SCALING UP
RULE-BASED SOFTWARE DEVELOPMENT ENVIRONMENTS

Naser S. Barghouti Gail E. Kaiser

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

Abstract

Rule-based software development environments (RBDEs) model the software development process in terms of rules that encapsulate development activities, and assist in executing the process via forward and backward chaining over the rule base. We investigate the scaling up of RBDEs to support (1) multiple views of the rule base for multiple users and (2) evolution of the rule base over the lifetime of a project. Our approach is based on clarifying two distinct functions of rules and chaining: maintaining consistency and automation. By definition, consistency is mandatory whereas automation is not. Distinguishing the consistency and automation aspects of RBDE assistance mechanisms makes it possible to formalize the range of compatible views and the scope of mechanizable evolution steps. Throughout the paper, we use the MARVEL RBDE as an example application.

Appeared in *International Journal on Software Engineering & Knowledge Engineering*, World Scientific, 2(1):59-78, March 1992.

1. Introduction

Process-centered software development environments assist their users in carrying out the software development process [Perry 89, Katayama 90]. In order to accommodate long-term projects involving teams of software personnel, it is essential to support *multiple views* and *evolution*.

Multiple views are needed for several purposes. Individual users can express personal tailoring of localized aspects of the process; managers can impose their different management styles regarding the rigidity of the process on their groups. Views for different user roles or lifecycle phases can emphasize the subset of functionality appropriate to those roles or phases, respectively.

Evolution is inescapable due to changes in management philosophy and optimization of the development process [Humphrey 89], reorganization of the project components as their number and complexity grows, and changes to the tool set as new tools become available and older tools are discarded.

The problems of multiple views and evolution are related, because they are both concerned with divergent versions of the process definition, the data format, and the tool set of an

environment. But in the evolution case the divergence is over time, whereas there may be multiple views at the same point in time. Multiple views necessarily co-exist with one another, while evolution is not necessarily backward compatible and in fact may require upgrading or invalidation of previous views. Note that we are concerned here with the evolution of the software development process, project organization schema and tool set, as opposed to the evolution of the product being developed or of the underlying environment framework.

Different process-centered environments represent the process using different formalisms. Arcadia [Taylor 88] uses an extension of Ada as a process programming language [Sutton 90a]. HFSP employs a form of attribute grammars [Katayama 89]. MELMAC combines several perspectives on the process into a representation similar to Petri nets [Deiters 90].

It seems likely that the approach to multiple views and evolution will depend to a large extent on the process modeling formalism. This paper describes our investigation of the problems of multiple views and evolution in the context of process-centered environments that use *rules* to define the software process. We chose rules because they provide a natural primitive formalism for defining software development activities, in terms of their conditions and effects. Other more abstract formalisms have been elegantly defined on top of rules [Deiters 90, Kaiser 91], but this issue is outside the scope of this paper.

Rule-based software development environments (RBDEs) are a subclass of process-centered environments. In RBDEs, each step in the process is modeled by a rule that specifies the *condition* for initiating the step and the *effect* of completing the step. The schema (or data model) is typically specified in an object-oriented style, as a set of classes. The rule conditions and effects operate on attributes of objects, instances of the classes. Example RBDEs include Grapple [Huff 88], Darwin [Minsky 91], Workshop [Clemm 88], Oikos [Ciancarini 93] and ALF [Legait 89]. In an RBDE, a view is a subset of the rules, and the main problem introduced by adding multiple views to an RBDE lies in the inter-operability of views with respect to shared objects. Evolution consists of modifications to rules, and the most significant difficulty is upgrading existing objects so they can be correctly manipulated by the new or modified rules in the environment.

We analyzed the RBDE paradigm, to find clues toward a solution for these problems. We discovered that rules serve two distinct purposes: to express *consistency constraints* and to

express opportunitiesfor *automation*. Maintaining consistency is mandatory whereas carrying out automation is optional. Distinguishing between these two functions of rules results in an elegant approach to supporting both multiple views and evolution. In particular, an RBDE can support multiple simultaneous views if they share a common set of consistency constraints so no view can make changes resulting in inconsistency with respect to another view, while there are no such restrictions on overlapping notions of automation. Further, mechanical evolution of the consistency model is always feasible towards weaker levels of consistency or the addition of new constraints completely disjoint from the previous process and data models (considering initial values), but again there are no such limitations on evolution of the automation model.

We first give an overview of RBDEs, using our own MARVEL environment [Kaiser 88a] as an example, and describe a chaining algorithm for ‘‘enacting’’ software processes. We also present an example that we use throughout the paper. Next, we show how chaining has both consistency and automation aspects, and then introduce a ‘‘maximalist’’ chaining mechanism that formalizes the distinction between consistency constraints and automation opportunities. An extension of this mechanism has been implemented in the first multi-user MARVEL, version 3.0 [Programming Systems Laboratory 91a, Programming Systems Laboratory 91b], while the previous single-user versions of MARVEL, culminating in version 2.6, did not distinguish these concerns [Kaiser 90]. In the next two sections, we explain how the separation of consistency from automation may be exploited to support multiple views and evolution. The paper ends with a comparison of related work concerned with multiple views and evolution in software development environments, and a summary of our contributions.

2. MARVEL Overview

Every software project assumes a specific development process and a particular organization for its components, as well as a set of suitable software tools. Since these may be quite different for different projects, it is inappropriate to build a single software development environment that has a fixed development process, data model and tool set. Instead, RBDEs provide a common kernel that can be *tailored* to the particular project by providing a project-specific process, data model and tool set.

In the MARVEL RBDE, a project administrator specifies a model of the development process in terms of rules (the *project rule set*) and a model of the project’s data in terms of object-

oriented classes (the *project type set*), and writes envelopes that interface to external tools (the *project tool set*). These descriptions are then loaded into the MARVEL kernel, tailoring it as a MARVEL environment. The organization of the project components is abstracted into a hierarchy of complex objects, each an instance of one of the administrator-defined classes. These objects are manipulated by the software development tools installed on the operating system. The tools do not manipulate the objects' attributes directly, but operate on files and directories that are mapped to these attributes [Kaiser 88b].

2.1. Rules

```
reserve[?c:FILE]:
    :
(?c.reservation_status = Available)
{ RCS reserve ?c.contents ?c.version }
(?c.reservation_status = CheckedOut);

edit[?c:CFILE]:
    :
(?c.reservation_status = CheckedOut)
{ EDITOR edit ?c.contents }
(and (?c.compile_status = NotCompiled
     (?c.time_stamp = CurrentTime));

compile[?c:CFILE]:
    :
(?c.compile_status = NotCompiled)
{ COMPILER compile ?c.contents ?c.object_code
                   ?c.error_msg "-g" }
(and (?c.compile_status = Compiled)
     (?c.object_time_stamp = CurrentTime));
(?c.compile_status = Error);

dirty[?m:MODULE]:
(forall CFILE ?c suchthat (member [?m.cfiles ?c]))
    :
(?c.object_time_stamp > ?m.time_stamp))
{ }
(?m.archive_status = NotArchived);
```

Figure 2-1: Example Rules

The development process of a project is modeled in terms of rules. Most rules control the execution of a development *activity*, typically the invocation of a tool. They specify the condition under which it is appropriate to initiate the tool and the possible effects of the tool on the values of objects' attributes. Since such rules invoke external tools that might either succeed or fail, they typically have multiple mutually exclusive sets of effects. For example, a compiler might succeed in producing object code, or fail and produce error messages. Thus, the rule encapsulating the compilation activity must specify two sets of effects, one to be asserted in case of success and the other in case of failure.

Some rules are not associated with activities (i.e., the activity part is left empty), in which case there can be only a single effect, an expression that is a logical consequence of the condition of the rule. Such rules define relations among attributes of the same or different objects and accordingly derive new values for attributes of the objects. All rules are parameterized to take as arguments one or more objects, each of which is an instance of some class. Four MARVEL rules for a C programming environment are shown in figure 2-1.

Each rule has a name followed by a list of parameters enclosed in square brackets "[...]". Rule names may be overloaded. Each parameter has a name beginning with "?" and a type, one of the classes defined in the project type set. Following the parameter list is the condition, which consists of a set of bindings followed by ":" and a property list. Bindings attach a local variable, whose name also begins with "?", to the set of objects that are of the type specified after the variable name and which satisfy the logical clause following "`suchthat`".

The property list gives a complex expression of logical predicates that must be true of one (for existential) or all (for universal) of the objects bound to a variable. For historical reasons, quantifiers are currently stated as part of the bindings but are applied as part of the property list. The activity invocation is enclosed in curly braces "{...}". It consists of a tool name, the name of an operation of that tool (some tools support more than one operation), and a sequence of attributes supplied as arguments. These arguments may include attributes of the bound variables in addition to attributes of the parameters. Following is the set of effects, each terminated by ";". Each effect is a conjunction of logical predicates that assign values to named attributes of the parameter objects; it is currently not possible to assign values to attributes of bound variables.

The classes for the rules of figure 2-1 are shown in figure 2-2. Each class definition starts with the name of the class followed by ":: `superclass`" and the list of its superclasses. Multiple inheritance is supported. Then follows a list of attribute definitions, each terminated by ";". A definition consists of a name followed by ":" and the type of the attribute; the type may optionally be followed by "=" and an initialization value. MARVEL supports several built-in attribute types: `string`, `integer`, and `real` are self-explanatory. `text` and `binary` refer to text and binary file types, respectively, and the ‘‘initializations’’ in these cases give the file name suffixes used in the underlying file system. `user` represents a userid and `time` a time stamp. An enumerated type is a list of possible values enclosed in parentheses "(...)". The "`set_of`" construct allows an aggregate of arbitrarily many instances of the same class.

```
RESERVABLE :: superclass ENTITY;
    locker : user;
    reservation_status : (CheckedOut,Available,None) = None;
end

FILE :: superclass RESERVABLE;
    time_stamp : time;
    contents : text;
end

HFILE :: superclass FILE;
    contents : text = ".h";
end

CFILE :: superclass FILE;
    contents : text = ".c";
    includes : set_of link HFILE;
    compile_status : (Archived,Compiled,NotCompiled,Error)
                        = NotCompiled;
    object_code : binary = ".o";
    object_time_stamp : time;
end

MODULE :: superclass RESERVABLE;
    time_stamp : time;
    archive_status : (Archived,NotArchived)
                        = NotArchived;
    cfiles : set_of CFILE;
    modules : set_of MODULE;
end
```

Figure 2-2: Example Classes

A "`link`" attribute refers to a named and typed relation from an instance of this class to an instance or a set of instances of the given class. Each class definition is terminated by "`end`".

MARVEL's process and data models are strongly typed. The project type set must include definitions for all classes and attributes mentioned in the conditions and effects of the rules in the project rule set. Consider the `edit` rule in figure 2-1, which applies to instances of class `CFILE`, and whose condition checks whether the value of the `reservation_status` attribute is equal to `CheckedOut`. Thus, the definition of `CFILE` must contain an attribute called `reservation_status` of an enumerated type including the `CheckedOut` value. The rule set must be self-consistent in the sense that no two rules assume different types for the same attribute of the same class. For example, the rule set would not be self-consistent if it contains another rule that also applies to `CFILE` but assumes `reservation_status` is a `string`.

MARVEL analyzes the specifications of the project rule set and the project type set when they are loaded into the kernel by the project administrator. If these specifications are not both

self-consistent and consistent with each other, they are rejected, with appropriate error messages. The administrator should debug the specifications and attempt to load them again. Thus, loading is analogous to compilation with strong typing. When loading succeeds, the tailored RBDE presents the environment end-users (the software developers) with commands corresponding to the project rule set and an objectbase defined by the project type set. Thus environments for different projects are likely to have different user commands as well as different objectbase structures. When a user requests the execution of a command on a set of objects, MARVEL selects the rule that matches the command. Rule names can be overloaded, and application of rules is disambiguated using the types of the actual parameters, subtyping and inheritance [Barghouti 90].

For example, the user might request the `edit` command on an instance of `CFILE`, which matches a rule whose condition specifies that the object's `reservation_status` attribute must have the value `CheckedOut` in order to initiate the activity. The rule's activity invokes the edit operation of the EDITOR tool and passes the given attributes by reference, i.e., they may be modified by the tool. The first effect is applied if the tool returns the status code 0, setting the `CFILE`'s `compile_status` attribute to `NotCompiled` and `object_time_stamp` to the value of the built-in `CurrentTime` variable. The second effect is applied for status code 1. The purpose of effects are to indicate any results of invoking the activity on the state of the parameters' attributes not passed as arguments to the envelope. The distinction between arguments and effects is required for chaining.

2.2. Chaining

RBDEs assist their users by applying forward and backward chaining to opportunistically fire rules, usually in order to automatically initiate development activities. When a command is requested by the user, the corresponding rule's activity cannot be invoked unless its condition is satisfied. If the condition is not satisfied, the RBDE applies backward chaining to fire other rules whose effects might satisfy the condition. (These are attempted in an arbitrary order determined by the implementation.) It cannot be known a priori whether firing a given rule, even if its condition is satisfied, will produce the desired effect since there are multiple effects, one of which is selected only as a result of actually executing the activity. The result of this backward chaining is either the satisfaction of the original condition or the inability to satisfy it given the current objectbase state. In the latter case, the user is informed that the RBDE cannot execute her command.

When the condition is satisfied, the activity is executed, and after it terminates, the RBDE asserts one of the rule's effects. This might satisfy the condition of other rules, collectively called the *implication set* of the rule. The RBDE fires these rules (in an order determined by the implementation). The effects of these rules may in turn cause further forward chaining, until no additional rules are triggered. In MARVEL, the parameters of rules fired through backward and forward chaining are bound through a logical inversion mechanism [Heineman 91].

Consider the `edit` rule in figure 2-1. Say a user requests to `edit` an instance of class `CFILE` but the `reservation_status` attribute of this object does not have the value `CheckedOut`, as required by the condition of the `edit` rule. Instead of rejecting the user's command, MARVEL tries to fire the `reserve` rule, one of whose effects changes the value of the `reservation_status` attribute to `CheckedOut` (only the success effect is stated since the failure effect results in no changes). If `reserve` succeeds, `edit`'s condition becomes satisfied and its activity is invoked. Once the editing session terminates and its effect is asserted, MARVEL tries to fire all the rules whose condition became satisfied, including the `compile` rule. If `compile` terminates successfully, it triggers the `dirty` rule on the `MODULE` containing the original `CFILE` object.

3. Consistency and Automation

The chaining algorithm described above, which is implemented in MARVEL 2.6, does not necessarily reflect the project administrator's intentions in specifying the various conditions and effects. There is no way for the administrator to state: (1) whether or not an unsatisfied condition of a rule warrants rejecting the user's command that triggered the rule, even though it might be logically possible to infer the condition; (2) whether or not an unfulfilled implication of the effect of a rule warrants rejecting the command, when it is not logically possible to automatically fulfill the implications; and (3) whether or not the actions performed during chaining are definite or tentative (i.e., can be undone). This is relevant if backward chaining fails to satisfy the desired condition. In particular, backward chaining is always attempted, forward chaining is executed until no additional rules can be triggered (i.e. infinite cycles are possible, but this is necessary to support edit/compile/debug cycle), and all actions are definite. The problem lies in the inability to distinguish between consistency and automation.

Considering the example of the previous section, the administrator might like to specify that compiling a C file after it has been edited is not mandatory, while outdating a module after one of its C files has been compiled is obligatory. The rules as given make it seem that both are optional, or alternatively both are obligatory, which is not the case. There is no way to specify that the former reflects an opportunity for automation but the latter is a consistency constraint.

3.1. A Maximalist Chaining Mechanism

Some RBDE rule languages, such as CLF's AP5 [Cohen 89], distinguish between consistency rules and automation rules. The MARVEL Strategy Language (MSL) rule language used in MARVEL 3.0 distinguishes between consistency and automation predicates in both the condition and effects of rules, and a single rule may contain both kinds of predicates. Both AP5 and MARVEL 3.0 implement what we call a *maximalist* assistance mechanism. In the rest of this paper, we assume the maximalist mechanism implemented in MARVEL 3.0 (henceforth simply MARVEL), which subsumes the AP5 mechanism.

MARVEL combines consistency and automation as follows: (1) if an automation predicate in the condition of a rule is not satisfied, MARVEL tries to make it satisfied by backward chaining; (2) if a consistency predicate in the condition is not satisfied, MARVEL refuses to execute the activity; (3) the assertion of a consistency predicate in the effect of a rule mandates that MARVEL must fire all the rules in the implication set atomically; and (4) the assertion of an automation predicate in the effect of a rule causes MARVEL to try to carry out the implications of the predicate. Backward and/or forward automation may be turned off, if desired by the user, but consistency chaining is by definition mandatory.

```
compile[?c:CFILE]:
    :
(?c.compile_status = NotCompiled)
{ COMPILER compile ?c.contents ?c.object_code
                   ?c.error_msg "-g" }
(and (?c.compile_status = Compiled)
     [?c.object_time_stamp = CurrentTime]);
(?c.compile_status = Error);
```

Figure 3-1: Consistency and Automation Predicates

To illustrate, consider the modified `compile` rule in figure 3-1. The predicate enclosed in square brackets "[...]" is a consistency predicate, whereas those in parentheses "(...)" are

automation predicates (thus MARVEL is compatible with old rule sets, with all predicates treated as automation). The second predicate in the first effect of the `compile` rule ("[?c.compile_status = Compiled]") is now marked as a consistency predicate. This causes the `dirty` rule to become a consistency implication of `compile`.

The distinction between automation and consistency predicates is depicted by representing rules and chains as a graph. This graph consists of nodes that represent rules, and three kinds of edges: automation forward edges, consistency forward edges and automation backward edges; there are no consistency backward edges. An automation forward edge from rule `R1` to rule `R2` exists if one of the effects of `R1` contains an automation predicate that implies a predicate (of either kind) in the condition of `R2`.

Figure 3-2: Rule Graph

There is a consistency forward edge from rule `R1` to rule `R2` if there is a consistency predicate in one of the effects of `R1` that implies a predicate in the condition of `R2`. There exists an automation backward edge from `R2` to `R1` if an automation predicate in the condition of `R2` is implied by an automation predicate in one of the effects of `R1`. The graph for the two unchanged rules of figure 2-1 and the modified rules in figure 3-1 is shown in figure 3-2.

Each rule `R` has a set of consistency implications consisting of all the rules that are connected

to R via a forward consistency edge emanating from R. The set of loaded rules is *complete* if and only if for each rule loaded by the RBDE, all the rules in its consistency implication set are also loaded. It must also be the case that the loaded data model defines all of the classes and attributes accessed by these rules. In the MARVEL implementation, an extended form of this graph-based model, with edges distinguished by their specific predicates, is generated when the project rule set is loaded. This network is used to limit search during backward and forward chaining.

Consider the following scenario. A user requests to `edit` a C file "f.c". Once the editing session is terminated and the effect asserted, the RBDE fires all those rules whose condition became satisfied by the `edit` rule. One of these rules is `compile`, since there is an automation predicate in common between the effect of `edit` and the condition of `compile`. Thus MARVEL attempts to `compile` "f.c". If the compilation is successful, its first effect is asserted, and MARVEL fires all the rules in the implication set of this effect. The set includes the `dirty` rule, which is fired on the module "mod" containing "f.c". If the condition of `dirty` is not satisfied, say because "mod" has been reserved by another user, then the `compile` rule cannot take effect. The `edit` rule, however, will not also be undone since the `compile` rule was only an automation implication of `edit`, not a consistency implication.

Distinguishing between automation and consistency predicates enables us to define a *consistency model* for a development process. Given a rule set, the consistency model of the software process is defined by the consistency forward edges in the rule graph. Thus, in order to preserve data consistency in a particular project, the RBDE must guarantee that for each rule, all the consistency implications of each rule's asserted effect will be carried out. All other rules in the rule graph define the *automation model* for the development process. These notions of consistency and automation form the basis for our approach to supporting multiple views and automating evolution in RBDEs.

Our approach is based on two observations: (1) all co-existing views must maintain either the identical consistency model or disjoint consistency models; and (2) for the RBDE to be able to automatically transform the objectbase to a new consistent state following an evolution step, there must be a mechanical transformation from the current consistency model to the new consistency model. We argue for (1) by noting that if a forward consistency edge does not exist in one view, and therefore is not followed when its predicate is asserted, then another view containing that forward consistency edge will see what it terms is an inconsistent

objectbase. (2) is true by the conventional definition of consistency, i.e., if the objectbase can reach some state that could not have been reached had the new consistency model been in place as it was developed, then it is inconsistent. In the next two sections we elaborate on our approach to solving the multiple views and evolution problems.

4. Multiple Views

A *view* in an RBDE consists of subsets of the project rule set, the project type set and the project tool set. At any time, an RBDE must have some view loaded in order to function. Different views might be loaded by the same user during different phases of the development process; different users might also load different views at the same time to fulfill different roles. There are two correctness criteria for views: (1) the type set of the view must match the assumptions of its rule set; and (2) the set of rules in the view must be *complete* with respect to the project rule set, consistency off all the rules in all views. (A third criterion, compatibility of the rule and type sets with the project tool set, is outside the scope of this paper; see [Gisi 91].)

The first criterion is local to a view in the sense that it determines whether or not the view is self-consistent. The conditions and effects of the rules in a view manipulate (read and update) the attributes of instances of the classes in the view's type set. Thus, each rule assumes that its view includes definitions of those attributes that the rule manipulates. If the view's type set does not meet the assumptions of the view's rule set, then there is a *discrepancy*.

A discrepancy can make it impossible for the RBDE to evaluate the conditions of some rules because the attributes manipulated in these conditions are either missing or of the wrong type, so these rules could never fire because their conditions would always be unsatisfied. This possibility can be tolerated only if none of the affected rules is in the consistency implication set of any rule in the rule set of the view. Similarly, a discrepancy could also make it impossible for the RBDE to assert the effects of some rules, resulting in a situation where the activity of a rule has terminated but the RBDE cannot assert the appropriate effect. This situation is not well-defined regardless of whether the predicates involved are consistency or automation, and in either case cannot be permitted.

Thus, an RBDE cannot load any view that does not meet at least the first of the criteria listed above (that the rule set and the type set match). The analysis is not affected by any other views, loaded or otherwise. The second criterion mentioned above, on the other hand, is

greatly affected by the consistency model of the project rule set. A view is said to be *complete* if and only if for each rule `R` in the rule set of the view, the rule set includes all those rules that are members of the consistency implication set of `R`.

The notion of completeness is based on only the consistency forward edges in the graph of the project rule set. Automation forward edges are not relevant, since they represent optional behavior. For example, there is an automation forward edge between the two rules `edit` and `compile`. The inability to fire the `compile` rule after `edit` terminates successfully for any reason, including the non-existence of `compile` in the current view, does not invalidate the success of `edit`. Thus, a view containing `edit` but not `compile` is still complete.

A consistency forward edge, in contrast, is interpreted as a consistency implication. The consistency forward edge between `compile` and `dirty` means that if `compile` is fired and the relevant effect asserted, then the changes that its activity introduces in the objectbase cannot be made permanent unless `dirty` is also fired and terminates successfully (and so do any consistency implications of `dirty`). Thus, if a view contains `compile` but not `dirty`, the edge representing this consistency implication would be missing from the rule graph of the view. This could lead to inconsistencies if the changes introduced by `compile`'s activity become permanent (assuming all other implications are met) even though the `dirty` rule cannot be fired (because it is not in this view). Thus any other view that assumes that the module is marked as outdated whenever any of its C files is recompiled will see an inconsistent objectbase.

The two correctness criteria for a view can be formalized as follows:

> **Definition 1:** A view is said to be *complete* iff its rule set and type set are consistent with each other and if for each node in the rule graph of the view, the consistency forward edges are identical to the consistency forward edges for the corresponding node in the graph of the project rule set consisting of all views.

Given this definition, the RBDE can determine that a view that consists of the rules shown in figure 4-1 is incomplete because it is missing the `dirty` rule and hence the forward consistency edge from `compile` to `dirty`. Similarly, if the `dirty` rule were present but there were only an automation forward edge, the view would still be incomplete.

```
reserve[?c:FILE]:
    :
(?c.reservation_status = Available)
{ RCS reserve ?c.contents ?c.version }
(?c.reservation_status = CheckedOut);

edit[?c:CFILE]:
    :
(?c.reservation_status = CheckedOut)
{ EDITOR edit ?c.contents }
(and (?c.compile_status = NotCompiled)
     (?c.time_stamp = CurrentTime));

compile[?c:CFILE]:
    :
(?c.compile_status = NotCompiled)
{ COMPILER compile ?c.contents ?c.object_code
                   ?c.error_msg "-g" }
(and (?c.compile_status = Compiled)
     (?c.object_time_stamp = CurrentTime));
(?c.compile_status = Error);
```

Figure 4-1: Incomplete View

5. Evolution

As the process or data model is changed over time, the correctness criteria we required for views (that the data model matches the process model and that the process model is complete) must be reevaluated to ensure that they still hold. Technically speaking, these restrictions need not be enforced if the rules that have been modified or added are not in the consistency implication set of any other rules. Since by definition automation chaining is optional, then inability to chain because the project type set does not match the project rule set might be an acceptable reason to not carry out some automation chains. This does not seem satisfactory, however: automation chains should be prevented only when the user because the user has turned off automation, not because of typing errors that should have been detected when the rules were loaded.

To illustrate, consider the `compile` rule of figure 3-1. This rule is connected by a consistency forward edge to the `dirty` rule of figure 2-1. The condition of the `dirty` rule contains a predicate that checks if any C file contained in the module has been compiled more recently than the last archival of the module. If this predicate is satisfied, the module's archive is outdated by assigning the value `NotArchived` to the module's `archive_status` attribute.

```
edit[?c:CFILE]:
    :
(?c.reservation_status = CheckedOut)
{ EDITOR edit ?c.contents }
(and (?c.compile_status = NotCompiled)
     [?c.time_stamp = CurrentTime]);

dirty[?m:MODULE]:
(forall CFILE ?c suchthat (member [?m.cfiles ?c]))
    :
(?c.time_stamp > ?m.time_stamp));
{ }
(?m.archive_status = NotArchived);
```

Figure 5-1: Non-Mechanizable Evolution

Consider the scenario where the users of the environment tailored by these rules complain that the RBDE does not always outdate the module archives when it should. The administrator discovers the reason is that the `dirty` rule outdates a module whenever any of its C files is recompiled correctly, but not when the compilation discovers errors. So she modifies the `edit` and `dirty` rules, as shown in figure 5-1. A new predicate "(?c.time_stamp > ?m.time_stamp)" replaces "(?c.object_time_stamp > ?m.time_stamp)" in the condition of `dirty`, while the "(?c.timestamp = CurrentTime)" predicate in the effect of the `edit` rule is changed to a consistency predicate "[?c.time_stamp = CurrentTime]". This creates a new consistency forward edge between `edit` and `dirty`. Now the module will be outdated whenever the source code has been edited.

If this modified rule is loaded, then potentially all instances of `MODULE` might become inconsistent because the value of their `archive_status` attribute might not reflect the new consistency implication between `edit` and `dirty`. To upgrade the objectbase, the RBDE must fire the `dirty` rule on all instances of `MODULE`. For the cases where the condition is satisfied, the `archive_status` can be set to `NotArchived` and we can be confident that this is the correct value. However, in the cases where the condition is not satisfied, but the current value of `archive_status` is `NotArchived`, the RBDE must determine what is the correct value of `archive_status`. In general, this is impossible to do automatically (since there are two possible values, `Archived` and `INotArchived`). Therefore, this change to the `dirty` rule should not be allowed.

This example demonstrates the need for a concept of a *legal evolution step*. An evolution step is a change either to the definition of a single rule (or a single class).

Definition 2: An evolution step is said to be *legal* iff (1) the resulting project type set and project rule set are self-consistent and consistent with each other; and (2) all objects in the objectbase can be mechanically transformed to meet the new consistency implications specified by the modified rule set.

The example above is not a legal evolution step in the general case since it is not always possible to mechanically transform an objectbase to a consistent state. However, this example may represent a legal evolution step on some objectbases, simply because the condition of the `dirty` rule happens to be satisfied for all instances of `MODULE` in that particular objectbase. There is no method for determining whether or not this evolution step is legal a priori; it is necessary to attempt the potentially very costly transformation of the objectbase.

A more realistic approach would be to restrict the kinds of changes allowed to those that can be statically analyzed to determine whether or not an evolution step should be attempted. Intuitively, if the consistency implications specified by the modified rules are weaker than the old ones, then all objects in the objectbase will definitely meet the new criteria. If the only change to the rule set is to relax a rule (i.e., make it apply to a narrower set of objects) or to completely remove it, then the objectbase would not violate any consistency requirements. Thus, a more useful definition is:

Definition 3: An evolution step is *legal* iff the consistency implications specified by the rule set after the evolution step are either weaker than the implications before the evolution step is carried out, or are independent of them.

```
edit[?h:HFILE]:
    :
(?h.reservation_status = CheckedOut)
{ EDITOR edit ?h.contents }
(?h.time_stamp = CurrentTime);

compile[?c:CFILE]:
(exists HFILE ?h suchthat (linkto [?c.includes ?h]))
:
(or (?h.time_stamp > ?c.object_time_stamp)
    (?c.compile_status = NotCompiled))
{ COMPILER compile ?c.contents ?c.object_code ?c.error_msg
                   "-g" ?h.contents }
(and (?c.compile_status = Compiled)
     [?c.object_time_stamp = CurrentTime]);
(?c.compile_status = Error);
```

Figure 5-2: Mechanizable Evolution

The problem then reduces to defining ‘‘weaker’’ and ‘‘independent’’ consistency implications. The set of consistency implications specified by a rule set can be represented by

the rule graph after removing all the automation edges from the graph, but keeping the consistency forward edges, to produce a *consistency graph*. For each set of consistency implications *I*, the corresponding graph is denoted by *G(I)*.

> **Definition 4:** A set of consistency implications *I′* is said to be *weaker* than another set *I* iff *G(I')* is a subgraph of *G(I)*. *I′* is said to be *independent* of *I* iff *G(I')* and *G(I)* are disjoint.

In an RBDE supporting only automation predicates, with no consistency predicates, there are no restrictions at all on modifying, adding and/or removing rules at any time. There is no notion of consistency in the objectbase that could become corrupted. The change in the rules simply causes a change in the optional automation behavior of the RBDE. In a maximalist RBDE with both automation and consistency predicates, rules containing only automation predicates can always be removed, but there are limitations on addition and modification of rules containing consistency predicates since this might introduce a new forward consistency edges from some consistency rule to an automation rule.

To illustrate how an RBDE can decide whether or not an evolution step is legal, consider the `compile` rule of figure 5-2. The condition of the rule was modified so that the `compile` rule is fired on a C file whenever one of the `HFILE`s it includes is edited. This change is necessary for practical programming in C, since any C source file might include several ".h" files. (The `edit` rule that applies to `HFILE` is also shown in figure 5-2, but would have been added in an earlier evolution step.)

Figure 5-3: Consistency Graphs

The RBDE determines if this evolution step is legal by comparing the consistency graph of the rule set including the modified `compile` rule, shown in figure 5-3B, with the consistency

graph of the original rule set, shown in figure 5-3A. The consistency edges in the two graphs are identical, which means that the evolution step fulfills the legality criteria and can be permitted. In contrast, the consistency graph of the rule set after the illegal evolution step discussed earlier in figure 5-1, shown in figure 5-3C, is not a subgraph of the original graph. Therefore, the evolution step should not be permitted.

6. Related Work

The RPDE^3 project has had substantial experience with supporting real changes [Ossher 90]. The ease with which these changes were accommodated rests on three pillars: the system's architecture, its extended object-oriented technology, and structured representation of the programs manipulated by the system. RPDE^3 supports multiple display views, but these are hard-wired into the code of the system, as is the software process. Evolution of the process was one of the the several kinds of changes singled out as particularly difficult due to the limitations of their object-oriented technology [Harrison 89].

The Field environment [Reiss 90] integrates tools through a centralized message server that routes messages among tools. Tools send messages to the server to announce changes that other tools might be interested in, and the server broadcasts those messages to tools that have previously registered matching patterns. Existing tools are supported by adding an interface to each tool to send and receive messages. Field makes it relatively easy to evolve the tool set, but is not concerned with process modeling and there is no common project data model.

Forest [Garlan 90] extends the Field approach by adding a policy tool to a simulated message server. The policy tool supports a simple process model consisting of condition/action rules and a simple data model consisting of state variables. The policy tool fires the first rule whose condition matches each incoming message; the possible actions include sending a message on to a tool, sending a new message to the message server, or doing nothing. Forest enforces access control over which categories of users can change which rules, as well as the membership of these categories. It is not concerned with logical restrictions on evolution or the impact of changes on consistency of the global state reflected in the data accessed by the tools, and does not address multiple views.

In MELMAC [Deiters 90], an object type and activity view, a process view, a feedback view and a project management view are combined into a common representation called a FUNSOFT net. These views represent different perspectives on the process, and are not

related to our notion of multiple views for multiple users. The FUNSOFT net, essentially a hierarchical Petri net, consists of some nodes representing primitive activities while other nodes must be refined by recursive nets. The nodes may have preconditions and postconditions. MELMAC supports evolution of the process model by introducing the notion of modification points, which are attached to nodes whose refinements can be changed on the fly. It is not clear, however, whether there are any restrictions on what kinds of changes are allowed, and if so how they are handled, to ensure consistency in the project database.

Darwin [Minsky 91] uses *laws*, written as Prolog rules, to regulate the activities that users can carry out but it never initiates activities on behalf of users. A Prolog interpreter enforces these laws. The laws govern what changes can be made to the laws themselves as well as to programs, and thus a single elegant mechanism supports both process enactment and process evolution. But Darwin does not provide any mechanism for legislating that new laws are consistent with old ones with respect to the existing objectbase, and does not address the problem of multiple views.

APPL/A [Sutton 90a] is the process programming language used in the Arcadia environment. It extends Ada with persistent relations, triggers, predicates, and transaction statements. APPL/A is specifically concerned with accommodating inconsistency resulting from, among other things, evolution of the process program. A flexible consistency model, FCM [Sutton 90b], is defined in terms of predicates and transactions. Evolution of the process model may lead to adding new predicates, introducing inconsistencies in the project objectbase. The transactions may be used to control the enforcement of these predicates, by determining the conditions for relaxing consistency and providing a scope for restoring consistency. APPL/A seems to provide the closest match to MARVEL's consistency model. FCM defines a complementary framework for tolerating inconsistency.

7. Conclusions

We considered the problems of scaling up rule-based software development environments to support multiple views and evolution. Our solution is based on distinguishing two functions of rules and chaining: (1) to express, enforce and maintain consistency constraints and (2) to express and carry out opportunities for automation. By definition, consistency is mandatory whereas automation is not. Our earlier work on MARVEL did not recognize any distinction between consistency and automation, and thus our original approach to multiple views and

evolution [Kaiser 88b] could lead to inconsistent objectbases. The new MARVEL 3.0 extends the maximalist assistance mechanism presented in this paper with additional directives to turn off forward and/or backward chaining in and/or out of individual predicates, to provide the administrator with powerful control over consistency implications. Now by separating the consistency and automation aspects of RBDE assistance mechanisms, we make it possible to formalize the range of compatible views and the scope of mechanizable evolution steps. Although neither multiple views nor evolution has been implemented yet, the distinction between consistency and automation is used in MARVEL 3.0 as part of a semantics-based concurrency control mechanism to support multiple users.

Acknowledgments

We would like to thank Israel Ben-Shaul, George Heineman and Timothy Jones, who worked with us on implementing the multi-user version of MARVEL. George Heineman and the anonymous referees also provided many suggestions to improve earlier versions of this paper. Prof. Kaiser's Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368, CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research. Dr. Barghouti is now a Member of the Technical Staff at AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974, United States.

MARVEL 3.0 is available for licensing by educational institutions and industrial sponsors. Contact Prof. Kaiser. The multi-user client/server architecture and concurrency control issues for MARVEL are described in [Ben-Shaul 91] and [Barghouti 92], respectively. An extended abstract of this paper appeared under the title "Scaling Up Rule-Based Development Environments" in *Third European Software Engineering Conference*, A. van Lamsweerde and A. Fugetta (eds), Lecture Notes in Computer Science 550, Springer-Verlag, October 1991, pp. 380-395.

References

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

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

[Ben-Shaul 91] Israel Z. Ben-Shaul.
An Object Management System for Multi-User Programming Environments.
Master's thesis, Columbia University, Department of Computer Science, April, 1991.
CUCS-010-91.

[Ciancarini 93] Paolo Ciancarini.
Coordinating Rule-Based Software Processes with ESP.
ACM Transactions on Software Engineering and Methodology 2(3):203-227, July, 1993.

[Clemm 88] Geoffrey M. Clemm.
The Workshop System — A Practical Knowledge-Based Software Environment.
In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 55-64. ACM Press, Boston MA, November, 1988.
Special issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.

[Cohen 89] Donald Cohen.
Compiling complex database transition triggers.
In James Clifford, Bruce Lindsay and David Maier (editors), *1989 ACM SIGMOD International Conference on the Management of Data*, pages 225-234. Portland OR, June, 1989.
Special issue of *SIGMOD Record*, 18(2), June 1989.

[Deiters 90] Wolfgang Deiters and Volker Gruhn.
Managing Software Processes in the Environment MELMAC.
In Richard N. Taylor (editor), *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.

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

[Gisi 91] Mark A. Gisi and Gail E. Kaiser.
Extending A Tool Integration Language.
In 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.

[Harrison 89] William H. Harrison, John J. Shilling and Peter F. Sweeney.
Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 85-94. ACM Press, New Orleans LA, October, 1989.
Special issue of *SIGPLAN Notices*, 24(10), October 1989.

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

[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 and of *Software Engineering Notes*, 13(5), November 1988.

[Humphrey 89] Watts S. Humphrey.
Managing the Software Process.
Addison-Wesley, Reading MA, 1989.

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

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

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

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

[Katayama 90] Takuya Katayama (editor).
6th International Software Process Workshop: Support for the Software Process.
IEEE Computer Society Press, Hakodate, Japan, 1990.

[Legait 89] 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.

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

[Ossher 90] Harold Ossher and William Harrison.
Support for Change in RPDE^3.
In Richard N. Taylor (editor), *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 218-228. Irvine CA, December, 1990.
Special issue of *Software Engineering Notes*, 15(6), December 1990.

[Perry 89] Dewayne Perry (editor).
5th International Software Process Workshop: Experience with Software Process Models.
IEEE Computer Society Press, Kennebunkport ME, 1989.

[Programming Systems Laboratory 91a]
Programming Systems Laboratory.
Marvel 3.0 User's manual.
Technical Report CUCS-033-91, Columbia University Department of Computer Science, October, 1991.

[Programming Systems Laboratory 91b]
Programming Systems Laboratory.
Marvel 3.0 Administrator's manual.
Technical Report CUCS-032-91, Columbia University Department of Computer Science, October, 1991.

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

[Sutton 90a] Stanley M. Sutton, Jr., Dennis Heimbigner and Leon J. Osterweil.
Language Constructs for Managing Change in Process-Centered Environments.
In Richard N. Taylor (editor), *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 206-217. Irvine CA, December, 1990.
Special issue of {\it Software Engineering Notes
, 15(6), December 1990.}

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

[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 issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.