Consistency and Automation in Multi-User Rule-Based Development Environments

Naser S. Barghouti and Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027
naser@cs.columbia.edu, (212) 854-8182
kaiser@cs.columbia.edu, (212) 854-3856

Technical Report CUCS-047-90
31 October 1990

Abstract

We investigate the scaling up of a class of single-user software development environments, which we call *rule-based development environments* (RBDEs), to support multiple developers cooperating together on a project. RBDEs model the software development process in terms of rules that encapsulate activities, and execute forward and backward chaining on the rules to provide assistance in carrying out the development process. There is a spectrum of assistance models, ranging from pure automation to strict consistency preservation. We describe three problems whose solutions are dependent on the choice of assistance model: (1) multiple views; (2) evolution; and (3) concurrency control. We discuss how the two extremes of the spectrum restrict the possible approaches to multiple views and evolution. In order to explore different aspects of the concurrency control problem across multiple points on the spectrum of RBDEs, we develop a *maximalist* assistance model and propose an approach to synchronization of cooperating developers within the context of this model.

Copyright © 1990 N. S. Barghouti and G. E. Kaiser

1. Introduction

Software development environments aim to assist developers of software projects in carrying out the development process. A particular class of environments, which we call *rule-based development environments* (RBDEs), model the software development process in terms of rules [1, 2, 3]. RBDEs encapsulate development activities as rules and provide assistance to developers by applying forward and/or backward chaining on the rules to automatically perform software development activities and/or inform the developers when particular activities should or should not be done.

RBDEs are built on top of object management systems that abstract project components as objects and store them in a project objectbase. Most existing RBDEs are single-user; the few that support multiple users do so by isolating the users, restricting cooperation [4, 5, 6]. Our goal is to scale up RBDEs to allow cooperation among multiple users concurrently accessing the shared objectbase. By ‘‘cooperation’’, we mean the users concurrently access and modify the same objects or related objects while their work is in progress, in order to share knowledge and change these objects in a coordinated fashion to achieve a common goal such as integrating a subsystem.

We have identified three major problems that arise in scaling up RBDEs. The first is the concurrency control problem, which is concerned with the synchronization of concurrent accesses to the project objectbase by either human users or rules executing on their behalf, collectively termed *agents*. This problem is complicated by the need for collaboration among the agents in some situations, when concerted effort is required, and thus conventional concurrency control schemes are not appropriate. The second problem is concerned with multiple views, when different users have different views of the project objectbase and different rule sets. The third is the evolution problem, which is concerned with introducing changes to either the data model or the process model. The solutions to these three problems are dependent on the kind of assistance provided by the RBDE.

There are many different assistance models that can be implemented by an RBDE. For example, if an RBDE aims to automate some of the chores that developers would have otherwise had to do manually, it would try to automatically invoke activities on its own, determined according to the status of the project objectbase and the users’ activities. If the RBDE implements a model that aims to maintain strict consistency during the development process, it would prevent user activities that would corrupt the status of the project objectbase. We explain how the choice of consistency versus automation in the RBDE’s assistance model restricts the possible approaches to the problems of multiple views and evolution.

In between the two extremes, there are a variety of assistance models that combine automation with consistency preservation. Rather than consider the concurrency control problem for specific points on the assistance spectrum, we construct a hypothetical *maximalist* assistance model that integrates automation with consistency, and propose a solution in the context of this model. Our solution is based on synchronization among concurrent *tasks*, where a task is the complete set of rules automatically executed during the backward and forward chaining resulting from an initial rule triggered by a user.

We have previously introduced the notion of tasks and devised a protocol for detection of potential concurrency conflicts between tasks, for a class of automation assistance models similar to our own RBDE, MARVEL, based on the treatment of rules as multimethods associated with an object-oriented data model [7, 8]. We introduced a particular automation assistance model based on rules in our earlier publications on the single-user MARVEL environment [9, 10]. We have also constructed other approaches to concurrency control for cooperation among developers [11, 12], but these did not consider the problems of rules or rule chains (tasks).

In this paper, we briefly describe the general idea of rule-based development environments, and present two assistance models, one a pure consistency model and the other a pure automation model. We then show how the choice of assistance model impacts multiple views and evolution in an RBDE. In section 4, we present our maximalist assistance model, and then in section 5 we explain tasks and the corresponding concurrency control protocol. We conclude by summarizing the effects of the choice of assistance model on the problems of scaling up RBDEs.

2. Rule-Based Development Environments

Each software project assumes a specific organization for its components, a specific set of software development tools, and a specific development process. The organization of the project components can be abstracted into a hierarchy of complex objects, each with a set of attributes whose values encode information about the status of the component. Some software development tools do not directly manipulate the objects' attributes but operate on files that are mapped to these attributes, while others execute on their own temporary copies of objects. The changes that a tool causes in the files or the private copies must be mapped to changes in the values of the objects' attributes in the project objectbase.

The development process of a project is specified in terms of two kinds of rules, which we call *activation rules* and *inference rules*. Activation rules control the invocation of tools on objects by specifying the conditions under which it is appropriate to invoke the tools and the effects of the tools' invocation on the values of the objects' attributes. In contrast, the condition of an inference rule is a logical expression that specifies current values for objects' attributes and the single effect of the rule derives new values for attributes of the same or other objects; tools are not associated with inference rules. Both kinds of rules are parameterized to apply to one or more classes of objects, and are thus similarto multi-methods in object-oriented programming. Two activation rules and one inference rule are shown in figure 2. When not relevant, we ignore the distinction between these two classes and just say ''rules''.

In order to assist in the development of a software project, an RBDE is provided with a specification of the project's data model, in terms of object classes (the *project type set*), and its process model, in terms of development rules (the *project rule set*). The classes in the project type set must include definitions for all attributes mentioned in the conditions and effects of the project rule set. For example, if the condition

```
# reserve:  reserve a file type object.  In the C/Marvel example, you
#           can use this rule on FILE, CFILE, HFILE and DOCFILE,
#           because of the inheritance mechanism.

reserve[?f:FILE]:

    :
    (?f.reservation_status = Available)

    { RCS reserve ?f }

    (and (?f.reservation_status = Checked_out)
         (?f.owner = CurrentUser));
    (?f.reservation_status = Available);


# deposit: deposit an object.  This rule works on the same objects
#          as the reserve rule.

deposit[?f:FILE]:

    :
    (and (?f.owner = CurrentUser)
         (?f.reservation_status = Checked_out))

    { RCS deposit ?f }

    (?f.reservation_status = Available);
    (?f.reservation_status = Checked_out);

# arch_proj: archive all the libraries in this project.  This rule is
#            as inference rule (one with an empty activity) that
#            causes arch_lib to be executed (via chaining) for all
#            the libraries in the PROJECT.

arch_proj[?proj:PROJECT]:

    (forall LIB ?l suchthat (member [?proj.libraries ?l]))
    :
    (?l.archive_status = Archived)

    { }

    (?proj.archive_status = Archived);
    (?proj.archive_status = Error);
```

Figure 2-1: Example Rules Based on MARVEL Notation

of a rule `r` that applies to instances of class `C` checks if the value of an attribute `a` is greater than the integer `5`, then `r` assumes that the definition of `C` contains an attribute called `a` of type integer. The classes for the first two rules of figure 2 are shown in figure 2.[1] The rule set must be self-consistent in the sense that there are no conflicting assumptions about the classes. A conflict occurs when two rules assume different types for the same attributes of the same class: there cannot be another rule `r'` that also applies to `C` but assumes that `a` is a string.

A project administrator writes these specifications and loads them into the RBDE, which tailors its ob-

[1]The definitions include attributes not used in the rules shown, and an example of inheritance, which is not addressed in this paper.

```
Class PROGRAM ::
    status : (Built,NotBuilt,Error) = NotBuilt;
    debug_status : (OK,NeedsDebugging) = OK;
    cfiles : set_of CFILE;
end

Class FILE ::
    owner : user;
    timestamp : time;
    reservation_status : (Checked_out,Available,Error) = Available;
    contents : text;
end

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

Figure 2-2: Example Classes Based on MARVEL Notation

jectbase to the data model and its behavior to carry out the process model. The tailored RBDE presents the end users (i.e., the software developers) with commands corresponding to the set of loaded rules. Thus, the environments for different projects might have different user commands as well as different objectbase structures.

When a user requests the execution of a command on a specific object, the RBDE sends a message to the object, causing the selection of a rule that matches the command. The rule's condition for firing depends on the *state* of the object, which is an assignment of a set of values to the object's attributes. The state of the objectbase is defined by an expression written in the first order predicate logic, which describes both the states of objects in the objectbase and the existence or non-existence of objects. For example, the user might request the `edit` command on an object, which might match a rule whose condition specifies that the object has to have been ''checked out'' (i.e., its attributes must reflect that it is in the ''checked out'' state), the specific editor to invoke, and the effects of invoking the editor on the state of the object.

The RBDE's decision to fire a rule depends on the assistance model implemented by the RBDE. More specifically, the assistance model determines: (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 automatically satisfy 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 implication; and (3) whether or not the actions performed during a chaining cycle are definite (i.e., irreversible) or tentative (i.e., can be undone).

There is a spectrum of assistance models that may be implemented by an RBDE. The two extremes are a strict consistency preserving model and a pure automation model. The first model understands the rules to specify what is allowable and what is not in regards to development activities. The RBDE permits the activities requested by its users only if these activities do not violate any of the rules provided by the project administrator. In contrast, the pure automation model interprets the rules to specify which activities can be automatically invoked as a result of initiating or completing an activity requested by a user. The RBDE automatically carries out as many activities as it can, but this does not affect the validity of activities requested directly by users.

2.1. A Consistency Preserving Model

The consistency preserving assistance model is implemented in systems like Darwin [2], whose rules are considered laws that cannot be violated. Every rule has a condition and a set of one or more effects. The rule cannot be invoked unless its condition is satisfied. If the condition is not satisfied, the RBDE applies backward chaining to invoke rules, whose effects might *verify* that the condition is satisfied. Verifying that a condition is satisfied means that the RBDE attempts to fire rules that derive new objectbase states (which can be thought of as equivalent to facts in Darwin) from ones already known by the RBDE so that the condition is shown to be satisfied. Thus, during a backward chaining cycle in a consistency model, only inference rules that do not invoke activities are considered.

Although Darwin supports only backward chaining, a generalization that supports both backward and forward chaining is possible: Once the condition of the rule is satisfied and its activity invoked, the effect of executing the activity would not be made permanent unless all the implications of this effect can be carried out. These implications correspond to rules whose conditions become satisfied because of the assertion of the effect of the original rule, and in turn all of the implications of these rules, collectively the *implication set*. If any of the implied rules cannot be fired, the effects of the original rule and all the rules in the implication set have to be undone. Thus a chain of rule executions in this model transforms that project objectbase from one consistent state to another.[2]

> **Definition 1:** The objectbase is said to be in a consistent state if the system is quiescent (i.e., no rules are currently in progress) and there are no rules whose condition is satisfied.

The intuition behind this definition is that if there is a rule whose condition is satisfied, then that rule must be fired and the system would not be in a quiescent state. If the system is in a quiescent state, then it must be because there are no rules whose conditions are satisfied. This definition ensures that all implications of the effects of an activity requested by the user are fully satisfied; if this is not possible, then the user's requested is rejected.

---

[2]Another possible consistency model supports neither backward nor forward chaining, but this is a trivial case from the point of view of the three problems addressed in this paper.

2.2. An Automation Model

A pure automation model differs from the consistency model presented above in two main respects: (1) carrying out the implications of a rule is not considered mandatory; and (2) if a condition of a rule is not satisfied, backward chaining is applied to try to *make* it satisfied (as opposed to verifying that it is already satisfied).

The set of rules specify which activities can be automatically invoked as a result of initiating or completing a command requested by a user. The implications of this command do not have to be successfully carried out for the command to be completed. The RBDE simply attempts to fire all the rules whose condition contains a predicate that *became* satisfied as a result of the assertion of the effect of the user's command. The inability to fire any of these rules (because a condition might contain other predicates that are still unsatisfied even after the assertion) does not invalidate what has been performed so far by the original rule or other implications. Thus, the changes that each rule introduces are definite, rather than tentative as in the consistency model.

Unlike in the consistency model, there are no restrictions on the kind of rules that can be invoked in a backward chaining cycle that is initiated in an attempt to satisfy the condition of the rule corresponding to a user's command. It might be the case that one of the rules in a backward chaining cycle might invoke an external tool with more than one effect, to specify the multiple possible results of the tool's execution. In this case, which one of the rule's effects to assert cannot be determined before the execution of the tool is completed, and thus it may or may not satisfy the condition of the original rule. In either case, however, the changes that the tool introduced (to the file system, printer and mail spooler, user I/O, etc.) cannot be undone. Although invocation of such tools might be unacceptable in a consistency model, this is not an problem in an automation model.

Consider the `edit` command described earlier. If a user requests to edit an object but the object is not ‘‘checked out’’ yet, instead of rejecting the user's command as in a consistency model, the RBDE would try to automatically fire a rule whose effect will change the state of the object to be ‘‘checked out’’ (perhaps by invoking a tool to check it out). If firing that rule actually results in changing the state of the object to a ‘‘checked out’’ state (the rule might have more than one effect and thus it is not guaranteed that the desired one will be asserted), the activity of the `edit` rule will be invoked. Once the editing session is completed and an effect has been asserted, the RBDE will try to fire all the rules whose condition became satisfied. Unlike in the consistency model, if any one of these (or all of them) cannot be fired, the changes that the *edit* activity introduced to the object (i.e., the checking out as well as the edits to the object) will not be undone.

3. Interaction Between Assistance Models and RBDE Functionality

Many existing RBDEs support assistance models that combine aspects of automation and consistency preservation. Some RBDEs, such as CLF's rule system [13], distinguish between consistency rules and automation rules, while another possibility is distinguishing between consistency predicates and automation predicates in the condition and effects of the same rule. The choice of assistance model impacts the functionality of the RBDE, in particular, supporting multiple views, evolution of the process and data models, and synchronization of multiple users.

In any assistance model, potential chaining cycles in the project rule set form a graph consisting of nodes and two kinds of edges, forward edges and backward edges. Each node represents a rule, a forward edge between nodes `r1` and `r2` means that a predicate in an effect of `r1` may cause a predicate in the condition of `r2` to be satisfied. Similarly, a backward edge between `r1` and `r2` means that a predicate in the condition of rule `r1` might be satisfied by a predicate in one of the effects of `r2`. The distinction among assistance models can be characterized in terms of the interpretation placed on the edges in the graph. For example, a consistency model would interpret a forward edge as a consistency implication whereas an automation model would interpret the same forward edge as a potential opportunity for automation.

3.1. Multiple Views

A *view* in an RBDE consists of a subset of the project rule set and a corresponding subset of the project type set. At any time, an RBDE must have a 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 to fulfill different roles. In a single-user RBDE, only one view will be loaded at a time; in a multi-user RBDE, different users might have different views loaded by the same RBDE simultaneously. There are two correctness criteria for views: (1) how well the type subset matches the assumptions of the rule subset; and (2) how complete is the subset of rules in the view.

The conditions and effects of each rule loaded by an RBDE manipulate (read and update) the attributes of objects in the project database, which are instances of the loaded classes. Thus, each rule assumes that the RBDE has also loaded a corresponding set of classes that include at least the attributes that the rule manipulates. If the set of classes loaded by an RBDE does not meet the assumptions of all the rules that are loaded, then there is a discrepancy between the loaded process model and the loaded data model. This discrepancy can result if the data model contains incomplete or incorrect classes, or if different rules make conflicting assumptions.

In a strict consistency preserving model, such discrepancy cannot be allowed because it would mean that the condition of rule `r` would never be satisfiable since it would not be able to check the value of an attribute that either does not exist or is of a different type. But since all rules in a consistency model are considered consistency rules, the discrepancy entails that any rule that implies `r` would also not be executable.

If the RBDE supports a pure automation model, a discrepancy would mean that some of the rules either would never fire (if the discrepancy invalidates the condition) or would have invalid effects. Although the former can be tolerated, since in an automation model the inability to fire a rule does not impact any other rule, the latter results in a situation where the activity of a rule has completed but the RBDE cannot assert the appropriate effect because the attributes that the predicates of the effect manipulate either do not exist or are of the wrong type. This situation is not well-defined in the RBDE model regardless of the assistance model, and thus cannot be allowed.

Thus, regardless of the assistance model implemented in an RBDE, it can only load a view that meets at least the first of the criteria listed above (that the rule set and the type set match). This is not affected by whether one or more views are loaded simultaneously as long as the rule sets of all of these views are subsets of the project rule set and the type sets are subsets of the project type set. The second criteria, on the other hand, is greatly affected by the assistance model because it is the assistance model that defines what the edges in the graph for the whole project rule set actually mean.

In an automation model, a forward edge between `r1` and `r2` means that when the activity of `r1` terminates, and if the appropriate effect is asserted, the RBDE should try to fire `r2`. The inability to fire `r2` for any reason, including the non-existence of `r2` in a view, does not invalidate the success of `r1`. Thus, a view containing `r1` but not `r2` is considered legal in an automation model.

In a consistency model, however, a forward edge is interpreted as a consistency implication, meaning that if `r1` is fired, the changes that its activity introduces in the database can not be made permanent unless `r2` is fired and completes successfully (and so do `r2`'s own implications). Thus, if a view contains `r1` but not `r2`, the edge representing the consistency implication would be missing from the rule graph of the view. Say that changes introduced by `r1`'s activity could become permanent (assuming all other implications are met) even though `r2` (which is not loaded) cannot be fired. If `r2` were then loaded after this point, its condition might be satisfied (because of the effect of `r1`), but the system would be quiescent, violating the definition of consistency given earlier.

> **Definition 2:** A rule set in a view is said to be legal iff for each node in the rule graph, the forward edges and the backward edges are identical to the edges for the same node in the graph of the complete project rule set.

This definition is satisfied if the graph of the view's rule set is one or more disjoint connected components (i.e., no forward or backward edges enter or leave the subgraph) of the project rule graph. The highlighted subgraph in figure 3-1 represents a legal set of rules for a view to have.

Figure 3-1: Connected Component Representing Legal Rule Subset

3.2. Evolution of the Process and Data Models

When the process model or the data model is changed, while the RBDE is off-line (not in active use),[3] the initial correctness criteria we required (that the data model matches the process model and the process model is complete) must be reevaluated to make sure they still hold. In addition, the RBDE must guarantee that the change does not invalidate any of the active views. This must be enforced regardless of the assistance model. There are other aspects of evolution, however, that depend on the assistance.

In a strict consistency preserving model, the status of objects in the objectbase at any point is a consequence of carrying out the consistency implications of each rule (in terms of invoking other rules). Say that the process model of a specific project includes two rules, `r1` and `r2`, which manipulate instances of two classes, `MOD` and `PROC`. `MOD` has several attributes, among them `status`, which is supposed to reflect the compilation status of instances of `MOD`; `PROC` has a similar attribute, `updated`. Assume that all instances of `PROC` are contained in instances of MOD. Rule `r1`, which applies to instances of `PROC`, specifies that if the tool `T` is invoked on an instance of `PROC` then the `updated` attribute of the instance must be assigned the value ‘‘True’’ after the tool invocation completes. `r2` specifies that for all instances of `MOD`, if any of the instances of `PROC` they contain has its `updated` attribute set to ‘‘True’’, then the `status` attribute of the instance of `MOD` must be assigned the value ‘‘NotUpToDate’’. Thus, at any point, the RBDE guarantees that if any `PROC` has its `updated` attribute set to ‘‘True’’, then the containing `MOD` is ‘‘NotUpToDate’’. This is considered a consistency implication according to the loaded set of rules.

Now assume that the rule `r2` is changed to specify that instead of assigning `status` to ‘‘NotUpToDate’’, it instead assigns the value ‘‘NeedsCompilation’’. Under a strict consistency mode, all instances of `MOD` that contain an instance of `PROC` whose `updated` attribute is set to ‘‘True’’ are now

[3] On-line changes are much more complicated, and must be handled by special mechanisms spanning the concurrency control and evolution support. We do not discuss this possibility in this paper.

inconsistent, since they violate this new consistency implication. The RBDE must try to change accordingly the value of the `status` attribute of all of these instances. But if it fails to do so, say because there is another rule that disallows this change in some way, the original change to `r2` must not be allowed.

This example shows that there is a need for the notion of a *legal evolution step* in a consistency preserving model. An evolution step is a single change either to the definition of a single class, or to the condition, the activity, and/or the effects of a single rule.

> **Definition 3:** In a consistency model, an evolution step is said to be legal if all objects in the objectbase can be transformed to meet the new consistency implications specified by the modified rule set.

Verifying that every object in the objectbase meets the new requirements is impractical in general (or at least highly undesirable since it is bound to be very costly). A more realistic approach might be to come up with a restriction on the kinds of changes allowed. Intuitively, if the consistency implications specified by the modified rules are weaker than the old one, then all objects in the objectbase will definitely meet this criteria. In the example above, if the only change done to the rule set was to relax rule `r2` (i.e., make it apply to a narrower set of objects) or completely remove it, then the instances of `MOD` would not violate any consistency requirements. Thus, a more practical definition is:

> **Definition 4:** In a consistency model, an evolution step is said to be legal if the consistency implications specified by the rule set after the evolution step is completed are either weaker than the implications before the evolution step is carried out, or are independent of them.

The problem then reduces to finding acceptable definitions of ‘‘weaker’’ and ‘‘independent’’ consistency implications. As described earlier, consistency implications are represented by forward edges in the rule graph. The set of consistency implications specified by a rule set can be represented by the rule graph after removing all the backward edges from the graph, to produce a *consistency graph*. For each set of consistency implications *I*, the corresponding graph is denoted by *G(I)*.

> **Definition 5:** A set of consistency implications *I′* is said to be weaker than another set *I* if *G(I’)* is a subgraph of *G(I)*.

> **Definition 6:** A set of consistency implications *I′* is said to be independent of another set *I* if *G(I’)* and *G(I)* are disjoint.

In a pure automation models, in contrast to the consistency model, there are no restrictions at all on modifying, adding and/or removing rules at any time, provided that the RBDE is effectively off-line (i.e., is quiescent and there are no users other than the one changing the process model). 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.

RBDEs that support a combination of consistency and automation may or may not have a clean separation between consistency rules and automation rules. If not, the evolution facilities is necessarily specific to the RBDE. If there is a clean separation, however, the notion of a *registered objectbase* provides an appropriate mechanism for preserving consistency. A registered objectbase consists of the set of consistency rules and the classes that these rules assume; these rules and classes are always loaded in the RBDE. The discussion above concerning a consistency preserving model applies only to the registered objectbase in the sense that compatible changes to the set of consistency rules or the set of classes contained in the registered objectbase are allowed. Thus the automation rules can change arbitrarily, but changes to consistency rules must be controlled. When an automation rule bridges a pair of otherwise disjoint connected components in the consistency rule graph, there is still no problem, since the executing the automation rule is by definition optional.

3.3. Multi-Agent RBDEs

We assumed above that there is only one thread of execution (agent) in the RBDE, i.e., the RBDE executes one action at a time, where the action might be the evaluation of a rule's condition, or the invocation of an activity followed by the assertion of the logical expression representing its effect on the objectbase (or an evolution step as described above). Under this single-agent model, the user requests a command and waits until it, and all chains resulting from it, have been completed. This is clearly wasteful from the point of the view of the user, who remains idle for the entire duration of the execution of a chain of rules. It is a very simple and clean model, however, from the point of view of the transition of objects from one state to another.

To reduce the amount of time a user has to wait, the RBDE could execute a user's command (and all its implications) in the background, and allow the user to request another command while this background processing is in progress. If that is allowed, then the RBDE must manage several threads of execution concurrently, one for each chain of rules resulting from a user's command. One of the complications that would result is the possibility that rules on different threads might cause conflicting transitions of the objectbase from one state to another, where one thread assumes that the objectbase is in a particular state (since that is the state it left it in last), while a parallel thread might have changed it to a different state. The exact nature of possible conflicts between concurrent agents depends on whether the RBDE supports a consistency model, an automation model or a combined model.

The problem of conflicting transitions by multiple concurrent agents of course also occurs in multi-user RBDEs that allow more than one user to concurrently access the project database. We discuss the concurrency control problem for multi-agents, applicable to both single-user and multi-user RBDEs, after presenting our Maximalist model.

4. A Maximalist Assistance Model

We will consider the concurrency control problem for multi-user RBDEs in the context of a *maximalist* assistance model. Our maximalist model combines consistency and automation in the sense that: (1) if the condition of a rule is not satisfied, the RBDE tries to make it satisfied by backward chaining; and (2) after the activity of any rule has terminated, the RBDE *must* carry out the implications of the activity's effect after it has been asserted. Backward chaining is complicated by the fact that it might fail (because of the uncertainty of which effect will be asserted). In this case, all the implications of the rules that were executed during the backward chaining cycle must be reversed. Thus, the RBDE must have a way of logging these effects.

The combination can be at the level of rules or at the level of predicates. Our maximalist model supports two kinds of predicates in the conditions and effects of rules: consistency predicates and automation predicates. If the condition and all the effects of a rule contain only automation predicates, the rule is an *automation rule*; if all the predicates are consistency predicates, the rule is a *consistency rule*. Otherwise it is a *combined rule*.

The rule graph for the maximalist model contains three kinds of edges: automation forward edges, consistency forward edges, and 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`. Similarly, there is a consistency forward edge between rule `r1` and rule `r2` if there is a consistency predicate in `r1` that implies a predicate in the condition of `r2`. There exists a backward edge from `r2` to `r1` if an automation predicate in the condition of `r2` is implied by a predicate in one of the effects of `r1`.

The consistency forward edges emanating from a node define the implications of the rule represented by the node. More formally, each rule *r* has a set of implications *I(r)*, which consists of all the rules that are connected to *r* by a forward consistency edge. In the maximalist model, the set of loaded rules is complete if for every rule that is loaded, all the rules on its implication set are also loaded. It must also be the case that the data model contain all the classes and attributes accessed by these rules.

To illustrate the maximalist assistance model, consider the set of rules shown in figure 4-1: the predicates in square brackets are consistency predicates and those in parentheses are automation predicates. The objectbase is in the state shown in figure 4-1. The graph representing the chaining network derived for the set of rules is shown in figure 4-2. The edges define the forward and backward chains that are possible when each rule is initiated.

Assume that the user selects a command that triggers rule `r1` on object `a`. The precondition of the rule is not satisfied because the attribute `z` of object `a` does not have the right value. Since it is the automation predicate of the condition that is not satisfied, the RBDE will trace the backward edges of `r1` to try to

Figure 4-1: Rule Set and Objectbase State

Figure 4-2: Rule Graph

make the predicate true. Since there is only one backward edge, it is pursued and the RBDE tries to fire rule `r2`, whose condition is satisfied at that point and thus its activity is executed. When `r2`'s activity terminates, one of the effects of `r2` is asserted. Let us suppose that second effect is the one chosen (the choice, as described earlier, depends on the results of the activity). This effect has two predicates, a consistency predicate and an automation predicate.

Asserting the consistency predicate of the second effect of `r2` leads to firing `r3`, according to the rule graph, and asserting the automation predicate satisfies the original predicate in the condition of `r1` that initiated the backward chain. But before concluding that the backward chain succeeded in satisfying the condition of `r1`, enabling the firing of `r1`, the consistency implications of `r2`'s effect (i.e., firing `r3`) must be satisfied first. But since the condition of `r3` is not and cannot be satisfied in the current state of the objectbase, the RBDE concludes that `r2`'s firing was invalid and thus has to be retracted. This implies reversing the state of the objectbase to the state it was in before `r2` was fired, thus causing `r1`'s condition to remain unsatisfied. Since there is no other backward edge that the RBDE can trace to satisfy `r1`'s condition, it concludes that the rule cannot be fired. The user's request is thus rejected.

One interesting complication is the side effects of `r2`'s activity. Although the rule was retracted, there is no way to undo the side effects of the rule's activity, such as the changes it introduced to the file system, because the RBDE has no way of monitoring the execution of the tool that the activity invokes if that tool is an external tool (e.g., a compiler resident on the operating system). The situation is further complicated by the fact that the tool might have done an irreversible activity, such as sending mail to the manager of the project or interacting with the user in the case of an interactive tool. This problem cannot be solved by restricting backward chaining to inference rules, as was done in section 2.1, because the forward chaining implied by the rules fired during backward chaining might include activation rules. There is no general solution to the problem of tools with irreversible side effects.

5. Concurrency Control

The last problem we address in this paper is that of synchronizing concurrent accesses by multiple agents to the project objectbase so that the synchronization requirements of the project under development, whatever those may be, are not violated. The synchronization requirements are project-specific in the sense that each project allows specific situations of cooperation between agents, depending on the software process employed, which are achieved through the interleaving of the concurrent execution threads of these agents. Note that the *synchronization* requirements prescribed by project administrators, which determine allowable interleavings among execution threads, must be compatible with any *consistency* requirements of the project (in the case of a consistency-preserving assistance model).

We have developed a multi-level framework to deal with the concurrency and cooperation problems in multi-agent RBDEs. This framework addresses the problems at three different levels: (1) interleaving of commands requested by different users; (2) interleaving of concurrent execution threads (rule chains) that are initiated by each user command; (3) interleaving of the execution of the three parts of rules. The flexibility of the concurrency control protocols decreases from level 1 to level 3. We first describe the primitives representing the three levels, and then present the concurrency control policies applied to levels 2 and 3. The concurrency control protocol at the top level may be based on any of a number of schemes described in the literature for cooperative transactions [14], and we do not discuss this here.

5.1. Concurrency Control Primitives

The basic building block of our framework is the rule. A rule consists of three parts, each of which is an atomic operation. But since these parts represent a logical unit, they have to be executed as a unit; the validity of the rule's activity and effect depends on the satisfaction of the condition. If the objectbase is changed be some other rule in such a way so as to make the condition false while the activity is in progress, the results of the activity, and thus the assertion of an effect, will be invalid. Thus, the RBDE must guarantee that the effect of an activity is not asserted if the condition has become unsatisfied while the activity was in progress. Notice that we do not consider the backward chaining that might result from an unsatisfied condition to be part of the evaluation of the condition, nor the forward chaining that might result from an asserted effect to be part of the assertion, from the point of view of atomicity.

Since chaining is caused by changes to the state of objects in the objectbase, there is an implicit assumption that during a chain that resulted from a particular state of the objectbase, the state of the objectbase must not be changed to another state that negates the original state. Thus, the whole chain must be encapsulated in a unit in order for the RBDE to be able to reason about how the changes that were introduced by concurrent agents affect the validity of the chain. We encapsulate each rule chain in a unit, which we call a *task*. Like a transaction in traditional database management [15], a task has a read set and a write set. Like nested transactions [16], tasks are made up of subtasks; leaf nodes of the nested transaction are individual rules that do not cause any more chaining. The read set of a task is composed of all the objects that are read during the evaluation of the conditions of all the rules fired in the task.

The third and last primitive that is needed is a unit to describe a single synchronization requirement. Such a unit is necessary because each project has its specific synchronization requirements that must be specified and provided to the RBDE rather than being built-in. We use the concept of *control rules* to define synchronization requirements. Each control rule has a condition that describes a conflict situation, and a repair that prescribes how to resolve the specific conflict. Both the condition and repair are specified in terms of objects, rules and tasks. Figure 5-1 shows an example of a control rule.

```
(ControlRule test_conflict
  IF  (and (in-progress RULE ?r1 USER ?u1 TASK ?t1)
           (start RULE ?r2 USER ?u2 TASK ?t2)
           (not-equal ?t1 ?t2)
           (not-equal ?u1 ?u2)
           (conflict ?r1 ?r2))
  THEN
       (abort ?r2)

  COMMENT: '' If a conflict has been detected between two rules in two
              different tasks that are initiated by two different
              users, then abort the more recent rule.'')
```

Figure 5-1: Example of a Control Rule

We have developed a two-level concurrency control protocol that synchronizes concurrent tasks by applying control rules. The bottom level of the protocol, called the *rule scheduler*, operates on rules and guarantees serializability of concurrent rules. The top level, the *task controller*, synchronizes concurrent tasks. This layer is itself decomposed into two components: (1) a layer that detects potential conflicts using a relatively low-overhead serializability-based algorithm; and (2) a layer that resolves these potential conflicts using a more expensive knowledge-based algorithm.

5.2. The Rule Scheduler

The simplest way to guarantee the atomicity of a rule execution is to treat the rule as an atomic transaction and to apply a conventional serializability-based concurrency control protocol to guarantee the atomicity of concurrent rules. The details of the protocol depend on the specification of the effects of rules. In the RBDE model presented in section 2, the effect of a rule specifies which attributes of which objects to change in order to reflect the changes introduced by a tool. If these objects are known when the rule is fired (i.e., after binding the parameters of the rule to objects) and before the activity is initiated, then the set of objects that are written by the rule can be derived when the rule is fired.

In the more general case, however, the full set of objects that are read is known only after the evaluation of the rule's condition is completed, and the write set is known only after the invocation of the rule's activity is completed, at which point it is known which objects among those read were actually changed by the activity. In either case, either a variation of the traditional two-phase locking protocol, or a variation of optimistic concurrency control that supports merging rather than rollback of user activities (such as done in Sun's NSE [17]) can be used. We give the details of only the locking protocol here.

When a rule is fired, the rule tries to acquire a read lock for every object in its read set before evaluating its condition. If it cannot do so, the rule execution is rejected and the rule terminates unsuccessfully. Otherwise, the evaluation is carried out, and if the condition is not satisfied, the rule execution terminates unsuccessfully. In the case where all the read locks can be acquired and the condition is satisfied, the activity is invoked. By definition, the objects accessed by either the activity or the rule's effects must be in the set of objects that are either read while evaluating the condition, or bound when the parameters of the rule are evaluated.

If an activity operates on external files or local copies of objects, there is not a need to lock any object it accesses; after the execution of the activity is completed, however, the rule must then acquire a write lock on every object mapped to the copies or files written by the activity. If it is not possible to obtain all the necessary write locks, any locks held are released and the rule execution terminates unsuccessfully. If the activity completes successfully, all the read locks are released, the appropriate effect is asserted by replacing objects in the objectbase by any copies written by the activity and changing the values of the other objects accessed in the effect, and only then are the write locks released.

Although this protocol guarantees atomicity of a rule, it can lead to wasted efforts in the case when a write lock cannot be acquired after the activity working on copies of objects has completed and the RBDE attempts to assert the effect on the objectbase itself, leading to the discarding of all the work that an activity performed on local copies. This can be avoided by acquiring a write lock on all objects in the read set of the rule (which is known before the activity is invoked). If these locks can be acquired, then the rule is guaranteed to complete successfully because the objects that are written by the activity and the effects are a subset of the locked set of objects. This scheme might be too restrictive, however, in cases where the write set is a small subset of the read set, such as during a search, but in the absence of knowledge about the write set before the invocation of the activity, we cannot do any better than one of these two schemes.

Note that if the activity manipulates the objectbase directly, then write locks must be obtained on all its arguments before it begins in any case, so the second scheme is followed. However, the assumption that rules must be executed atomically is not appropriate for rules that invoke external tools because the execution of these tools might last for a long time (e.g., an interactive tool). In this case, one of the techniques used for long transactions, as mentioned above for cooperation among users, might be more appropriate for the rule scheduler.

5.3. The Task Controller

The task controller is decomposed into a conflict detection layer and a conflict resolution layer. We have previously explained the details of the first layer of the task controller and sketched the second layer, for an automation assistance [8]. In this section, we explain how the maximalist model impacts the second layer of the task controller.

We have developed a nested incremental locking (NIL) protocol for detecting conflicts between concurrent tasks in an automation model. The protocol models each task as a nested transaction, whose read set is incrementally formed from the read sets of all the rules that are fired within the task. Similarly, the write set is formed from the write set of component rules. NIL treats the top-level task as a normal atomic transaction for the purpose of *detecting* conflicts only (and not for *resolving* these conflicts). It thus detects any potential conflict that occurs whenever two tasks attempt to acquire two locks on the same object and at least one of the locks is a write lock.

Internally, however, subtasks are executed concurrently and their actions are synchronized by an internal mechanism. A subtask can fail and be restarted or replaced by another subtask without causing the whole nested task to fail or restart. This is useful in an automation model because the RBDE might decide to fire a rule, only to find out that its condition cannot be satisfied; in this case, the rule has to be abandoned and other rules explored. Automation assistance should not be invalidated because of such failures.

The NIL protocol applies also to the maximalist model. The conflict resolution mechanism that we developed for an automation model, however, must be changed in order to support the maximalist model.

In our previous work, we based the conflict resolution mechanism on control rules, and did not place any restrictions on what the repair of a control rules can prescribe because only automation was affected. In the maximalist model, however, there are consistency implications introduced by the consistency predicates of rules, and these implications cannot be violated. Thus, the repairs of control rules must be carried out in accordance with these implications.

To illustrate what we mean, consider the example described in section 4. Say that a user requests a command that leads to the backward chaining cycle, `t1`, described in section 4. At the same time, another task, `t2`, initiated by another user is in progress. Suppose that the control rule shown in figure 5-1 is triggered because the conflict detection layer detected a potential conflict between the rule `r3` in task `t1` and another rule in task `t2`. Suppose that `r3` was executed more recently than the rule it conflicts with. In this case, the repair rule specifies that rule `r3` should be aborted.

In an automation model, the conflict resolution mechanism would have to abort `r3` only. In the maximalist model, however, `r3` and `r2` must both be aborted by undoing the changes that both rules introduced, because firing `r3` resulted from a consistency implication of `r2` and if this implication cannot be carried out, `r2` must be rolled back. If `r3` had been fired due to chaining through an automation predicate in one of `r2`'s effects, then `r2` would not have to be aborted.

In order to rollback a rule, the conflict resolution mechanism must keep track of all the changes made by every rule within a task. This can be done by logging or other conventional mechanisms, but it is important to remember that many results of external tools simply cannot be rolled back.

6. Conclusions

In this paper, we investigated the spectrum of assistance models that might be implemented in a rule-based development environment and how the chosen assistance model affects the problem of scaling up of the RBDE. The spectrum ranged from strict consistency preservation to pure automation, and included many possible combinations in between, considering how predicates of rule conditions and effects were interpreted and whether backward and/or forward chaining are supported.

We focused on three problems:

- The multiple views that might be employed by the same user over the lifecycle of a project and by different users who carry out different roles on the project;
- The evolution of both the data model and the process model in terms of both the interactions between the data and process models and their interactions with the prior state of the objectbase before an evolution step; and
- The concurrency control mechanisms suitable for detecting and resolving synchronization conflicts among rules and rule chains, where the specific concurrency control policies employed may be specific to the rule-based process model employed by the project.

We described how the feasible approaches to solving the multiple views and evolution problems are restricted by the assistance model chosen, and then presented a maximalist model combining consistency and automation in order to construct our solution to the concurrency control problem. Our concurrency control mechanism consists of a standard protocol for detecting potential conflicts among rule chains, or tasks, and then employs control rules to determine how to resolve the conflict, including ignoring it, a semantic repair, or rolling back one of the tasks, according to the requirements of the software process.

Acknowledgments

We would like to thank Michael Sokolsky, who collaborated with us on designing and implementing the single-user MARVEL, Israel Ben-Shaul, who is working on the multi-user object management system for MARVEL, and George Heineman, who is working with us on developing an RBDE model that combines consistency preservation and automation. The single-user MARVEL version 2.6 is available for licensing to educational institutions and industrial sponsors; contact Israel Ben-Shaul, israel@cs.columbia.edu, 212-854-2930 for information.

References

[1] *CLF Manual*
University of Southern California, Information Sciences Institute, Marina del Rey CA, 1988.

[2] N. H. Minsky and D. Rozenshtein.
A Software Development Environment for Law-Governed Systems.
In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, New York NY, 1988.
Published as a special issue of *SIGPLAN Notices*, 24(2):65-75.

[3] K. E. Huff and V. R. Lesser.
A Plan-based Intelligent Assistant that Supports the Software Development Process.
In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, New York NY, 1988.
Published as a special issue of *SIGPLAN Notices*, 24(2):97-106.

[4] D. Wile and D. Allard.
Worlds: An Organizing Structure for Object-Bases.
In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, New York NY, 1988.
Published as a special issue of *SIGPLAN Notices*, 24(2):16-26.

[5] K. Benali et al.
Presentation of the ALF Project.
In *9th International Conference on System Development Environments and Factories*. Berlin, May, 1989.

[6] W. Deiters and V. Gruhn.
Managing Software Processes in the Envireonment MELMAC.
In *ACM SIGSOFT 4th Symposium on Software Development Environments*, pages . ACM Press, Irvine CA, December, 1990.
To Appear.

[7] N. S. Barghouti and G. E. Kaiser.
Multi-Agent Rule-Based Development Environments.
In *5th Annual RADC Knowledge-Based Software Assistant (KBSA) Conference*. Syracuse NY, September, 1990.
To Appear.

[8] N. S. Barghouti and G. E. Kaiser.
An Object-Oriented Framework for Modeling Cooperation in Multi-Agent Rule-Based Development Environments.
IEEE Expert , December, 1990.

[9] G. E. Kaiser, P. H. Feiler and S. S. Popovich.
Intelligent Assistance for Software Development and Maintenance.
IEEE Software 5(3):40-49, May, 1988.

[10] G. E. Kaiser, N. S. Barghouti and M. H. Sokolsky.
Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel.
In *23rd Annual Hawaii International Conference on System Sciences*, pages 131-140. Kona HI, January, 1990.

[11] C. Pu, G. Kaiser and N. Hutchinson.
Split Transactions for Open-Ended Activities.
In *Proceedings of the 14th International Conference on Very Large Databases*, pages 26-37. Morgan Kaufmann, San Mateo CA, August, 1988.

[12] G. E. Kaiser.
A Flexible Transaction Model for Software Engineering.
In *Proceedings of the 6th International Conference on Data Engineering*. IEEE Computer Society Press, Los Angeles CA, February, 1990.

[13] D. Cohen.
Compiling Complex database transition triggers.
In *1989 ACM SIGMOD International Conference on the Management of Data*. ACM Press, New York, NY, 1989.
Published as a special issue of *SIGMOD Record*, 18(2):225-234.

[14] N. S. Barghouti and G. E. Kaiser.
Concurrency Control in Advanced Database Applications.
Technical Report CUCS-425-89, Columbia University Department of Computer Science, New York, NY, March, 1989.

[15] K. Eswaran, J. Gray, R. Lorie and I. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[16] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, Cambridge MA, 1985.

[17] E. W.Adams, M.Honda, and T. C. Miller.
Object Management in a CASE Environment.
In *11th Int'l Conf. Software Eng.*, pages 154-163. Computer Society Press, Washington DC, May, 1989.