modeling formalisms, such as the Petri net-like structures in Melmac [13] and the task graphs in the Articulator [23], which currently rely on largely manual mechanisms. But have not yet seriously investigated either possibility.

We first present a general overview of the MARVEL system, including how consistency versus automation is specified in the process model. We outline those aspects of transaction management that motivate much of the distinction between consistency and automation; in particular, a consistency chain is treated as a transaction whereas an automation chain is a series of independent transactions. We next describe the design and implementation of the initial data model Evolver, focusing on the differences between MARVEL and Orion. The rest of the paper concentrates on process evolution. We explain precisely what is meant by consistency strengthening and weakening, and then present the corresponding extensions to the Evolver to support process model evolution. We give a small but practical example we recently applied to C/Marvel.

---

[1] As explained in Section 7, some rules are not directly available to arbitrary users, but just to the administrator, so the Evolver must operate using the administrator's privileges.

## 3. MARVEL Background

Our goal in the MARVEL project is to develop a kernel for *process-centered environments* [20, 14] that guide and assist teams of users working on large-scale software development efforts. The generic kernel must be tailored by an *administrator* who provides the data model, process model, tool envelopes and coordination model for a specific organization or project. The data model classes define an objectbase containing the software system under development. Multiple inheritance, status attributes of primitive and enumerated types, file attributes of text and binary types, aggregate composite objects, and non-hierarchical links between objects can be declared. Figure 3-1 shows two representative classes from C/Marvel. It should be self-explanatory, except that the ‘‘initialization’’ of a file attribute indicates its filename extension rather than its default contents.

```
# Module is the organizing force in the data model.  It groups together
# lex files, yacc files, and C files.  NOTE:  This definition is recursive,
# that is, modules can contain other modules.  A Module can only be linked
# to ONE afile.  This prevents duplication in libraries.

MODULE :: superclass ARCHIVABLE, PROTECTED_ENTITY;
    lfiles   : set_of LFILE;
    cfiles   : set_of CFILE;
    yfiles   : set_of YFILE;
    doc      : set_of DOC;
    modules  : set_of MODULE;
    afiles   : link AFILE;
end

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

CFILE :: superclass COMPILABLE, REFERENCED, FILE;
    status         : (New, NotAnalyzed, ErrorAnalyze, Analyzed,
                      NotCompiled, ErrorCompile, Compiled,
                      NotArchived, ErrorArchive, Archived) = New;
    contents       : text = ".c";
end
```

**Figure 3-1:** `MODULE` and `CFILE` Classes from C/Marvel

The process or workflow is specified in a *process modeling language* [17]. Each process step is encapsulated in a *rule* with a name and typed formal parameters; these are used to generate the command menu available to users. Each rule is composed of a condition, an optional activity, and a set of effects. The condition has two main parts: local *bindings* to query the objectbase to gather implicit parameters (e.g., included ".h" files when compiling a ".c" file) and a *property list* that must evaluate to true prior to invocation of the step.[2] Each member of the set of effects asserts one of the step’s possible results. If there is no activity, then by definition there can be only one effect; if there is an activity, then in general the invoked tool may have several possible results corresponding one-to-one with the given effects. Forward and backward chaining over the rules enforces consistency in the objectbase and automates tool invocations. Enforcement and automation are two forms of ‘‘enaction’’, the term used in the community for any computer-aided support for process. A MARVEL rule for compiling (cc) a C file is shown in Figure 3-2.

MARVEL’s process enaction is user-driven. When a user enters a command, the environment selects the

[2]The bindings and property list are optional, but we ignore in this paper the relatively simple cases where either or both are omitted.

```
# *** rule name and typed parameters ***

compile [?c:CFILE]:

    # *** bindings ***

    (and (exists PROJECT ?p  suchthat (ancestor [?p ?c]))
         (forall INC     ?i  suchthat (member   [?p.include  ?i]))
         (forall HFILE   ?h  suchthat (member   [?i.hfiles   ?h]))):

    # *** property list ***

    # 0. If an INC object is archived, don't automatically chain here
    # 1a. If this CFILE has been analyzed or
    # 1b. had a previous compilation error.
    # ---------------------------------------------------------------
    (and no_forward (?i.archive_status = Archived)                # 0.
         (or (?c.status = Analyzed)                               # 1a.
             no_chain (?c.status = ErrorCompile)))                # 1b.

    # *** activity ***

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

    # *** effects ***

    # 0.a Chain to archive the module in which it is contained.
    # 0.b Update the object code time stamp
    # 1.  Update status of object.  Chain to dirty rules
    # ----------------------------------------------------------
    (and (?c.status = Compiled)                             # 0.a
         (?c.object_time_stamp = CurrentTime));             # 0.b
    (?c.status = ErrorCompile);                             # 1.
```

**Figure 3-2:** `CFILE` Compile Rule From C/Marvel

rule with the same name and ‘‘closest’’ signature to the provided actual parameters considering multiple inheritance [4].[3] The condition of this rule is said to be *satisfied* if its property list evaluates to true for every binding of a local variable (in the case of a "forall" binding) or to at least one binding of a local variable (in the case of an "exists" binding), considering the parameters supplied to the rule. If the condition is not satisfied, backward chaining is attempted. If the condition is already satisfied or becomes satisfied during backward chaining, the activity is initiated. After the activity has completed, the appropriate effect is asserted. This triggers forward chaining to any rules whose conditions become satisfied by this assertion. The asserted effects of these rules may in turn satisfy the conditions of other rules, and so on. Eventually no further conditions become satisfied and forward chaining terminates. MARVEL then waits for the next user command.

Each rule binding may optionally include a complex clause (nested conjunctions and disjunctions) of *predicates*. In the examples of this paper, the bindings include only individual *structural* predicates, which navigate the objectbase to obtain ancestors or descendants of specified types, containers or members of aggregate attributes, and objects linked to or from other objects. However, *associative* predicates can also be employed in the bindings, as they are in the property lists. Each property list is a complex logical clause of any combination of structural and associative predicates. In this paper, only

[3] When there are several equally ‘‘close’’ rules, they are considered in an implementation-specific order; the details would overly complicate the discussion, so are omitted.

associative predicates are shown in the examples, and these include only cases of testing equality between an attribute and a literal value. However, the full range of relational operators can be used to compare the attributes of the same or different objects as well as literals. Finally, an effect consists of a conjunction of predicates; in these, only the equality operator is meaningful, which is asserted by assigning the specified value to an attribute. However, MARVEL's built-in add, delete, move, copy, rename, link and unlink operations may also be used in effect predicates.[4]

Individual predicates in a rule's property list and in its effects are each annotated as either "consistency" or "automation"; predicates in a rule's bindings are not annotated. By definition, all forward chaining from a consistency predicate in an asserted effect to rules with satisfied conditions is mandatory. In contrast, forward chaining from an automation predicate is optional, and can be ‘‘turned off’’ wholesale by a user command or explicitly restricted through "no_forward", "no_backward" or "no_chain" directives on individual automation predicates. Backward chaining is meaningful only into (unrestricted) automation predicates, since any consistency predicates that could satisfy would already have done so during consistency forward chaining. It is important to understand that only automation *chaining* is optional; users are still obliged to follow some legal process step sequence implied by the conditions and effects of rules, whether through manual selection of commands or automation chaining. We discuss in another paper [19] the specification of alternatives, iteration and synchronization through the conditions and effects of rules; the process is not in any sense limited to a deterministic sequence of steps. Possible chains are compiled into a network when the kernel is tailored [18]. The process engine chains among rules with different or multiple parameters by ‘‘inverting’’ local bindings [16].

An automation predicate is enclosed in parentheses "(...)", and may optionally be preceded by one of the chaining directives. A consistency predicate is enclosed in square brackets "[...]", and directives restricting chaining are not valid. The condition of the rule shown in Figure 3-2 permits backward chaining from it to archive all the `INC` (include) components of the `PROJECT` (condition predicate 0), but forward chaining into it from the `arch` (archive) rule, not shown, is prevented by the no_forward directive. Forward chaining into this rule is permitted after analyzing (lint) a C file (condition predicate 1a), which becomes the parameter `?c`. Chaining is prohibited into or out of the `compile` rule from another instance of the `compile` rule, due to the no_chain directive (condition predicate 1b). The `compile` operation of the `BUILD` tool and its corresponding envelope are defined separately, not shown. If this rule had no activity, then the curly braces "{...}" would be empty.

If all the predicates in the effects of this rule were annotated as consistency instead of automation (in which case the chaining directives would be removed), then a compilation resulting in correct object code (effect predicate 0) would necessarily chain into a `build` rule, not shown, to link the program being modified, and a compilation resulting in error messages (effect predicate 1) would necessarily chain to repeat the same compilation rule (condition predicate 1b)! To avoid such undesirable behavior, the administrator should use consistency annotations only to indicate attribute change propagation: for example, when an object has been modified, other objects that depended on its previous state might be marked as `dirty`. Further details of the rule language and process engine are explained in Section 7 as part of our process evolution example.

Conventional file-oriented tools are integrated into the process without source modifications, or even recompilation, through an *enveloping* language based on shell scripts [15]. A non-empty rule activity

[4]The latter functionality is quite new, and we would not (yet) claim that any facilities described in this paper work properly when built-in operations are used in effects.

specifies an envelope and its input and output arguments, which may be literals, status attributes and/or (sets of) file attributes. In addition to output arguments, each envelope returns a distinguished value to indicate which of the specified effects should be asserted. Further details of activities and envelopes are outside the scope of this paper.[5]

Multiple users of a MARVEL environment are supported by a client/server architecture [9]. A client provides the user interface, checks the arguments of commands, and forks operating system process to execute tool envelopes. Two distinct X11 windows user interfaces support graphical browsing and ad hoc queries on the objectbase, one with relatively primitive but portable functionality implemented directly in Xlib and the other with advanced capabilities that depend on the XView toolkit. There is also a command line interface for terminals and batch scripts.

The process engine, transaction management system and object management system reside in a central server. Scheduling is FCFS, with rule chains interleaved at the natural breaks provided when clients execute rule activities. In particular, the process engine executes the chain initiated by a client's command until an activity is encountered. Sufficient information to execute the activity is then returned to that client, and the server retrieves the next client request from its input queue. When an activity terminates, the client places its results at the end of the server's queue and waits its turn until the server resumes its in-progress chain where it left off. Since essentially all long duration operations are performed in the clients rather than the server, the ''wait'' is never noticeable.[6]

The XView client supports multiple threads of control that the user can switch among manually using different subwindows; to do multiple things at the same time using the Xlib or command line client, a user must invoke multiple clients from different windows. Clients may run on the same or different machines as their server, but the enveloping facility assumes a shared file system for accessing the contents of file attributes. The external view and internal architecture of a generic MARVEL environment are illustrated in Appendix I.

The transaction management system consists of three layers that support concurrency control among multiple clients. The bottom layer is a table-driven lock manager that detects potential access conflicts, e.g., when one rule within an ongoing rule chain holds a lock on a particular object and another rule in a concurrent chain attempts to obtain a conflicting lock on the same object. The middle layer resolves such conflicts through a semantics-based protocol that understands the distinction between consistency and automation chains indicated by the process model [6]. Consistency is all-or-nothing: if a consistency chain cannot be completed due to a lock conflict when a rule attempts to access an object, the entire chain should be aborted (rolled back). In contrast, rule chains purely for automation can be terminated after the previous rule when there is a conflict. Each rule directly selected by a user or triggered by automation chaining starts a new top-level transaction, with all rules fired during subsequent consistency chaining treated as actions within this transaction. A consistency chain can lead to initiation of an automation chain, starting a new transaction, and an automation chain can lead to a consistency chain, which is considered part of the same transaction as its last rule. The middle layer also supports failure recovery, in the sense that incomplete consistency chains or individual automation rules are rolled back after a crash.

---

[5]MARVEL provides special facilities that permit extensible tools, such as emacs, to request additional parameters from the objectbase incrementally over long sessions; this is done by simulating what process enaction treats as multiple independent rules with different parameters, but they all initiate the same instance of the activity.

[6]The only exceptions are special administrator functions, such as evolution, during which no other users would be permitted to access the server anyway.

However, the current implementation reflects a seemingly arbitrary restriction on consistency chains: Only the original rule from which the chain emanates can contain a non-empty activity. All subsequent rules in the chain must have empty activities. This is not overly limiting in practice, since the purpose of consistency chaining is generally to propagate changes from attributes of the parameters and local bindings of the original rule to dependent attributes of other objects, where all necessary computation can be done in the conditions and effects of the rules without any tool invocations. Our rationale for this restriction was to conserve the file server space available to the MARVEL project.

Consider a rule with a non-empty activity. If this individual rule, or a consistency chain containing this rule, were to abort, its activity would have to be rolled back or undone. This can be accomplished by saving all input and output arguments of the activity before it begins, so that they can be restored later if necessary. For example, if the activity is to edit a text file, then that file must first be copied; if the activity modifies an RCS file, then the entire RCS file must be copied; and if the activity generates an executable file, the previous version of the executable must be saved. If the activity is later rolled back, the new version of the file would be discarded and the old one restored. Activities with external side-effects, such as sending electronic mail or printing files, obviously cannot truly be rolled back.[7]

We found the potential space expense of logging the contents of file attributes accessed by long duration transactions to be prohibitive. Our restriction on consistency chains, together with the differential treatment of automation chains explained above, means that the arguments of only <u>one</u> rule must be saved for the duration of a top-level transaction. This restriction is enforced when the process model is translated into the internal rule network used by the process engine, by representing all edges from a consistency predicate in an effect into any rule with a non-empty activity as if they were automation predicates. Otherwise, nothing in the MARVEL system is concerned with this restriction, and it would be easy to remove. It is important to note that this scheme does not in any way impact the ability to include multiple versions and version management explicitly in the process model, and in fact the C/Marvel and P/Marvel environments both employ RCS for text file attributes.[8]

MARVEL's *coordination modeling language* permits the administrator to tailor the top layer of the transaction manager with ‘‘coordination rules’’ (which should not be confused with process rules). When the semantics-based protocol above would abort a rule chain, the coordination rule base is inspected to attempt to find an alternative resolution. Each coordination rule specifies a scenario where relaxation of transaction serializability can be tolerated, and corresponding actions that increase concurrency and enable cooperation [7]. Coordination modeling seriously complicates both failure recovery and process evolution, because it may introduce commit and abort dependencies among top-level transactions [10] and the possibility of relatively long-lived ‘‘temporary’’ inconsistency in the objectbase [2]. This is the subject of ongoing research.

## 4. Data Model Evolver

To evolve an objectbase, the administrator simply edits the data model and runs the Evolver tool. One alternative considered in the design phase was to alter classes through an interactive dialog with the Evolver. This was considered too cumbersome, and instead we adopted the approach of automatically

---

[7] Like nearly all transaction systems that we know of, MARVEL ignores this problem and pretends that such activities can be rolled back. We refer the reader to Pausch's analysis of interactive transactions, which pose a similar difficulty [25].

[8] Since the MARVEL kernel is entirely generic, and knows nothing about any tools employed in the process, another version management tool such as SCCS [26] could equally well be incorporated.

matching old and new versions, which any renaming of classes or attributes specified on the command line.

### 4.1. Capabilities

The following types of changes to the data model are supported:

1. Add an attribute to a class, and (implicitly) to its subclasses.
2. Delete an attribute from a class, and from its subclasses (this may cause the class and its subclasses to ‘‘reinherit’’ a previously overridden attribute with the same name from one of its superclasses). Reordering the attributes of a class is recognized as such, so is not treated as a set of adds and deletes. Deletion of status attributes is supported, but file attributes cannot be deleted, nor can aggregate attributes (representing one or more child objects) or link attributes. There is really no good reason to disallow deletion of links, and this should be remedied eventually.
3. Rename an attribute in a class, and in its subclasses.
4. Change the type of an attribute in a class, and its subclasses, including changes to the list of values for an enumerated type. Changing the type of an attribute is permitted only for status attributes, that is, both the old and new types are either primitive or enumerated. When an attribute’s type is changed, data may be lost, because the value is reset to the default for the (new) type (when the only change is to modify the list of enumerated values, data is not lost if the old value remains in the list). No type coercion of the old type to the new type is performed, although this is in principle feasible for some (but not all) changes.
5. Change the default value of an attribute in a class, and its subclasses (except when overridden by a subclass).
6. Add a class.
7. Delete a class is not supported, see operation 10.
8. Rename a class.
9. Add a class to the list of superclasses of a class.
10. Delete a class from the list of superclasses of a class. A reordering of the superclasses for a class is recognized as such, rather than treated as a set of adds and deletes. Deleting a class from the list of superclasses is permitted <u>only</u> if the deleted superclass consists entirely of status attributes. This is simply an extension of the restriction on operation 2 above to inheritance. The prohibition against deleting classes, operation 7, is really overkill. We could have instead applied the analogous restriction as for superclasses: only classes consisting solely of status attributes could be deleted.

The theme running through all the restrictions above is prevention of changes potentially affecting file attributes. This makes sense for an object management system whose primary purpose is to store product data resident in conventional files. The last thing we want to do is make it "easy" for the administrator to (perhaps unintentionally) delete files.[9] Although the Evolver saves the old objectbase, as explained below, saving the old hidden file system would be very expensive in both time and space. We should make clear, however, that it is *possible* to perform all the operations prohibited by the Evolver, but they

[9] The access control mechanism makes it possible for the administrator to configure the objectbase to allow or disallow arbitrary users, members of user groups, and owners to delete objects.

must be done "manually" using other facilities of MARVEL outside the scope of this paper.[10]

### 4.2. Operation

A MARVEL environment stores its objectbase in a binary representation using the Unix dbm utility, for fast retrieval, but the MARVEL system provides ascii2bin and bin2ascii tools. These are used for converting between different machine architectures, as well as by the schema evolution component of the Evolver — which operates directly on the ascii version.

## 5. Consistency

We refer to the consistency-related portion of the process model as the *consistency model*, although it should be understood that this consistency model is not stated separately but instead is projected through consistency annotations on a subset of the effect predicates in the process model. The consistency model guarantees the following: if a consistency predicate is asserted in an effect of a rule (we call this rule the *source rule*), and if that assertion causes the condition of another rule to become satisfied (we call this other rule the *destination rule*), then the activity (if any) of the destination rule must be executed and subsequently an effect of the destination rule must be asserted. This means that if the activity of the destination rule cannot be executed and/or the resulting effect of the destination rule cannot be asserted, then the effect of the source rule must be undone and the source rule must be rolled back.[11] This definition is recursive in the sense that once the destination rule has completed, if there are any consistency predicates in the effect of the destination rule, the destination rule now becomes the source rule for any consistency subchains resulting from the asserted effect. During normal process enaction, the only reasons a consistency chain might not be able to execute in its entirety as a transaction is because of concurrency control conflicts or system failures.

The *consistency graph* of a process model is a directed graph consisting of vertices labeled by rules and edges labeled by predicates. The source of an edge is a rule whose effect contains a consistency predicate that is the label of the edge. The destination of the edge is a rule that has a matching predicate in its condition. This graph then represents all possible attempts at consistency forward chaining. If the condition of the destination rule is satisfied, then the consistency model guarantees that an effect of the destination rule is also asserted. If the condition of the destination rule is evaluated and found to be unsatisfied, the rule is not fired and does not become part of the consistency chain; this ‘‘failure’’ should not abort the preceding consistency chain.

A change to the process model *strengthens consistency* if it adds a new edge to the consistency graph. A change that strengthens consistency often but not necessarily makes the objectbase inconsistent. For example, if we add a consistency predicate to the effect of a rule that results in adding an edge to the consistency graph, and that rule is not the destination of any edges in the consistency graph (i.e., it is not the destination rule of a pre-existing consistency chain) then the new effect need not be asserted.

A change to the process model *weakens consistency* if it removes a pre-existing edge from the consistency graph. Our definition of weakening consistency includes certain changes that would require an update to the objectbase. For example, if we remove a predicate from a conjunct in the condition of a rule, this may

---

[10]We intend to investigate evolution-oriented facilities to *move* file attributes, since this comes up frequently in practice.

[11]As explained in Section 3, activities in such destination rules are not supported in the implementation, solely due to a restriction of the transaction management system.

remove an edge from the consistency graph; however, this change may also allow the condition to be satisfied where it could not be satisfied previously. If such is the case, and there still exists an edge into this rule in the consistency graph, then the consistency model may require that we now fire the rule to assert one of its effects.

There are certain changes that neither strengthen nor weaken consistency. Like the changes that affect consistency, these changes may or may not require an update of the objectbase.

- We say that certain changes to a rule *weaken its condition* if the condition becomes easier to satisfy. This happens when a predicate is removed from a conjunct or added to a disjunct in the property list of a rule.
- We say that certain changes to a rule *strengthen its condition* if the condition becomes harder to satisfy. This happens when a predicate is added to a conjunct or removed from a disjunct in the property list of a rule.
- We say that certain changes *strengthen the effects* of a rule if new predicates could be asserted. This may happen either by adding more predicates to one effect (always in a conjunct) or by adding predicates to be asserted in a new effect (for a new return value from an activity).
- We say that certain changes *weaken the effects* of a rule if the opposite occurs (i.e., predicates are removed from the effect of a rule or an entire effect is removed).

We now describe exactly what kinds of changes may result in an *inconsistent objectbase*.

1. Changes to a rule (or the addition of a rule) that is the destination of an edge in the consistency graph whose source is a pre-existing consistency effect could possibly make the objectbase inconsistent. This is a direct result of what is guaranteed by the consistency model. An effect of a rule need be asserted only if the condition was satisfied by a consistency predicate asserted in the effect of another rule. In other words, if there is no consistency chain into this rule, then we are not forced to assert one of this rule's effects. This means that if new rules are added to the process model, and if these new rules are disjoint from the old rules (in the sense that no pre-existing consistency effect chains into the new rules), then the addition of those rules does not make the objectbase inconsistent.
2. Removing predicates from the effect of a rule (weakening the effects of the rule) can never make the objectbase inconsistent since we can only remove consistency edges (weaken consistency) and/or lessen the effects that can be asserted. It is true that the objectbase may then be in a state that could not have been reached had the new rules always been in place, but the objectbase is nonetheless consistent with the new rules.
3. Adding predicates to the effects of a rule (strengthening the effects) can add a new edge to the consistency graph (strengthen consistency) if the predicate is a consistency predicate. However, the addition of either a consistency or automation predicate to the effects of a rule can make the objectbase inconsistent, and force us to assert the new effect on the objectbase, if, as stated earlier, the modified rule is the destination of a consistency chain from a pre-existing consistency effect.
4. Removing predicates from the condition of a rule may cause a number of changes.
    a. Removing a predicate from the condition may weaken consistency (regardless of whether it was a consistency predicate or an automation predicate) because there may have been a consistency effect from a different rule that chained into the predicate.
    b. Removing a predicate from the condition of a rule weakens the condition if it was part of a conjunct, regardless of the possible impact on the consistency graph. Also, if there is a pre-existing consistency chain into this rule, then the objectbase might

now be inconsistent.

c. Removing a predicate from the condition of a rule strengthens the condition if it was part of a disjunct. Again, this is independent of any impact this has on the consistency graph. This change cannot make the objectbase inconsistent, although it may be in a state that could never have been reached had the new process model always been in place.

5. Adding predicates to the condition of a rule has the complementary effects of those above.

a. Adding a predicate to the condition may strengthen consistency (regardless of whether it is a consistency predicate or an automation predicate) because there may now be a consistency predicate from another rule chaining into this new predicate. If this is the case, the objectbase may be inconsistent, but since there is no way to know a priori whether the condition is now satisfied, an update must be attempted.

b. Adding a predicate to the condition of a rule strengthens the condition if it was part of a conjunct, regardless of the possible impact on the consistency graph.

c. Adding a predicate to the condition of a rule weakens the condition if it was part of a disjunct. Again, this is independent of any impact this has on the consistency graph.

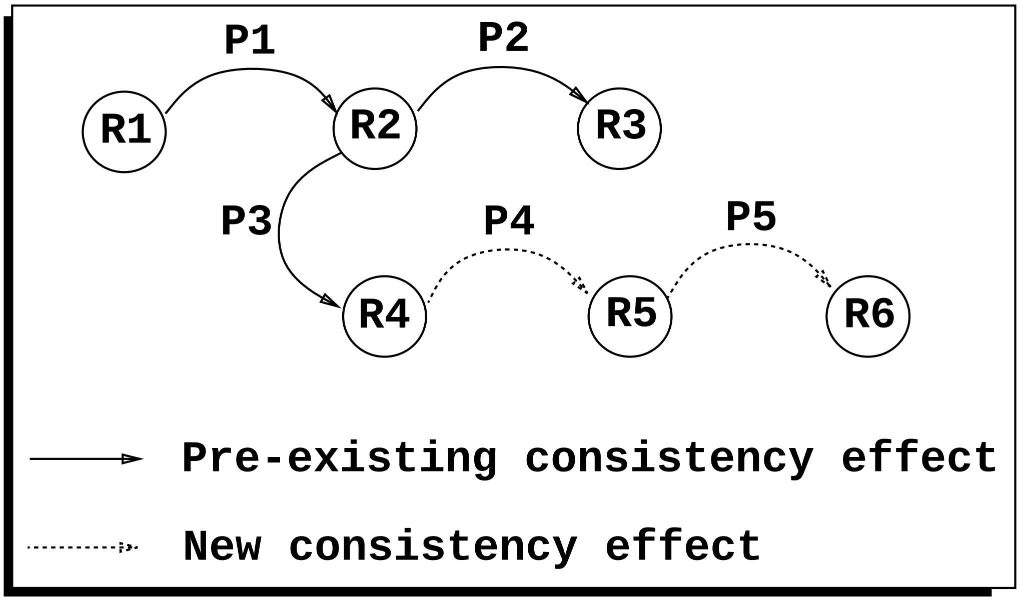


**Figure 5-1:** A Sample Consistency Graph

Sometimes adding new edges to the consistency graph can ‘‘increase inconsistency’’ without actually making the objectbase inconsistent. This can happen only if the source rule of the new edge has only new consistency effects chaining into it. The source rules of those new effects may also have only new consistency effects chaining into them, and so on. However, at some point, the chain must originate form a pre-existing consistency effect. For example, in Figure 5, rule R6 is the destination of the edge P5. This resulted from the addition of the consistency predicate P5 to the effect of rule R5 and possibly also from the addition of P5 to the condition of R6. Since rule R5 has only new consistency predicates chaining into it, the new consistency edge P5 by itself does not make the objectbase inconsistent. However, by looking at the graph, we notice that R5 is the destination of a new edge P4, but this edge comes from a rule that has a pre-existing consistency effect P3 chaining into it. In this example, the new edge came about because the predicate P4 was added to the rule R4. Since a pre-existing consistency effect chains into R4, the addition of P4 may make the objectbase inconsistent. However, if P4 were asserted in an attempt to make the objectbase consistent, P5 still would not have been asserted, and the objectbase would still be inconsistent. That is why the new edge P5 makes the objectbase ‘‘more inconsistent’’: it does not create an inconsistency unless the objectbase is already inconsistent. This would never become a problem unless ‘‘persistent’’ inconsistency is permitted, e.g., by the experimental coordination manager mentioned in Section 3; the problem can be ‘‘solved’’ by requiring all such inconsistency to be removed prior to process evolution.

## 6. Process Model Evolver

Following schema evolution, the Evolver implements process evolution in two distinct phases. The first phase consists of comparing the new process model to the old process model in order to determine if any changes may make the objectbase inconsistent. If this is the case, the administrator is warned and the ‘‘offending’’ rule changes are displayed. The administrator may choose to retract the new process model rather than going ahead with the corresponding objectbase update. The second phase automates the necessary rule invocations to make the objectbase consistent with respect to the new process model. The Evolver generates a MARVEL script that attempts to fire the appropriate rules through the normal process engine.

### 6.1. Capabilities

The first phase of the Evolver warns the administrator of any process model changes that could potentially lead to an inconsistent objectbase. Since the actual objects are not checked, there may be some rule changes that produce a warning but do not actually make the objectbase inconsistent. The following kinds of changes result in warnings:

- The most basic way to make the objectbase inconsistent would be to add a new rule that a pre-existing consistency effect chains into.
- If new predicates are added to the effects of a rule and a pre-existing consistency effect chains into the changed rule, the current objectbase may not reflect the assertion of these new predicates and so it could be inconsistent.
- If a new predicate is added to the property list of a rule and a pre-existing consistency effect chains into this newly added predicate, then we have added a new consistency edge into the rule.
- If the condition to a rule is weakened (i.e., it becomes easier to satisfy), then that condition may be satisfied in instances where it was not satisfied before the change, and the objectbase may be inconsistent if a pre-existing consistency effect chains into the rule.

Changes to the *bindings* can also weaken the condition of a rule, and thus also result in warnings:

- A universally quantified variable must satisfy the relevant predicates in the property list of the rule for all objects bound to it; therefore, the condition can only be weakened when the number of objects bound to the variable is decreased. This means that removing predicates from a disjunct or adding predicates to a conjunct in the binding of a universally quantified variable should produce a warning if a pre-existing consistency effect chains into the rule.
- The opposite is true for an existentially quantified variable. An existentially quantified variable needs to satisfy the property list of the rule for only one of the bindings. Therefore, the condition can be weakened only when the number of objects bound to the variable is increased. Therefore, a warning should be generated whenever a predicate is added to a disjunct or removed from a conjunct in the binding of an existentially quantified variable and a pre-existing consistency effect chains into the rule.

- Changing an existential quantifier to a universal quantifier can weaken the condition since predicates in the property list that involve a universally quantified variable are vacuously true if no objects are bound to the variable. Note that removing an existential binding entirely also falls into this category. Once again, if a pre-existing consistency effect chains into the rule, a warning should be generated.

In the special case described earlier concerning a chain of new consistency edges that originates from pre-existing consistency effect (see Figure 5), the Evolver generates a warning only about the first change in the chain that introduces an inconsistency. This should not pose a problem since, as discussed, the other changes do not make the objectbase inconsistent by themselves, and if the objectbase is updated by firing the rule given in the warning, then this will start a consistency chain that insures that all the necessary updates are done to maintain consistency.

### 6.2. Operation

Also during the first phase, the Evolver adds the ‘‘offending’’ rule to a queue every time a warning is generated. Since the same rule may be the destination of more than one consistency edge, the Evolver always checks for a duplicate rule in the queue before inserting a new one, distinguishing rules by signature.[12]

Once the analysis is complete and the second phase begins, the Evolver generates a batch script of MARVEL commands according to the following simple algorithm. For each offending rule,

1. For each parameter of the rule, create a list of all objects in the objectbase that are instances of the parameter's class or one of its subclasses. (Any schema evolution should already have been completed.)
2. For a rule with multiple parameters, generate the appropriate cross-product from the lists above.
3. For each element in the cross-product, output a script command to fire the rule with those objects as parameters.

The batch script may then be executed.

## 7. Example

The C/Marvel data model divides the environment's objectbase into a shared repository, called the "master area", and a collection of private workspaces, each called a "miniproject". As part of the process, a user initiates a code change by first reserving relevant objects in the master area and then copying them to a miniproject. The user does all editing and testing in the miniproject. Archived libraries in the master area may be linked together with modified code for testing, if objectbase links have been established from objects in the miniproject to the appropriate objects in the master area. Once the changes and test results are satisfactory, the user copies the objects back to the master area and deposits them. Whenever an object representing a source or header file is deposited, the affected archives and executables in the master area are marked as outdated in a mandatory consistency chain, and may be rebuilt then by an optional automation chain or later according to an explicit user command.

In our example, the administrator modifies this process so that outdated or reconstructed archives in the

---

[12] MARVEL permits multiple rules with the same signature but different conditions, so this discussion is somewhat oversimplified.

master area also propagate to all the miniprojects that link to them. The goal is to encourage users to incorporate changes in other parts of the system as soon as they have been deposited, rather than continuing testing using the old versions. We do not intend to claim the new process is necessarily better than the original process for any or all software development projects, just that it is representative of relatively simple but realistic process evolution.

```
arch [?a:AFILE]:
   (and (exists LIB     ?l suchthat (member [?l.afiles ?a]))
        (exists PROJECT ?p suchthat (member [?p.lib    ?l]))
        (forall MODULE  ?m suchthat (linkto [?m.afiles ?a]))):

   # 0. Backward chains to archive each individual MODULE
   # 1. Guarantees this only happens when PROJECT is UnRestricted or
   # 2. CompileAll is set.
   # -----------------------------------------------------------------
   (and (?m.archive_status = Archived)                          # 0.
        (or no_chain (?p.status = UnRestricted)                 # 1.
            no_chain (?p.status = CompileAll)))                 # 2.

   { }

   # 0. Archive this AFILE
   # ----------------------
   (?a.archive_status = Archived);

build [?mp:MINIPROJECT]:
    (and (forall CFILE        ?c    suchthat (member [?mp.files      ?c ]))
         (forall YFILE        ?y    suchthat (member [?mp.files      ?y ]))
         (forall LFILE        ?l    suchthat (member [?mp.files      ?l ]))
         (exists EXEFILE      ?exe  suchthat (member [?mp.exec     ?exe ]))
         (forall MACHINE_EXEC ?me   suchthat (member [?exe.machines ?me ]))
         (forall AFILE        ?a    suchthat (linkto [?mp.afiles     ?a ]))
         (forall MACHINE      ?mc   suchthat (member [?a.machines   ?mc ]))):

    (and (?c.status = Compiled)
         (?l.status = Compiled)
         (?y.status = Compiled))

    { LOCAL build_local ?c.object_code ?l.object_code ?y.object_code
                        ?mc.afile ?me.exec ?exe.history "NO"}

    (?mp.build_status = Built);
    (?mp.build_status = NotBuilt);
```

**Figure 7-1:** Old C/Marvel `arch` and `build` Rules

Two of the old C/Marvel rules, `arch` and `build`, are shown in Figure 7-1. The parameter to `arch` is an `AFILE` object, which represents a Unix archive (".a") file. The job of the bindings is to access the ancestor `PROJECT` object containing this `AFILE` and all the `MODULE` objects linked to this `AFILE`. In the MARVEL objectbase, one object may "contain" other objects in a composite object hierarchy and an object may be "linked to" other objects anywhere else in the objectbase. A `PROJECT` is the shared repository or master area for a software development team. A `MODULE` contains a semantically related set of source and object files, and is linked to the archive for its object code.

The property list of the `arch` rule checks that all of the `MODULE`s linked to the `AFILE` have been `Archived`, and also that the containing `PROJECT` is in either one of the two states `UnRestricted` or `CompileAll`. A `PROJECT` would be in one of these two states if either a source file or header file had been deposited into the master area since the last time the `PROJECT` had been built (i.e., all its executables manufactured), and the `PROJECT` had not yet been rebuilt. This portion of the property list is there to prevent unnecessary work. If the property list already evaluates to true, or can be satisfied through backward chaining to other rules that archive all the relevant MODULEs, then the AFILE itself is said to have been Archived. No tool invocation is actually needed, since the condition directly implies the effect. Rules with no activity are often used in MARVEL process models to indicate relationships among attributes of different objects, and these rules are fired only to propagate changes to such attributes.

The parameter to the build rule is a MINIPROJECT object, which represents a private workspace for an individual software developer. The bindings here are rather complex. They gather up all the C, yacc and lex source files contained in the MINIPROJECT, the MINIPROJECT's executable, and any AFILEs to which it is linked — perhaps in the master area or another miniproject. The MACHINE_EXEC and MACHINE objects are used to maintain archives and executables for different machine architectures; C/Marvel currently supports SparcStations, DecStations and IBM RS6000s, which are the different hardware platforms on which MARVEL itself runs.

The build rule's property list requires that all the source files have been Compiled since the last time the MINIPROJECT was built. If so, the activity invokes the build_local envelope to rebuild the MINIPROJECT, that is, generate the executable being developed by the user. When the envelope terminates, one of the rule's two effects is asserted. If the build was successful, the MINIPROJECT is said to be Built; if the tool produces errors, however, it is NotBuilt.

Figure 7-2 gives the new arch and build rules, which have been modified in the new process model, and Figure 7-3 shows the update and restore rules, which have been added. The only difference in arch is to change its one effect predicate from automation ("(...)") to consistency ("[...]"). This means that when this effect predicate is asserted, consistency forward chaining must be attempted to every rule with a matching predicate in its property list. The new update rule has such a predicate. (It is not significant that the object in both cases is represented by the symbol "?a", only that the symbol is specified as an instance of the same class, AFILE; the object could have been bound to "?a" in one rule and to "?z" in the other.)

The hide directive on the update rule prevents it from being displayed in the user command menu. This is to avoid cluttering the menu with rules that are intended to be invoked only implicitly through forward or backward chaining, never through direct selection by a user. The parameter of update is a MINIPROJECT. When an archive in the master area causes arch to forward chain to update, the binding is ‘‘inverted’’ to find all MINIPROJECT objects that are linked to the particular AFILE object on which the consistency effect was asserted. The update rule is then instantiated separately for each such object.

Then for each of the instantiated rules, the property list is evaluated. If the MINIPROJECT is either INC_NotBuilt or NotBuilt, it is set to NotBuilt. Basically, NotBuilt means that the MINIPROJECT is out of date because one or more of the archives it imports from the master area has been updated since it was last built. INC_NotBuilt means that one of these archives has been outdated but not yet reconstructed; this is set by the new restore rule when an imported archive becomes NotArchived. The effective difference is that a MINIPROJECT can be directly rebuilt when its status is NotBuilt, but it is first necessary to reconstruct the relevant archives in the master area if its status is INC_NotBuilt. (NotBuilt and INC_NotBuilt are also overloaded with other possible meanings that are not relevant here.)

The modified build rule considers the relationship between a MINIPROJECT and the AFILEs it is linked to. If a source file within the workspace is recompiled <u>and</u> the imported archives have already

```
arch [?a:AFILE]:
   (and (exists LIB     ?l suchthat (member [?l.afiles ?a]))
        (exists PROJECT ?p suchthat (member [?p.lib    ?l]))
        (forall MODULE  ?m suchthat (linkto [?m.afiles ?a]))):

   # 0. Backward chains to archive each individual MODULE
   # 1. Guarantees this only happens when PROJECT is UnRestricted or
   # 2. CompileAll is set.
   # -------------------------------------------------------------
   (and (?m.archive_status = Archived)                      # 0.
        (or no_chain (?p.status = UnRestricted)             # 1.
            no_chain (?p.status = CompileAll)))             # 2.

   { }

   # 0. Archive this AFILE
   # ---------------------
   [?a.archive_status = Archived];


build [?mp:MINIPROJECT]:
    (and (forall CFILE        ?c    suchthat (member [?mp.files      ?c ]))
         (forall YFILE        ?y    suchthat (member [?mp.files      ?y ]))
         (forall LFILE        ?l    suchthat (member [?mp.files      ?l ]))
         (exists EXEFILE      ?exe  suchthat (member [?mp.exec     ?exe ]))
         (forall MACHINE_EXEC ?me   suchthat (member [?exe.machines ?me ]))
         (forall AFILE        ?a    suchthat (linkto [?mp.afiles     ?a ]))
         (forall MACHINE      ?mc   suchthat (member [?a.machines   ?mc ]))):

    (or (and (?c.status = Compiled)
             (?l.status = Compiled)
             (?y.status = Compiled)
             no_forward (?a.archive_status = Archived))
        (?mp.build_status = NotBuilt))

    { LOCAL build_local ?c.object_code ?l.object_code ?y.object_code
                        ?mc.afile ?me.exec ?exe.history "NO"}

    # 0. Assert proper build
    # 1. Problems occurred.  Have no_chain() to prevent interfering with
    #    normal chaining on this predicate.
    # ---------------------------------------------------------------
    (?mp.build_status = Built);                                 # 0.
    no_chain (?mp.build_status = NotBuilt);                     # 1.
```

**Figure 7-2:** Modified C/Marvel `arch` and `build` Rules

been constructed, then the `MINIPROJECT` can be built. Alternatively, if an imported archive has just been reconstructed, so the status has become `NotBuilt`, it is also appropriate to now rebuild the `MINIPROJECT`. The no_chain directive was added to the second effect to prevent a cycle, that is, if the `build` is unsuccessful, it is futile to immediately try again.

Now we consider evolution of a C/Marvel objectbase from the old process model to the new one. The data model has not changed, and the only changes in the process model are those described above. We say "a" C/Marvel objectbase rather than "the" C/Marvel objectbase because there can be an arbitrary number of environment instances, each with its own objectbase, employing the same process. In any case, we always test an evolved process and/or data model on a toy objectbase before applying it to the C/Marvel objectbase containing MARVEL's own source code!

The command to the Evolver is "evolve . be_local.load", where "." means the current directory contains the relevant objectbase and associated files, and "be_local.load" is the top-level MSL file that directly or

```
hide update[?mp:MINIPROJECT]:
    (forall AFILE ?a suchthat (linkto [?mp.afiles ?a])):

    # 0. Forward chain once Master area library rebuilt.
    # 1. Restrict access to this rule until either restore is fired ...
    # 2. (only to prevent rule termination) or update itself fired.
    #    Note: (2) makes this rule idempotent.
    # -----------------------------------------------------------------
    (and (?a.archive_status = Archived)                           # 0.
         (or no_backward (?mp.build_status = INC_NotBuilt)        # 1.
             no_chain (?mp.build_status = NotBuilt)))             # 2.
    { }
    (?mp.build_status = NotBuilt);

#
# won't force to recompile local area.  Just notifies.
#
hide restore[?mp:MINIPROJECT]:
    (forall AFILE ?a suchthat (linkto [?mp.afiles ?a])):

    # 0. Don't BC, but still want to come here when master area
    #    is made dirty.
    # ---------------------------------------------------------
    no_backward (?a.archive_status = NotArchived)

    { }

    no_chain (?mp.build_status = INC_NotBuilt);
```

**Figure 7-3:** New C/Marvel `update` and `restore` Rules

indirectly imports all the other MSL files that collectively define C/Marvel.[13] "MSL" stands for MARVEL Strategy Language, the combined notation in which data and process models are written. The MSL input to the MARVEL kernel consists of one or more files, where a top-level file directly or indirectly imports all the other files. Individual MSL files thus represent data and/or process model ‘‘modules’’. The names of each MSL file are listed in the Evolver output, given in Appendix II, as they are encountered.

The Evolver first detects that there were no changes to the data model, so no schema evolution is required in this case. It then attempts to match rules in the old and new process model pairwise, and requests help interactively from the administrator when there is an ambiguity. It discovers the ‘‘change’’ to `restore`, which strengthens consistency because it is the destination of a consistency effect of another rule — `touchup` — that has not been shown. The relevant portion of the new process model's consistency graph is shown in Figure 7-4; the relevant portion of the old process model's consistency graph was exactly the same except for the new vertex for `restore` and the new edge from `touchup` to `restore`.

The Evolver also finds the change to `update`, which is the destination of the new consistency predicate in `arch`. The tool does not print out anything about `arch`, since it is the source not the destination of strengthened consistency, and does not mention `build` because the no_forward directive effectively removes the incoming consistency edge.[14]

The Evolver generates the MARVEL batch script shown in Figure 7-5 to trigger the offending `restore`

[13] The top-level file is not called "cmarvel" or something else more intuitive than "be_local" for historical reasons that involved an alternative "be_global" process.

[14] The rationale for this, like most other instances of chaining directives, is subtle — so omitted here.

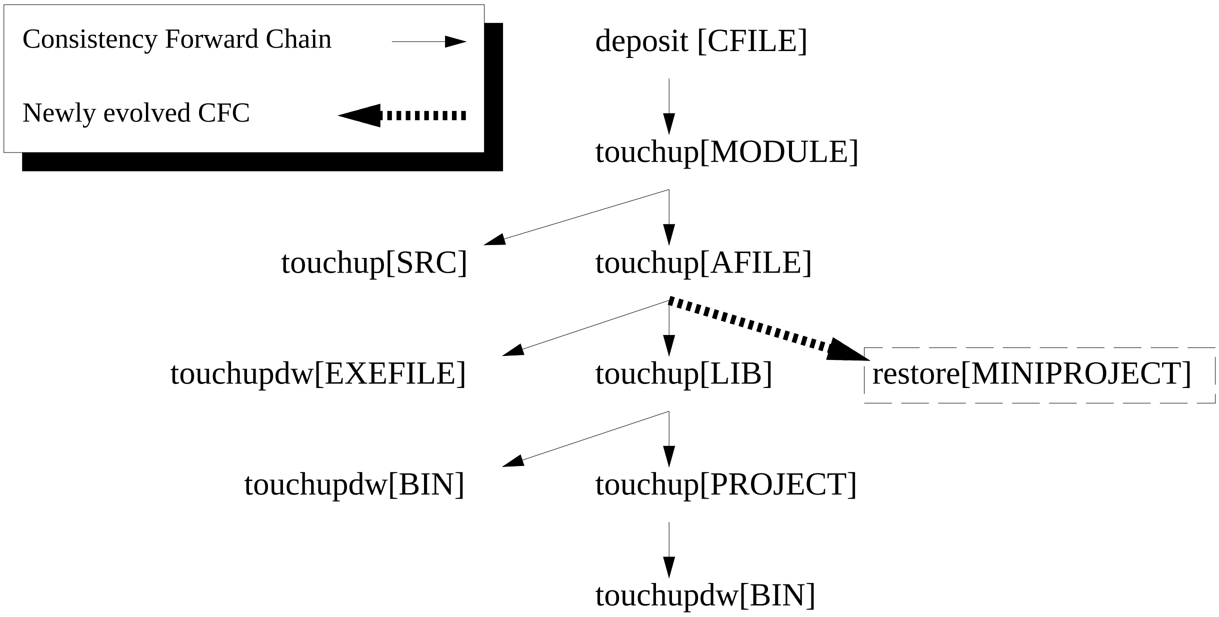


**Figure 7-4:** Consistency Subgraph with `restore`

```
# marvel script
#

#restore [ Env ]
restore <19>

#restore [ Make.Env ]
restore <32>

#update [ Env ]
update <19>

#update [ Make.Env ]
update <32>
```

**Figure 7-5:** Batch Command Script Generated by the Evolver

and `update` rules on all affected objects. This would be all objects that are instances of the `MINIPROJECT` class or any subclasses of `MINIPROJECT` (there are no such subclasses in C/Marvel). In the testing environment, there are only two `MINIPROJECT`s, named "Env" and "Make.Env". Their internal OIDs (object identifiers) are 19 and 32, respectively. The script uses OIDs rather than names (provided in comments for convenience) because in the general case there may be multiple instances of the same class with the same name (consider the "main.c" CFILE needed for every program). However, it is not usually necessary for human users of MARVEL to know the OIDs of objects when entering commands, because they can click on the intended objects in the graphical user interface.

The batch script is normally executed in a MARVEL command line client spawned by the Evolver. For the purposes of this paper, we actually executed the script using an XView client. When the new `update` rule is applied to the "Env" `MINIPROJECT`, the chaining window gradually displays the output shown in Figure 7-6 as each rule is fired. The corresponding portion of the rule network, containing automation backward chaining edges as well as the new consistency edge, is shown in 7-7. Note this is <u>not</u> a consistency graph, but the relevant part of the full rule network used by the process engine to represent all

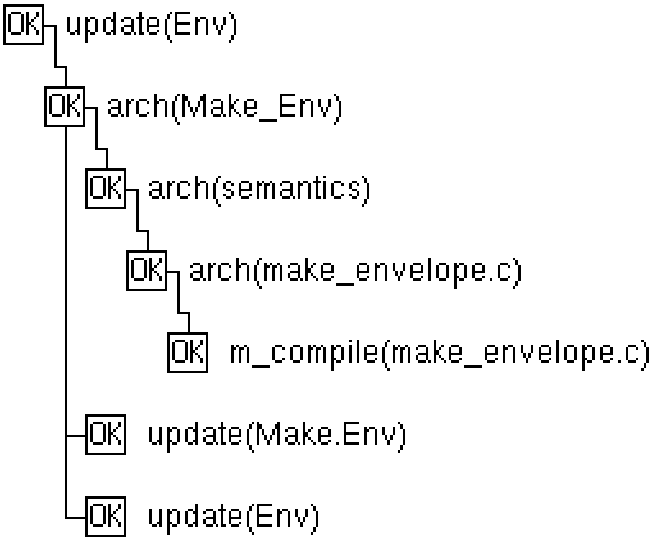


**Figure 7-6:** XView Client Chaining Display for `update <19>`

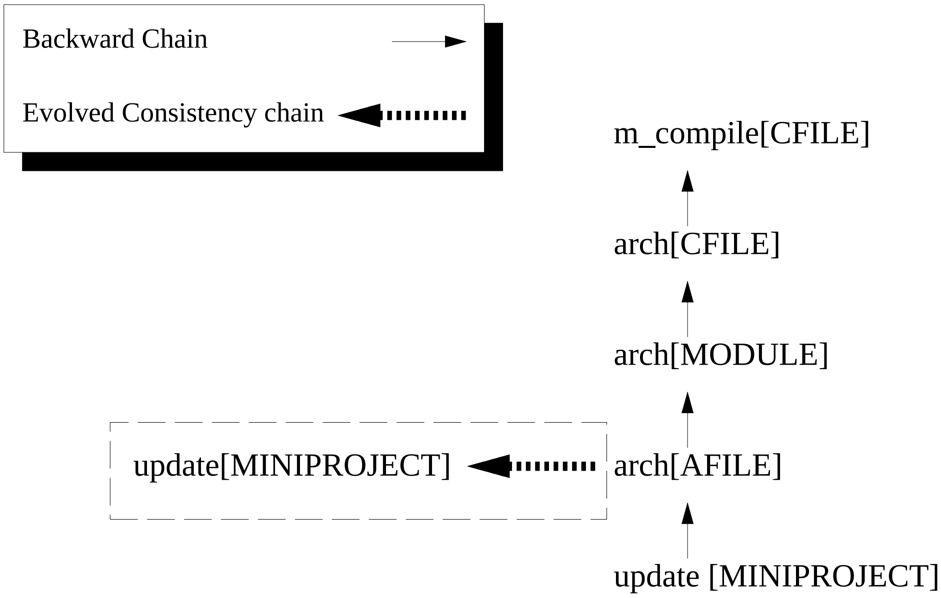


**Figure 7-7:** Subnetwork Containing `update` Rule

possible forward and backward chaining. The actual objectbase is displayed at the end of Appendix II.

`update` on "Env", represented by the topmost icon, *automation backward chains* to the `arch` rule for each of the `AFILE`s that "Env" is linked to. This is done to attempt to set the `AFILE`'s status to `Archived`, since this predicate in the property list is not already true. The only linked `AFILE` is named "Make_Env"; it should not be confused with the "Make.Env" `MINIPROJECT`. So `arch` is invoked on the "Make_Env" `AFILE` in the master area; this corresponds to the second icon.[15]

[15]The rather confusing object naming scheme results from the use of an existing objectbase developed for internal testing purposes, rather than the construction of a new objectbase solely for this paper.

This rule in turn backward chains to another `arch` rule for each `MODULE` linked to this `AFILE`. MARVEL is not just instantiating the same rule with a different parameter, but instead a distinct rule also named `arch`, with a parameter of class `MODULE`. Recall that MARVEL permits overloading of multiple rules with the same name but different signatures (different orders and/or types of parameters). The "Make_Env" `AFILE` is linked to the "semantics" `MODULE`, also in the master area, so the third icon represents this rule firing.

The `arch` rule parameterized by a `MODULE`, not shown, requires that all source files contained in the `MODULE` also be `Archived`. Since this is not the case for one C file, "make_envelope.c", yet another `arch` rule is applied, the fourth icon. This in turn backward chains to the `m_compile` rule, whose activity actually compiles "make_envelope.c", represented by the fifth icon. This backward chain then unwinds to the second icon, since all the relevant conditions have been satisfied.

Now, the first `arch` invoked on the "Make_Env" `AFILE` *consistency forward chains* to trigger the `update` rule on all `MINIPROJECT`s that link to it. This is first instantiated with the "Make.Env" `MINIPROJECT`, the sixth icon. The `build` rule is not triggered due to the no_forward directive. And, as it turns out, the consistency forward chain from `arch` also invokes the `update` rule on "Env", shown by the seventh and final icon, even though it was another instance of this same update rule on the same parameter object that triggered the whole process enaction in the first place.

Thus when the backward chain from the initial `update` to `arch` unwinds, the former rule has already been executed. But the process engine goes ahead and executes it again, since the condition is now satisfied and there is no chaining directive to prevent this. The no_chain directive on the `NotBuilt` predicate would be relevant only if assertion of a matching effect were the reason for the attempted execution (and this did in fact prevent a cycle when the `update` was completed on "Env" earlier). Note that the later `update` on "Make.Env" in the batch script will still be executed (generating a separate chaining picture, not shown), even though it has incidentally already been executed.

## 8. Summary

We described a schema and process evolution tool, called the Evolver, implemented for the MARVEL process-centered environment. The schema evolution facilities are based on those developed for the Orion object-oriented database management system, but the process evolution mechanism is new. The Evolver analyzes the differences between the new and installed process models of an existing environment, detecting each case where the notion of consistency defined by the process model has been strengthened or weakened. The Evolver then automatically updates the environment’s objectbase to guarantee that the objects are not inconsistent according to the new process model. This is accomplished by determining the set of rules affected by changes related to consistency, and generating a list of all possible combinations of parameters for these rules. Then the instantiated rules are applied by MARVEL’s normal process engine as if they were user commands. The Evolver can be used while the old process is in progress, provided no actual chaining is being performed. Development may then continue using the new process. We have had several months of experience using the Evolver to make repeated changes in the C/Marvel process that supports our own continuing development of MARVEL, as well as numerous test cases, and have generally found the tool to be both practical and useful.

## Acknowledgments

Except as noted, all facilities mentioned in this paper are already implemented in the internal version MARVEL 3.0.5, which will be demonstrated at CASCON ’92 and SIGSOFT ’92, and will be released as part of MARVEL 3.1 in early 1993. The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants and fellowships from AT&T, BNR, Bull, DEC, IBM, Paramax 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. Non-profit institutions and potential industrial sponsors may send email to MarvelUS@cs.columbia.edu for information about licensing MARVEL.

## References

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

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

[3] Jay Banerjee and Won Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD Annual Conference on the Management of Data*, pages 311-322. San Francisco CA, May, 1987. Special issue of *SIGMOD Record*, 16(3), December 1987.

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

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

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

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

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

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

[10] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In Hector Garcia-Molina and H.V. Jagadish (editors), *1990 ACM SIGMOD International Conference on Management of Data*, pages 194-203. Atlantic City NJ, May, 1990. Special issue of *SIGMOD Record*, 19(2), June 1990.

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

[12] Reidar Conradi, Espen Osjord, Per H. Westby and Chunnian Liu. Initial Software Process Management in EPOS. *Software Engineering Journal* 6(5):275-284, September, 1991.

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

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

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

[16] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti and Israel Z. Ben-Shaul. Rule Chaining in MARVEL: Dynamic Binding of Parameters. *IEEE Expert* , December, 1992. In press.

[17] 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.

[18] 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.

[19] Gail E. Kaiser, Steven S. Popovich and Israel Z. Ben-Shaul. *A Bi-Level Language for Software Process Modeling*. Technical Report CUCS-016-92, Columbia University Department of Computer Science, September, 1992. Submitted for publication.

[20] Takuya Katayama (editor). *6th International Software Process Workshop: Support for the Software Process*. IEEE Computer Society Press, Hakodate, Japan, 1990.

[21] Won Kim, Nat Ballou, Jorge F. Garz and Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems* 9(1):31-51, January, 1991.

[22] Amaury Legait, Flavio Oquendo and Dan Oldfield. MASP: A Model for Assisted Software Processes. In Fred Long (editor), *Lecture Notes in Computer Science*. Number 467: *Software Engineering Environments International Workshop on Environments*, pages 57-67. Springer-Verlag, Chinon, France, 1989.

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

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

[25] Randy Pausch. *Adding Input and Output to the Transactional Model*. PhD thesis, Carnegie Mellon University, August, 1988. CMU-CS-88-171.

[26] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering* SE-1:364-370, 1975.

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

[28] Andrea Skarra and Stanley Zdonik. The Management of Changing Types in an Object-Oriented Database. In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 483-495. ACM, Portland OR, September, 1986. Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[29] Michael H. Sokolsky and Gail E. Kaiser. A Framework for Immigrating Existing Software into New Software Development Environments. *Software Engineering Journal* 6(6):435-453, November, 1991.

# I. Generic MARVEL Architecture

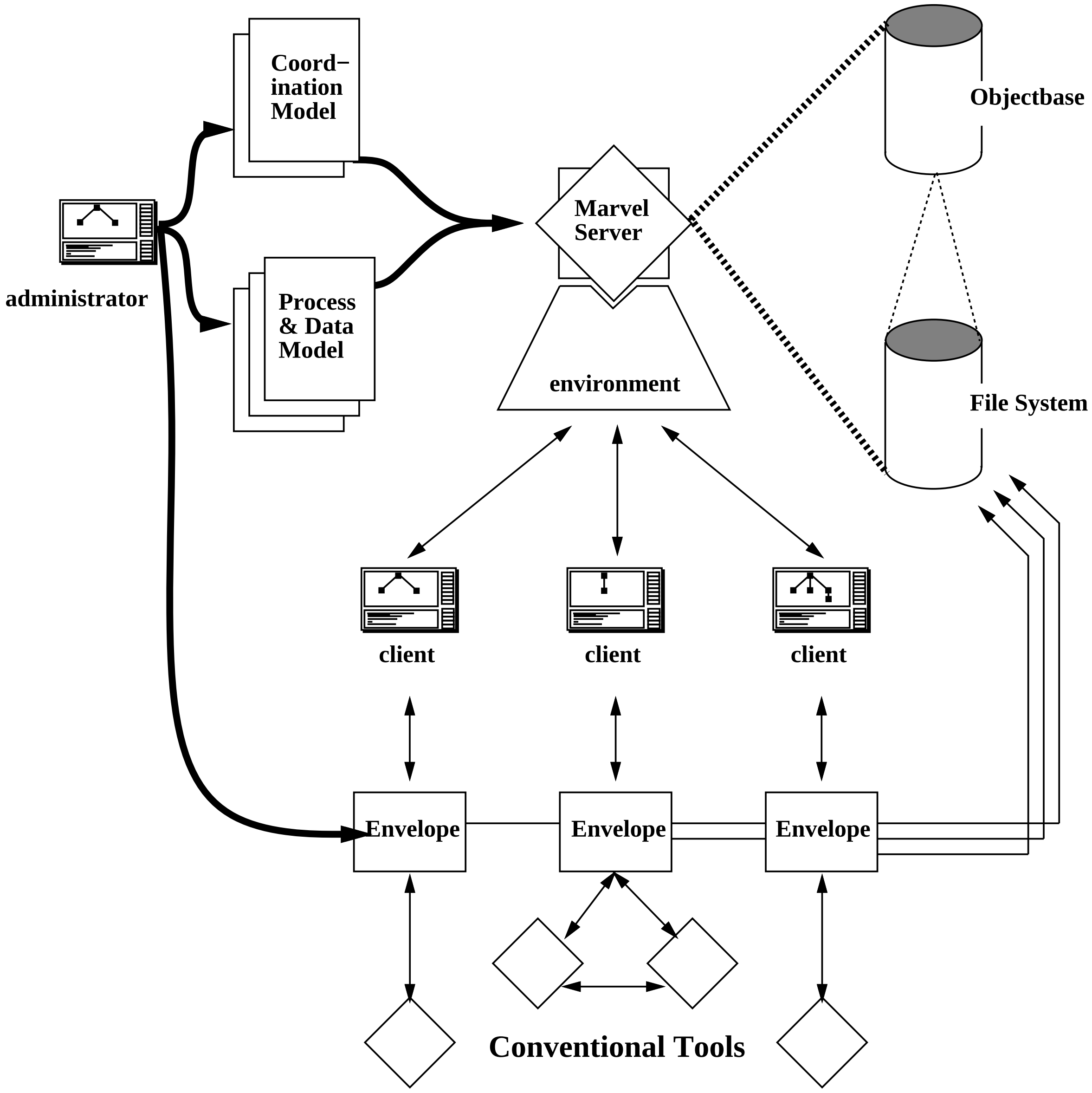


**Figure I-1:** Generic Marvel 3.1 Environment

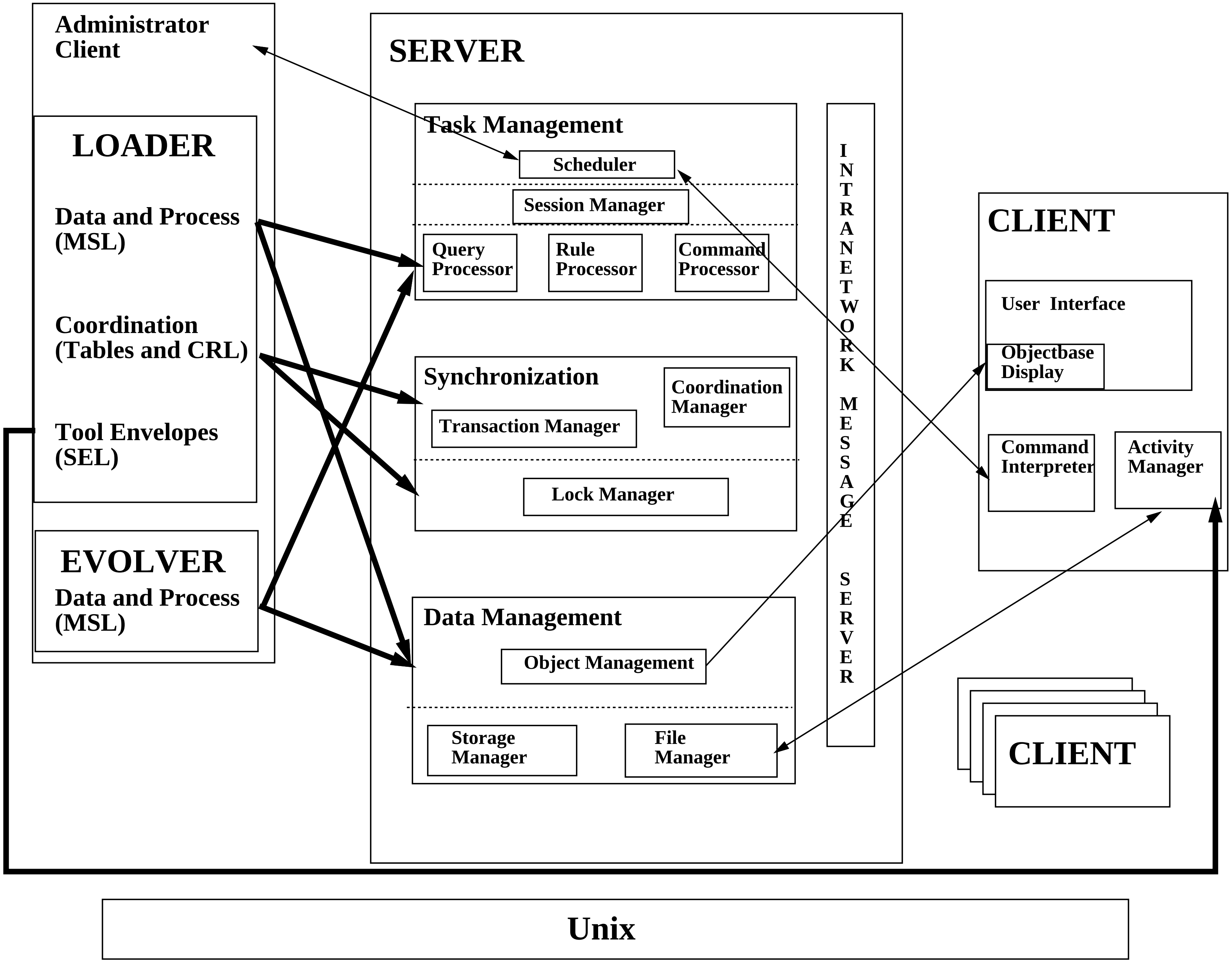


**Figure I-2:** Marvel 3.1 Architecture

## II. Raw Evolver I/O

The following is the complete input and output of the Evolver session for the example. Text in *italics* is entered by the MARVEL administrator. The rest is output by the Evolver (except the "$" prompt comes from Unix).

```
$ evolve . be_local.load

     * * * Evolver starting * * *
Running parser on be_local.load ...
parsing:be_local

Running parser on std_built_ins.load ...
parsing:std_built_ins

Running parser on data_model.load ...
parsing:data_model

Running parser on protection.load ...
parsing:protection

Running parser on archive.load ...
parsing:archive

Running parser on build.load ...
parsing:build

Running parser on print.load ...
parsing:print

Running parser on marvel.load ...
parsing:marvel

Running parser on local_edit.load ...
parsing:local_edit

Running parser on rcs.load ...
parsing:rcs

Running parser on doc.load ...
parsing:doc

Running parser on touch.load ...
parsing:touch

Running parser on mail.load ...
parsing:mail

Running parser on release.load ...
parsing:release

Running parser on owner.load ...
parsing:owner

Running parser on permissions.load ...
parsing:permissions

Running parser on local.load ...
parsing:local

Running parser on new.load ...
parsing:new

Running parser on protection.load ...
parsing:protection

Running parser on data_model.load ...
parsing:data_model
```

```
Running parser on std_built_ins.load ...
parsing:std_built_ins

Running parser on archive.load ...
parsing:archive

Running parser on build.load ...
parsing:build

Running parser on print.load ...
parsing:print

Running parser on marvel.load ...
parsing:marvel

Running parser on rcs.load ...
parsing:rcs

Running parser on local_edit.load ...
parsing:local_edit

Running parser on doc.load ...
parsing:doc

Running parser on touch.load ...
parsing:touch

Running parser on mail.load ...
parsing:mail

Running parser on release.load ...
parsing:release

Running parser on owner.load ...
parsing:owner

Running parser on permissions.load ...
parsing:permissions

Running parser on local.load ...
parsing:local

Running parser on new.load ...
parsing:new

Running parser on be_local.load ...
parsing:be_local

Loader: done parsing
Warning: Archaic use of object as parameter in activity invocation for rule notify
Warning: Archaic use of object as parameter in activity invocation for rule arch
Warning: Archaic use of object as parameter in activity invocation for rule locked
Warning: Archaic use of object as parameter in activity invocation for rule locked
Warning: Archaic use of object as parameter in activity invocation for rule locked
Warning: Archaic use of object as parameter in activity invocation for rule mail

[NO CLASSES CHANGED]

[NO NEW CLASSES ADDED]
No changes to the data model have been made.
Objectbase unchanged.
More than one rule with same name and same parameters found
Please match the following OLD rule:
RULE: touchup [ ?Proj : PROJECT ]

CONDITION:
(exists INC ?i suchthat (MEMBER [?Proj.include ?i]))
:
```

```
[?i.archive_status = INC_NotArchived]


EFFECTS:
0: (AND [?Proj.build_status = INC_NotBuilt]
        [?Proj.status = CompileAll])


With one of the following NEW rules:
Rule 1:
RULE: touchup [ ?Proj : PROJECT ]

CONDITION:
(exists LIB ?l suchthat (MEMBER [?Proj.lib ?l]))
:
[?l.archive_status = NotArchived]


EFFECTS:
0: (AND [?Proj.build_status = NotBuilt]
        [?Proj.status = UnRestricted])



Rule 2:
RULE: touchup [ ?Proj : PROJECT ]

CONDITION:
(exists INC ?i suchthat (MEMBER [?Proj.include ?i]))
:
[?i.archive_status = INC_NotArchived]


EFFECTS:
0: (AND [?Proj.build_status = INC_NotBuilt]
        [?Proj.status = CompileAll])



Choose one of the rules (1-2 or 0 for none):
2

Match with Rule 2? [y/n]

y
Rule change to
 restore [ ?mp : MINIPROJECT ]
strengthened consistency and may make
the objectbase_insconsistent.
More than one rule with same name and same parameters found
Please match the following OLD rule:
RULE: touchup [ ?l : LIB ]

CONDITION:
(exists AFILE ?a suchthat (MEMBER [?l.afiles ?a]))
:
[?a.archive_status = NotArchived]


EFFECTS:
0: [?l.archive_status = NotArchived]


With one of the following NEW rules:
Rule 1:
RULE: touchup [ ?l : LIB ]

CONDITION:
```

```
(exists AFILE ?a suchthat (MEMBER [?l.afiles ?a]))
:
[?a.archive_status = INC_NotArchived]


EFFECTS:
0: [?l.archive_status = INC_NotArchived]



Rule 2:
RULE: touchup [ ?l : LIB ]

CONDITION:
(exists AFILE ?a suchthat (MEMBER [?l.afiles ?a]))
:
[?a.archive_status = NotArchived]


EFFECTS:
0: [?l.archive_status = NotArchived]



Choose one of the rules (1-2 or 0 for none):
2

Match with Rule 2? [y/n]

y
More than one rule with same name and same parameters found
Please match the following OLD rule:
RULE: touchdw [ ?m : MODULE ]

CONDITION:
(exists SRC ?s suchthat (MEMBER [?s.modules ?m]))
:
[?s.archive_status = INC_NotArchived]


EFFECTS:
0: [?m.archive_status = INC_NotArchived]


With one of the following NEW rules:
Rule 1:
RULE: touchdw [ ?child : MODULE ]

CONDITION:
(forall MODULE ?m suchthat (MEMBER [?m.modules ?child]))
:
[?m.archive_status = INC_NotArchived]


EFFECTS:
0: [?child.archive_status = INC_NotArchived]



Rule 2:
RULE: touchdw [ ?m : MODULE ]

CONDITION:
(exists SRC ?s suchthat (MEMBER [?s.modules ?m]))
:
[?s.archive_status = INC_NotArchived]


EFFECTS:
```

```
0: [?m.archive_status = INC_NotArchived]


Choose one of the rules (1-2 or 0 for none):
2

Match with Rule 2? [y/n]

y
Rule change to
 update [ ?mp : MINIPROJECT ]
strengthened consistency and may make
the objectbase_insconsistent.
Rule changes have made the objectbase inconsistent.

Writing file strategy.new . . .
strategy --> strategy.old . . .
strategy.new --> strategy . . .
Running ascii2bin . . .
New objectbase/strategy installed.
     * * * Evolver complete * * *

$
```

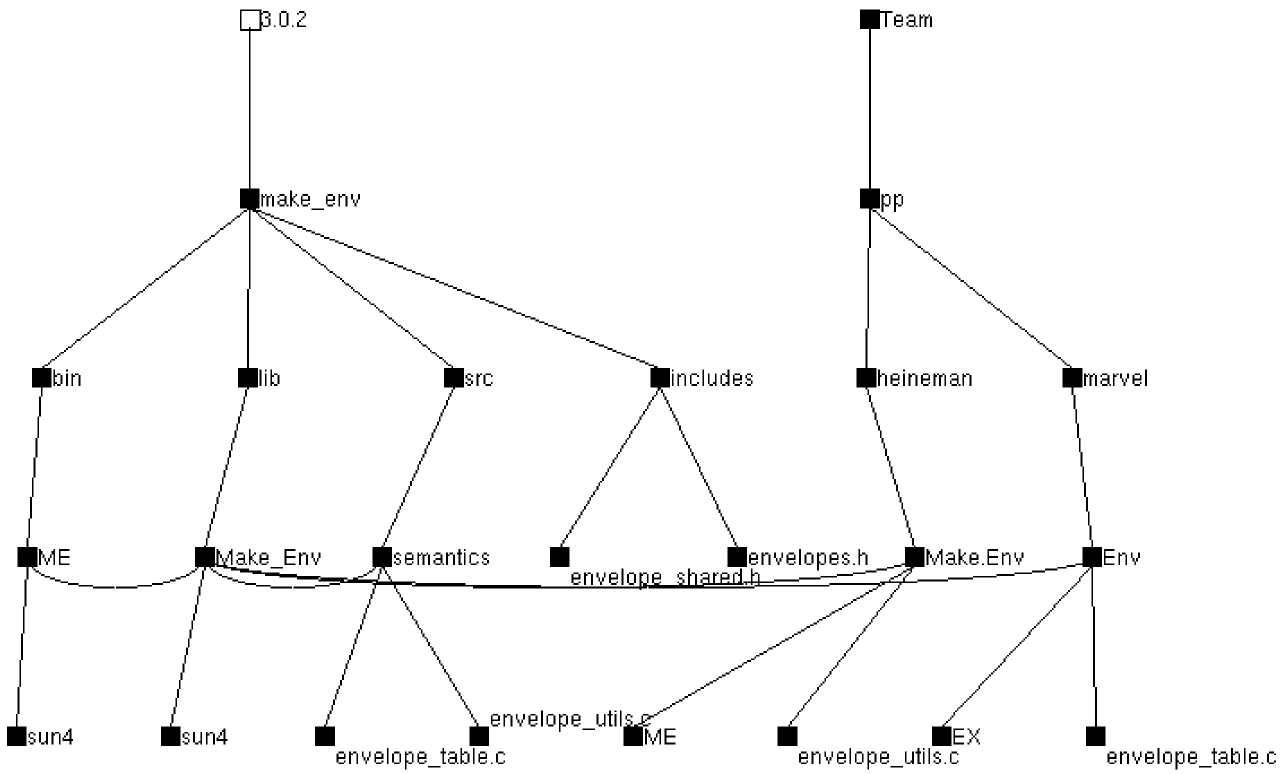


**Figure II-1:** Display of Small C/Marvel Objectbase