Concurrency Control in Rule-Based Software Development Environments

Naser S. Barghouti

Technical Report CUCS-001-92

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences.

COLUMBIA UNIVERSITY
1992

Concurrency Control in Rule-Based Software Development Environments

Naser S. Barghouti

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences.

COLUMBIA UNIVERSITY
1992

ABSTRACT

Concurrency Control in Rule-Based Software Development Environments

Naser S. Barghouti

This dissertation investigates the *concurrency control problem* in software development environments (SDEs). The problem arises when multiple developers perform activities that *concurrently* access the project's components, stored as database objects. The interleaved execution of the developers' activities leads to *interference* if they access overlapping sets of objects concurrently. An SDE can ensure that activities never interfere by modeling their execution in terms of atomic transactions and allowing only serializable schedules. This prevents *cooperation*, which requires some degree of interference between the activities of multiple developers.

To allow cooperation, an SDE must be provided with semantic information about development activities. In *rule-based SDEs*, the necessary information is readily available in the set of rules that defines the *process model* of a project. The rules are loaded into the SDE, which provides process-specific assistance through a rule chaining engine. A single user command might lead the chaining engine to initiate a rule chain. The concurrency control problem in rule-based SDEs manifests itself in terms of interference between concurrent rule chains. We present a mechanism that extracts semantic information from the process model to solve the concurrency control problem without obstructing cooperation.

The mechanism is composed of two modules: (1) a conflict detection module, which models activities as nested transactions and uses *two-phase locking* to detect interference; and (2) a conflict resolution module, which employs two protocols to resolve interference. The first protocol, *SCCP*, uses the process model to implement a priority-based scheme that aborts the "least important" of the interfering transactions. The second protocol, *PCCP*, overrides SCCP by consulting process-specific *control rules*, written by the project administration. Each control rule describes a specific interference and the actions that resolve it. We have implemented SCCP and parts of PCCP in MARVEL, a multi-user rule-based SDE developed at Columbia.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisor Gail Kaiser for her guidance, support and encouragement for the duration of this research. Gail has taught me a lot about how to conduct research, write papers, present talks, and be critical in my thinking. I would also like to thank the other members of my thesis committee: Bob Balzer, Dan Duchamp, Sal Stolfo, and Alex Wolf. They have all spent many hours reviewing sections of this dissertation. In addition, I would like to acknowledge my colleague George Heineman, who read a complete draft of the dissertation and provided many corrections and suggestions for improvements. I would also like to acknowledge my office mates Shu-Wie Chen and Felix Wu for putting up with me during the worst of times. Thanks are also due to past and present members of the Marvel project: Michael Sokolsky, Israel Ben-Shaul, George Heineman, Timothy Jones, Mark Gisi, and Kevin Lam, who implemented the parser for control rules. I would also like to thank my colleagues, especially Wenwey Hseush, Steve Popovich, Timothy Balraj, Paul Michelman, Bulent Yener, and others, too many to mention here, who have provided me with friendship and support the duration of this research.

To my homeland, Palestine

Chapter 1
Introduction

Every software project assumes a specific development process and a particular organization for its components. As software projects become larger and more complex, the importance of sophisticated ***software development environments***[1] (SDEs) that can assist in managing a project's data and carrying out its development processes increases. Early SDEs were limited in two respects: (1) they were inflexible because they assumed either no development process or a fixed hard-wired development process, and (2) their data management capabilities were generally limited to either supporting only one developer at a time or isolating multiple developers.

Since the development processes of different projects are different, SDEs must be flexible enough to tailor their assistance to support different processes. ***Process-centered SDEs*** aim at exactly that: they load an encoding of the development process of a project, and tailor their assistance to the specified process. The encoding is typically written by the project administrator or manager. Process-centered SDEs thus overcome one of the limitations of early SDEs.

Almost any large project involves ***cooperation*** among multiple developers who work together on developing project components (e.g., source code, documentation, etc.). Consequently, it is essential that SDEs allow multiple developers to work on the same project concurrently and in a cooperative manner, i.e., not in isolation. The effect of concurrency, however, is to allow multiple developers to interleave their access to the various components of the project. This interleaving can lead to ***interference*** that may corrupt the project components. Avoiding such interference is called the *concurrency control problem.* The problem has been studied extensively in traditional database sys-

---

[1] Appendix A is a glossary of acronyms and all terms shown in ***bold italics***.

tems (***DBSs***)[2]. Unfortunately, the traditional concurrency control mechanisms developed for DBSs are too restrictive for SDEs because they enforce serialization of access to data, making cooperation unacceptably difficult.

In this dissertation, we develop an advanced concurrency control mechanism that employs two protocols. The first uses information about the development process, specifically the consistency constraints of the process and the structure of data in a software project, to provide the default concurrency control policy. The second protocol overrides the default policy by using an explicit specification of how to resolve conflict situations that result from interference between two transactions. Instead of carrying out the default actions to resolve a conflict situation, the second protocol executes actions prescribed by the project administrator specifically to resolve that class of conflicts.

We have chosen the approach of constructing a specific multi-user process-centered SDE architecture and developing our concurrency control mechanism in the context of that architecture. Our architecture is based on the MARVEL system, but it differs from the implementation of MARVEL in several respects. The architecture is composed of a specification language, ***EMSL***, a rule-based development kernel, ***RBDE***, and a language for programming the concurrency control policy, ***CRL***.

One advantage of the approach we have followed is that we were able to implement our solution in a working system. In fact, most of the research described in this dissertation has been implemented in MARVEL. It is expected that all the constructs described as part of EMSL, RBDE and CRL will be implemented in MARVEL at some point, but not necessarily as part of this thesis work. The implementation of parts of the two concurrency control protocols is significant because most of the other mechanisms that have been proposed for advanced database applications, like SDEs, have either not been implemented at all or have been implemented only in toy environments. By partially implementing our solution, we have proven the practicality of the approach.

---

[2]As in [Bernstein et al. 87], we use the term database system, instead of the more conventional database management system, to denote any system that uses a database, including a simple file system with a transaction management facility.

In the rest of this introductory chapter, we first motivate the need for a programmable and flexible concurrency control policy in multi-user SDEs by giving a simple example. Variants of this example will be used throughout the dissertation. We then give an overview of the dissertation.

1.1. A Motivating Example

Three programmers, Bob, John and Mary, are working on the same software project[3]. They use the C programming language, and several software tools resident on the operating system, such as `emacs`, `cc`, and `lint`, to develop a program called `Prog`. `Prog` consists of three modules: `ModA`, `ModB` and `ModC`, a directory, `includes`, where all the header files reside, and three archive libraries: `LibA`, `LibB`, and `LibC`, where all object code is archived. Each library stores the archived object code of one module (e.g., `LibA` corresponds to `ModA`). Each of modules `ModA`, `ModB` and `ModC` consists of a set of C source files that comprise the main code of the project.

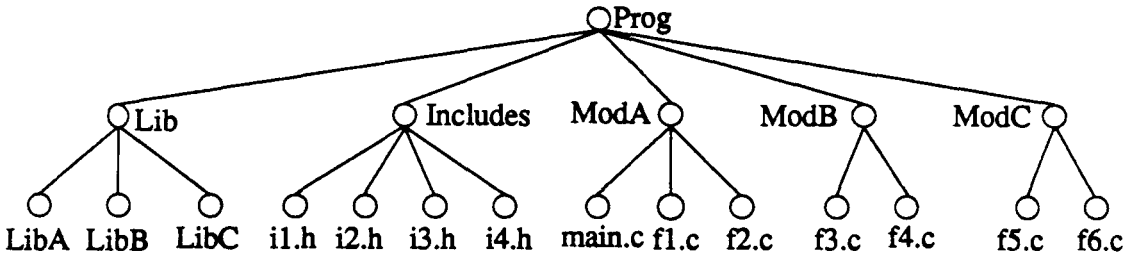

Figure 1-1: Organization of Example Project

Each of the C source files in these three modules includes some of the header files in the `includes` directory: `i1.h` is included in `main.c`, `f1.c`, `f2.c`, and `f3.c`; `i2.h` is included in `main.c`, `f1.c`, and `f2.c`; finally, `i3.h` is included in `f4.c`, `f5.c` and `f6.c`. Figure 1-1 depicts the organization and components of the project.

[3]We only need John and Mary in this example, but we introduce Bob because he will participate in the example later on in the dissertation.

While testing the project, John and Mary discover two bugs. John is assigned the task of fixing one bug that is suspected to be in module ModA. He starts browsing the C files in module ModA. Mary's task is to explore a possible bug in the code of module ModB, so she starts browsing the C files in ModB. After a while, John finds out that there is indeed a bug in ModA, caused by a bug in il.h in the includes directory. John modifies a few definitions in il.h, creating a new version of it, and then proceeds to finish his modifications of module ModA.

Mary finds a bug in the code of one of the C files in ModB; she modifies various parts of the module to fix the bug. Mary now wants to test the new code of ModB. She is not concerned with the modifications that John has made in ModA because ModA is unrelated to ModB. However, the source files in ModB include il.h; it makes sense for Mary to access the new version of il.h, since that version has the bug fixes that John had made. The modifications to the definitions in il.h might have introduced inconsistencies with the usage of these definitions in the code of the C files in module ModB. But since John is still not done with his modification task, of which the modifications in il.h were a part, Mary will either have to access il.h at the same time that John is accessing it, wait until he is done, or access the old version of the header file.

One trivial solution of the problem is not to have any concurrency control. Basically, Mary and John would be allowed to access any objects they want without any constraints. In this case, John and Mary may not be aware that they are overwriting each other's work when they access il.h concurrently. For example, John might change the definition of a data structure in il.h to fix a bug. This change might cause an inconsistency with Mary's changes to module ModB. Unaware that John had changed il.h, Mary might think that she has discovered a bug in il.h; she might go ahead and change the definition of the same data structure in il.h that John had changed previously, in order to remove the inconsistency with the way the data structure is used in the code of ModB. John would then suddenly discover that the bug he thought he had fixed has re-occurred. It should be clear from this simple example that having no concurrency control might lead to very confusing and chaotic situations. Yet, this is the current state of affairs in many development projects.

Another solution is to model both tasks in the above example as transactions and use a traditional concurrency control scheme, such as strict two-phase locking, to control access to objects. If locks are at the granularity of modules, then John and Mary will be allowed to concurrently lock modules `ModA` and `ModB`, respectively, since they work in isolation on these modules. Both of them, however, need to access the header files in `includes` at the same time to compile their code, which causes a locking conflict. One of the two tasks will be blocked, waiting for the other to finish. Even if the locks were at the granularity of files, John and Mary would not be able to access `il.h` in the manner described above. The locks will be released only after reaching a satisfactory stage of modification of the code, such as the completion of unit testing. Using other traditional serializability-based concurrency control schemes will not solve the problem because these schemes also require the serialization of Mary's work with John's, i.e., one appears to complete before the other starts.

A third solution might be to require John and Mary to "reserve" (also called checkout) files before accessing them. Reserving a file prevents other users from accessing it. Files are deposited (also called checkin) only when developers have finished working on them. In this case, John would reserve `il.h`, modify it, and then deposit it. John would also reserve `ModA` before modifying it. Suppose that after John had deposited `il.h`, Mary reserves both `ModB` and `il.h`. Since the changes John makes to `il.h` affect the files in `ModA`, it is likely that John would need to access `il.h` again if he discovers that some of the changes he made in `il.h` introduced inconsistencies with the code of `ModA`. However, since Mary has reserved `il.h`, John will not be able to access it until Mary deposit it. This reserve/deposit model prevents John from accessing `il.h` while Mary has it reserved.

A more satisfactory solution might be to support multiple versions of `il.h`. When John checks out `il.h`, Mary would still be allowed to access the last version of `il.h` while John works on a new version. But this solution requires Mary to later retest her code after the new version of `il.h` is released, which is really unnecessary if `il.h` does not change again. Supporting parallel versions also introduces the problem of merging. For example, Mary and John could have worked for a long time on parallel versions of

`il.h`. When it comes time to integrate their work, they might discover that some of the changes they made cannot be merged. Thus, any solution that assumes merging of parallel versions might be costly.

None of the solutions discussed above distinguish between cooperative development, which requires some degree of interference among the cooperating developers, and isolated development, in which interference should not be allowed. What is needed is a flexible concurrency control scheme that would allow both John and Mary to access objects concurrently, but yet can detect situations when they are "stepping on each other's toes" (e.g., undoing each other's work). The administrator of the project should be able to specify certain situations when the concurrency control scheme should allow John and Mary to access the same objects at the same time. This might be necessary in a cooperative development effort. Furthermore, it might be necessary during some of the phases of development (e.g., the release phase) to enforce stricter concurrency control policies. It would be advantageous to be able to change the concurrency control policy if the consistency requirements of the project change[4]. Both of these goals, flexibility and programmability, are achieved by the concurrency control mechanism we present in this dissertation.

The rest of this chapter is a synopsis of the dissertation. First, we briefly describe our multi-user SDE architecture and state the concurrency control problem in it. We then overview related work in three areas and show how our work fits into the ''big picture'' of these three areas. Next, we outline the main ideas of our concurrency control mechanism, which is the heart of this dissertation. Finally, we overview the contributions of this dissertation and its organization.

---

[4]Changing the consistency requirements of a project cannot, in general, be done on-line since there might be some in-progress transactions that support the old requirements. What we mean here is the ability to change the requirements off-line and have the concurrency control policy change accordingly next time the environment is in use.

1.2. A Software Development Environment Architecture

We construct an architecture composed of a specification language, EMSL, and a software development kernel, RBDE, that provides automated assistance to developers of a software project. EMSL provides constructs for writing a rule-based specification of the development process of a project and an object-oriented specification of the project's data model. The activities comprising the development process are prescribed in terms of ***rules***, while the structure and organization of the project's components are specified in terms of ***classes***. The administrator of a project writes these EMSL specifications and loads them into RBDE, which tailors its behavior and its data management capabilities accordingly. The tailored RBDE (the base RBDE plus the EMSL specifications) presents the developers with commands that are either built-in (i.e., shared by all RBDE environments) or correspond to a selected subset of the rules that were loaded. RBDE also creates a database for the project, where all project components will be stored as ***objects*** that are instances of the classes defined by the administrator. The objects inherit typed ***attributes*** from their classes.

```
edit [?c: CFILE]:
    # 1- the condition.
    :
    (and (?c.reservation_status = CheckedOut)
         (?c.locker = CurrentUser))

    # 2- the activity.
    { edit output: ?c.contents }

    # 3- the effects.
    (and (?c.status = NotCompiled)
         (?c.timestamp = CurrentTime));
```

Figure 1-2: Example EMSL Rule

Each rule in EMSL has three parts, as is shown in figure 1-2. The first part, the condition, must be satisfied before the second part, the development activity, is executed. The condition is a logical expression composed of predicates, each of which is essentially a query (i.e., read operation) on the values of objects' attributes in the database. The development activity is generally an invocation of a software tool that resides on the operating system (e.g., a compiler or an editor). Since a tool invocation might have

several possible results (e.g., a compilation might either succeed or discover errors), these results are mapped into the third part, a set of mutually exclusive effects. Each effect is composed of a set of assignment predicates, each of which changes (i.e., writes) the value of an object's attribute. The outcome of the rule's activity determine which effect to assert on the objects in the database.

RBDE provides assistance during the development of a software project. When a user requests a command, RBDE automatically fires a rule that implements the command by manipulating the components of the project (i.e., by changing the values of attributes of the objects representing these components). If the effect of a rule changes the values of objects' attributes in the database in such a way that the conditions of other rules become satisfied, all of those rules are fired automatically, in a forward chaining manner. Alternatively if the condition of a rule is not satisfied, backward chaining is performed to attempt to make it satisfied. If that fails and the condition remains unsatisfied, then the rule cannot be executed at this time. The details of the EMSL language and rule execution are presented in chapter 2.

1.2.1. Supporting Multiple Users in RBDE

Large-scale software development efforts often involve teams of developers who cooperate on the same project. Since RBDE stores all project components in a single database, these developers must share this common database. In order to allow the developers to access the components of the project in the database concurrently, RBDE implements a client/server model [Ben-Shaul 91] in which access to the project database is controlled by a centralized server that can service concurrent access requests by multiple clients. Each developer interacts with a client process in order to complete an assignment. Every request by a user to execute a command is sent to the server, which provides the client with access to the objects needed to execute the command. Concurrent requests by multiple users to execute commands might initiate the firing of multiple rule chains, which perform operations on objects in the shared project database. The server executes these rule chains concurrently in an interleaved fashion.

1.3. The Concurrency Control Problem

Concurrent processing presents several new technical challenges, one of which is interference between concurrent rule chains in terms of accessing objects in the database. By definition, interference between concurrent rule chains can violate the consistency of the objects accessed by the chains. Interference (also called *concurrency conflict* or simply *conflict*) occurs when objects that have been accessed by a rule, `r1`, are updated by another rule, `r2`, in a concurrent chain. This overlapping access might, for example, lead to changing objects that were read during the evaluation of the condition of `r1`. The validity of `r1`'s execution depends on the assumption that these objects will not be changed by rules in other rule chains, while the chain containing `r1` is still executing. Thus, if any of the objects is actually changed, `r1`'s execution might have to be invalidated, meaning that its results have to be undone. In addition, all the rules that were fired after `r1` in `r1`'s chain might have to be invalidated since the reason for firing them (i.e., the successful firing of `r1`) has now been reversed.

The concurrency control problem arises only if two rule chains access the same objects concurrently. There is a need to synchronize such concurrent access in order to avoid conflicts. Our solution to the concurrency control problem combines and integrates research results and concepts from three areas: SDEs, software process modeling, and database concurrency control[5]. The progression of research in these three areas points toward increasing cross-fertilization to build systems that can support the specific needs of applications involving teams of users. Before we explain our concurrency control mechanism, we give a brief overview of the relevant research in these three areas and show how our work fits into the "big picture."

---

[5]Note that we have also applied many ideas from Artificial Intelligence research. Incorporating AI in the picture, however, would complicate the discussion and make it even longer than it is now.

1.4. This Dissertation in Perspective

SDEs aim to increase programmer productivity and software reliability. Early research focused on exploiting general-purpose file systems and providing a collection of independent file-based tools for system building [Feldman 79], editing [Stallman 84], debugging [Linton 81, Bates and Wileden 83], and version control [Tichy 85, Rochkind 75]. The invocation of these tools, however, was left up to the user. In addition, the data generated by the tools was stored in files managed by the file system, which meant that these files could easily be accessed and corrupted by external tools.

These problems prompted the development of programming environments that appeared to be "intelligent" because they automated some of the tool invocation and provided object management systems to store and manage a project's data [Dart et al. 87]. The "intelligence" of these environments, however, was hard-wired in their code, making them difficult to change or adapt to different projects. One approach to overcome this difficulty has been to build SDEs that can tailor their behavior by loading a specification of the desired development process.

Providing a complete and efficient formalism for specifying the development process is the main problem addressed by the software process modeling community. Early research in that field produced "fixed" models that prescribed a particular cycle for software development, such as the waterfall model [Royce 87]. This approach had two main disadvantages: (1) no single model was found to be appropriate for all development efforts, which vary in their development processes; and (2) the models were meant only as guidelines to be enforced manually by project personnel and management, and not as executable encodings that can be enforced automatically. To solve the first problem, researchers proposed meta-models in which more than one process model can be described, such as the spiral model [Boehm 88].

The second problem was solved by the idea of *process programming*, which treats software processes as programs that can be executed [Osterweil 87, Perry 89a, Katayama 90]. Several process-centered SDEs that followed this approach were proposed and constructed, including Arcadia [Taylor et al. 88, Sutton 90], HFSP [Katayama 89], and

MELMAC [Deiters and Gruhn 90]. One approach to process programming that has gained popularity recently is ***rule-based process modeling***, in which each step in the process is prescribed in terms of a rule. Some of these steps can then be carried out automatically through a rule chaining engine. The RBDE architecture is based on this approach.

Process-centered SDE provide a mechanism for tailoring their behaviors to the needs of a particular project. However, having the development process explicitly encoded does not alone solve the problem of supporting multiple users, a requirement for any large-scale software development effort. The basic problem is the inability to allow concurrent access to project components by multiple users while still maintaining the consistency of these components. This problem arises in all multi-user SDEs, e.g., Darwin [Minsky and Rozenshtein 90], the CommonLisp Framework [Balzer 87, CLF Project 88], Oikos [Ambriola et al. 90], and MELMAC [Deiters and Gruhn 90]. In general, the proposed solutions fall into two categories: (1) checkout/checkin mechanisms in which developers work on different versions of the whole database or parts of it, which are then merged together; and (2) transaction mechanisms that basically force the serialization of the tasks of concurrent developers. This second approach has also been used in rule-based production systems in which multiple rules are fired in parallel [Stolfo 84, Ishida and Stolfo 85, Schmolze 89, Pasik 89, Miranker et al. 90, Kuo et al. 90]. The serialization approach is based on research done by the database community.

The database community has studied the concurrency control problem extensively in the context of traditional DBSs. The two main concepts that were developed to solve this problem are the ***transaction*** and ***serializability*** [Eswaran et al. 76, Bernstein et al. 87]. A transaction groups together operations comprising a complete task, and when executed transforms the database from one consistent state to another. When multiple transactions are executed concurrently, their operations are interleaved. Such interleaving, called a *schedule*, is consistent (i.e., does not result in interference among the concurrent transactions) if the transactions comprising it are executed serially one after the other. It can then be established that a *serializable* schedule, one that is computationally equiv-

alent to a serial schedule, is also consistent. Using transactions and serializability, the concurrency control problem in conventional database systems reduces to that of testing for serializable schedules. We explain the concepts of transaction and serializability in more detail in chapter 3.

It was recognized, however, that serializable transactions were too restrictive for advanced database applications, such as CAD/CAM systems and software development environments, that involve long interactive database sessions and cooperation among multiple users [Bernstein 87]. *Semantics-based concurrency control* [Salem et al. 87, Garcia-Molina and Salem 87, Beeri et al. 88, Kutay and Eastman 83, Korth and Speegle 90, Pu et al. 88] and *cooperative transaction models* [Bancilhon et al. 85, El Abbadi and Toueg 89, Dowson and Nejmeh 89, Klahold et al. 85, Skarra and Zdonik 89, Kaiser 90] were proposed to overcome some of the limitations of serializable transactions. As with software process models, none of the new transaction models was found appropriate for all applications. Consequently, Chrysanthis and Ramamritham have recently developed a high-level formalism in which many transaction models can be defined [Chrysanthis and Ramamritham 90]. Other researchers have developed mechanisms for explicitly programming the concurrency control policy [Skarra 91]. For a survey of advanced transaction mechanisms, the reader is referred to [Barghouti and Kaiser 91a].

We have applied the concept of a high-level coordination modeling formalism to rule-based SDEs. The result is a language, CRL, in which the project administrator can program a range of concurrency control policies, and a conflict resolution protocol, which provides the runtime environment that can implement any of these policies in RBDE. Figure 1-3 depicts the progression of research in the three areas discussed above and shows how our work fits into the "big picture."

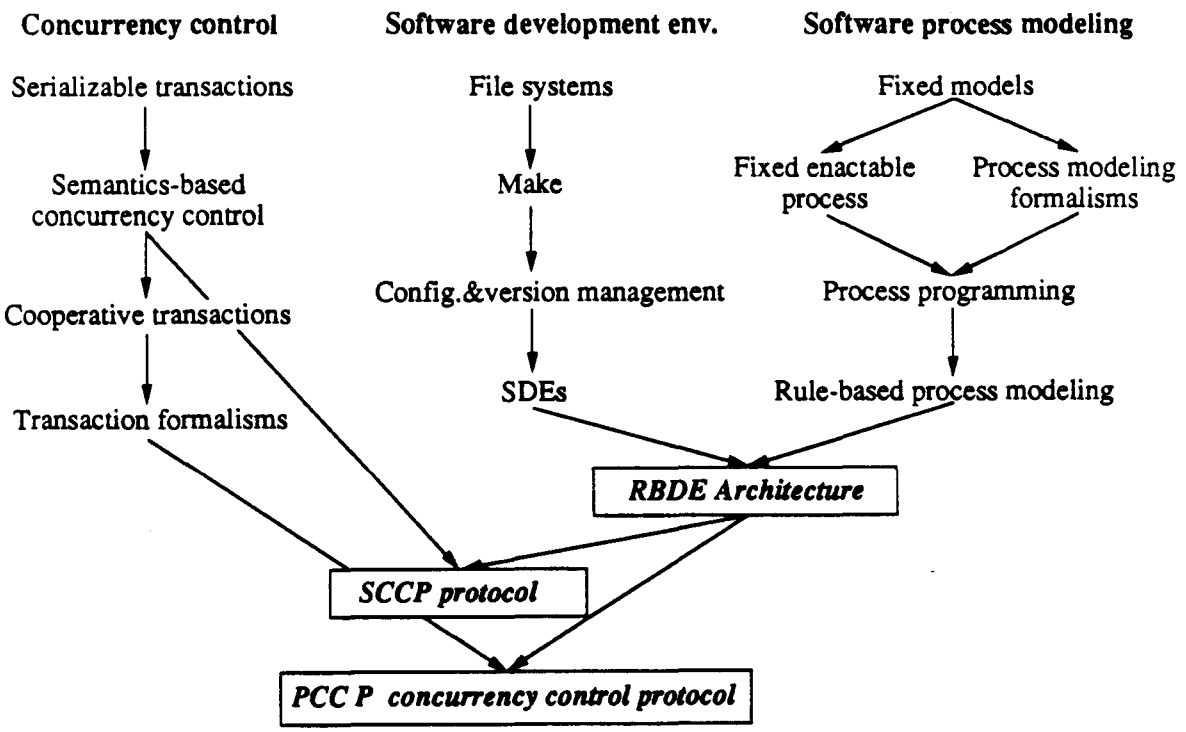

Figure 1-3: The Big Picture

1.5. Overview of our Approach

We develop a concurrency control mechanism composed of a *conflict detection* module and a *conflict resolution* module. The first module employs a transaction model, which encapsulates each individual rule in an atomic transaction, and each rule chain in a ***nested transaction*** [Moss 85], where each rule becomes a subtransaction. A ***nested granularity locking*** (NGL) protocol, which is based on multiple granularity locking [Kim et al. 87], is then used to detect any interference between concurrent rule chains. Interference is defined as any non-serializable interaction among the nested transactions encapsulating the concurrent chains.

Conflict resolution is carried out by a Scheduler that employs two protocols: (1) a

semantics-based concurrency control protocol (***SCCP***), which implements the default concurrency control strategy; and (2) a programmable concurrency control protocol (***PCCP***), which implements a project-specific concurrency control policy. When NGL detects a conflict between two transactions, the Scheduler first tries to resolve the conflict by using PCCP, which attempts to resolve the conflict according to the project-specific policy. If such a policy was not specified, or if PCCP cannot resolve the conflict for reasons explained in chapter 6, then the Scheduler applies SCCP, which guarantees that the conflict will be resolved according to a hard-wired priority scheme.

The SCCP protocol, which builds on the mechanisms of semantics-based concurrency control [Garcia-Molina 83, Salem et al. 87], uses semantic information about the consistency constraints of a project in order to resolve conflicts detected by NGL. To specify the consistency constraints of a project's development process, EMSL provides constructs that can be used to distinguish between *consistency predicates* and *automation predicates* in both the conditions and effects of rules; a single rule might contain both kinds of predicates. The consistency predicates define the consistency constraints of a project since they prescribe to RBDE the mandatory steps that RBDE must perform after firing a rule. Automation predicates prescribe to RBDE the optional steps that RBDE can automatically perform in response to firing a rule.

SCCP guarantees that the consistency constraints of a project will be maintained by disallowing any concurrent interaction that violates these constraints (i.e., that prevents the mandatory steps from being carried out). SCCP resolves a conflict by aborting one of the interfering transactions. Aborting a transaction, however, does not automatically require that the whole chain of which the transaction is a part be aborted. If the aborted transaction is part of a nested transaction that encapsulates the execution of a rule chain initiated by a consistency predicate, then the whole nested transaction must be aborted to guarantee the preservation of the consistency constraint implied by the predicate. Otherwise, only one transaction (one of the conflicting transactions) needs to be aborted.

SCCP determines which of the two interfering transactions to abort by using information about the kind of predicates that triggered the the transactions and the nature of the commands encapsulated by the transactions (e.g., whether they are interactive or not). Thus,

in the absence of a project-specific concurrency control policy, RBDE enforces serializability among rule chains, but uses semantic information to abort the least important of the two interfering transactions and to avoid unnecessarily rolling back other transactions.

In the multi-user RBDE architecture, however, it might be the case that the consistency constraints need to be relaxed temporarily in order to allow for cooperation between multiple developers. For example, it is sometimes necessary to let one developer access a a file that is being modified by another developer, in order for the first developer to be able to produce an executable; the first developer might be willing to ignore the modifications being done by the second developer because he needs the executable urgently in order to give a demonstration of the program to a funding agency. It might also be the case that some aspects of consistency might be relaxed for some development teams but not for others. Thus, there is a need to specify non-serializable interactions, which, although they violate the consistency constraints, should be allowed when the rule chains are executed in a concurrent fashion; the consistency constraints can be re-established later after the execution of the rule chains is completed.

We present a *control rule language* (CRL), which provides constructs for specifying legal (not necessarily serializable) concurrent interactions in multi-user RBDE. The specification is written in terms of *control rules*, which describe specific conflict situations and prescribe actions to resolve conflicts in such situations. The PCCP conflict resolution protocol provides the runtime environment for control rules in order to resolve conflicts detected by the NGL protocol. The actions prescribed by control rules include aborting, and thus undoing, one of the conflicting rule chains (the one that SCCP would not have aborted, for example), suspending one of the chains until the other is completed, or prematurely terminating, without undoing, one of the chains; a notification action is also provided to inform users of what the control rule prescribed. Prematurely terminating a rule chain that was initiated by a consistency predicate can leave objects in an inconsistent state with respect to the consistency constraints of the software process. RBDE ***marks*** these objects and stores information about the nature of the inconsistency (i.e., the value of an attribute is inaccurate) and how to re-establish consistency. This is similar to Balzer's notion of tolerating inconsistency [Balzer 91].

Although control rules can prescribe actions that enable RBDE to temporarily tolerate the inconsistency of some objects, they have no way of guaranteeing that the consistency of these objects will be re-established later. We introduce the notion of ***obligations***, which when satisfied re-establish the consistency of objects that were left inconsistent because of a concurrency conflict. In order to provide a context for obligations (i.e., a scope in which obligations much be satisfied), we must provide for a unit on top of rule chains. Such a unit, called a ***user session***, is provided by grouping sets of user commands, and the rule chains initiated by these commands, that together achieve a development task. Then, a control rule can prescribe adding an obligation to one or both of the user sessions that caused a conflict in order to guarantees the re-establishment of an object's consistency before the user sessions are completed. Our notion of obligations builds on previous notions [Minsky and Lockman 85, Perry 87].

To further specialize the concurrency control policy, we extend EMSL with another construct: ***development domains***, which can be used to define teams of developers. The purpose of domains is to allow the administrator to write control rules that apply to specific development teams or to members of the same team, for any team. These control rules use the fact that the two developers that caused the conflict are in the same team to prescribe an additional kind of action: merging the two conflicting rule chains into one chain. This action is sometimes desirable when two team members are closely cooperating, which causes them to access the same objects frequently. By merging their chains into one, their access to objects is treated as if it was requested by one developer, and thus both developers will be able to interleave their access to the same set of objects without introducing any conflicts. The merge action is similar to the join-transaction operation introduced by Pu *et al.* [Pu et al. 88]. The notion of development domains is similar to the group paradigm proposed for cooperative transaction management [El Abbadi and Toueg 89, Dowson and Nejmeh 89, Skarra and Zdonik 89]. The details of the CRL language and PCCP are explained in chapter 6.

The components of the concurrency control mechanism and the interactions among the different protocols are depicted in figure 1-4. The transaction manager and the lock manager form the conflict detection module. The Scheduler and control rules form the conflict resolution module.

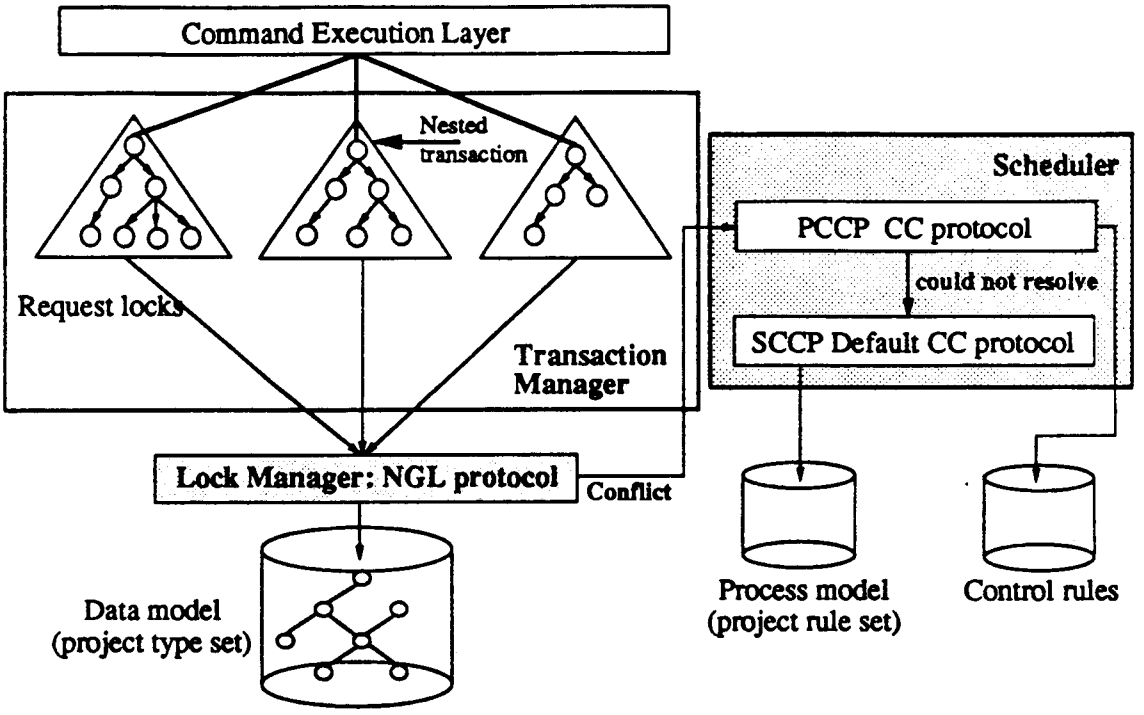

Figure 1-4: Components of the Concurrency Control Mechanism

1.6. Contributions

There are four main contributions of the research described in this dissertation:

1. The first contribution is the database model of RBDE and the decomposition of the concurrency control mechanism into a serializability-based conflict detection protocol and a semantics-based conflict resolution protocol. By separating the concurrency control problem into a conflict detection module and a conflict resolution module, we are able to minimize the overhead of monitoring concurrent activities. One of the previous cooperative transaction mechanisms that provides for programmability of the concurrency control policy [Skarra 91] requires monitoring all database activities and matching each activity to a step in a state machine. The state machine encodes required legal concurrent interactions. This monitoring and matching adds significant overhead that renders the idea impractical. We instead allow all serializable activities to proceed without much delay, as

will be explained in chapter 4. Due to the concept of modularity and given that a large-scale project usually involves development teams that do not interact much with each other, we expect most access to the project database to be serializable. Only in the relatively few cases of non-serializable interactions do we require some searching to determine which control rule to fire.

2. The second contribution is the characterization of the semantic information that the concurrency control mechanism in process-centered SDEs can use to provide a project-specific concurrency control policy. This semantic information can be extracted from the encoding of the development process. We have identified seven pieces of semantic information that can be extracted from the process model: (1) the distinction between maintaining consistency and providing automated assistance; (2) the distinction between interactive and non-interactive activities; (3) the time elapsed since the beginning of each on-going activity; (4) the identity of each on-going activity (e.g., compiling, editing, sending mail, etc.); (5) the human developer who initiated each activity; (6) the development task of which the activity is a part; and (7) the development team to which each developer belongs. We believe that these seven pieces of information can be made available in every process-centered SDE. This semantic information is the basis for supporting concurrency and cooperation in our architecture.

3. The third is the default SCCP protocol, and the separation between consistency predicates and automation predicates, which enable us to implement a semantics-based concurrency control policy. By providing language constructs in EMSL for distinguishing between consistency predicates and automation predicates, we enable the project administrator to define the consistency constraints of the software process explicitly. The SCCP protocol then guarantees the maintenance of these constraints. Thus, SCCP replaces the traditional correctness criterion of serializability with the semantic correctness criterion defined by the consistency constraints. SCCP also avoids aborting and rolling back transactions unnecessarily. In particular, SCCP does not requires cascaded rollbacks of transactions that encapsulate automation rule chains, as will be explained in chapter 5.

4. The fourth contribution is the programmable concurrency control protocol, PCCP, and the CRL language, which together provide a mechanism for implementing project-specific and flexible concurrency control policies in SDEs. The CRL language for writing control rules provides a mechanism for programming the concurrency control policy of a project rather than having a fixed policy built into the environment. This ability is important for two reasons. First, it encourages experimentation with several concurrency control policies in the same project in order to find the most appropriate one. Such experimentation might require some measurements, by which the relative advantages and disadvantages of different policies can be evaluated; we do not suggest any specific measurements but provide the underlying mechanisms needed for trying different policies. Second, the range of actions provided by control rules enables the implementation of both cooperative and traditional concurrency control policies. Some of these policies have never been implemented before. CRL provides a platform for coding these policies. We show how several policies can be specified in terms of control rules in chapter 6.

1.7. Organization of the Dissertation

The rest of the dissertation is organized as follows: We present the details of EMSL and RBDE in chapter 2. Although the work described in chapter 2 is not, strictly speaking, part of this thesis work (since it was done earlier in the context of the MARVEL project), it provides the platform on which we develop our concurrency control mechanism. In chapter 3, we construct a database model of command execution in RBDE and explain the concurrency control problem that results from allowing concurrent rule chains in RBDE. We also present a list of requirements that any satisfactory solution to this problem must meet; this list is based on requirements that other researchers have proposed in the literature [Bernstein and Goodman 81, Bancilhon et al. 85, Yeh et al. 89].

We describe our solution in four chapters. Chapter 4 describes the nested transaction model and the NGL locking protocol for detecting serializability conflicts between concurrent rule chains. In chapter 5, we extend EMSL with consistency predicates, and

introduce the SCCP protocol. We also explain how consistency predicates modify the chaining algorithms. Next, we present the details of CRL, and explain how PCCP uses control rules to resolve conflicts in chapter 6. Chapter 7 extends our mechanism with support for work units and teamwork . We explain the concepts of user sessions and development domains, and discuss how information about these concepts can be used in control rules to provide delayed conflict resolution in terms of obligations and team-oriented conflict resolution.

Finally, we conclude by summarizing the thesis work, describing its implementation, and discussing its contributions, its limitations, and suggestions for future work. In this final chapter we also evaluate how closely our thesis work meets the requirements we present in chapter 3, and compare our results to the results of two other theses that have addressed the concurrency control problem in cooperative environments.

Related work is discussed throughout the dissertation. In particular, we briefly overview some other process-centered SDEs in chapter 2. We explain the notions of transaction and serializability in chapter 3 and use them to formalize the concurrency control problem in RBDE. Granularity locking, which forms the basis for the conflict detection module, is explained in chapter 4. In chapter 5, we overview the concept of semantics-based concurrency control and describe some semantics-based mechanisms that are similar to the SCCP protocol. We discuss some cooperative transaction mechanisms and show how we can implement them in CRL in chapter 6. In chapter 7, we compare the notion of user sessions to sagas, and the concept of development domains to the group paradigm in group-oriented mechanisms. We also explain how our notion of obligations builds on previous notions of obligations.

Chapter 2

Multi-User Rule-Based Software Development Environment

In this chapter we construct an architecture for software development environments based on the MARVEL system developed at Columbia. The architecture is composed of a specification language, EMSL, and a software development environment kernel, RBDE. RBDE is process-centered in the sense that it can be tailored to support a specific development process by loading a rule-based specification of the process written in EMSL. EMSL also provides constructs for specifying the data model of a project in terms of a class hierarchy. Although many papers have been written about MARVEL, none of the papers gives a precise and detailed description of either the rule execution model or the MSL specification language, on which EMSL is based. These details are essential to understanding the concurrency control problem and our solution. This is the reason we decided to include this detailed chapter.

We first give some background information about the concept of process-centered SDEs. We then present the overall architecture, followed by a description of EMSL. We concentrate on the details that are relevant to the rule execution model and rule chaining. We first assume a single-user rule execution model, where only one rule chain is executed at a time. We then extend the architecture to support multiple concurrent rule chains. It is in this multi-user model that the concurrency control problem arises.

2.1. Background: Process-Centered SDEs

Large-scale software development involves managing large amounts of data in the form of source code, object code, documentation, test suites, etc. Traditionally, developers of such projects have managed this data either manually or by using special-purpose tools, such as `make` [Feldman 79] and `rcs` [Tichy 85], which manage the configurations and versions of the programs being developed. Releases of the finished project are typically stored in different directories manually.

The only common interface among software tools used in traditional development efforts is the file system, which stores project components in text or binary files regardless of their internal structure. This significantly limits the ability to manipulate these objects in ways that depend on their structure. It also causes inefficiencies in the storage of collections of objects, and leaves data, stored as a collection of related files, susceptible to corruption due to incompatible concurrent access by external tools.

SDEs aim to solve these problems by: (1) abstracting away from file systems and providing higher-level data management capabilities; (2) providing a platform for integrating sets of software tools; and (3) automating parts of the software development process by invoking tools automatically. The first objective can be achieved by utilizing database technology to store and manage the data generated and manipulated by SDEs. The second objective can be achieved by using information about the inputs, outputs and side effects of tools. The third objective of SDEs can only be achieved by using knowledge about the specific development process of a project in order to provide specialized automated assistance to the project's developers.

A development process is comprised of a set of activities, some of which require the invocation of software tools resident on the operating system. Different processes often differ from each other in three main respects: (1) the set of tools required to carry out the activities comprising the process, (2) the prescription of how to use software tools, and (3) the restrictions on when to use these tools to perform the activities. Because of these differences, there is no single specification that models all development processes adequately. This points to the need for tailorable SDEs that provide assistance in carrying out specific development processes rather than SDEs with a hard-wired process. The term *process-centered* is used to describe such SDEs.

Different process-centered environments represent the software development process using different formalisms. Arcadia uses an extension of Ada as a process programming language [Sutton 90]. HFSP employs a form of attribute grammars [Katayama 89]. MELMAC combines several perspectives on the process into a representation similar to Petri nets [Deiters and Gruhn 90]. In one class of process-centered SDEs, called *rule-based SDEs*, the software process is specified in terms of rules that resemble planning systems rules.

Several rule-based SDEs have been proposed and constructed. Each typically provides for one or two forms of enaction of the development process [Perry 89a, Katayama 90]. The CommonLisp Framework [CLF Project 88] supports both consistency maintenance and automation of the software development process through consistency and automation rules, respectively [Cohen 86, Cohen 89]. Darwin [Minsky and Rozenshtein 88] restricts what programmers can do by treating rules as constraints and automating the enforcement of these constraints; Darwin uses Prolog as a base language. Grapple [Huff 89] uses rules to suggest plans and do plan recognition in order to monitor a user-specified process model. Other rule-based SDE models include E-L [Cheatham 90], Workshop [Clemm 88], Oikos [Ambriola et al. 90], and ALF [Benali et al. 89].

The details of the process definition formalisms and process enaction mechanisms are essential for understanding and solving the concurrency control problem in process-centered SDEs. However, since the SDEs mentioned above implement a variety of process specification and enaction mechanisms, it is very difficult to develop a general statement of the concurrency control problem in all of these SDEs. It is also difficult to construct a concurrency control mechanism that is appropriate for all of them. Thus, we have taken the approach of constructing a specific SDE architecture and solving the problem in that architecture. The main ideas of our solution can be used to develop concurrency control mechanisms for any process-centered SDE.

2.2. The Architecture

We construct an SDE architecture based on three concepts: (1) object-oriented data modeling; (2) rule-based modeling of the software development process; and (3) high-level integration of commercial off-the-shelf tools. EMSL provides constructs for specifying the data model in terms of object-oriented class definitions. The set of classes that defines the data model is termed the ***project type set***. EMSL also provides constructs for defining the activities that comprise the development process in terms of rules. Each rule applies to specific sets of classes in the project type set. The complete set of rules is termed the ***project rule set***.

Tool integration is carried out by defining interfaces between RBDE and the tools,

which are called ***envelopes***. The set of envelopes defined for a project is called the *project tool set*. Envelopes are written in an extension of the Unix Shell language called SEL. SEL defines constructs to manipulate files in the database and invoke tools. The details of SEL and envelopes are irrelevant to concurrency control and thus they are not discussed here. For a description of envelopes and SEL, see [Gisi and Kaiser 91].

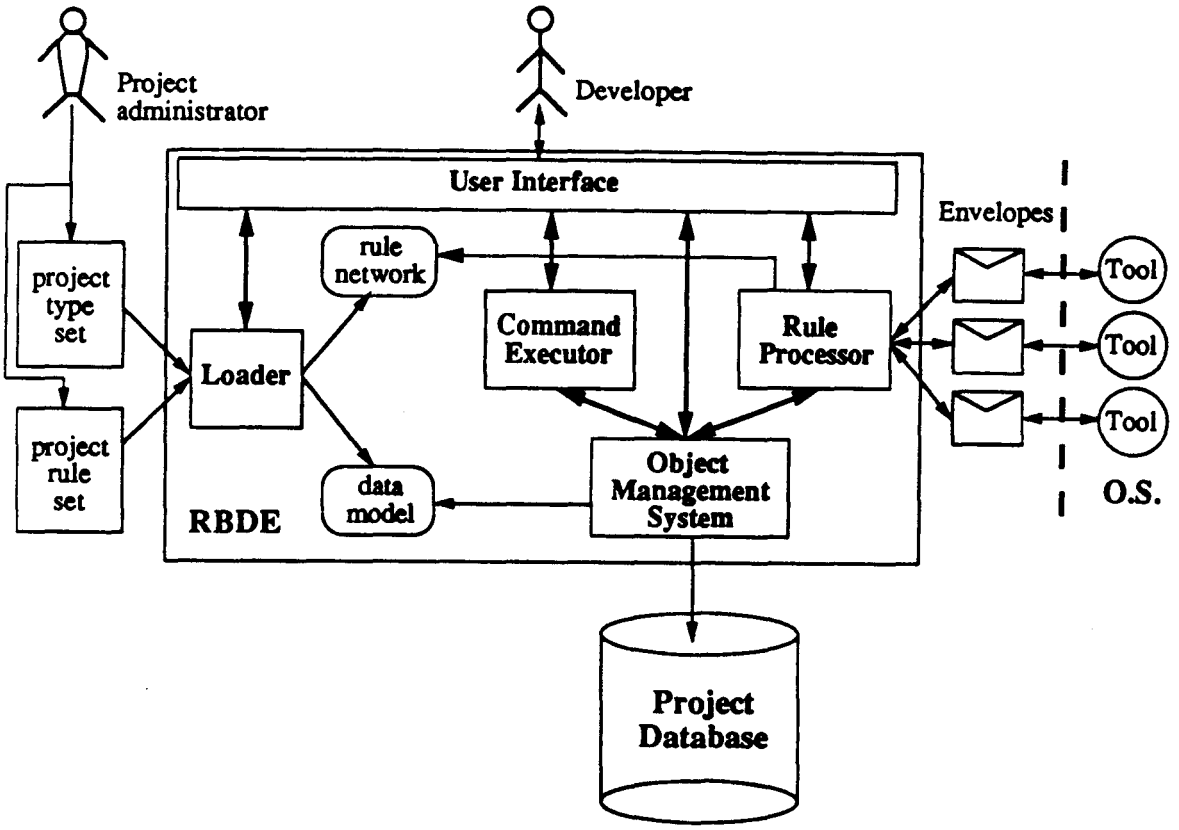

Figure 2-1: Overall System Architecture

The administrator of the project under development writes EMSL specifications of the project's data and process models, and loads these specifications into RBDE. Given this information, RBDE knows

- the structure and organization of the project's data,
- the relations that exist between different kinds of components,

- the activities that can be invoked on each kind of component, and
- the condition for invoking each activity and the effects of the activity on the project components it manipulates.

This knowledge is used by the various parts of RBDE to provide automated assistance. Figure 2-1 depicts the various components of the single-user RBDE architecture[6]. This architecture will be revised in subsequent chapters to add multi-user support and transaction management.

The Loader loads and analyzes the EMSL specifications of the project type set and the project rule set. If either of the two sets is inconsistent, or if the two sets are inconsistent with each other, as explained later, the two sets are rejected, with appropriate error messages. The administrator debugs the specifications and attempts to load them again. When RBDE succeeds in loading the two sets, the tailored RBDE environment presents the end users (the software developers) with commands corresponding to a selected subset of the rules in the project rule set, as will be clarified later. The rule set and the type set are project-specific. Consequently, the environments for different projects are likely to have different user commands, different behaviors, and different object management support.

All RBDE environments, however, share common built-in commands, which apply to all objects. These built-in commands must be directly requested by the user and cannot be performed automatically by RBDE. There are five built-in commands that are relevant to our discussion here: `add`, `delete`, `move`, `copy`, and `link`. `Add` creates an instance of one of the classes in the project type set and inserts the instance in the object hierarchy[7]; `delete` deletes an object and all of its subobjects; `move` moves an object

---

[6]Note that there is no direct correlation between the size of the project type set and the database; in other words, a small number of classes can be used to produce a huge number of instance. Similarly, the size of the project type set is not directly related to the size of the project rule set in the sense that the administration can write many rules for a small set of classes or vice versa.

[7]In some AI systems, there is an automatic classifier that inserts a new instance (or class) in the class hierarchy without having the user specifically indicate to which class the instance belongs. Such an automatic classifier might be feasible in RBDE given the object-oriented framework; a discussion of this topic, however, is outside the scope of this dissertation.

from one location to another; `copy` duplicates an object; finally, `link` creates a relationship between two objects. The details of built-in commands are irrelevant here except as far as their access to the database is concerned. Note that `add` (or `delete`) is the only mechanism provided for creating (or deleting) new objects. The command executor (CE) is responsible for executing built-in commands.

The object management system (OMS) uses the class definitions in the project type set to create, store and manipulate objects representing the project's data. The OMS is also responsible for storing the objects on the file system to provide persistence. A third component, the rule processor (RP), uses the specification of the project rule set and tool set to provide automated assistance. The RP is responsible for selecting (matching) rules corresponding to user commands, executing these rules, and initiating rule chains whenever possible.

2.3. The EMSL Specification Language

EMSL, which stands for Extended Marvel Strategy Language for historical reasons, is an object-oriented language that provides constructs for defining object classes and rules as methods of these classes. We will describe only the constructs of EMSL that are relevant to the rule execution model.

2.3.1. Class Definitions

The data model is specified in terms of classes, each of which defines a set of typed ***attributes*** that can be inherited from multiple superclasses; inheritance conflicts are resolved according to a pre-defined strategy whose details are irrelevant here. The class hierarchy that results is similar to an IS_A hierarchy in semantic networks. The actual components of the project are then created as instances of these classes. Figure 2-2 depicts example class definitions that a project administrator might write in EMSL to describe the organization and structure of the example project presented in chapter 1. Each class definition starts with the name of the class followed by the list of its superclasses. Then follows a list of attribute definitions, each of which consists of a name followed by the type of attribute; the type may optionally be followed by an initialization

```
PROGRAM :: superclass ENTITY;
    modules : set_of MODULE;
    libraries : set_of LIB;
    includes : INCLUDE;
    status : (Built, NotBuilt, Error, None) = None;
end

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

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

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

CFILE :: superclass FILE;
    contents : text = ".c";
    error_msg : text = ".err";
    includes : set_of link HFILE;
    status : (NotCompiled, Compiled, NotArchived,
              Archived, Error, Initial) = Initial;
    object_code : binary = ".o";
    test_status : (Tested, NotTested, Failed) = NotTested;
    object_timestamp : time;
    archive_timestamp : time;
    libs : set_of link LIB;
end

INCLUDE :: superclass ENTITY;
    hfiles : set_of HFILE;
    archive_status : (Archived,NotArchived,INotArchived)
                          = NotArchived;
end

LIB :: superclass ENTITY;
    afile : binary = ".a";
    archive_status : (Archived,NotArchived,INotArchived)
                          = NotArchived;
    timestamp : time;
end

MODULE :: superclass RESERVABLE;
    archive_status : (Archived,NotArchived,None)
                          = NotArchived;
    test_status : (Tested, NotTested, Failed) = NotTested;
    cfiles : set_of CFILE;
    modules : set_of MODULE;
    lib : link LIB;
end
```

Figure 2-2: Class Definitions in EMSL

value. EMSL provides four kinds of built-in attribute types: *status*, *data*, *structural*, and *link* attributes[8].

Status attributes are used to store information about the state of an object. There are seven types of status attributes: `integer`, `string`, `boolean`, `real`, `user`, `time`, and `enumerated`. The first four are self-explanatory; `user` represents a user id and `time` stores a time stamp. An enumerated type is a list of possible values enclosed in "( )". For example, in figure 2-2, every instance of `FILE` has one status attribute, `timestamp`, which can be used to store the modification timestamp of a `FILE` object. Every instance of `RESERVABLE` has three status attributes: `locker`, whose value is a user id, `purpose`, which is a string, and `reservation_status`, which can take exactly one of the values listed between parentheses (i.e., `CheckedOut`, `Available`, or `None`); this attribute is initialized to "None" whenever a new instance of `RESERVABLE` (or any of its subclasses) is created. The `locker` attribute might be used to store the operating system-assigned id of the user who reserved the object. The `purpose` attribute might be used to store a sentence describing the reason for reserving the object, and the `reservation_status` attribute might reflect whether or not the object is reserved.

Data attributes store the pathnames of files containing the data of an object. There are two types: `text` and `binary`, which refer to text and binary file types, respectively. The initializations of attributes of this kind give the file name suffixes used in the underlying file system. For example, an instance of `CFILE` (i.e., a C source object) will have its data stored in an attribute called `contents`, which is a text file attribute inherited from `FILE` but modified so that the file will have the suffix ".c". Every instance of `CFILE` will also have an attribute called `object_code`, a binary file with the suffix ".o", in which the derived object code of the `contents` attribute will be stored.

---

[8]Status, data and structural attributes were called *small*, *medium* and *large* attributes, respectively in previous papers on MARVEL.

2.3.2. Composite Objects

Structural attributes define hierarchical relations (i.e., IS_PART_OF relations) between objects, and form the basis for composite objects. Structural attributes are defined in terms of the `set_of` construct, which is an aggregate of arbitrarily many instances of the same class. The class definitions of figure 2-2 define structural attributes for `PROGRAM`, `MODULE` and `INCLUDE`. For example, each instance of `INCLUDE` is defined to contain a set of instances of `HFILE` via the `hfiles` attribute. Using structural attributes, a whole object hierarchy (tree), which reflects the desired organization of data in the specific project, can be created.

Strict object hierarchies cannot represent all kinds of relationships between objects in a software project. For example, although there might not be a structural relationship between a C source file and a header (".h") file in a typical C programming project (i.e., one does not contain the other), we might want to represent the fact that the C file includes the header file. EMSL provides `link` attributes to model this kind of relationship between objects. EMSL supports typed links between two objects (i.e., binary relationships between two instances of specific classes); each link is syntactically unidirectional (i.e., it is defined in only one object), but semantically, a link is bidirectional. For example, `includes` is a set of links from a `CFILE` object to `HFILE` objects. This attribute can be used to specify which header files are included in a C source file. The link might also be used to know which `CFILE` objects include a specific `HFILE` object. This kind of information might be needed, as will be discussed shortly, for writing a rule specifying that when a header file is modified, RBDE should outdate all C files that include it.

Figure 2-3 shows an object hierarchy and several links obtained by instantiating the classes shown in figure 2-2. The hierarchy represents the project organization depicted in figure 1-1, where each module contains a set of C source files and all the header files are stored under one directory. Each `MODULE` object has a `link` to a `LIB` object, which has an `afile` attribute where the object code of the C source file contained in the module should be archived. Not all links are shown for simplicity.

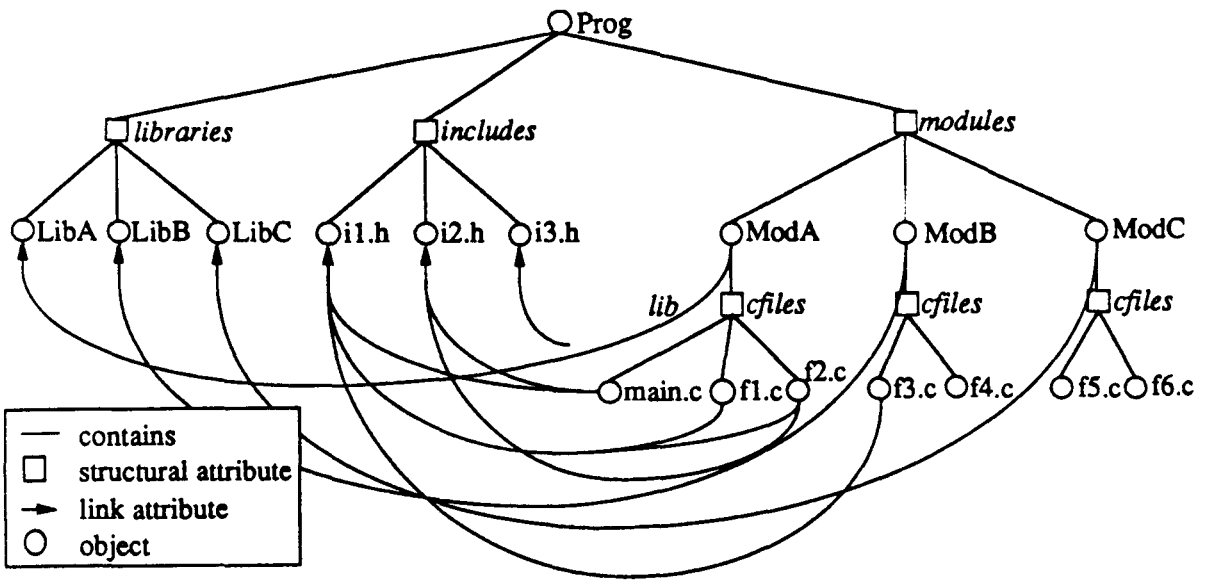

Figure 2-3: Composite Object Hierarchy and Links

Given a data model written using the constructs described thus far, the project administrator can now write a specification of the project's software development process. For example, given the hierarchy in figure 2-3, the project administrator might want to specify that a C source file needs to be compiled if either its contents were edited or if one of the header files it includes has been edited after the last compilation of the C file.

To prescribe this behavior to RBDE, the administrator must specify the "edit" activity for both `CFILE`s and `HFILE`s and the "compile" activity for `CFILE`s. RBDE can then use this specification to compile a `CFILE` automatically without the user having to explicitly request the compilation. Similarly, the administrator can specify all the activities that comprise the development process of a project; RBDE uses the specifications to provide project-specific assistance by automatically invoking software tools.

2.3.3. Definition of Rules

The software development process of a project is modeled in terms of two kinds of rules: ***activation rules*** and ***inference rules***. An activation rule controls the initiation of a software development activity, typically the invocation of a software tool resident on the operating system. It specifies the condition under which it is appropriate to invoke the tool and the possible effects of the tool on the values of the attributes of objects that are passed as arguments to the tool. Since a tool might have several possible outcomes, activation rules typically have a set of multiple mutually exclusive effects, each of which maps one of the possible results of the tool to changes in the values of status attributes. For example, a compiler might succeed in producing object code, or fail and produce error messages — it is not generally possible to determine which without invoking the compiler. The activation rule encapsulating the compilation activity must specify at least two sets of effects, one to be asserted in case of success and the other in case of failure.

In contrast, inference rules are not associated with activities and each has a single effect, which is a logical expression that is a consequence of the condition of the rule. Inference rules define relations among attributes of the same or different objects. They derive new values for objects' attributes based on the current values of the same or other attributes. Inference rules thus define implications, which are used later on to define the consistency model of a particular project; defining the consistency model of a project will be explained in chapter 5.

All activation rules and some inference rules form the user command menu of the tailored RBDE environment. The administrator might define some inference rules to be "hidden" in the sense that they are used solely by RBDE during chaining (explained later). Thus, each user command, other than built-in commands, corresponds to a set of rules; selecting a command causes one of the corresponding rules to be fired. The selection of the appropriate rule depends on the arguments passed to the command.

In addition to inference and activation rules, the administrator can write rules that specialize any one of the built-in commands provided by RBDE for particular classes of

objects. These rules basically add conditions for invoking the built-in command for a particular class in the project type set, and specify the effects of executing the built-in command on the values of the objects' attributes. For the purpose of our discussion in this chapter, these rules are treated like activation rules in terms of rule selection and assertion of effects. Unlike activation rules, however, the activity part is implicit in the name of the rule, which is the same name as a built-in command. For example, the administrator might write a rule that specifies that a user can add a CFILE object to a module only if no other CFILE in the same module has the same name. If a user requests to add an instance of CFILE to a MODULE, RBDE automatically fires this rule, and will only allow the add built-in command to be executed if the condition of the rule is satisfied.

```
     # Rule name and parameter list
1.   rulename [?param1 : CLASS1; ?param2 : CLASS2; ... ]:

     # Binding part of condition.
2.   (bind (?var1 to_all CLASS3 suchthat characteristic exp.)
3.         (?var2 to_all CLASS4 suchthat characteristic exp.)
           ( ... ))
4.   :

     # Property list part of condition.
5.   (property list)

     # Activity
6.   { <Envelope> argument1 ... argumenti;
                  output: argumenti+1 ... argumentn  }

     # Effects
7.   (effect1);
8.   (effect2);
     (...);
```

Figure 2-4: A Rule Template in EMSL

Figure 2-4 shows a template for EMSL rules. Each rule has a name followed by a list of parameters enclosed in square brackets. Rule names can be overloaded: several rules with the same name can apply to different sets of classes. Each parameter has a name and a type, one of the classes defined in the project type set. All variables used in the body of a rule, including parameters, begin with a "?" to distinguish them from iden-

tifiers. Following the parameter list is the condition of the rule, which is composed of a set of bindings (lines 2-3) followed by a property list (line 5). The activity invocation (line 6) is enclosed in curly braces "{...}". The outputs of the activity are specified after the keyword `output`; all the arguments are considered inputs to the activity. Following is the set of effects (lines 7-8), each of which is a conjunction (and) of assignment predicates that assign values to named attributes of the parameter objects. All lines beginning with a "#" are comment lines.

The binding part can be empty, which means that only the parameters are used as variables in the body of the rule. The condition can be empty in the case of an activation rule, which means that the activity of the rule is always performed when the user requests the command corresponding to the rule. The rule's activity is empty in the case of an inference rule. The effects of a rule are empty only in two cases. The first is in the case of an activation rule that invokes a tool that does not change any of the contents of objects in the database (i.e., a tool like `cat` that reads data attributes of objects and outputs information directly to the user). The effects are also empty if a tool changes the data attributes of objects but these changes are not mapped explicitly to changes in the values of status attributes of the same objects. Such a case is dangerous because RBDE is left unaware (RBDE's chaining mechanism considers only status attributes) of the changes that a tool made to the contents (data attributes) of objects.

Figure 2-5 shows an actual rule that prescribes the compile activity discussed earlier. This rule is taken (and adapted to EMSL syntax) from a MARVEL environment for C programming. The rule specifies that in order to compile an instance of `CFILE`, the following condition must be satisfied: either the `CFILE`, or an `HFILE` that is contained in one of the `INCLUDE` objects linked to the `CFILE`, has been edited after the last compilation. If the condition is satisfied, the "compile" envelope is invoked and several of the attributes of the `CFILE` and the `HFILE`s (if they exist) are passed to it. The attributes that will be written by the tool are specified after the keyword `output`; the distinction between arguments that are only read and those that are written by the tool will be used in chapter 4 by the transaction manager.

If the compilation succeeds, then RBDE must change the values of the `status` and

```
1. compile [?f:CFILE]:

   # If C source file has been edited but not yet compiled,
   # or if any of the header files it is linked to has been
   # changed after the last compilation of the CFILE,
   # then RBDE can compile it.

2. (bind (?h to_all HFILE suchthat (linkto [?f.includes ?h])))
3. :
4.
5. (exists ?h):
6. (or (?f.status = NotCompiled)
7.     (?h.timestamp > ?f.object_timestamp))

8. { compile ?f.contents ?h.contents "-g";
             output: ?f.object_code ?f.error_msg }

9. (and (?f.status = Compiled)
10.     (?f.object_timestamp = CurrentTime));
11.(?f.status = Error);
```

Figure 2-5: Example EMSL Rule

`object_timestamp` attributes of the `CFILE` to reflect that. If the compilation failed, then the `status` attribute is assigned the value "Error". In the rest of this section, we explain further details of the rule sublanguage of EMSL and the rule execution model. We use the "compile" rule as an example throughout the rest of the chapter.

2.3.3.1. Rule Parameters

After RBDE matches a user's command to a rule, the RP (rule processor) must select one of possibly many rules that correspond to the command (because of overloading of rule names). Rule selection is based on the fact that rules are parameterized to take as arguments one or more objects, which may be instances of one or more classes. Rules are thus similar to multi-methods in object-oriented programming [Bobrow 86]. Selecting a rule involves binding an actual object to each parameter of the rule and then firing the rule. For example, if a user requests from RBDE to compile an object, `main.c`, which is an instance of class `CFILE`, RBDE will select the compile rule shown in figure 2-5 and bind the `?f` parameter to `main.c`. Note that there might be several rules whose name is "compile" but which apply to different classes of objects. Further details of rule selection are not relevant to our discussion here; the reader is referred to [Marvel 91, Barghouti and Kaiser 90].

When a rule has been selected for execution, the first thing RBDE does is evaluate the rule's condition. EMSL provides a subset of first-order logic (FOL) [Chang and Lee 73] for writing the conditions of rules. In order to facilitate the evaluation of a condition, we diverge slightly from the notation provided by first-order predicate logic.

2.3.3.2. Rule Conditions as Logical Expressions

In EMSL, rule conditions are essentially read-only queries on the objects in the database. As shown in figure 2-4, a condition has two parts: a binding part, and a property list. The separation between the binding part and the property list is made to distinguish between binding a variable to a set of objects and testing if some predicates are true for these objects. This is essential for the chaining model, which will be explained later in this chapter.

```
(bind (?v1 to_all CLASS1 suchthat (and (or (func1)
                                           (func2)
                                       (pred1))))
      (?v2 to_all CLASS2 suchthat (fun3))))
```

As shown above, the binding part consists of a list of binding expressions. Each binding expression consists of a variable, and a *characteristic expression.* The characteristic expression, if not empty, characterizes the set of objects that should be bound to the variable, which is specified to be of a particular type (class). An empty characteristic expression means that the variable is bound to all instances of the specified class. The characteristic expression is a combination of conjunctions (AND) and disjunctions (OR) of either *structural functions* or *simple comparison predicates.*

Structural functions return a set of objects as their value. EMSL defines three structural functions: `member`, `ancestor` and `linkto`. These functions are called "structural" because their values are obtained by traversing the structure of the database through set and link attributes of objects. Simple comparison predicates are of the form `(?v.att <op> value)`, where `?v` is the variable of the binding expression, `att` is one of the status attributes of the specified class of `?v`, `<op>` is one of six comparison operators (described shortly), and `value` is a constant value of the same type as `att`. The evaluation of the binding part of the condition of a rule is described in detail in section 2.4.1.1.

The second part of the condition, called the *property list*, is a formula of the following form:

$(Q_1x_1)...(Q_nx_n)(M)$,

where every (Q_ix_i), $i = 1,...,n$ is either $(\forall x_i)$ or $(\exists x_i)$, and M is a formula containing no quantifiers. $(Q_1x_1)...(Q_nx_n)$ is called the *prefix* and M is called the *matrix*. The prefix is thus a list of variables that are quantified either *universally* or *existentially*. The matrix is a complex expression of predicates; we call the predicates in the property list of a condition *property predicates* to distinguish them from assignment predicates in the effects of a rule and the simple comparison predicates in the binding part.

Each quantified variable in the prefix of the property list must have already been bound in the binding part to a set of objects before evaluating the property list (i.e., the property list is a *safe* formula). The quantifiers are used strictly for the purpose of evaluating the truth value of the property list and not to bind variables. More specifically, the universal and existential quantifiers in front of variables in the prefix of the property list are used during the evaluation of predicates in the matrix, as will be explained in section 2.4.1.2.

Six comparison predicates can be used in the matrix of the formula: ''='' (equal), ''>'' (greater than), ''<'' (less than), ''!='' (not equal), ''<='' (less than or equal), and ''>='' (greater than or equal). The ''='' and ''!='' predicates are overloaded for all seven types of status attributes (i.e., integer, string, real, boolean, time, user, and enumerated). The others are overloaded for all types of status attributes except string and enumerated. There are no facilities for defining additional predicates.

All of these predicates are two-place; their truth value can be directly evaluated by RBDE. The predicates can either be simple comparison predicates, as described above, or they can be of the form `(?v1.att1 <op> ?v2.att2)`, where both `?v1` and `?v2` are quantified variables.

```
1. compile [?f:CFILE]:

2. (bind (?h to_all HFILE suchthat (linkto [?f.includes ?h])))
3. :
4.
5. (exists ?h):
6. (or (?f.status = NotCompiled)
7.     (?h.timestamp > ?f.object_timestamp))
```

An example of a property list is shown in lines 5-7 above (this example is the same as the `compile` rule shown in figure 2-5; we repeat it here for clarity). Both variables in the formula would have been bound before the evaluation of the property list: `?f` when selecting the rule since it is the parameter, and `?h` when evaluating the binding part. The property list specifies that either the value of the `status` attribute of the object bound to `?f` is `equal` to "NotCompiled", or the `timestamp` attribute of at least one of the objects bound to `?h` is greater than (i.e., more recent) than the `object_timestamp` attribute of the object bound to `?f`. The formula of the property list specifies the condition that must be satisfied before the activity of the rule can be executed.

2.3.3.3. The Activity

The activity part of an activation rule specifies which external tool to invoke, what arguments to pass to the tool, and which of these arguments will be changed by the tool. The activity specification consists of an envelope name and a set of arguments to the envelope. The envelope is responsible for passing the arguments in an appropriate form to the tool. Activities are of the form specified by line 6 in figure 2-4. The argument list of an activity can consist of the attributes of objects that are either parameters to the rule or bound to variables. Literals are also allowed in the arguments.

For example, consider the activity on line 8 of the `compile` rule in figure 2-5. This activity specifies that the "compile" envelope should be invoked and that five arguments should be passed to it. The first argument is the value of the `contents` attribute of the object bound to the parameter (`?f`), which is the pathname of the C source text file. The second argument is the set of the `contents` attributes of all the objects bound to `?h`; the value of each of these attributes is the pathname of a text file containing the header file represented by the `HFILE` object. The third argument is a literal,

"-g", which the envelope might pass as a flag to the compiler tool. The data attributes passed as the first two arguments, as well that literal "-g", are only read by the tool. The last two arguments, in contrast, are both written (i.e., changed) by the tool, and thus they are preceded by the keyword "output". These two arguments are the value of the data attribute, `?f.object_code`, which is the pathname of the binary file where the object code will go, and the value of `?f.error_msg`, which is the pathname of a text file where error messages from the compiler should go.

Activities are treated as black boxes that operate outside the direct knowledge of RBDE, and only communicate via arguments and returned results. The envelope might change the contents of some of the data attributes passed to it. For example, the "compile" envelope will change the contents of the `object_code` attribute passed to it (to put the new object code in it). These kinds of changes should normally be mapped to changes to the values of the status attributes of the objects, and included as part of the effects of the rule. If they are not, RBDE cannot use them as basis for chaining, as will become clear in section 2.4.

2.3.3.4. Rule Effects

Each effect is a conjunction of assignment statements, hereafter called *assignment predicates*, that assign values to the attributes of objects bound to the parameters of the rule. The general form of an assignment predicate is as follows: `(?v1.att <op> value)`, where `?v1` is one of the bound variables, `<op>` is an assignment operator, and `value` is either a literal, an object or an attribute. Assignment operators apply to either status attributes or link attributes. For status attributes, there is one assignment operator, "=". For link attributes, there are two: `linkto` to create a link between two objects and `unlink` to remove an existing link; `unlink` does not require a `value`.

```
1.  compile [?f:CFILE]:

    . . .

9.  (and (?f.status = Compiled)
10.      (?f.object_timestamp = CurrentTime));
11. (?f.status = Error);
```

A rule might have multiple effects corresponding to the different possible results of a tool invocation. For example, the `compile` rule has two effects that are shown above. The first effect is a conjunction (and) of two assignment predicates: the first assigns the value "Compiled" to the `status` attribute of the object bound to the parameter of the rule, `?f`; the second assigns the current time (system time) to the `object_timestamp` attribute of the object bound to `?f`. The second effect simply assigns the value "Error" to the `status` attribute of the object bound to `?f`. Which effect will be asserted depends on the result returned by the envelope specified in the activity part.

The EMSL language we described above is strongly typed. 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, say the project rule set includes a rule `r` that applies to instances of class `C`, and its condition checks if the value of an attribute `a` is greater than the integer `5`. Then `r` requires that the definition of `C` contains an attribute called `a` of type integer. This checking also guarantees that the rule set is self-consistent in the sense that no two rules assume different types for the same attribute of the same class.

2.4. Rule Execution Model

In the previous few sections, we presented the details of the EMSL language and showed how the project administrator can use the constructs of the language to define the data and process models, and integrate development tools. We now explain the rule execution model in RBDE. We first explain how one rule is fired and then present the rule chaining algorithms, which are the basis for automated assistance in RBDE.

After matching the user command to a rule, the first thing that RBDE does is to select a rule, which involves binding the parameters of the rule to actual objects. If the rule corresponds to a user command, the parameters are bound to the objects selected by the user. If the rule is fired during chaining, then binding of the rule's parameters is a more complicated process. In this case, we apply an algorithm that analyzes the logical expressions in the conditions and effects of rules to bind the parameters to actual objects of the same type (or a subtype) as the parameter. This approach has been recently im-

plemented in MARVEL to replace a more limited heuristic approach. The details of the heuristic approach to binding parameters as well as a discussion of the logical approach can be found in [Heineman et al. 91]. In the rest of this chapter, we assume that when a rule is fired, all of its parameters would have already been bound successfully to actual objects.

2.4.1. Evaluation of the Rule's Condition

After the parameters of a rule have been bound, RBDE evaluates the condition of the rule. The evaluation is divided into two steps: binding all the variables in the binding part and then evaluating the truth value of the property list.

2.4.1.1. Evaluating the Binding Part

The binding part is a list of binding expressions, each of which binds one variable. The RP evaluates the binding expressions of the binding part in turn. A variable that is bound in a binding expression can be used as a constant in a following binding expression. Binding one variable involves evaluating the characteristic expression of the binding expression and binding the variable to all the objects returned by the evaluation. The characteristic expression can be represented as an AND/OR tree. Every non-terminal node is either an AND node or an OR node. All the terminal nodes are either structural functions or simple comparison predicates.

To bind a variable, the AND/OR tree is evaluated. Evaluating a node in the AND/OR tree of the binding expression means that the node is assigned a set of objects (possibly empty). An AND node is evaluated by first evaluating all of its children and then taking the intersection of the object sets assigned to the children. Evaluating an OR node involves taking the union of the object sets assigned to the child nodes. Finally, the structural function at each terminal node is evaluated directly, assigning the node to a set of objects (possibly empty).

As mentioned earlier, EMSL supports only three structural functions: `member`, `ancestor`, and `linkto`. Each is a two-place function. `Member` and `linkto` are of the form: `(f [?v1.att, ?v2])`, where `f` stands for either `member` or `linkto`;

one of ?v1 or ?v2 is an unbound variable while the other is a variable that has already been bound in either a previous binding expression or as a parameter; att is an attribute of type set_of in the case of member and of type link in the case of linkto.

If ?v1 is the bound variable, then member returns all the objects that are members of the att attribute of any of the objects bound to ?v1. Alternatively, if ?v2 is the bound variable, then member returns all the objects that include any of the objects bound to ?v2 in their att set attribute. The processing of linkto is similar to that of member except that link attributes are used instead of set_of attributes.

```
rule [?m : MODULE]:

 (bind (?c to_all CFILE suchthat (member [?m.cfiles ?c]))
       (?h to_all HFILE suchthat (linkto [?c.includes ?h])))
```

To illustrate, consider the two binding expressions above. In the first expression, the variable ?m corresponds to the parameter of the rule. This variable will be bound to an actual object when the rule is fired. Say that it was bound to ModB in figure 2-3 on page 30. Then, when evaluating the expression, the variable ?c will be bound to all members of the cfiles attribute of ModB. Thus, ?c will be bound to two objects: f3.c and f4.c. When evaluating the second expression, the variable ?h will be bound to all instances of class HFILE that are linked to either f3.c or f4.c through the includes attribute. In other words, ?h will be bound to i1.h and i3.h.

The processing of ancestor is slightly different than that of member and linkto. Ancestor is of the form: (ancestor [?v1 ?v2]). If ?v1 is the bound variable, then ancestor returns the descendants of all the objects bound to ?v1 that are of the class specified for ?v2, or one of its subclasses. Alternatively, if ?v2 is the bound variable, then ancestor returns all the ancestors of the objects bound to ?v2 that are of the same class (or a subclass) as that specified for ?v1. An object is not considered an ancestor or descendant of itself.

In terms of the database, the binding expressions are read-only queries. From the discussion above, it should be clear that the binding of variables might entail reading the values of the attributes of more than the objects bound to the parameters of the rule. For

example, processing the binding part of the `compile` rule leads to reading the `includes` attribute of the `CFILE` object bound to the parameter. In fact, the structural functions in the binding part might access every object in the object hierarchy.

By evaluating the binding part of a rule, the RP collects all the objects that will be accessed during the execution of the rule. All of these objects will be read by the rule; some of them will also be written by the activity or in the effects. This fact is used by the transaction manager, as will be explained in chapter 3. Note that the ability to predict the set of objects that will be read and written by a rule is based on the assumption that activities (tools) are treated as "black boxes", whose inputs and possible outputs are known before they start executing. This assumption is not true for interactive tools that access new objects incrementally. We discuss some ideas as to how to relax this assumption in chapter 8.

2.4.1.2. Evaluating the Property List

After binding all the variables of a condition, RBDE evaluates the property list of the condition. As explained earlier, the prefix of the property list is a list of existential and universal quantifiers over the variables bound in the binding part. The quantifiers are used during the evaluation of the predicates in the matrix of the property list, as will be explained shortly. Existential quantifiers serve an additional purpose. It is possible to existentially quantify a variable without using that variable in the matrix of the property list. In this case, the existential quantifier is used to check if the set of objects bound to a specific variable is non-empty. If it is empty, the condition is said to be UNSATISFIABLE because there is no way to make it satisfied except by creating an object that can be bound to the variable. But since only the user can create objects, RBDE will not be able to satisfy the condition automatically and thus it informs the user that the execution of the rule could not be continued.

For example, consider the condition of the `build` rule shown in figure 2-6. Say that this rule was fired and that its parameter, `?prog`, was bound to an object, `Prog`. The condition specifies that the activity of the rule can be invoked only if the `status` attribute of `Prog` is equal to "Ready" and if `Prog` contains at least one `CFILE` object as a descendant. There are two reasons why this condition might not be satisfied. The first

```
build [?prog: PROGRAM]:
   (bind (?f to_all CFILE suchthat (ancestor (?prog ?f))))
   :
   (exists ?f):
      (?prog.status = Ready)

   { ... }

   ...;
```

Figure 2-6: Example of Existential Quantifiers

is if the value of the `status` attribute of `Prog` is not equal to "Ready". In this case, the condition evaluates to FALSE, meaning that the condition is not satisfied but that RBDE might possibly be able to make it satisfied automatically through chaining (explained in section 2.5). The second possibility is that `?f` was bound to an empty set of objects, which means that `Prog` does not contain any `CFILE` objects. In this case, the condition is said to be UNSATISFIABLE because there is no way that RBDE can make it satisfied automatically since only the user can create an object.

If the condition is not UNSATISFIABLE, RBDE goes on to evaluate the matrix of the property list. The matrix is a complex logical expression, which again can be represented by an AND/OR tree. The evaluation of this AND/OR tree is similar to the evaluation of the AND/OR tree of the binding expressions but differs in some respects. The first difference is that the terminal nodes are predicates and not structural functions. The evaluation of these predicates is quite different from that of structural functions. The second difference is that the value of each node is either TRUE or FALSE rather than a set of objects.

As explained in section 2.3.3.2, EMSL supports only six simple predicates: "=", ">", "<", "!=", "<=", and ">=". Evaluating a predicate requires that each variable in the predicate be replaced by an actual object. A predicate can involve at most two variables, both of which would have been bound either as parameters or in the binding part. For example, the predicate `(?p.status = ?m.status)` involves the two variables `?p` and `?m`; and the predicate `(?p.status != Compiled)` involves just the variable `?p`. Each variable is bound to a set of objects as described above.

Replacing the variables of a predicate with actual objects bound to these variables is termed *instantiation* of the predicate. If the predicate involves only one variable, `?v1`, which has been bound to *n* objects, then in order to evaluate the truth value of the predicate, it must be instantiated *n* times, once for each object bound to the variable. The value assigned to the predicate depends on the evaluations of these instantiations and on whether the variable is quantified universally or existentially in the prefix of the property list.

Input: A predicate, *P*, of the form (?v1.att1 <op> value)
Output: True or False.

```
/* ?v1 is existentially quantified */

For each object, obj, bound to ?v1 Do
   Begin
     instantiate P, replacing ?v1 with obj1;
     evaluate the instantiation of P;
     If evaluation returns TRUE Then
        Begin
          P := TRUE;
          stop the evaluation for this node and return;
        End;
   End;

P := FALSE;
```

Figure 2-7: Evaluating a Predicate with One Variable

If the variable in the predicate is quantified existentially, then the predicate evaluates to TRUE if any of its instantiations evaluates to TRUE. Otherwise, the predicate evaluates to FALSE. The exact procedure is in figure 2-7. If the variable in the predicate is quantified universally, then the predicate evaluates to TRUE only if all instantiations of it evaluate to TRUE; otherwise it is assigned the value FALSE. Evaluating a predicate with a universally quantified variable is very similar to the evaluation procedure shown in figure 2-7, except that the evaluation stops as soon as an instantiation of the predicate evaluates to FALSE.

A predicate involving two variables is of the form `(?v1.att op ?v2.att)`. In this case, there are four possibilities: each variable could have been quantified either existentially or universally. If both `?v1` and `?v2` are quantified universally, then the

```
Input:  A predicate, P, of the form (?v1.att1 <op> ?v2.att2)
Output: True or False.

 /* Both ?v1 and ?v2 are universally quantified. */

 For each object, obj1, bound to ?v1 Do
   Begin
     For each object, obj2, bound to ?v2 Do
       Begin
         instantiate P (?v1 := obj1 and ?v2 := obj2);
         evaluate the instantiation of P;
         If evaluation returns FALSE Then
           Begin
             P := FALSE;
             stop evaluation for this node and return FALSE;
           End;
       End;
   End;

 P := TRUE;
```

Figure 2-8: Evaluating a Predicate with Two Variables

node representing the predicate is assigned the value TRUE only if the instantiations of the predicate for each combination of the objects bound to ?v1 and ?v2 evaluate to TRUE. The evaluation stops as soon as a FALSE value is returned by any of the instantiations. The evaluation of this case is shown in figure 2-8.

The evaluation of a predicate in which both ?v1 and ?v2 are existentially quantified is similar. All we have to do is find one object bound to ?v1 and one bound to ?v2 such that the predicate evaluates to TRUE. The evaluation stops as soon as such a combination is found. If after going through all the objects bound to ?v1 and all those bound to ?v2, none of the evaluations returned TRUE, then the terminal node representing the predicate is assigned the value FALSE.

The evaluation of a predicate in which one of ?v1 is quantified universally whereas the other is quantified existentially is slightly more complicated. Consider the case where ?v1 is quantified universally and ?v2 is quantified existentially. The terminal node representing a predicate of this kind will be assigned the value TRUE if and only if for each object bound to ?v1, there is an object bound to ?v2, such that the predicate evaluates to TRUE. The evaluation stops as soon as we find an object bound to ?v1 for which we could not find any object bound to ?v2 that will make the predicate evaluate

to TRUE. The evaluation of the case in which ?v1 is quantified existentially and ?v2 is quantified universally is similar.

```
1.  compile [?f:CFILE]:

2.  (bind (?h to_all HFILE suchthat (linkto [?f.includes ?h]))
3.  :
4.
5.  (exists ?h):
6.  (or (?f.status = NotCompiled)
7.      (?h.timestamp > ?f.object_timestamp))
```

Consider again the condition of the compile rule shown above. The variables ?f and ?h will have been bound before evaluating the predicates in the property list as follows: ?f will be bound to main.c; and ?h will be bound to i1.h and i2.h. The first predicate of the property list, (?f.status = NotCompiled) is evaluated by checking if the value of the status attribute of main.c is equal to "NotCompiled". If it is, then the predicate evaluates to TRUE. Since the complex expression is an OR of two predicates, then the whole expression evaluates to TRUE in this case and we are done.

If the value of the status attribute of main.c is not equal to "NotCompiled", then we have to check if the timestamp attribute of any of the objects bound to ?h is greater than the timestamp of main.c. This is done by going through the list of objects bound to ?h and evaluating the predicate for each one of them. If the predicate evaluates to TRUE, then we stop. Otherwise, if after going through the three objects, we found that the predicate evaluates to FALSE for each one of them, the node representing the predicate is assigned the value FALSE. Since both child nodes of the OR node that is the root of the tree are FALSE, the whole tree evaluates to FALSE.

If the final result of the evaluation of the AND/OR tree is TRUE, then the condition is satisfied and the RBDE can invoke the activity of the rule. If the result is FALSE, however, then the condition is not satisfied. If the condition is not satisfied because of the evaluation of an instantiation of a predicate in the property list returned FALSE, RBDE stores the predicate and the objects used in the instantiations. RBDE will initiate a backward chain in order to try to make this predicate satisfied for the specific objects.

Chaining will be discussed in more detail later in this chapter, but first we complete the discussion of the rule execution model for an individual rule by explaining how the rule's effects are asserted.

2.4.1.3. Assertion of Effects

If the condition of an activation rule is satisfied, the rule's activity is invoked by executing the envelope corresponding to the operation specified in the activity line of the rule. When the activity terminates, the envelope returns a status code (an integer) to the RBDE. The RBDE uses this integer to determine which effect of the rule to assert. Only one of the effects of the rule will be asserted. In theory, there is no limit to the number of effects an activation rule can have; in practice an activation rule will have only a few effects. An inference rule has only one effect.

```
routine ASSERT_EFFECT (which_effect : integer; rule);
  Begin

    If which_effect > number of effects of rule OR
       which_effect < 1 Then
         return NULL;

    get the effect corresponding to which_effect;
    chaining_list := empty;

    For each assignment predicate in the effect Do
      Begin
        Assert the predicate;
        If assertion changed the value of any attribute Then
           add predicate to chaining_list;
      End;

    return chaining_list;

  End.
```

Figure 2-9: Asserting the Effects of a Rule

Assertion of an effect entails carrying out all of the assignment statements of the conjunction that makes up the effect. The details of the routine that RBDE calls to assert one of the effects of a rule are shown in figure 2-9. The routine is passed an integer that indicates which one of a rule's effects to assert. This routine inserts all of the assignment predicates that actually changed the value of an attribute on a list called `chaining_list`. This list is used for forward chaining, as will be discussed in the

next section. Forward chaining is thus driven by both the occurrence of an event (the firing of a rule) and changes to the data. If an event occurs but it does not change any data, no forward chaining results.

```
routine EXECUTE_RULE (rule);

  Begin
     Evaluate the condition of rule;
     If condition is UNSATISFIABLE Then
        return UNSATISFIABLE;

     If condition is FALSE Then
        return UNSATISFIED;

     If rule is an activation rule Then
        Begin
           Execute the activity;
           which_effect := return value of envelope;
        End;
     Else if rule is an inference rule Then
        which_effect := 0;

     chaining_list := ASSERT_EFFECT (which_effect, rule);

     If chaining_list is not empty Then
        return TRUE;
     Else
        return FALSE;
  End.
```

Figure 2-10: Complete Rule Execution Algorithm

Finally, the complete algorithm for executing one rule is shown in figure 2-10. The algorithm evaluates the condition of the rule; if it is not satisfied, the routine returns the reason (either UNSATISFIABLE or UNSATISFIED). Otherwise, if the rule is an activation rule and its condition is satisfied, its activity is executed by invoking the corresponding envelope. Once the activity invocation is complete, the variable *which_effect* is assigned the return value. This variable is used to determine which of the effects of the rule to assert. In case of an inference rule, the first and only effect is always asserted. If the assertion actually changed the values of any attributes, TRUE is returned. Otherwise, FALSE is returned. The return value is used by the rule chaining algorithm to determine whether or not to initiate forward chaining.

Note that the algorithm supports cycles, e.g.,

`edit->compile->edit->compile`-etc, which occur naturally in software development. Additional conditions that guarantee that infinite cycles do not occur can be encoded in the algorithm. To do that, the rule processor should allow cycles as long as there exists a way out of the cycle through multiple effects. The rule execution algorithm will be revised in section 2.7 when we extend the RBDE architecture to a client/server model to provide multi-user support.

2.5. Automated Assistance in RBDE

Given the rule execution algorithm presented above, RBDE can determine whether or not the firing of one rule can lead to the firing of another rule. For instance, while discussing the algorithms for evaluating the condition of a rule, we mentioned that RBDE can distinguish between a condition that cannot be satisfied automatically and one that can possibly be satisfied if other rules were executed. This knowledge is the basis for initiating ***backward chaining***. Similarly, the routine for asserting the effects of a rule indicates whether or not it has actually changed the values of any attributes. If it has, it is possible that the conditions of some rules that were not satisfied before have become satisfied because of the assertion. This is the basis for ***forward chaining*** in RBDE. To distinguish the rule that directly corresponds to the user command from the rules fired during chaining, we call the former the ***original rule***.

Given these two pieces of information, the RP in RBDE implements the rule chaining algorithm shown in figure 2-11. This algorithm executes one rule and the complete rule chain initiated by the rule without interruption. In section 2.7, we show that this chaining model is too restrictive in a multi-user environment and we subsequently revise it to allow the interleaving of multiple rule chains. In the rest of this section, however, we assume that only one rule chain is executed at any one time.

The rule chaining algorithm first attempts to execute the rule corresponding to a user command (by calling EXECUTE_RULE, shown in figure 2-10). If the execution did not succeed because the condition cannot be satisfied (i.e., the condition evaluation algorithm returns UNSATISFIABLE), the user is informed that the command cannot be executed at this time, and the cycle terminates. Otherwise, if the rule's condition is not

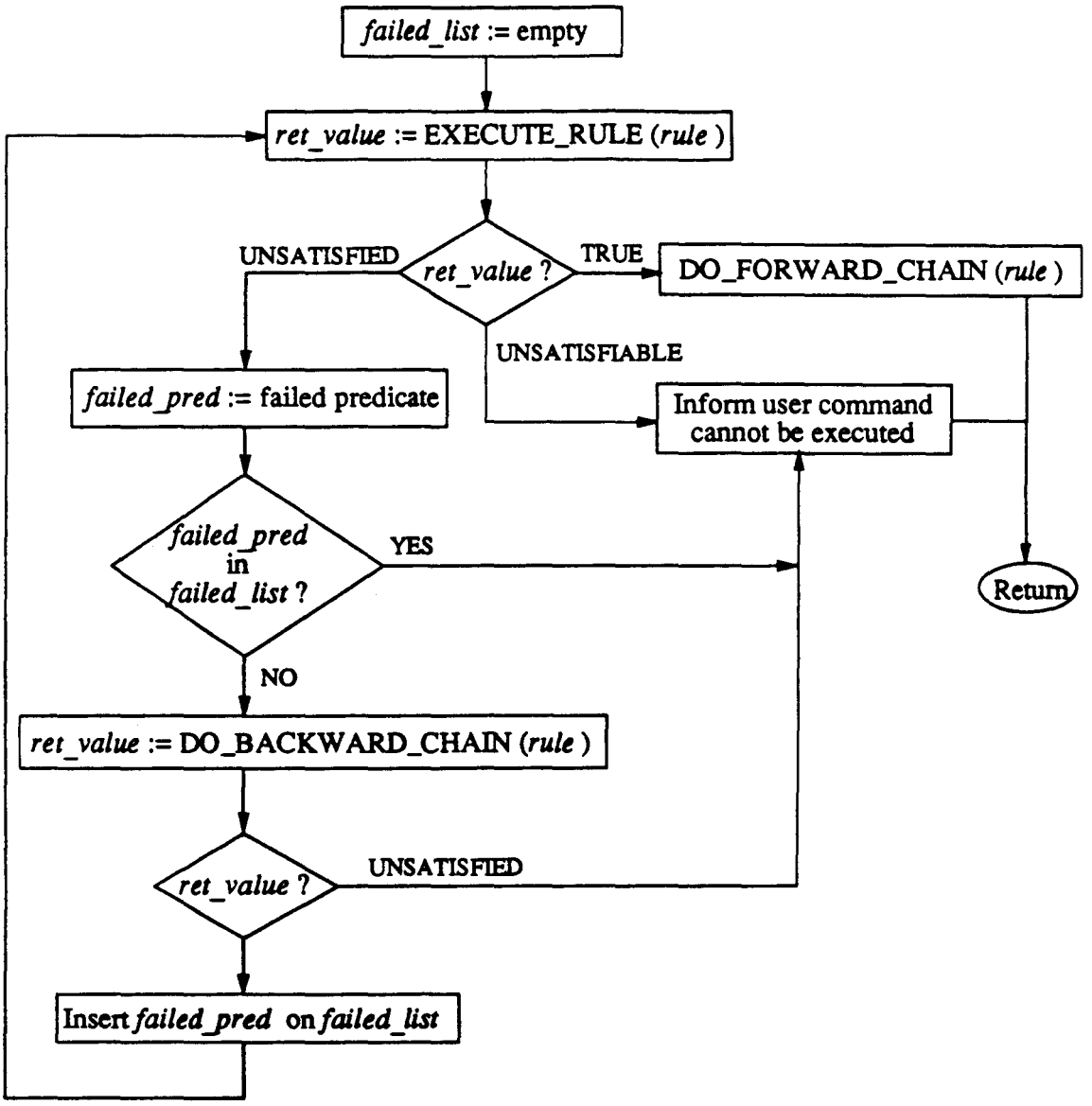

Figure 2-11: Rule Chaining Algorithm

satisfied (i.e., the evaluation algorithm returns FALSE), backward chaining is attempted. The algorithm for backward chaining is discussed in section 2.5.2. Backward chaining can either succeed in satisfying the condition of the rule, in which case the execution cycle is restarted to try to execute the rule again, or it can fail, in which case the execution cycle terminates. Finally, if the rule is successfully executed, a forward chaining cycle is initiated. The forward chaining algorithm is presented in section 2.5.3.

Forward and backward chaining together provide the main mechanisms for automated assistance in RBDE. Forward chaining is similar to what is implemented in OPS5 production systems, except that in RBDE all the rules whose conditions become satisfied are invoked rather than just one, as in SOAR [Laird 86]. Backward chaining in RBDE is similar to backward chaining in theorem provers, constraint systems and some production systems, but there is a peculiarity to the RBDE mechanism. Activation rules in RBDE invoke external tools, which can have side effects on the file system that RBDE cannot monitor. If such rules are part of a backward chaining cycle, then the actions performed by these rules cannot be reversed if the chaining fails in satisfying the condition that triggered it. Also, since tools can have several possible results, it is impossible to simulate a backward rule chain before executing it. In most other rule-based systems, either the backward chaining cycle is simulated before being attempted, or the effects of a failed backward chaining cycle can be reversed. RBDE cannot simulate backward chaining because the results of invoking tools cannot be determined, in general, without executing these activities.

RBDE provides two forms of assistance based on rule chaining:

1. **Change propagation:** If the assertion of one of the effects of a rule changes the values of the attributes of one or more objects, forward chaining is applied to propagate these changes by causing the values of attributes of the same or other objects in the database to change.

2. **Automation of tool invocation:** Both forward and backward chaining can be used to perform activities that would otherwise be done manually. By requesting one development activity (e.g., editing a document), several other activities (e.g., formatting the document and printing it) might be performed automatically by RBDE on behalf of the developer who requested the first activity.

2.5.1. Compiling Forward and Backward Chains

In order to provide assistance without unnecessarily delaying the user, RBDE tries to optimize the chaining mechanism. Rather than searching for which rules to fire during chaining, RBDE (specifically, the Loader component of it) compiles all the possible forward and backward chains between rules into an internal representation when it first loads the rule set of the project. The compilation of these chaining possibilities at load time is based on the notion of *change implication.*

Definition 1: An assignment predicate, `P1`, in the effect of a rule is said to **imply** a predicate, `P2`, in the property list of another rule iff:

1. `P1` is of one of two forms: either `(?x.att1 = value1)`, or `(?x.att = ?q.att2)`
2. `P2` is of one of two forms: either `(?y.att <op> value2)` or `(?z.att3 <op> ?y.att)`,
3. and both `?x` and `?y` are of either the same type (i.e., they are bound to instances of the same class) or one is of a subtype of the other's type.

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

     { reserve output: ?f.contents ?f.version }

     (and (?f.reservation_status = CheckedOut)
          (?f.locker = CurrentUser));

edit [?c: CFILE]:
     :
     (and (?c.reservation_status = CheckedOut)
          (?c.locker = CurrentUser))

     { edit output: ?c.contents }

     (and (?c.status = NotCompiled)
          (?c.timestamp = CurrentTime));
```

Figure 2-12: Example EMSL Rules

To illustrate, consider the rules in figure 2-12. The first assignment predicate in the effect of the `reserve` rule, `(?f.reservation_status = CheckedOut)`, *implies* the predicate, `(?c.reservation_status = CheckedOut)`, in the condition of the `edit` rule. Note that the `reserve` rule applies to all `FILE` objects, and

FILE is a superclass of both CFILE and HFILE. The two predicates operate on the same attribute of the CFILE class of objects. Thus, the Loader can determine that if the condition of the edit rule is not satisfied (because the first predicate in its condition is not satisfied), it might be possible to make this predicate satisfied if the reserve rule is fired automatically. Alternatively, asserting the effect of reserve might lead to firing the edit rule.

From this example, it should be clear that given the EMSL constructs we have described so far, forward and backward chaining are completely symmetric. In other words, every forward chain from rule r1 to rule r2 has a corresponding backward chain from rule r2 to rule r1. This is not always desirable. In the example above, although it might be desirable for edit to backward chain to reserve, it is certainly awkward that reserving an object should lead to invoking the editor on it automatically. To solve this problem, EMSL provides three prefixes that the administrator can attach to predicates in the conditions and effects of rules: no_forward, no_backward, and no_chain.

If a predicate in the property list of a rule is preceded by the no_backward prefix, then the predicate cannot initiate backward chaining if it is not satisfied. Similarly, if a predicate in the effects of a rule is preceded by the no_forward prefix, then the predicate cannot initiate forward chaining. Attaching a no_backward prefix to an assignment predicate in the effect of a rule prevents backward chaining into the predicate. Similarly a no_forward prefix attached to a predicate in the property list of a rule prevents forward chaining into the predicate. A no_chain prefix attached to a predicate prevents chaining from or into the predicate.

```
reserve[?f:FILE]:
    :
    no_backward (?f.reservation_status = Available)

    { reserve output: ?f.contents ?f.version }

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

Figure 2-13: Revised "Reserve" Rule

Using these prefixes, the administrator can specify that the edit rule can backward

chain to `reserve` but not vice versa. A revised version of the `reserve` rule that achieves this is shown in figure 2-13. Note that it was not necessary to change the `edit` rule.

These three prefixes provide us with the ability to distinguish between predicates that can cause chaining, which we call ***chaining predicates***, and those that cannot, called *non-chaining predicates*. To put it more formally, a chaining predicate is defined as follows:

Definition 2: A predicate, `P`, is said to be a *chaining predicate* if:

1. `P` is in the property list of a rule and it is not preceded by a `no_chain` or a `no_backward` prefix, or
2. `P` is an assignment predicate in the effect of a rule and it is not preceded by a `no_chain` or a `no_forward` prefix.

Every other predicate is a *non-chaining* predicate. Note that an assignment predicate with a `no_backward` prefix is still a chaining predicate.

Given this definition of chaining predicates, we must now revise the definition of change implication as follows:

Definition 3: An assignment predicate, `P1`, in the effect of a rule is said to **imply** a property predicate, `P2`, in the property list of another rule iff:

1. `P1` is a chaining predicate, and it is of one of two forms: either `(?x.att1 = value1)` or `(?x.att = ?q.att2)`
2. `P2` is in one of two forms: either `(?y.att <op> value2)` or `(?z.att3 <op> ?y.att)`,
3. and both `?x` and `?y` are of either the same type (i.e., they are bound to instances of the same class) or one is of a subtype of the other's type.

In order to simplify the compiling of possible chains, the loader creates two tables, the *Rule Table*, which contains an entry for each rule in the rule set, and the *Predicate Table*, which contains an entry for each predicate in the conditions and effects of every rule. The entries in the rule table point to the entries in the Predicate Table. Immediately after loading the project rule set, the forward and backward chains in the Predicate Table are empty. The Loader then executes the routine shown in figure 2-14 to fill in these chains.

```
routine COMPILE_CHAINS();

  Begin

    /* Predicate Table is a global variable. */

    For each predicate, p1, in the Predicate Table Do
      Begin
        If p1 is not a chaining predicate Then
          continue;

        /* Check p1 against every predicate in the table. */

        For each predicate, p2, in the Predicate Table Do
          Begin
            If p1 = p2 Then
              continue;

            If p1 is an assignment predicate Then
              Begin
                If p2 is an assignment predicate Then
                  continue;
                Else If p1 implies p2 Then
                  Begin
                    If p2 is preceded by either no_forward
                                              or no_chain Then
                      continue;
                    add forward chain from p1 to p2;
                  End;
              End;

            Else If p1 is a property predicate Then
              Begin
                If p2 is a property predicate Then
                  continue;
                Else If p2 implies p1 Then
                  Begin
                    If p2 is preceded by either no_backward
                                              or no_chain Then
                      continue;
                    add backward chain from p1 to p2;
                  End;
              End;
          End;
      End;
  End.
```

Figure 2-14: Algorithm for Compiling Forward and Backward Chains

This routine checks each predicate in the Predicate Table to determine if this predicate can chain to any other predicate in the table. The routine makes sure that a predicate does not chain to itself.

After compiling the forward and backward chains, the predicates in the Predicate Table

will be inter-connected by forward and backward chains. At runtime, the RBDE does not need to match asserted assignment predicates against the conditions of rules. It simply has to follow the chains that have been compiled in order to perform the appropriate type of chaining. To complete the rule chaining model, we now present the chaining algorithms that RBDE uses to provide automated assistance.

2.5.2. Backward Chaining

When a rule is selected for execution, the first thing RBDE does is evaluate the rule's condition. If the condition is satisfied (i.e., TRUE), the activity of the rule can be executed immediately. If the condition is UNSATISFIABLE, then the activity cannot be performed and the user is informed of the problem. Otherwise if the evaluation of the condition returns FALSE, then an instantiation of a predicate in the property list of the rule's condition must have been evaluated to be FALSE. If the predicate's entry in the Predicate Table has any backward chains, RBDE initiates a backward chaining cycle in order to attempt to make the predicate satisfied.

The backward chaining algorithm is shown in figure 2-15. Given a rule whose condition is not satisfied, the algorithm first gets the first predicate *P* that caused the condition not to be satisfied. Then, for each predicate connected to *P* by a backward chain, the routine gets the rule containing this predicate: if that rule's condition is satisfied, it is inserted in the *ready* list; otherwise if the condition is not satisfied, the rule is inserted in the *backward* list. Note that the rules whose conditions are UNSATISFIABLE are skipped. Next, RBDE attempts to execute the rules in the *ready* list. If any of these rules was successful in making *P* (the predicate that initiated the backward chain) satisfied, the chaining stops. Otherwise, RBDE initiates one more level of backward chaining recursively on the rules in the *backward* list. Eventually, if every possible rule has been fired, but *P* still evaluates to FALSE, the algorithm returns UNSATISFIED to indicate that it cannot satisfy the predicate.

```
routine DO_BACKWARD_CHAINING (failed_rule);

  Begin

    /* Get the predicate that initiated backward chaining. */

    P := failed predicate in failed_rule;

    /* Insert rules on ready and backward lists. */

    For each predicate in the backward chain of P Do
      Begin
        rule := the rule containing the predicate;
        If rule is not already on ready or backward lists Then
           Begin
              Evaluate the condition of rule;
              If condition is SATISFIED Then
                 Add rule to the ready list;
              Else If the condition is FALSE Then
                 Add rule to the backward list;
           End;
      End;

    /* Process the rules on the ready list. */

    For each rule in ready Do
      Begin
        ret_value := EXECUTE_RULE (rule);
        If ret_value = TRUE Then
          Begin
            Evaluate P;
            If P is TRUE Then
               return SATISFIED;
          End;
      End;

    /* Process the rules on the backward list. */

    For each rule in backward Do
      Begin
        ret_value := DO_BACKWARD_CHAINING (rule);
        If ret_value = SATISFIED Then
          Begin
            ret_value := EXECUTE_RULE (rule);
            If ret_value = TRUE Then
              Begin
                Evaluate P;
                If P is TRUE Then
                  return SATISFIED;
              End;
          End;
      End;

    return UNSATISFIED;
  End.
```

Figure 2-15: Backward Chaining Algorithm

2.5.3. Forward Chaining

```
routine DO_FORWARD_CHAINING (rule);

   Begin
     get chaining_list of rule;

     For each predicate, p1, in chaining_list Do
       Begin
         forward_list := forward chains of p1;
         For each predicate, p2, in forward_list Do
            Begin
              r := rule containing p2;
              ret_value := EXECUTE_RULE (r);
              If ret-value = TRUE Then
                 DO_FORWARD_CHAINING (r);
            End;
       End;
   End.
```

Figure 2-16: Forward Chaining Algorithm

The forward chaining algorithm is simpler than the backward chaining one. Given a rule that was just successfully executed, the algorithm gets the *chaining_list* of the rule (which is produced by ASSERT_EFFECT in figure 2-9 on page 47). Then, for each predicate in this list, the routine gets the rule containing the predicate, executes that rule, and initiates any further forward chaining caused by that rule's execution. Thus, the algorithm is recursive and depth-first in the sense that it executes one rule and all the chaining resulting from that rule before going on to the next possible rule in the forward chain. The algorithm is shown in figure 2-16. As can be seen from the algorithm, forward chaining is a "best-effort" activity. It is always successful by definition even if none of the rules in the chain can be executed.

Finally, the complete command execution cycle implemented by RBDE is shown in figure 2-17. Most of the algorithms and routines presented thus far are very similar to those implemented in MARVEL; higher-level descriptions of data modeling, process modeling, and rule chaining in MARVEL appears in several papers [Kaiser et al. 88a, Kaiser et al. 88b, Barghouti and Kaiser 88, Kaiser et al. 90]. The presentation above, however, is much more detailed and precise than that of any of these papers. As mentioned earlier, these details are necessary for explaining and solving the concurrency

```
routine COMMAND_CYCLE ();

  Begin
    While (TRUE) Do
      Begin
        get the next command request from the user interface;
        If user requested a built-in command Then
           If command is Quit Then
             exit;
           Else
             call command executor to execute the command;

        Else If user requested command from rule menu Then
          Begin
            rule := rule corresponding to the command;
            ret_value := EXECUTE_RULE (rule);
            If ret_value = FALSE Then
              inform user that command cannot be executed;
          End;
      End;
  End.
```

Figure 2-17: The Command Execution Algorithm in RBDE

control problem; without the details presented above, it is difficult to explain the multi-user command execution model, in which the concurrency control problem arises.

We have now completed the presentation of all the algorithms needed for executing commands in RBDE. The command execution model we presented above executes built-in commands, individual rules, and complete rule chains one at a time. In order to scale up the command execution model to support multiple users, we must extend our architecture to support the execution of multiple concurrent rule chains. In the rest of this chapter, we extend RBDE to handle multiple concurrent rule chains through a client/server model. This model was the result of a Master's thesis by a member of the MARVEL project [Ben-Shaul 91]. We only present the aspects that are relevant to concurrency control.

2.6. Client/Server Architecture

The main extension needed to handle multiple users in RBDE is to separate all access to the objects in the database of one project within a server process that executes independently from the user processes. There can be at most one server process running per project database. The server includes the OMS (object management system), the CE (command executor), and the RP (rule processor). The transaction manager and the lock manager, which will be responsible for implementing the concurrency control mechanism we develop in subsequent chapters, will also reside in the server.

Each user process, called a *client*, handles interaction with one user and the execution of the activities of rules fired by the user's commands. Multiple clients can establish communication channels with a single server, and the server will service all of these clients concurrently. A user requests a command within the client process. The client will then send the request to the server. The clients and the server communicate with each other through message passing. A client sends a message to the server; this message is inserted on the server's queue; the server processes the messages in the queue in a first-come-first-serve fashion. After processing a client's message, the server sends a message to the client instructing it what to do next.

The messages sent between clients and servers have three fields. The first field, `client_id`, is used to store the unique identifier that the server assigns to the client. The second field, `kind`, indicates the kind of the message, and the third field, `contents`, stores the contents of the message. The messages that the client sends to the server are of two kinds: (1) to request the execution of a command; or (2) to request the server to continue executing a rule (or a rule chain). The need for the second kind of messages and how these messages are processed by the server will be explained shortly. Processing the first kind of messages requires that the server call either the RP or the CE to execute the command.

One major decision we had to make was whether to have the RP and the CE in each of the clients or in the central server. The execution of either a built-in command or a rule requires access to objects in the database. As described earlier in this chapter, the RP is

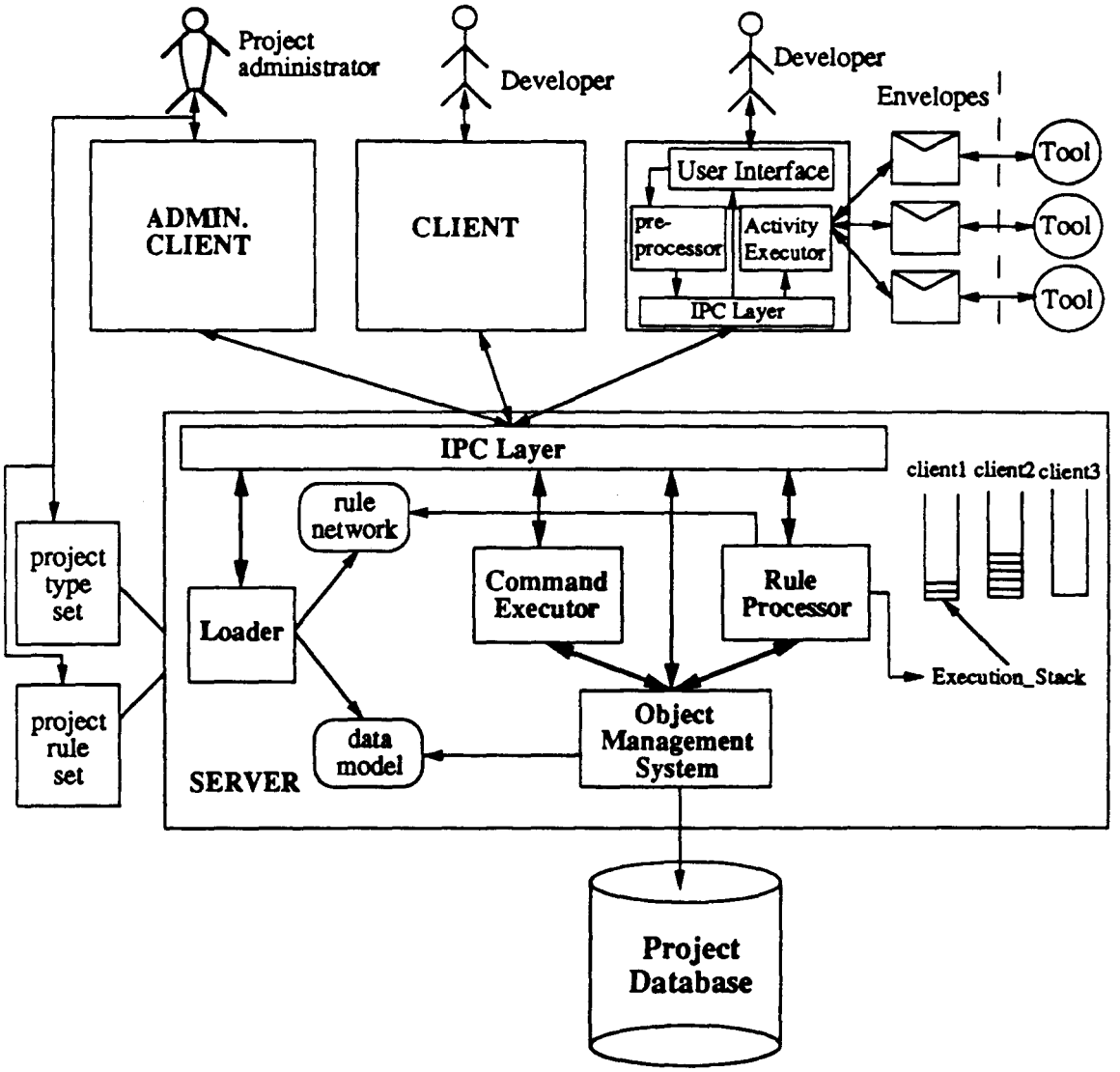

Figure 2-18: Client/Server RBDE Architecture

responsible for collecting all the objects that are bound to the variables in the binding part of the rule. The execution of a rule (i.e., evaluation of a rule's condition and the assertion of one of its effects) might require access to possibly many objects in the database. In the extreme case, the execution of a rule might require access to the whole database. In contrast, the execution of each built-in command involves access to a fixed

number of objects. For example, the `copy` command requires accessing three objects, the source object we need to copy, the new object that duplicates the source object, and the parent of the new object (the destination object).

There are no advantages at all to having the command executor in the clients. Having the RP in each client, on the other hand, would potentially enable us to have a private rule set for each client. However, in order for the client to execute a rule, it would have to request access to all the objects bound to the variables in the binding part of the rule via the server. The server would have to collect all of these objects and send them to the client. The size and number of objects needed per rule execution could make the communication costs between the clients and the server very expensive. Because of these problems, as well as to simplify transaction management, as will be explained in chapter 4, we decided to put both the RP and the CE in the server. Both components form the *command execution layer*. The problem of supporting multiple rule sets with this architecture is discussed in [Barghouti and Kaiser 91b].

The server maintains a context for each client and performs the commands requested by a client within that client's context. Each client's context is basically an execution stack, which is used to maintain information about the execution of the command requested by the client, and the progress of the rule chain that might have been initiated by the client's previous message. For example, if the execution of a client's request initiates a forward chain to several rules, then these rules are pushed on the stack. The RP executes these rules, one by one, by popping the rule on top of the stack and executing it. Each client can request only one command at a time. Before a client can request a command, that client's execution stack must be empty; this occurs only when the client's previous command and any chaining resulting from it have been completed.

When a client sends a message to the server and when the message is selected from the server's queue, the server restores the client's context by making the client's execution stack the global execution stack, pointed to by the global variable *Execution_Stack*. The RP uses the stack pointed to by *Execution_Stack* to store rules that will be executed during chaining. Thus, the RP actually does not have to know about the existence of multiple execution stacks belonging to different clients. In the following discussion, un-

less otherwise specified, the "stack" refers to the global execution stack. The overall client/server architecture is depicted in figure 2-18.

Whenever a rule is pushed on the stack, the state of its execution is stored with it. A rule is pushed on the stack for two reasons: either it is still waiting to be fired (i.e., its execution has not started yet but it was inserted on the stack because it is a possibility for chaining), or its condition has been evaluated but its effects haven't been asserted yet because the activity is still being executed by the client. In either case, the rule is either a rule that was fired directly in response to a user command, or a part of a forward or backward chain. Based on this, rules can be in one of three states: (1) `original_rule`, which indicates that this is the rule that initiated a backward chaining cycle; (2) `back_chain`, which indicates that the rule is a part of a backward chaining cycle; and (3) `forward_chain`, which indicates that the rule's condition is satisfied, but the assertion of its effects is awaiting the completion of the execution of the rule's activity; note that a rule can be in a `forward_chain` although it is a part of a backward chaining cycle. Once the assertion of one of the rule's effects is completed, the rule is removed from the stack since its execution would have been completed. We now explain rule execution in the client/server architecture in more detail.

2.7. Concurrent Rule Execution Model

Having the RP in the server means that access to objects is direct since the object management system is also part of the server. Thus, the evaluation of a rule's condition can be done as an atomic operation by the server without any communication with the clients. The RP is also able to carry out a whole rule chain atomically if that chain involves only inference rules (rules with empty activities) since it does not need to wait for the client to execute any activities. Processing activation rules, however, adds some complications.

In the rule execution algorithm of section 2.4 (figure 2-10 on page 48), the RP executes each activation rule as a unit. In other words, the evaluation of the condition, the execution of the activity and the assertion of one of the effects are all done by the same routine. In the client/server model, however, the rule's activity is executed by the client,

whereas the evaluation of the rule's condition and the assertion of the effects are done in the server. In order to increase server throughput, we divide the execution of an activation rule into two phases: one to evaluate the condition and possibly initiate backward chaining, and the other to assert the effects and possibly initiate forward chaining.

```
routine START_RULE_EXECUTION (rule);

  Begin
     Evaluate the condition of rule;
     If condition is UNSATISFIABLE Then
        return DONE;

     If condition is FALSE Then
       Begin
         set state of rule to original_rule;
         push rule on Execution_Stack;
         ret_value := DO_BACKWARD_CHAINING (rule);
       End;

     Else If condition is TRUE Then
        If rule is an inference rule Then
          Begin
            chaining_list := ASSERT_EFFECT (0, rule);
            If chaining_list is not empty Then
               ret_value := DO_FORWARD_CHAINING (rule);
          End;
        Else
          Begin
            set state of rule to forward_chain;
            push rule on Execution_Stack;
            ret_value := CONTINUE;
          End;

     return ret_value;
  End.
```

Figure 2-19: The First Phase of Rule Execution in the Server

The algorithm for the first phase of rule execution is shown in figure 2-19. Say that a user requests a command that causes RBDE to fire a rule, R. The rule's condition is evaluated; if it is not satisfied, then the state of R is set to `original_rule`, R is pushed on the client's execution stack, and backward chaining is initiated. By setting the state of R to `original_rule` before inserting it on the client's execution stack, the RP will know, when it gets to R, that it is the original rule that initiated the backward chaining cycle.

If R's condition is satisfied, then if R is an activation rule, it is pushed on the execution

stack of the client after setting its state to `original_rule`. If R is an inference rule, then its first and only effect is asserted, and if applicable, forward chaining is initiated. The algorithms for forward chaining and backward chaining that were presented earlier must be revised to take into account the client/server model. We will discuss these revised algorithms shortly.

When the first phase ends, the server sends a message to the client informing it that it can go ahead and execute R's activity. The execution of an activity can take an arbitrarily long time. Instead of waiting idly during that time, the server can continue processing some other client's request in the meanwhile. To do that, the server switches to the other client's context and continues the processing from the point at which it stopped. But before we explain context switching, let us complete the description of the second phase of rule execution, so assume for now that there is only one client, which means that the server just waits for that client to finish executing the activity of the rule R.

When the client completes the execution of R's activity, it sends the server a message that includes the results of the tool's execution and requests the assertion of one of the rule's effects. This request is queued until the server can process it, at which time the server restores the client's context (by making it the global Execution_Stack). The server then calls the RP to continue from the point at which it stopped processing rule R. Since the information that the server had stored in the context indicates that R has finished the first phase of its execution, the RP starts the second phase in the execution of R.

During the second phase, which is shown in figure 2-20, the RP continues the execution of a rule from where it left off. This point is always after the end of the first phase, which is after the evaluation of the rule's condition. However, the rule could have been part of either a forward chain (including the original rule corresponding to the user's command) or a backward chain. The state of the rule will indicate which one of these is the case. Depending on the state of the rule's execution, the RP will either continue (or initiate) forward chaining, or continue backward chaining.

```
routine CONTINUE_RULE_EXECUTION (rule : activation rule,
                                 which_effect : status code);
  Begin

     get the state of rule;
     If state = back_chain Then
       ret_value := CONTINUE_BACKWARD_CHAIN (rule, which_effect);
     Else
       Begin
         chaining_list := ASSERT_EFFECT (which_effect,rule);
         If chaining_list <> empty Then
           ret_value := DO_FORWARD_CHAINING (rule);
         Else
           ret_value := CONTINUE;

     return ret_value;
  End.
```

Figure 2-20: The Second Phase of Rule Execution in the Server

The forward and backward chaining algorithms presented in section 2.5 have to be slightly revised in the client/server model. In particular, backward chaining must be split into two phases. The first phase of backward chaining is initiated from the START_RULE_EXECUTION algorithm shown in figure 2-19, which starts the execution of a rule chain. This phase continues until a rule, R, whose condition is satisfied, is found. At this point, the activity of R must be executed. The RP sets the state of R to back_chain and pushes the rule on the stack. By setting the state of R to back_chain, the server will know when it continues executing R that it was a part of a backward chain. The details of the revised chaining routines, specifically CONTINUE_BACKWARD_CHAIN, are not terribly relevant to the concurrency control problem. As far as our discussion here is concerned, forward and backward chaining push additional rules on the client's execution stack (context).

Now that we have briefly explained the rule execution algorithm for one client, we return to our discussion of context switching among several clients' contexts. The server switches between the contexts of different clients at one of two points: (1) after the RP in the server has finished evaluating the condition of an activation rule within a chain, at which point the rule's activity is passed to the client to execute; or (2) when the execution of a rule chain is completed and there is nothing more that the server can do with

respect to this chain. In the latter case, the server cleans up the context of the client and informs the client that the execution of the command requested by the user has been completed. The execution stack, which will be empty, will be re-activated when the same client sends a new request to the server.

While the client is executing a rule's activity, the server can check if there are any pending messages in its request queue. If there are, then the server switches to the context of the client that sent the first message on the queue. The message that a client sends to the server can be a request to execute a built-in command, to execute a command corresponding to a rule, or to continue the execution of a rule within the client's current rule chain. The first two requests can be made only if the client's context is empty, which means that the RP has completed the execution of the previous command requested by the same client. The third request implies that the RP was still in the middle of executing a rule chain, which was initiated by some previous request from the same client.

As mentioned earlier, the context of a client is an execution stack that contains the current rule chain of the client. The rule that the RP was executing last will be on top of the stack. Thus, when the server receives a message requesting the continuation of a rule's execution, it restores the context of the client that sent the message. Restoring a client's context means that the client's execution stack becomes the global execution stack, pointed to by the variable *Execution_Stack*. The RP always assumes that it should continue executing the rule on top of the stack pointed to by *Execution_Stack*.

The complete algorithm performed by each client process is shown in figure 2-21. Each client first establishes a communication line with the server of the project database, and gets a unique client identifier from the server. This identifier is attached to every message the client sends to the server. The message's contents are either the name of the command or the return code from the envelope, which executed a rule's activity. After sending a message to the server, the client waits for the server's response. The server's response will direct the client to do one of three things: (1) exit, meaning that the user has requested the `quit` built-in command; (2) get the next command from the user; or (3) execute the activity of a rule by invoking a particular envelope.

```
routine CLIENT();

  Begin

    establish communication line with server;
    get unique client_id from server;

    While (TRUE) Do
      Begin

        get user command;

        /* Form message to request command from server. */

        msg.client_id := client_id;
        msg.kind := EXECUTE_COMMAND;
        msg.contents := user command;

        send msg to server;

        /* wait until the server sends back a message. */

        receive server_msg from server;

        While server_msg.kind <> DONE  Do
          Begin
            If server_msg.kind = QUIT Then
               exit;  /* end the client's process */

            activity := server_msg.contents;
            return_code := EXECUTE_ACTIVITY (activity);
            msg.kind := CONTINUE_RULE;
            msg.contents := return_code;
            send msg to server;

            /* wait till the server sends back a message. */
            receive server_msg from server;
          End;
       inform user that execution of command is completed;
    End;
  End.
```

Figure 2-21: The Client's Main Algorithm in RBDE

The complete algorithm that the server process executes is shown in figure 2-22. The two routines START_RULE_EXECUTION and CONTINUE_RULE_EXECUTION carry out the first phase and the second phase of rule execution, respectively. Further details of the client/server architecture can be found in [Ben-Shaul 91].

It should be clear from the algorithms above that concurrent user commands can cause multiple rule chains to execute concurrently in the server. The execution of the concur-

```
routine SERVER_CYCLE ();

  Execution_Stack is a global variable;

  Begin
    While (TRUE) Do
      Begin
        If request queue is empty Then
           continue;

        message := next message from the request queue;
        client_id := message.client_id;
        Execution_Stack := restore context of client_id;

        If message.kind = EXECUTE_COMMAND Then
          Begin
            If message.contents is a built-in command Then
              Begin
                call command executor to execute the command;
                If command is quit Then
                   msg.kind := QUIT;
                Else
                   msg.kind := DONE;
                send msg to client;
                continue;
              End;
            Else
              Begin
                rule := rule corresponding to the command;
                ret_value := START_RULE_EXECUTION (rule);
              End;
          End;
        Else if message.kind = CONTINUE_RULE Then
          Begin
            rule :=  top rule on client's stack;
            which_effect := integer returned by client;
            ret_value:= CONTINUE_RULE_EXECUTION(rule,which_effect);
          End;

        If ret_value = CONTINUE Then
          Begin
            get activity of the rule on top of client's stack;
            msg.kind := EXECUTE_ACTIVITY;
            msg.contents := activity;
          End;
        Else If ret_value = CYCLE_COMPLETED Then
          Begin
            clear client's context;
            msg.kind := DONE;
          End;
        send msg to client;
      End;
  End.
```

Figure 2-22: The Server's Main Algorithm in RBDE

rent chains of different clients is interleaved at the points when the server switches from

one context to another (i.e., when the first phase of the execution of an activation rule in a chain terminates). This interleaving might cause concurrency conflicts if the rule chains require access to overlapping sets of objects, possibly leading to a violation of the database consistency.

2.8. Assumptions

Throughout this chapter, we have made several assumptions that have direct implication on solving the concurrency control problem. Of course, the main assumption we have made is the rule-based SDE model itself. We do not necessarily advocate this model over other kinds of SDEs; we consider the model a given for our thesis work. In the RBDE rule execution model, we have made several assumptions that can be relaxed if we were to generalize and extend RBDE. We discuss these assumptions here. In chapter 8, we give some ideas as to how many of these assumptions can be relaxed. The assumptions we have made are:

1. The rule set is consistent. In other words, the administrator will not write two rules that contradict each other logically. All chaining behavior prescribed by the rules terminate at some points; thus, although the rules can specify cycles (e.g., edit-compile-debug-edit-etc), there must be a way out of each cycle (through multiple effects).

2. Tools are treated as "black boxes", meaning that we can determine their inputs and outputs. This enables us to predict the complete set of objects that will be read and written by activation rules.

3. The "black box" assumption also implies that the result of the tool is not known until the tool has been executed. A tool can thus have several possible results, which are reflected in multiple mutually exclusive effects in activation rules. Because of that, the exact set of rules that might be fired in a forward or a backward chaining cycle is unpredictable. Thus, we can predict only the set of all possible objects that might be read and written by a rule chain. This set is usually much larger than the actual set that will be accessed by the chain.

4. The only mechanism to create or delete objects is by requesting the `add` and `delete` built-in commands, respectively. Rules cannot add or delete objects in their effects.

5. The database in which project components are stored is centralized. The client processes, however, can be distributed across multiple machines. Distributing the database has significant implications on the concurrency control model we describe in this dissertation.

6. The centralized server executes both built-in commands and inference rules atomically. Interleaving of rules or rule chains occurs only when the activity of an activation rule is being executed. Implementing a different model in which the server uses time slicing to switch from one chain to another would invalidate several of the observations we make in chapters 3 and 5.

2.9. Summary

In this chapter, we presented a multi-user SDE architecture that is both process-centered and rule-based. The architecture is composed of a specification language, EMSL, a rule-based SDE kernel, RBDE, and a client/server model for supporting multiple users. A project administrator uses EMSL to specify three aspects of the project: (1) a prescription of the project's development process in terms of rules (the *project rule set*), (2) a description of the organization and structure of the project's data in terms of object-oriented classes (the *project type set*), and (3) the interface between the project rule set and external tools (the *project tool set*). These specifications are then loaded into RBDE, which presents the project's developers with a tailored RBDE environment.

The project's development process of a project is modeled by two kinds of rules: *activation rules* and *inference rules*. An activation rule controls the invocation of a tool by specifying the *condition* under which it is appropriate to invoke the tool and the possible *effects* of the tool on the values of objects' attributes. In contrast, the activity part of inference rules is empty, and each has a single effect, an expression that is a logical consequence of the condition of the rule. Both activation and inference rules are parameterized to take as arguments one or more objects, each of which is an instance of some class. When a user requests the execution of a command on a set of objects, RBDE selects the rule that matches the command.

RBDE provides assistance by applying forward and backward chaining to automatically fire rules, which in the case of activation rules initiate development activities. A 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. The result of this backward chaining is either the satisfaction of the original condition or the inability to satisfy it given the current state of the database. When the condition is satisfied, the activity is initiated, and after it terminates, the RBDE asserts one of the rule's effects. This might satisfy the condition of other rules. The RBDE fires these rules. The effects of these rules may in turn cause additional forward chaining.

When multiple developers cooperate on a project, they share a common database that contains all the components (source code, documentation, test suites, etc.) of the project. The developers may request commands that access objects in the shared database concurrently. To support concurrency, a centralized server controls all access to the project database. Each user interacts with a client process, which passes on the users' commands to the server.

The server maintains a context for each client, and employs context switching to process multiple user commands concurrently. The RP, which is a part of the server, executes multiple rule chains, resulting from multiple user commands, concurrently. The rule chains might interfere if they access overlapping sets of objects. The avoidance of such interference is the essence of the concurrency control problem.

In the rest of this dissertation, we present a database-oriented model of RBDE, and explain the concurrency control problem in terms of that model. We then proceed to explain the components of our concurrency control mechanism.

Chapter 3

The Concurrency Control Problem in RBDE

In the previous chapter, we presented a multi-user SDE architecture that allows multiple developers to request commands concurrently. Each of these commands might initiate a rule chain. The effect is to allow multiple rule chains to execute concurrently, interleaving their access to the project database. These concurrent chains interfere with each other if they access overlapping sets of objects. This interference can corrupt the objects in the database. In order to understand and solve the concurrency control problem in RBDE, we need to make the command execution model, presented in the previous chapter, more precise in terms of how rules and built-in commands access the database. Only then can we define the concurrency control problem more precisely.

In this chapter, we first illustrate the concurrency control problem in RBDE by means of an example. We then present a database-oriented model of command execution in RBDE. The main component of this model is the *agent*, which abstracts an individual user command. This notion is akin to the *transaction* concept in database systems, which we explain before defining agents. We then use the notion of *nested transactions* to define *nested agents*, which model rule chains. Next, we use the concepts of *serializability* and *serializable executions* to define the concept of *serializable schedules* of multiple agents. The notion of *interference* is then defined in terms of serializable schedules. Finally, we decompose the concurrency control problem in RBDE into three subproblems in order to simplify its solution.

3.1. Example of The Concurrency Problem

We illustrate the concurrency control problem by means of the example we presented in chapter 1. To remind the reader, Bob, John and Mary are working together to develop a program, `Prog`, composed of three source modules, three library modules and a directory, `includes`, that contains header files. Suppose that John and Mary want to test module `ModB` by running a test suite, `test1`. As was shown in figure 2-3 on page 30, `ModB` consists of two `CFILE` objects, `f3.c` and `f4.c`. When John and Mary request commands concurrently, John's command might trigger a chain of rules, one or more of which might interfere with one or more of the rules in the in-progress chain that Mary's command has triggered.

```
test [?f:CFILE ; ?t:TEST_SUITE]:

    # C source files can be tested only if compiled.
    :
    (?f.status = Compiled)

    { run-test ?t.contents ?f.object_code }

    (?f.test_status = Tested);
    (?f.test_status = Failed);


test [?mod:MODULE ; ?t:TEST_SUITE]:

    # A module can be tested only after all the contained
    # C source files have been tested.

    (bind ?f to_all CFILE suchthat (member [?mod.cfiles ?f]))
    :
    (forall ?f):
    (?f.test_status = Tested)

    { run-test ?t.contents ?mod.object_code }

    (?mod.test_status = Tested);
    (?mod.test_status = Failed);
```

Figure 3-1: Example Rules for Testing C Files and Modules

Suppose that John requests a command `test ModB test1`, which fires the second test rule shown in figure 3-1, at time *t1*. Suppose that the condition of the rule is not satisfied because the two `CFILE` objects contained in `ModB`, `f3.c` and `f4.c`, have not

been tested yet (i.e., the value of their `test_status` attributes is not equal to "Tested"). The unsatisfied condition triggers a backward chaining cycle to try to make the condition satisfied. The RP (rule processor) fires the first `test` rule of figure 3-1 on `f3.c` at time *t2* (since the RP executes each rule chain serially, the RP does not fire the `test` rule on `f4.c` except if the execution of `test f3.c test1` results in changing the value of the `test_status` attribute of `f3.c` to "Tested").

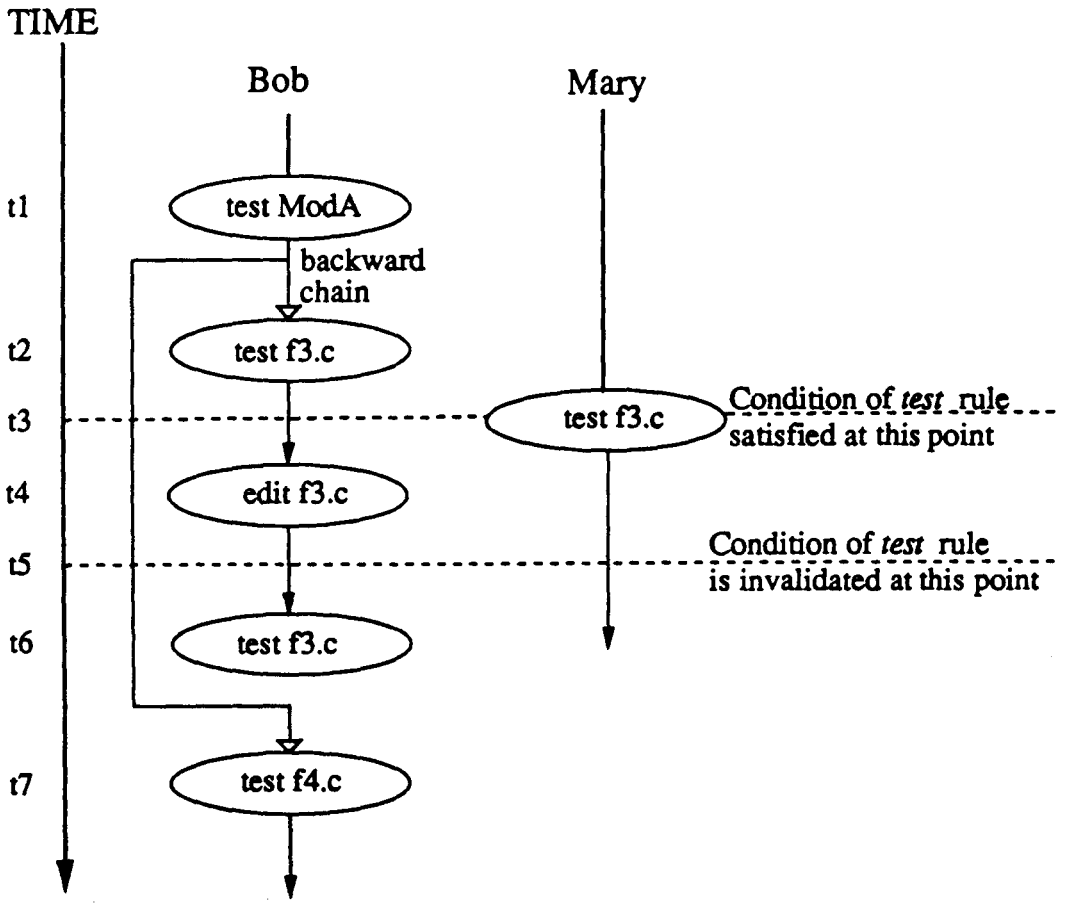

Figure 3-2: Example of Interference

While John is testing `f3.c`, Mary requests a command `test f3.c test1`, which fires the first test rule in figure 3-1, at time *t3*. The condition of the rule is satisfied at that time, causing the rule's activity to be invoked. Meanwhile, John discovers that `f3.c` has a bug so he issues the command `edit f3.c` at time *t4*. Editing `f3.c` will change the contents of `f3.c`; the effect of the `edit` rule will change the value of the

`status` attribute of `f3.c` to "NotCompiled", making the condition of Mary's `test` rule, which is already in-progress, unsatisfied. Allowing John to edit `f3.c` while Mary is testing might make the results of Mary's test obsolete. The sequence of activities is depicted in figure 3-2.

Interference also results from interleaving the execution of a rule chain with a built-in command. For example, say that after both `f3.c` and `f4.c` have been tested individually, John proceeds to test module `ModB`. After the condition of the `test` rule (that was fired on `ModB`) has been re-evaluated by the RP and found to be satisfied, John's client process invokes the `test` envelope to test `ModB`. While this activity is being executed in the client, Mary issues a built-in command to `add` an object, `f7.c`, to `ModB`. This built-in command will cause the condition of John's `test` rule to be UNSATISFIED, since the module now contains a `CFILE` whose `test_status` attribute is not equal to "Tested". Thus, John's testing of `ModB` will be invalidated.

The two examples above demonstrate the need for a concurrency control mechanism that can detect interference between concurrent rule chains. Once such interference is detected, RBDE can use semantic information about the project, the individual rules, and the rule chains to resolve the interference in a flexible manner. Before we can explain how interference is detected and resolved, however, we need to give a more formal model of command execution in RBDE, and define the concurrency control problem more precisely in terms of that model.

3.2. Related Work: Transactions in DBSs

In order to make the command execution model in RBDE more precise in terms of database access, we construct a database-oriented model of command execution in RBDE. The main construct of this model is the *agent*, which abstracts the execution of a user command in terms of its access to the project database. This notion of agent is akin to the *transaction* concept in database systems. We first explain the concept of transactions and then proceed to define our database model.

3.2.1. The Transaction Concept

In traditional DBSs, each database operation is abstracted to be either a read operation or a write operation, irrespective of the particular computation. Then, a DBS can guarantee that the database is always in a consistent state with respect to reads and writes independent of the semantics of the particular application.

To do that, the operations performed by a program accessing the database are grouped into sequences called *transactions* [Eswaran et al. 76]. Users interact with a DBS by executing transactions. In traditional DBSs, transactions serve three distinct purposes [Lynch 83]: (1) they are logical units that group together operations comprising a complete task; (2) they are atomicity units whose execution preserves the consistency of the database; and (3) they are recovery units that ensure that either all the steps enclosed within them are executed, or none are. It is thus by definition that if the database is in a consistent state before a transaction starts executing, it will be in a consistent state when the transaction terminates. Users interact with a DBS by invoking programs, each of whose execution is encapsulated by a transaction. Alternatively, one user program might initiate several transactions.

3.2.2. Nested Transactions in DBSs

A transaction, as presented above, is a set of primitive atomic actions abstracted as read and write operations. Each transaction is independent of all other transactions. In practice, there is a need to compose several transactions into one unit (i.e., one transaction) for two reasons: (1) to provide modularity; and (2) to provide finer grained recovery [Moss 85]. One way to compose transactions is gluing together the primitive actions of all the transactions by concatenating the transactions in sequence into one big transaction. This preserves consistency but decreases concurrency and prevents fine grained recovery because the resulting transaction is really a serial ordering of the subtransactions. What is needed is a composition that allows for the interleaving of the actions of the transactions to provide concurrent behavior, and at the same time, guarantees that the execution of the composition of transactions is carried out as a transaction in its own right.

The idea of nested spheres of control, which is the origin of the nested transactions concept, was first introduced by Davies [Davies 73] and expanded by Bjork [Bjork 73]. Reed presented a comprehensive solution to the problem of composing transactions by formulating the concept of nested transactions [Reed 78]. Reed defined a nested transaction to be a composition of a set of subtransactions; each subtransaction can itself be a nested transaction. To other transactions, only the top-level nested transaction is visible and appears as a normal atomic transaction. Internally, however, subtransactions are run concurrently and their actions are synchronized by an internal concurrency control mechanism. In Reed's design, timestamp ordering is used to synchronize the concurrent actions of subtransactions within a nested transaction. Moss designed a nested transaction system that uses locking for synchronization [Moss 85].

As far as concurrency is concerned, the nested transaction model mentioned above does not change the meaning of transactions (in terms of being atomic). The only advantage of nesting, in addition to modularity and composition of transactional abstractions, is performance improvement because of the possibility of increasing concurrency at the subtransaction level, especially in a multiprocessor system. We use the notion of nested transactions to model rule chains, as will be explained shortly.

3.3. A Database Model for RBDE

In RBDE, users interact with the system by requesting commands. A user command corresponds to either a built-in command or a rule. Built-in commands are executed by the CE (command executor) whereas rules are executed by the RP. The RP might initiate backward chaining before executing a rule and forward chaining after executing the rule. From the point of view of the user, the whole chain represents the execution of the requested command.

3.3.1. Database Access Units

From the point of view of the project database, the execution of an individual rule or a built-in command can be abstracted as a set of requests to access objects in the database in order to read the values of their attributes or change these values. To be more formal, accessing an object can be defined as either a read operation or a write operation as follows:

> **Definition 1:** A ***read operation***, `Read[o.att]`, returns the value stored in the `att` attribute of the object `o` in the project database.
>
> **Definition 2:** A ***write operation***, `Write[o.att, val]`, changes the value of the `att` attribute of object `o` to `val`.

A ***database operation*** is then defined as either a read operation or a write operation. Given this definition, we can abstract each of the five built-in commands we described earlier (e.g., `add`, `delete`, `move`, `copy`, and `link`) as a set of database operations. Similarly, each of the three parts of a rule can be abstracted as a set of database operations. For example, the condition of a rule is comprised of a set of read operations, and each of the effects is a set of write operations. The activity of an activation rule typically involves both read and write operations.

The OMS (object management system) of RBDE, which is responsible for storing and managing objects, executes each database operation *atomically*. Atomicity means that operations are either executed sequentially (i.e., one at a time) or that they appear as if they were executed sequentially.

Atomicity of operations does not guarantee the correctness of a rule's execution. In particular, the definition of rules in EMSL assumes that each of the three parts of a rule will be executed atomically. This leads us to the definition of a database access unit:

> **Definition 3:** A database ***access unit*** `U` is a set of database operations and an ordering relation $<$, where
>
> 1. $U \subset \{$ *Read*[*o.att*], *Write*[*o.att*,*val*] | *o* is an object in the database and *att* is an attribute of *o*$\}$, and
>
> 2. *Read*[*o.att*], *Write*[*o.att*,*val*] $\in U \rightarrow$ (*Read*[*o.att*] $<$ *Write*[*o.att*,*val*]) $\vee$ (*Write*[*o.att*,*val*] $<$ *Read*[*o.att*])

Each access unit is performed atomically.

Condition (1) says that an access unit is a set of read and write operations. Condition (2) says that the set of operations in an access unit is executed in some order. In our model, the only possible ordering is a *serial ordering*, where the set of operations of an access unit are executed in the order they are written.

Given this definition, we can abstract a built-in command as a single database access unit. Inference rules can also be abstracted as a single access unit consisting of all the read operations needed to evaluate the bindings and predicates of the condition and all the write operations performed when asserting the effect of the rule. Activation rules, in contrast, consist of exactly two access units: one consisting of all the database operations performed during the first phase of execution of the activation rule (i.e., before the activity is executed by a client), and the other consisting of all the database operations performed during the second phase of rule execution (i.e., to assert one of the effects of the rule). This means that an object's attribute whose value was read in the first unit could potentially change before the execution of the second unit.

3.3.2. Definition of Agent

Given the definition of an access unit, we can now abstract each user command as a set of access units. We term this abstraction an *agent*:

> **Definition 4:** An ***agent***, A, is a sequence of one or two database access units. An agent must appear as if it has been executed atomically. An agent consisting of *2* access units is denoted by the following expression:
>
> $A = U_1; U_2$, where U stands for "access unit".

This definition of agent can be used to abstract a built-in command or a rule from the point of view of the database. To illustrate, consider the `compile` rule shown in figure 3-3. The body of this compile rule is transformed by the RP into steps whose interpretation is equivalent to the agent shown in figure 3-4. All database operations are in **bold face**. As far as the database of RBDE is concerned, the execution of the `compile` rule involves only the database operations.

```
compile [?f:CFILE]:

  (bind (?h to_all HFILE suchthat (linkto [?f.includes ?h])))
  :
  (?f.status = NotCompiled)

  { compile ?f.contents ?h.contents "-g"
            output: ?f.object_code ?f.error_msg }

  (and (?f.status = Compiled)
       (?f.object_timestamp = CurrentTime));
  (?f.status = Error);
```

Figure 3-3: The Compile Rule

So far, we have treated each agent as if it were independent of all other agents. This is sufficient to model built-in commands and individual rules. However, to model a rule chain, there is a need to compose several agents into one unit in order to express that there is a causality relation between these agents, i.e., the execution of one agent is the reason for the execution of another. The notion of nested transactions in DBSs provides a nice "intuition" that we can use to compose multiple agents into one unit.

> **Definition 5:** A ***nested agent*** is a composition of a set of agents, each of which can itself be a nested agent. A nested agent must appear to have been executed atomically.

The top-level agent of a nested agent represents the original rule that initiated the rule chain. The nesting of agents does not affect the correctness criterion of an agent's execution. The execution of an agent, whether nested or not, produces correct results if all the access units comprising the agent appear to have been executed as an atomic unit. This is guaranteed if the server executes each agent to completion before starting the next agent. In the client/server model, the server executes multiple agents concurrently by interleaving the execution of agents at the granularity of access units. This can result in interference between these agents if their access units include operations that access overlapping sets of objects in the database. This is the essence of the concurrency control problem in RBDE.

```
Begin execution of compile rule

   /* FIRST ACCESS UNIT */

   /* First bind the parameter to an object. */
   ?f := object O;

   /* The binding part. */
   ?h := Read [O.includes];

   /* Evaluating the property list. */
   val := Read [O.status];
   If val <> NotCompiled Then
      return UNSATISFIED;

   /* Executing the activity. */
   source := Read [O.contents];
   For each object, h, bound to ?h Do
     Begin
       temp := Read [h.contents];
       includes := union (includes, temp);
     End;
   a.out := Read [O.object_code];
   error := Read [O.error_msg];

   ret := compile(source, includes, ``-g'', a.out, error);

   /* SECOND ACCESS UNIT */

   Write [O.object_code];
   Write [O.error_msg];

   /* Asserting one of the effects. */

   If ret = 0 Then
     Begin
       Write [O.status, Compiled];
       time := CurrentTime();
       Write [O.object_timestamp, time];
     End;
   Else
       Write [O.status, Error];
End.
```

Figure 3-4: The Agent Representing the Compile Rule

3.4. A Transaction Manager for RBDE

We model the execution of an agent in terms of a transaction. These transactions are created, managed and terminated by a transaction manager (TM). The RP and the CE must tell the TM when they are about to begin executing a command, so that the TM can start a transaction to encapsulate the execution of all the database operations of the command. When the command execution layer is done with the execution of a command, it must inform the TM so that the TM can terminate the corresponding transaction. The TM provides three transaction operations: `begin`, `commit`, and `abort`, which form the interface between the TM and the command execution layer (either the RP or the CE).

3.4.1. Transaction Operations

The command execution layer indicates to the TM that it is starting to execute an agent by issuing the `begin` operation. The TM creates a new transaction to encapsulate the execution of the agent in the database. Every database operation involved in the execution of the agent must be issued through the TM on behalf of the transaction encapsulating the agent. To identify transactions, the TM assigns every transaction a unique identifier at the time of the transaction's creation. The command execution layer must attach a transaction's unique identifier to every database operation belonging to the transaction.

The command execution layer indicates to the TM the termination of an agent's execution by issuing either the `abort` operation or the `commit` operation. By issuing a `commit` operation, the command execution layer tells the TM that the agent's execution has terminated normally. The `abort` operation, in contrast, prematurely ends a transaction and undoes all the steps that were already executed in the transaction. Undoing the steps of a transaction is called *rolling back* the transaction.

When the TM creates a new transaction (after receiving the `begin` operation), it creates a log file for the transaction, where all the steps (i.e., the database operations) of the transaction are recorded. For each database operation that accesses a status attribute, the

TM records the value of the attribute before and after the operation. A composite operation that accesses structure attributes, such as deleting a composite object, is recorded as a set of flat operations (operations that do not have suboperations). For example, deleting a MODULE object that contains three CFILE objects (as members of its cfiles attribute, for example) is recorded as four delete operations (one for each of the three CFILEs in addition to the delete operation on the MODULE object). The log file provides a mechanism for rolling back the transaction in case we want to undo its effect if the transaction is aborted.

Some database operations, such as editing the contents of a CFILE object, alter the contents of data attributes (i.e., text or binary files as described in section 2.3.3). In order to undo these operations, the TM must save copies of the data attributes of objects manipulated by the transaction before these attributes are changed (while executing the activity of a rule). Then, if the transaction must be rolled back, the contents of the data attributes before the transaction started can be reverted back to the saved copies. The copies are discarded once the transaction commits.

Given the three transaction operations, an agent's execution is defined, from the database's viewpoint, by a begin transaction operation, followed by a serial execution of a set of database operations, followed by either a commit or an abort. This is the reason we say that a transaction *encapsulates* an agent's execution. From the point of view of an RBDE user, a transaction is the execution of a user command. As explained in chapter 2, the execution of a user command may involve executing a rule chain. In this case, the "body" of the nested agent (i.e., the database operations comprising its execution) representing the chain is composed by the RP as it goes along. This is different from most traditional DBSs, where a transaction is the execution of a program, whose body is written before executing the transaction.

To be more formal, we introduce some notation that will be used in subsequent chapters. Our notation is similar to the one used by Bernstein *et al.* in [Bernstein et al. 87]. We use T_i to mean a transaction whose unique identifier is i. A read (or write) operation on object o issued by transaction T_i is denoted by $Read_i$[o.att] (or $Write_i$[o.att, val]). We use the letters P and Q to denote an arbitrary database operation (i.e., either

read or write). For example, `P`$_i$`[o.att]` stands for an operation `P` on object `o`, issued by transaction T_i. Note that when it is irrelevant which attribute of an object is accessed by an operation, we drop the specific attribute and simply use `P`$_i$`[o]` to denote an operation `P` on object `o`.

We define a transaction in RBDE as follows:

Definition 6: A ***transaction*** is a 6-tuple $T = (i, U_i, S, c, u, t)$, where

1. i is the unique identifier of the transaction,
2. U_i is the set of access units belonging to T. We use $U_{i,1}$ (or $U_{i,2}$) to denote the first (or second) access unit of T. Access units of a transaction are executed in a serial order,
3. S is the set of subtransactions (possibly empty) of T,
4. c is the command whose execution is encapsulated by T; c is either a built-in command or a rule,
5. u is the user who requested the command, leading to the creation of T; we say that u is the *owner* of T, and
6. t is the unique timestamp that the TM assigned to T.

Each of the elements of the tuple is called an *attribute* of the transaction. We use T_i as a shorthand notation for a transaction when we are concerned only with identifying a transaction. In subsequent chapters, we add three more attributes of transactions to capture more of the semantics available in RBDE.

Going back to our example above of the agent abstracting the `compile` rule (in figure 3-4), say that Mary requests a command that fires the `compile` rule on object `main.c`. Recall that `main.c` is linked to two `HFILE` objects, `i1.h` and `i2.h`. The transaction encapsulating the agent for this execution of the compile rule is shown in figure 3-5. Since we are not concerned with the actual values read or written from/to the attributes of objects, we do not show these values in the database operations.

As explained earlier, the server interleaves the execution of agents at the granularity of access units. Thus, the TM will also interleave the execution of transactions at the

```
/* Note: all computational code is omitted */

id := begin()

  /* First Access unit of transaction. */

  Read_id[main.c.includes]
  Read_id[main.c.status]
  Read_id[main.c.contents]
  Read_id[i1.h.contents]
  Read_id[i2.h.contents]
  Read_id[main.c.object_code]
  Read_id[main.c.error_msg]

  /* Second Access unit. */

  Write_id[main.c.object_code]
  Write_id[main.c.error_msg]
  Write_id[main.c.status]
  Write_id[main.c.object_timestamp]

commit(id);
```

Figure 3-5: The Transaction Encapsulating the Compile Agent of Figure 3-4

granularity of access units. The concurrency control problem then reduces to detecting interference between concurrent transactions. Before defining the problem more precisely, we overview the concept of *serializability* in database systems, and then define the problem in terms of that concept.

3.5. Background: Serializability

The concurrency control problem has been studied extensively in traditional DBSs. In DBSs, the problem arises when two or more transactions are executed concurrently, causing their database operations to be executed in an interleaved fashion. The interleaving of database operations from concurrent transactions results in a sequence of actions from both transactions, called an *execution* (or a schedule). An execution that gives each transaction a consistent view of the state of the database is considered a *consistent* execution. Consistent executions are a result of synchronizing the concurrent operations of transactions by allowing only those operations that maintain consistency to be interleaved.

An execution is guaranteed to be consistent if the transactions comprising the execution are executed serially. In other words, an execution consisting of transactions T_1, T_2, ..., T_n is consistent if for every i= 1 to n-1, transaction T_i is executed to completion before transaction T_{i+1} begins. We can then establish that a serializable execution, one that is computationally equivalent to a serial execution, is also consistent. Two executions E_1 and E_2 are said to be computationally equivalent if [Korth and Silberschatz 86]:

1. The set of transactions that participate in E_1 and E_2 are the same.
2. For each data item O in E_1, if transaction T_i executes read(O) and the value of O read by T_i is written by T_j, then the same will hold in E_2 (i.e., read-write synchronization).
3. For each data item O in E_1, if transaction T_i executes write(O) before T_j executes write(O), then the same will hold in E_2 (i.e., write-write synchronization).

The consistency problem in conventional database systems reduces to that of testing for serializable executions because it is accepted that the consistency constraints are unknown. Even though a DBS may not have any information about application-specific consistency constraints, it can guarantee consistency by allowing only serializable executions of concurrent transactions. This concept of serializability is central to all traditional concurrency control mechanisms. For further information on serializability, the reader is referred to [Bernstein et al. 87] and [Papadimitriou 86].

3.6. The Concurrency Control Problem in RBDE

We use the concept of serializability to state the concurrency control problem in RBDE more precisely. Interference cannot occur in the single-agent execution model because the set of access units that comprise each transaction are performed as an atomic unit. In the multi-agent RBDE model, however, the RP interleaves the execution of multiple agents (nested or flat) at the granularity of access units. Corresponding to this interleaved execution of agents is a transaction schedule (or execution) in which the execution of the access units of multiple concurrent transactions (encapsulating the agents) are interleaved.

A schedule involving a set of transactions, T_1 ... T_n, is said to be serial if for each transaction, T_i, all the access units of T_i are executed before executing any of the access units belonging to T_{i+1}. Serial schedules guarantee that transactions, and therefore the agents they encapsulate, are executed atomically; thus serial schedules cannot cause any interference.

Enforcing serial schedules, however, would revert the multi-agent model into a single-agent model. To avoid that, the multi-agent RBDE allows *concurrent* schedules. A concurrent schedule is one in which the server executes an access unit from an agent, A_i, followed by one or more access units of another agent, A_j, and then a second access unit from A_i, and so on. Note that a concurrent schedule can also involve interleaving of access units of more than two transactions. A concurrent schedule cannot cause interference if it is equivalent to a serial schedule. We use the same definition of computational equivalence given above in section 3.5.

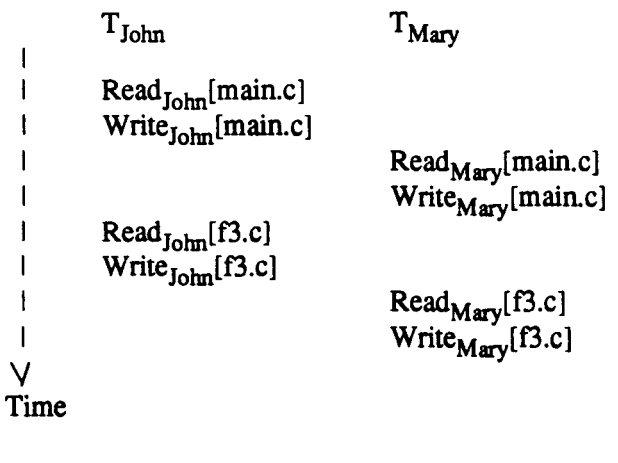

Figure 3-6: Serializable Schedule of Two Transactions in RBDE

To illustrate, assume that the two developers, John and Mary, are using the same RBDE to work on a project. In their client processes, both John and Mary issue user commands, which cause two concurrent agents, A_{John} and A_{Mary}, to execute in the server. The TM starts two transactions to encapsulate the execution of each of the two agents, T_{John} and T_{Mary}, respectively. The schedule of the execution of the two concurrent transactions shown in figure 3-6 is computationally equivalent to the serial schedule in

which T_{John} is executed to completion followed by T_{Mary}. Note that `main.c` is the name of an object and does not refer to attribute `c` of object `main`; we have dropped reference to which attribute is read or written, since it is irrelevant as far as the TM is concerned.

The two schedules, the one shown in figure 3-6 and the serial schedule T_{John}; T_{Mary}, meet the three criteria of equivalence listed above because: (1) both schedules involve the same transactions, T_{John} and T_{Mary}; (2) both of the objects read by T_{Mary}, `main.c` and `f3.c`, are written by T_{John} in both schedules; and (3) T_{Mary} executes both `write(main.c)` and `write(f3.c)` after T_{John} in both schedules.

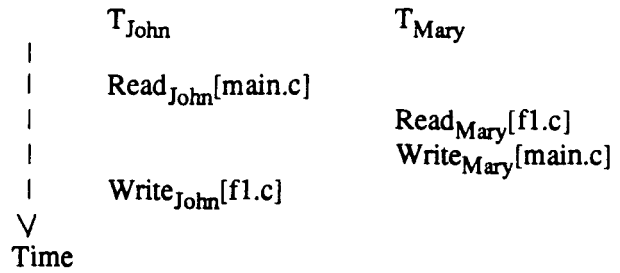

Figure 3-7: Schedule Showing Interference Between Two Transactions

The only kind of concurrent schedules that cause interference are non-serializable schedules, i.e., schedules that interleave the execution of database operations of multiple transactions in a fashion that cannot be done in a serial schedule. For example, consider the schedule depicted in figure 3-7. The interleaving of the database operations in this schedule cannot be produced by any serial schedule.

In general, given a schedule consisting of n transactions, $T_1, \ldots, T_n$, where each transaction, T_i, consists of a set of access units, each of which is denoted by $U_{i,j}$, and given two objects, O_1 and O_2, interference is caused by the the execution of an access unit, $U_{i,m}$, belonging to a transaction, T_i, followed by the execution of a set of access units, $U_{j,k} \ldots U_{j,l}$, belonging to another transaction, T_j, followed by the execution of an access unit, $U_{i,n}$, belonging to T_i, such that:

1. $U_{i,m}$ contains the operation $\texttt{Read}_i[O_1]$, and $U_{i,n}$ contains the operation

$\texttt{Write}_i[O_2]$, AND at least one of $U_{j,k}$ ... $U_{j,l}$ contains the operation $\texttt{Read}_j[O_2]$, and at least one of $U_{j,k}$ ... $U_{j,l}$ contains the operation $\texttt{Write}_j[O_1]$, OR

2. $U_{i,m}$ contains the operation $\texttt{Write}_i[O_1]$ and $U_{i,n}$ contains the operation $\texttt{Read}_i[O_2]$, AND at least one of $U_{j,k}$ ... $U_{j,l}$ contains the operation $\texttt{Write}_j[O_2]$, and at least one of $U_{j,k}$ ... $U_{j,l}$ contains the operation $\texttt{Read}_j[O_1]$.

Instead of disallowing all concurrent schedules, what is needed is a mechanism that can detect interference, but instead of disallowing interference automatically, the mechanism should resolve it in a flexible manner. More specifically, the mechanism should use semantic information about the project in order to determine the most suitable resolution of a detected interference. Based on this, the concurrency control problem can be divided into three subproblems:

- **Conflict detection**: before performing an access unit in a transaction, the TM must decide whether or not any of the database operations in the access unit might cause any interference with concurrent transactions. This subproblem involves implementing a mechanism in TM that can detect interference. This subproblem reduces to detecting non-serializable schedules.

- **Default conflict resolution**: once interference is detected, RBDE should be able to resolve it based on the *consistency constraints* of the project's development process. These constraints must be explicitly defined and maintained. The problem here is finding an appropriate formalism for explicitly specifying the consistency constraints, and devising a conflict resolution protocol that can enforce these constraints directly by disallowing any interference (non-serializable interactions) between concurrent transactions that violate these constraints. Note that not all non-serializable interactions are disallowed, only those that would prevent the consistency constraints from being carried out.

- **Programmable conflict resolution**: in some projects, non-serializable interactions that violate the consistency constraints might be required in order to allow specific situations of cooperation between agents. We call this specification the *coordination model* of the project. The problem here is

providing a language for programming the coordination model, and building the runtime environment for such a language, which would provide a way for programming a specific concurrency control policy to override the default policy.

3.7. Requirements on our Solution

Given the flexible consistency maintenance requirements of SDEs, it would be desirable for the concurrency control mechanism to support several requirements. The following requirements were compiled from the literature on advanced database applications, of which SDEs is one class.

1. **Long-duration operations**: software development often involves long-duration operations, such as editing and compiling. The transactions, in which these operations may be embedded, are also long-lived. Long transactions need different support than short transactions. In particular, blocking a transaction until another commits is rarely acceptable for long transactions. Only under specific circumstances (e.g., waiting until a short-lived activity finishes) is blocking acceptable. Aborting a long transaction might lead to wasting a lot of work, and thus the mechanism should only abort transaction when it is absolutely necessary.

2. **User control**: In an SDE, users request the execution of activities and they view the results of operations performed by the SDE automatically. In addition, users might be responsible for performing tasks that are nondeterministic and interactive in nature, such as fixing a bug. The nondeterminism results from the fact that software developers decide what activities they will perform as they go along in their tasks; in RBDE, even the sequence of rules in a chain cannot be known in advance when rules have multiple possible effects. The unpredictable nature of activities implies that the concurrency control mechanisms will not be able to determine whether or not the execution of a transaction will violate database consistency, except by actually starting to execute the transaction. This might lead to situations in which the user might have invested many hours running a transaction, only to find out later when he wants to commit his

work that some of the operations he performed within the transaction violated some consistency constraints. The user would definitely oppose deleting all of his work (by rolling back the transaction). He might, however, be able to explicitly reverse the effects of some operations in order to regain consistency. Thus, there is a need to provide more user control over transactions.

3. **Synergistic cooperation**: Cooperation among developers has significant implications on concurrency control. In SDEs and other design environments, several users might have to exchange knowledge (i.e., share it collectively) in order to be able to complete their work. The activities of two or more developers working on shared objects may not be serializable. They may pass the shared objects back and forth in a way that cannot be accomplished by a serial schedule. Also, two users might be modifying two parts of the same object concurrently, with the intent of integrating these parts to create a new version of the object. In this case, they might need to look at each others' work to make sure that they are not modifying the two parts in a way that would make their integration difficult. This kind of sharing and exchanging knowledge was termed *synergistic interaction* by Yeh *et al.* To insist on serializable concurrency control in design environments might thus decrease concurrency or, more significantly, actually disallow desirable forms of cooperation among developers.

4. **Complex objects**: In advanced applications, data is often defined in multiple levels of granularity. For example, an object representing a program in a software project might consist of modules, each of which containing procedures and documentation. If a user wants to gain exclusive access to the whole program (perhaps to build the executable of the program), he has to make sure that every subobject is made unavailable to other users. In this case, it is convenient to be able to lock the entire nested object in one operation rather than a separate operation for each subobject. There is thus a need for supporting operations on units of varying granularity.

5. **Teamwork**: Large-scale development efforts often involve teams of developers. Each team is typically assigned a development task that re-

quires access to various resources (tools, code, documentation, etc.). In general, different teams of developers might require different concurrency control policies. In addition, it is usually the case that team members cooperate more closely among each other than with members of other teams. Because of these two reasons, it is desirable to be able to specify team-oriented concurrency control policies.

6. **Tailorability of the concurrency control policy**: Every project requires a specific concurrency control policy that suits the needs of its development process. Even within the development process of a single project, several different policies might be needed at different phases of the project or within different programming teams. Rather than building-in a fixed number of policies into the SDE, it would be more advantageous to support a separation between policies and mechanisms, and to provide a programmable framework for implementing policies.

We present our solution to the concurrency control problem in chapters 4 — 7. In chapter 8, we evaluate how well our solution meets the requirements listed above.

3.8. Summary

In this chapter, we presented a database-oriented model of command execution in RBDE. From the point of view of the database, a user command is a set of database access units, each of which consists of a set of read or write operations. Database operations that must be executed together (atomically) as a unit are grouped into access units. For example, all the database operations involved in executing either a built-in command or an inference rule are grouped in one access unit; the server actually performs each access unit atomically. The execution of an activation rule involves exactly two access units: one for the first phase of rule execution and the other for the second phase of rule execution; the server executes each of these units atomically, but it can interleave the execution of the sequence of two units with other access units.

The TM (transaction manager) creates a transaction, with a unique identifier and a timestamp, to encapsulate the execution of an agent by a `begin` operation and a `commit` (or `abort`) operation. A *transaction* is a 6-tuple $T = (i, U_i, S, c, u, t)$, where i is the unique

identifier of the transaction, U_i is the set of access units belonging to T, S is the set of subtransactions of T, c is the command whose execution is encapsulated by T, u is the owner of T, and t is the unique timestamp that the TM assigned to T.

In the multi-agent RBDE model the RP interleaves the execution of multiple agents at the granularity of access units. Corresponding to this interleaved execution of agents is a transaction schedule (or execution) in which the execution of the access units of multiple concurrent transactions (encapsulating the agents) are interleaved. The only kind of concurrent schedules that can cause interference are non-serializable schedules, i.e., schedules that interleave the execution of database operations of multiple transactions in a fashion that cannot be done in a serial schedule.

Based on this, the concurrency control problem can be divided into three subproblems: conflict detection, default conflict resolution, and programmable conflict resolution. In the next four chapters, we will present mechanisms that solve all three subproblems.

Chapter 4
Detecting Interference

In the previous chapter, we defined interference in terms of the interleaving of the execution of concurrent transactions in the server. In this chapter, we present the conflict detection module of the concurrency control mechanism, which is responsible for detecting interference between concurrent transactions. We first describe briefly the traditional 2PL mechanism, which is used by most traditional DBSs to control the execution of concurrent transactions. We then describe the 2PL mechanism used by the TM in RBDE, and explain the interface between the TM and the LM. The NGL protocol implemented by the LM is based on the multiple granularity locking (MGL) protocol used in some DBSs. We briefly describe the MGL protocol, and then present the details of the NGL protocol.

4.1. Related Work: Detecting Conflicts in DBSs

The concurrency control problem in traditional DBSs reduces to that of detecting violations of serializability. In a typical DBS, the transaction manager includes a *scheduler*, which controls the order of execution of concurrent transactions [Bernstein et al. 87]. Schedulers employ various mechanisms to ensure that they detect any violations of serializability. If such a violation is detected, the scheduler determines how to resolve it.

Schedulers in traditional DBSs employ mechanisms that follow one of four main approaches to detect conflicts: 2PL (the most popular example of locking schedulers), timestamp ordering, multiversion timestamp ordering, and optimistic concurrency control. Some mechanisms add multiple granularities of locking and nesting of transactions. We have chosen 2PL as a basis for our transaction manager because it is the best understood of the locking schedulers. We briefly describe 2PL and then proceed to present the 2PL mechanism employed by the TM in RBDE. A comprehensive discussion of the concurrency control problem in DBSs can be found in [Bernstein et al. 87].

4.1.1. Locking Mechanisms

The 2PL introduced by Eswaran *et al.* is now accepted as the standard solution to the concurrency control problem in conventional DBSs. 2PL guarantees serializability in a centralized database when transactions are executed concurrently. Before a transaction can access an object it must first obtain a lock on the object. A lock controls access to an object. If a transaction acquires a lock on an object, then other transactions are prevented from acquiring locks on the same object. The mechanism depends on *well-formed transactions*, which (1) do not relock entities that have been locked earlier in the transaction, and (2) are divided into a growing phase, in which locks are only acquired, and a shrinking phase, in which locks are only released [Eswaran et al. 76]. During the shrinking phase, a transaction is prohibited from acquiring locks. If a transaction tries during its growing phase to acquire a lock that has already been acquired by another transaction, it is forced to wait until the transaction that holds the lock releases it. Forcing a transaction to wait is called *blocking*. Blocking transactions until locks are released can result in deadlock if transactions are mutually waiting for each other's resources.

2PL serves two purposes: to detect interference that causes locking conflicts and to resolve this interference by blocking the transaction that requested the conflicting lock. In our TM, we distinguish the two purposes of the scheduler, detecting interference and resolving it. We consider the conflict detection component of the scheduler to be a part of the TM. The TM employs a 2PL mechanism to detect interference but not to resolve it. The conflict resolution part of the scheduler, which we refer to as the *Scheduler*, implements two semantics-based protocols to resolve conflicts. In the rest of this chapter, we present the conflict-detection part of the TM; we leave the details of the Scheduler to the next two chapters.

4.2. A Two-Phase Locking Mechanism for RBDE

The TM in RBDE employs a locking mechanism similar to 2PL as a basis for detecting interference between concurrent transactions. Before executing any access units belonging to a transaction, the TM first obtains locks on all the objects that will be accessed during the transaction's execution. The rule processor determines the set of objects that will be accessed by a rule and passes this set to the TM. The locks are acquired by requesting them from the LM, which grants the locks on the specified objects to the requesting transaction only if none of the locks is incompatible with other locks currently held on the objects. The LM uses a locking protocol called NGL to detect lock conflicts; the details of NGL will be presented later in this chapter.

The execution of the access units comprising a transaction does not begin until all the necessary locks have been acquired[9]. The locks are released only when the transaction's execution completes. If the transaction is part of a nested transaction, then instead of releasing the locks, they are transferred to the parent transaction. The locks held by a nested transaction are released only when the execution of all the subtransactions and the top-level transaction of a nested transaction is completed.

If a locking conflict is detected by the NGL protocol, information about this conflict is passed to the Scheduler, which is part of the conflict resolution module we will present in chapters 5 and 6. The interaction between the TM, the LM, and the Scheduler is depicted in figure 4-1. In the rest of this section, we describe the details of the TM. In the next section, we present the LM and the NGL conflict detection protocol.

4.2.1. The Basic 2PL Algorithm in RBDE

We first discuss the 2PL mechanism we have devised for RBDE assuming that there are only flat transactions (no nesting). Incorporating nested transactions adds several complications. It is simpler if we discuss our mechanism without these complications first. Later on, we extend our mechanism to incorporate nested transactions.

---

[9]This is not exactly true but, as will be explained later, the effect of what is really implemented is the same as if all locks were acquired before executing an access unit.

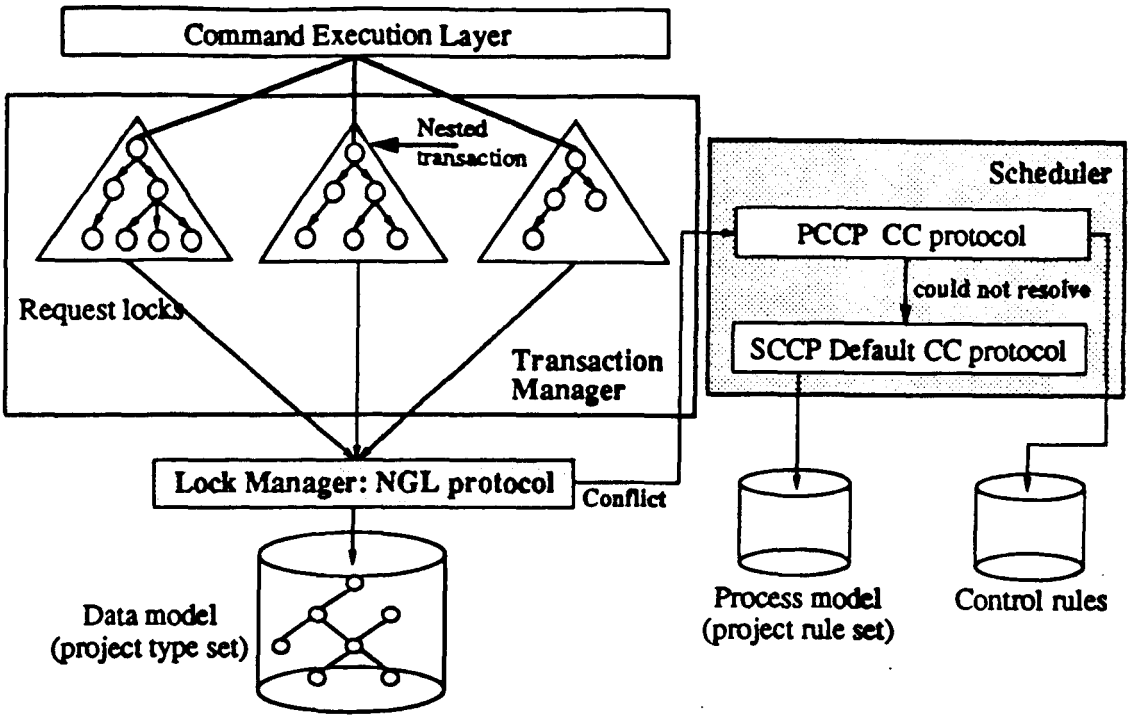

Figure 4-1: Interface Between the TM, the LM, and the Scheduler

The body of a transaction in RBDE (as explained in chapter 3) is composed of either one or two access units; each access unit consists of a set of read and write operations. Before allowing a transaction to execute an access unit, the TM must first acquire appropriate locks on behalf of the transaction on all the objects accessed by the operations that comprise the access unit. The TM maintains the list of locks held by each active (i.e., that has not been terminated) transaction in the server. The list of locks held by a transaction is called the *lock set* of the transaction.

Since the only two operations are read and write, we associate two types of locks with objects in the database: read locks, denoted by `R`, and write locks, denoted by `W`. Later on, we introduce additional types of locks when we present the granularity locking protocol.

We use $\texttt{Rl}_i\texttt{[o]}$ (or $\texttt{Wl}_i\texttt{[o]}$) to indicate that T_i holds a read (or write) lock on object `o`.

We use the two letters M and N to denote arbitrary lock types (for now, either W or R, but later on we will have more lock types). For example, $Ml_i[o]$ denotes that a transaction T_i holds a lock of type M on object o. Two locks $Ml_i[o]$ and $Nl_j[o]$ *conflict* if $i \neq j$ and the two lock types M and N are incompatible. In this case, we say that transactions T_i and T_j have *interfered* with each other.

We also use $Rl_i[o]$ (or $Wl_i[o]$) to denote a *request* by transaction T_i to obtain a read (or write) lock on object o. It will be clear from the context whether we are using $Rl_i[o]$ and $Wl_i[o]$ to denote locks or requests to obtain locks. We use $Ul_i[o]$ to denote the operation by which T_i requests *unlocking* or *releasing* its lock (either type of lock) on o. A transaction can hold at most one lock on an object. To incorporate the lock set of a transaction in the notation we developed in chapter 3, we extend a transaction to be a 7-tuple (rather than a sextuple as before) $T = (i, U_i, S, c, u, t, l)$, where l is the lock set of the transaction.

To execute an access unit, the transaction must request the operations in the access unit from the TM. The TM then requests the locks necessary for executing the operations on behalf of the transaction from the LM. The TM releases all of the transaction's locks once the transaction aborts or commits. The TM uses a 2PL mechanism to decide when to lock and unlock objects. The basic algorithm of the 2PL mechanism is as follows:

1. When the TM receives an access unit, U_i, from a transaction T_i, the TM preprocesses U_i and translates it into a set of lock requests; each operation $Q_i[o]$ (where Q is either Read or Write) is translated to a lock request, $Ml_i[o]$, where M is the type of lock required by the operation Q.
2. The lock requests are passed to the LM, one by one. The conflict detection protocol in the LM decides whether a lock request should be accepted or rejected. The details of the LM are presented later in this chapter.
3. If the LM grants all of the lock requests for U_i, the TM informs the transaction that it can go ahead and execute the operations in U_i. Otherwise if a lock request is rejected, the TM passes on the information returned by the LM to the Scheduler (which implements two different protocols, presented in chapters 5 and 6), which decides how to proceed.

4. Depending on what the Scheduler decides, the TM might have to either inform the transaction that it can go ahead and perform the operations in U_i, `abort` the transaction, or `commit` the transaction. The three options of the Scheduler will be discussed in chapter 5. Note that the default concurrency control policy in RBDE never blocks a transaction, thus avoiding deadlock.

5. Once the TM has obtained a lock on an object, the lock cannot be released until the transaction `commits` or `aborts` (releasing the locks at the end of a transaction is known as *strict* 2PL).

Unlike the traditional 2PL mechanism, the Scheduler might ask the TM to abort a transaction, which might lead to rolling it back; rolling back a transaction is done in the traditional 2PL mechanism only in the case of deadlock or a failure. Rollback is necessary when a transaction that has already executed some write operations is aborted. To decrease the probability of rollback, we require each individual transaction (i.e., each subtransaction within a nested transaction) to acquire all of its locks before executing any write operations. If the LM detects any locking conflict, and the Scheduler decides that the requesting transaction must be aborted, the transaction would not have executed any write operations yet, and thus rollback is not needed in this case. The only case where rollback is still required is when aborting one subtransaction in a nested transaction requires aborting, and thus rolling back, some of the ancestor transactions, which have finished executing all of their operations.

4.2.2. Interface Between the Rule Processor and the TM

In our 2PL algorithm, we assume that the TM receives access units belonging to a transaction one by one. This is sufficient for executing both built-in commands and inference rules atomically, since they are comprised of exactly one access unit. Transactions encapsulating activation rules, however, consist of two access units, which might not be executed one immediately after the other. If the TM obtains locks on a unit by unit basis, it could happen that after the first access unit, representing the condition, is performed, a locking conflict is detected by the LM while trying to acquire locks needed for the second access unit, representing the activity and the effect of an activation rule. In

this case, if the Scheduler decides that the transaction must be aborted (i.e., rolled back), the human user could have possibly wasted a lot of time executing the activity, only to discover that it must be rolled back.

To avoid rolling back the activity of an activation rule, we modify the interface between the RP (rule processor) and the TM so that the RP submits the access unit representing the effects immediately after evaluating the property list of the rule (if the condition is satisfied). Thus, the TM must obtain all the locks necessary for all the write operations of a transaction before the RP can proceed with executing the activity of the rule and asserting one of the effects (i.e., before the rule can execute any write operation). This guarantees that all locking conflicts are detected before the execution of a rule's activity, thus decreasing the possibility of having to roll back the activity of a rule. Rolling back activities, however, might still be necessary under certain circumstances, which we discuss in chapters 5 and 6. The Scheduler might decide to abort a transaction (including rolling back an activity) either while the activity of the rule encapsulated in the transaction is being executed, or even after the RP has finished executing the rule.

4.2.3. Determining Lock Types

The only issue remaining is how to decide what types of locks are needed for each access unit. The types of locks needed for the access unit that comprises the agent representing a built-in command are pre-defined by the command executor. For example, the `copy` built-in command requires a write lock on the destination object (since it will write it) and a read lock on the source object (since it only needs to read it in order to make a copy of it). The TM must obtain both of these locks before executing the `copy` operation. In order to determine the types of locks needed for the execution of a rule, we must define the write set and a read set of a rule. The TM must obtain write (or read) locks on every object in the write (or read) set of a rule.

The objects bound to the variables of a rule, including the parameters, are used during the evaluation of the property list of the condition. Since the evaluation of the property list involves reading the values of attributes of the objects bound to the variables, these objects comprise the read set of a rule. The write set is a subset of the read set. To be more precise, we define the read and write sets of a rule as follows:

Definition 1: The ***read set*** of a rule is the set of all objects bound to the variables of the rule, including the parameters.

Definition 2: The ***write set*** of a rule is the set of all objects bound to the variables used either as output arguments in the activity or in the left hand side of any assignment predicate in the effects of the rule.

Given that the RP can determine the read and write sets of a rule, the first phase of the rule execution algorithm, presented in chapter 2 (figure 2-19), should be revised to take into consideration communication with the TM. When the RP begins the execution of an original rule, it should first issue a `begin` operation to the TM to create a new transaction. Once a transaction has been set up, the RP should get the read set of the rule, by binding all the variables in the binding part and collecting the object identities returned by the functions in the binding part in one set.

After forming the read set, the RP should request read permission from the TM on all the objects in the read set. The TM receives this request and attempts to obtain read locks on all of these objects from the LM. Only when these locks are granted can the RP start evaluating the property list of the rule. Once the property list is evaluated, and if it is satisfied, the RP should collect all the objects that will be written in either the activity or the effects of the rule, forming the write set. It then requests write permission on all the objects in the write set. The TM again receives this request and attempts to obtain write locks on all of these objects. If the locks are granted, the RP can continue executing the rule. If one of the objects in the write set has already been locked with a read lock, the transaction requests *upgrading* the lock to a write lock; the LM grants the upgrade permission only if the new lock does not conflict with other locks already held on the object by other transactions.

The reason we delay obtaining locks on the write set of a rule until after the condition has been evaluated and found to be satisfied is to avoid locking objects with `W` locks unnecessarily. More specifically, if the condition of a rule was found to be unsatisfied, the transaction encapsulating the rule does not acquire write locks on the write set of the rule until after backward chaining, if any, is completed and succeeds in satisfying the condition. Other transactions can gain read access to the objects in the write set of the rule while backward chaining is in progress. This basically allows other transactions that only need to read these objects to proceed without interference.

The revised algorithm for the first phase of rule execution is shown in figure 4-2. All requests from the TM are prefixed by TM_ in the algorithm. Note that the transaction started in the first phase is terminated in the first phase only in the case when the condition of the rule is UNSATISFIABLE. In all other cases, the transaction is terminated either by the TM (if the LM informs the TM that a conflict was detected and the Scheduler decides to abort the transaction) or in the second phase of rule execution.

The only revision to the algorithm for the second phase of rule execution, shown in figure 2-20 in chapter 2, is to add a `TM_commit` operation before returning CYCLE_COMPLETE or DONE. The revised algorithm is shown in figure 4-3.

4.2.4. The Phantom Problem

In the revised algorithm for the first phase of rule execution (figure 4-2), the RP processes the binding part of a rule (in order to get the read set of the rule) before acquiring any locks. One seemingly possible problem with that is the *phantom problem.* This problem occurs in dynamic databases [Bernstein et al. 87], where a data item can be added dynamically while transactions are being executed. Say that a transaction T_i needs to lock all objects that belong to class `A` until T_i commits. Say that a new object, `o`, that is an instance of `A` is added after T_i has finished acquiring all of its locks. `o` would have been locked by T_i had it existed before T_i started. Thus, T_i must lock `o` as soon as it has been created. In general, T_i must lock all the non-existent (i.e., phantom) instances of `A` that might be created while T_i is executing. The problem is usually solved through a technique called *index locking* (e.g., locking class `A`). The details of the technique are not relevant here, but the gist of it is that in addition to locking data items, a transaction must lock an index that controls adding or removing data items that would interfere with the transaction's execution.

In our database model, the phantom problem cannot occur while the RP is collecting objects to bind to variables because the server is centralized and serial (i.e., executes one access unit at a time). This means that while the RP is evaluating the condition of a rule, `r`, another agent cannot make any changes to the structural or link attributes of any object, or add (or delete) any objects to (from) the database (because this can be done only

```
routine START_RULE_EXECUTION (rule);
  Begin

     /* First, start a new transaction. */

     tx_id := TM_begin ();
     bind all the variables in the binding part;

     /* Get the read and write sets of the rule. */

     read_set := UNION of all objects bound to variables;
     status := TM_read (tx_id, read_set)

     /* If TM has either aborted or committed the trans. */
     If status = TERMINATED Then
        return DONE;

     /* If all locks are granted, evaluate the condition. */
     Evaluate the condition of rule;
     If condition is UNSATISFIABLE Then
       Begin
         TM_commit (tx_id);
         return DONE;
       End;

     If condition is FALSE Then
       Begin
         set state of rule to original_rule;
         push rule on Execution_Stack;
         ret_value := DO_BACKWARD_CHAIN (rule);
       End;
     Else If condition is TRUE Then
       Begin

         /* Obtain locks on the write set. */
         write_set := get the write set of the rule;
         status := TM_write(tx_id, write_set);
         If status = TERMINATED Then
            return DONE;

         If rule is an inference rule Then
           Begin
             chaining_list := ASSERT_EFFECT (0, rule);
             If chaining_list <> empty Then
                ret_value := DO_FORWARD_CHAIN (rule);
           End;
         Else
           Begin
             set state of rule to be forward_chain;
             push rule on Execution_Stack;
             ret_value := CONTINUE;
           End;

     return ret_value;
  End.
```

Figure 4-2: Revised First Phase of Rule Execution

```
routine CONTINUE_RULE_EXECUTION (rule : activation rule,
                                 which_effect : status code);
  Begin

     get the state of rule;
     If state = back_chain Then
       ret_value := CONTINUE_BACKWARD_CHAIN (rule, which_effect);
     Else
       Begin
         /* write locks are acquired during first phase.  */

         chaining_list := ASSERT_EFFECTS (which_effect,rule);
         If chaining_list <> empty Then
           ret_value := DO_FORWARD_CHAINING (rule);
         Else
           ret_value := CONTINUE;
       End;

     If ret_value = DONE or ret_value = CYCLE_COMPLETE Then
        Begin
          tx_id := id of trans. encapsulating rule;
          TM_commit (tx_id);
        End;
     return ret_value;
  End.
```

Figure 4-3: Revised Second Phase of Rule Execution

by a built-in command, whose execution cannot be concurrent with the evaluation of a rule's condition).

```
(bind ?h to_all HFILE)
:
(forall ?h):
(?h.status = Edited)
```

The phantom problem can occur only if one or more objects are added or deleted while the client is executing the activity of a rule. Consider, for example, the condition shown above, which is the condition of an activation rule, `r`; we do not show the other irrelevant parts of `r`. When evaluating the binding part, which is composed of one binding clause, the RP must collect all instances of the class `HFILE` and bind all of these instances to the variable `?h`. Suppose that at the time `r` was fired, there were only three instances of `HFILE`, `i1.h`, `i2.h`, and `i3.h`. Thus, `?h` will be bound to these three objects, all of which form the read set of the transaction, T_i, encapsulating the execution of `r`. Before evaluating the property list, T_i acquires read locks on the three objects.

After acquiring these locks, the RP proceeds to evaluate the property list. Say that all of `i1.h`, `i2.h`, and `i3.h` have their `status` attribute assigned to the value "Edited". Therefore, the condition will be satisfied.

Since `r`'s condition is satisfied, the RP will proceed to lock the write set of the rule and then send a message to the client to execute `r`'s activity. Say that while `r`'s activity is being executed, another client requests to `add` an instance of `HFILE`, `i4.h`. The server starts a new transaction, T_j, acquires a write lock on `i4.h` and adds `i4.h` to the class `HFILE`. T_j then commits, releasing the lock on `i4.h`. This schedule appears to be problematic because T_i assumed that there are only three instances of `HFILE` when in fact a fourth one was added before T_i finished. The schedule, however, is serializable since it is equivalent to T_i followed by T_j (recall that we are considering only flat transactions here). The reason why this schedule, and in fact any schedules involving a phantom object in RBDE, is serializable is because in our database model: (1) each access unit is actually performed atomically by the server; and (2) the write set of any agent is a subset of the read set, on which the transaction obtains read locks before any write operation is performed. The second condition is not true in traditional DBSs, and is in fact the source of the phantom problem (see, for example, p. 65 of [Bernstein et al. 87]).

4.2.5. Recoverability and Strictness

Our 2PL mechanism enforces strict schedules since transactions release their locks only when they terminate. A schedule is said to be *strict* if the execution of a $Write_i$`[o.att, val]` (or $Read_i$`[o.att]`) operation is delayed until all transactions that have previously written `o` have either committed or aborted. The TM in RBDE guarantees strictness because if a transaction T_i contains the write operation, $Write_i$`[o.att, val]`, no other transaction can read the value that this operation assigned to `o.att` or change the value of the `att` attribute of `o`, except after T_i has either committed or aborted.

Strictness subsumes another desirable property called *recoverability*. Schedules are not recoverable if one transaction T_i reads a data item that another transaction T_j has written, and then after T_i commits, T_j aborts. Since T_j aborted, its write operations must be

rolled back. But then, T_i would have read a wrong value, possibly producing the wrong result.

A schedule is recoverable if for each transaction, T_j, that commits, T_j's commit follows the commit of every transaction T_i that has *read from* T_j. A transaction, T_i, *reads from* another transaction, T_j, if (1) T_i reads a data item o after T_j has written into it; (2) T_j does not abort before T_i reads o; and (3) every transaction (if any) that writes o between the time T_j writes it and T_i reads it, aborts before T_j reads o.

In our model, since locks are held until the completion of a transaction, no transaction can read from another transaction. This guarantees that the commit of every transaction follows the commit of every transaction that has read from T_i (of which there are none). Thus, our mechanism guarantees recoverability of a transaction.

The transaction model we described so far deals only with simple agents and ignores the nested agents resulting from forward and backward chaining. We now extend the TM and the locking mechanism to handle nested agents.

4.2.6. Extending 2PL to Nested Transactions in RBDE

The 2PL locking mechanism presented above guarantees the atomicity of transactions by acquiring locks on all the objects involved in an access unit before executing the access unit, and by releasing all the locks held by a transaction only when the transaction either commits or aborts. Nested transactions (nested agents) complicate the model.

Unlike simple agents, where it is possible to collect the read and write sets of the agent before the agent executes any write operation (i.e., before it changes any object in the database), it is impossible to determine the actual read and write sets of locks needed for the execution of a nested agent. The reason is that the composition of a nested agent is made up dynamically by the rule chaining algorithms. As explained in section 2.4, chaining is based on the assertion of one of the effects of a rule, which, in the case of an activation rule, cannot be determined until the activity of the rule has been executed. The best that can be done is to compute the union of all the possible read and write sets of all subagents in a nested agent. Locking these sets instead of the actual sets, however, would be a very conservative policy.

We encapsulate the execution of a nested agent by a nested transaction. The execution of each rule in the rule chain represented by a nested agent is encapsulated by a subtransaction. Before the RP fires a rule during chaining, it asks the TM to create a new subtransaction. To create a subtransaction, the `begin` operation must be modified so that it takes as a parameter the transaction id of the parent transaction. The forward and backward chaining algorithms must be revised so that they issue a `begin(tx_id)`, where `tx_id` is the identifier of the transaction encapsulating the execution of the rule that initiated the chaining, before inserting a new rule on the execution stack. Upon receiving the operation `begin(tx_id)`, the TM creates a new subtransaction of T_{tx_id}. Thus, the forward and backward chaining algorithms of the RP dynamically create the subtransactions of a nested transaction as they go along.

The 2PL locking mechanism described above applies for each subtransaction as before. However, aborting or committing a transaction is more complicated than before. All subtransactions of a transaction must be aborted before aborting the transaction. Before a transaction can be committed, all of its subtransactions must have either been committed or aborted. When a transaction is committed, all of its locks are inherited by its parent transaction: the TM requests the LM to *transfer* the locks of a child transaction to its parent. To transfer a lock, $Ml_i[o]$, from transaction T_i to a transaction T_j, the LM releases $Ml_i[o]$ and sets a new lock, $Ml_j[o]$, on the same object, `o`, in one atomic operation.

Let us illustrate by going through a backward chaining example because it is the more complicated of the two kinds of chaining. In particular, backward chaining is composed of two phases: in the first phase, the RP inserts rules on the stack because their conditions are not satisfied, and in the second phase, the RP removes rules from the stack in order to execute their activities. Say a client requests a command corresponding to the rule `r1` from the server. The RP fires `r1` by requesting the `begin` operation from the TM, which creates a transaction T_1 to encapsulate the execution of `r1`. Suppose that the condition of `r1` is not satisfied; the RP inserts `r1` on the execution stack and initiates backward chaining, say by firing `r2`. The TM creates a subtransaction, T_2, of T_1 to encapsulate the execution of `r2`.

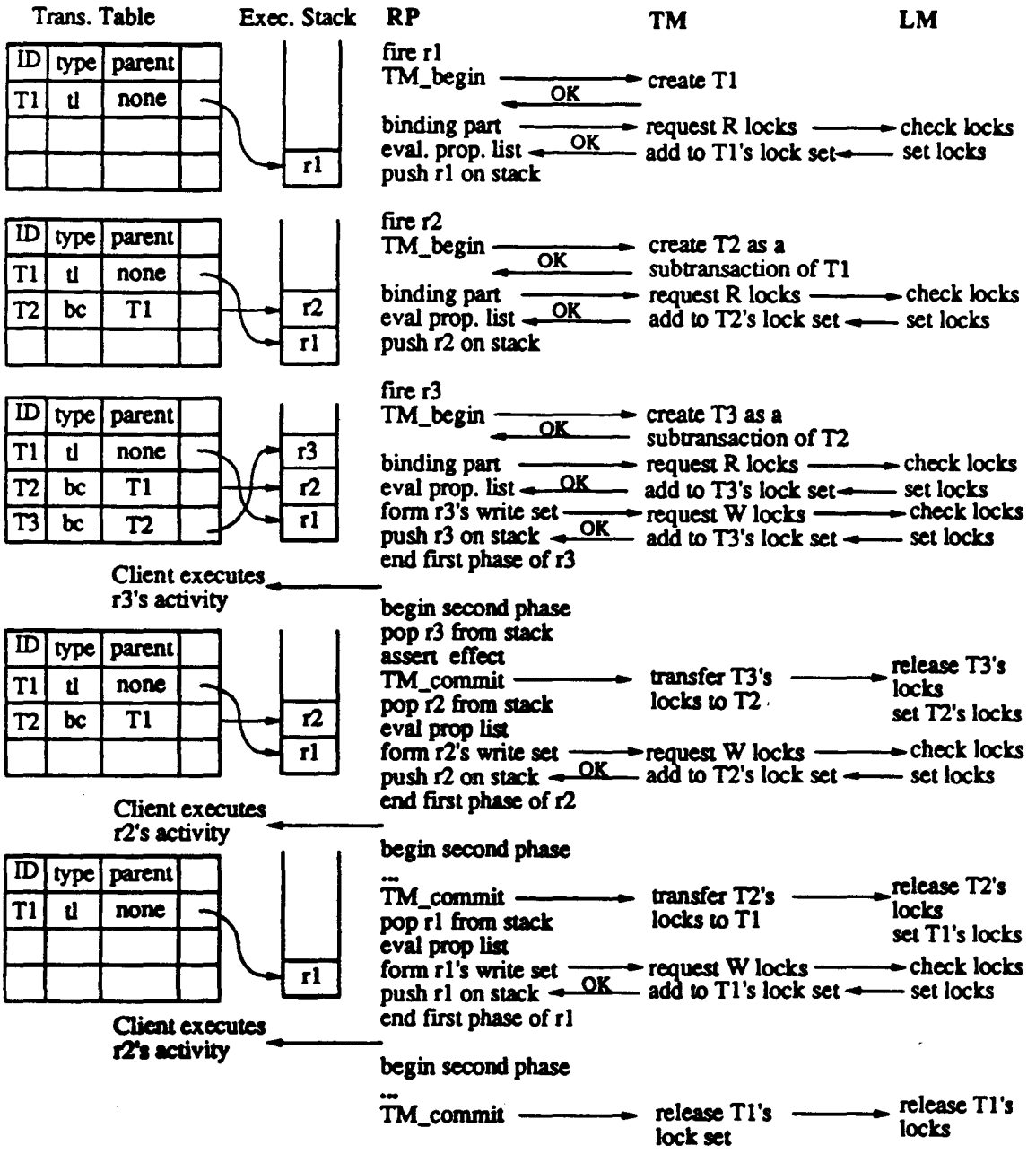

Figure 4-4: Execution of Three Rules by the RP, TM, LM, and the OMS

Suppose that r2's condition itself is not satisfied, so the RP inserts r2 on the stack and initiates further backward chaining, say by firing r3. Again T_3 is created as a subtransaction of T_2. Now say that r3's condition is satisfied. The RP can then proceed to execute the activity of r3, beginning the second phase of backward chaining. Once r3's activity has been executed and one of its effects asserted, the TM can commit T_3; T_2 inherits T_3's locks. If asserting one of r3's effects actually does make the condition of r2 satisfied, the RP pops r2 from the stack and executes the activity of r2. Finally if r2's execution makes the condition of r1 satisfied, the RP pops r1 from the stack and executes it. The complete sequence of events involving the RP, the TM, and the LM is shown in figure 4-4; the figure also shows the states of the transactions and the contents of the execution stack at various points.

During the first phase of backward chaining, T_1, T_2 and T_3 request only read locks (in order to evaluate the conditions of the corresponding rules). During the second phase, the transactions request write locks in order to execute the activities and assert the effects of the rules. Note that other transactions (outside the nested transaction) that need only read locks on objects accessed during the first phase of the backward chain can proceed concurrently with the first phase of backward chaining without interfering.

It will often be the case that rules fired during backward chaining access the same object, o. In this case, all of T_1, T_2 and T_3 will obtain read locks on o in the first phase. Then, in the second phase, T_3 will request a write lock on o, which is incompatible with the read locks held by T_1 and T_2. However, since T_3 is a descendant of both T_1 and T_2, the interference is not considered "serious". In order to avoid these kinds of conflicts, the TM transfers locks from parent transactions to subtransactions during the second phase of backward chaining. For instance, when T_3 requests a write lock on o, the TM releases the read locks that T_2 and T_1 hold on o. The rationale for this is that when T_3 commits, it will transfer all of its locks, including the write lock on o, to T_2 (and T_2 will transfer all of its locks to T_1). Thus, it is guaranteed that T_1 will retain its lock on o after T_3 and T_2 commit.

By inheriting the locks held by a subtransaction, the parent transaction guarantees that the child transaction's changes to objects will not be visible to other transactions outside

the nested transaction until the whole nested transaction commits. This is essential to guarantee correct execution of transactions. If a subtransaction makes its changes visible by writing these changes to the database and releasing its locks, then if the parent transaction aborts, the child must be aborted. But the child would have already released all of its locks so its operations cannot be rolled back. For this reason, committing a subtransaction transfers the locks that the subtransaction held to the parent transaction.

In chapter 5, we analyze the different kinds of rule chains a bit closer. The analysis results in a relaxation of some of the restrictions mentioned above on committing nested transactions. In particular, it will be shown that some transactions (those encapsulating activities whose execution is not affected by other activities in the same chain) can commit and release their locks before their subtransactions have committed; this leads to increased concurrency.

We have now completed the presentation of the TM of RBDE. As explained before, the TM works in conjunction with the LM to determine whether two transactions interfered with each other. We will now present the conflict detection protocol used by the LM to detect such conflicts.

4.3. The Lock Manager

The LM in RBDE is responsible for setting and releasing locks on objects when requested to do so by the TM. As explained earlier, the TM receives three kinds of lock-related requests from the command execution layer: a request to obtain read permission on the read set of a transaction, a request to obtain write permission on the write set of a transaction, or a request to commit a transaction, which entails releasing all the locks held by the transaction. To process a request to commit a transaction, the TM in turn asks the LM to release the lock set of the transaction.

4.3.1. Lock Operations

To handle the TM's requests, the LM maintains a lock table[10] and provides two operations, `lock` and `unlock`, which form the interface between the TM and the LM. The lock table has an entry for each locked object in the database. Each entry stores the locks currently held on a single object. A lock is a pair, <`type`, `tx-id`>, where `type` is the type of lock and `tx_id` is the identifier of the transaction that holds the lock. Several transactions can hold compatible locks on a single object. The Scheduler we present in chapter 6 might force the LM to set incompatible locks on the same object. For the rest of this chapter, however, we assume that if more than one lock is set on an object, then the locks must be compatible.

The `lock` operation is of the form `lock[tx_id, obj_list, type]`, where `tx_id` is the identifier of the transaction on whose behalf the TM is requesting the operation, `obj_list` is a list of objects, and `type` is the type of lock to set on all of the objects in `obj_list`. The `unlock` operation is of the form `unlock[tx_id, obj_list]`, where `obj_list` is a list of objects, on which T_{tx_id} currently holds locks.

To process the operation, `unlock [tx_id, obj_list]`, the LM simply goes through the `obj_list` and for each object, `o`, it removes the lock held by T_{tx_id} on `o` (regardless of the type of lock). The `lock` operation is more complicated. Assuming that there are only the two lock types presented earlier (read and write locks), the algorithm for processing the `lock` operation is shown in figure 4-5. The algorithm basically guarantees that if a `W` lock is set on an object, no other lock can be set on the object; multiple `R` locks can be set on the same object simultaneously.

The algorithm in figure 4-5 assumes that objects in the database are independent of each other. But as explained in section 2.3.2, each object in RBDE is a part of an object hierarchy. Accessing one object has implications on the objects above it and below it in the hierarchy.

---

[10]Note that the lock table must be persistent. In the current implementation, the locks are stored with the objects in the database rather than in a lock table.

```
routine LOCK (tx_id: transaction identifier;
              obj_list : list of objects to lock;
              type : either R or W);
  Begin
    For each object, O, in obj_list Do
      Begin
        If there are no locks held on O Then
          continue;

        If there is only one lock held on O Then
          Begin
            cur_id := id of transaction that holds the lock;
            cur_type := type of current lock;
            If type = W Then
              If tx_id <> cur_id Then
                return CONFLICT;

            /* Conflicts between a transaction and one of its
               ancestors are resolved by NGL protocol, shown
               in figure 4-7 below. */

            Else If type = R and cur_type = W Then
              If tx_id <> cur_id Then
                return CONFLICT;
              Else
                remove O from obj_list;
          End;

        Else if more than one lock are set on O Then
          If type = W Then
            return CONFLICT;
          Else If T_tx_id already holds a lock on O Then
            remove O from obj_list;
      End;

    /* If we reach here, then no conflicts were detected */

    For each object, O, in lock_list Do
       Add <tx_id, type> to O's entry in the lock table;

    return OK;
  End.
```

Figure 4-5: Setting Locks on Flat Objects

The protocol shown in figure 4-5 is correct, even when dealing with composite objects. To understand why, consider the object hierarchy shown in figure 4-6; this hierarchy is the same as the one presented in chapter 2. Say that John requests a command to `edit main.c`. To execute this command, the RP starts a transaction T_{John}, which obtains a write lock on `main.c`. While John is editing the `contents` attribute of `main.c` (a text file), Mary comes along and requests to `delete ModA`. To execute this built-in

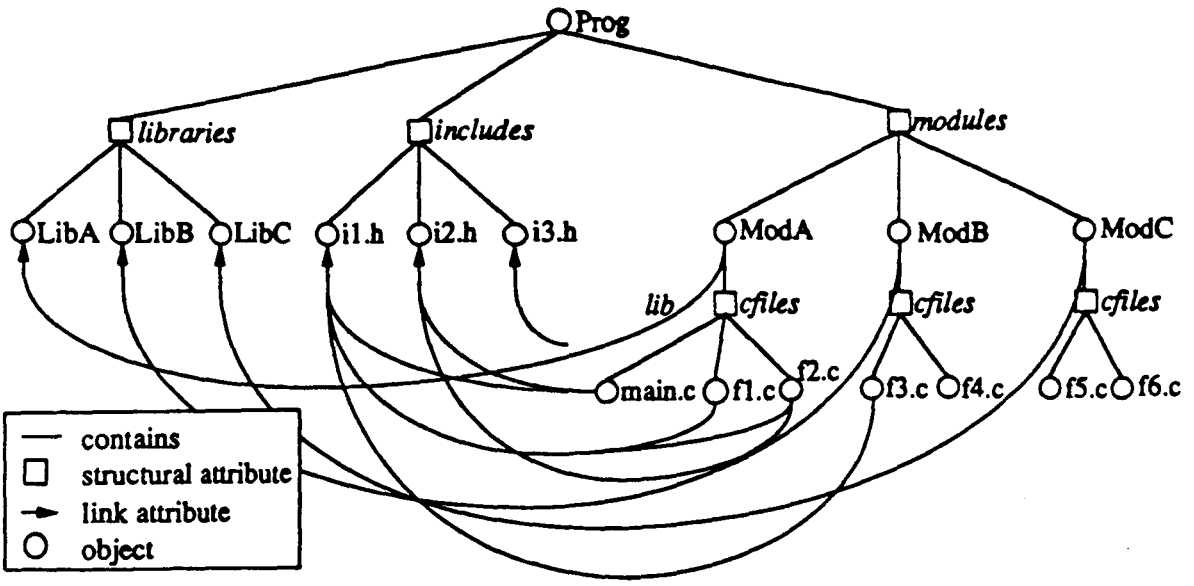

Figure 4-6: Composite Object Hierarchy

command, the CE (command executor) starts a transaction T_{Mary}. Since deleting `ModA` requires deleting all of its children first, T_{Mary} proceeds to obtain write locks on `main.c`, `f1.c` and `f2.c`. At this point T_{Mary}'s request to lock `main.c` will be rejected because it conflicts with T_{John}'s lock on `main.c`.

Given the locking conflict, Mary will not be able to delete `ModA`, which is the correct result. Now if we consider a much deeper hierarchy with many more objects, which is much more likely in a large-scale project, the protocol above will perform poorly. Transactions lock nodes explicitly in either `W` or `R` mode, which in turn locks descendants implicitly. Before granting a transaction a lock on an object, the TM would have to follow the path from the object to the root to find out if any other transaction has explicitly locked any of the ancestors of the object. This is clearly inefficient since it increases the overhead of locking and makes the performance of the LM unacceptable. In order to improve the performance of the protocol, we employ a nested granularity locking (NGL) protocol, which is based on the multiple granularity locking (MGL) protocol. We first describe MGL and then present the details of our NGL protocol.

4.3.2. Related Work: Multiple Granularity Locking

Gray *et al.* presented a *multiple granularity locking* (MGL) protocol that aims to minimize the number of locks used while accessing sets of objects in a database [Gray et al. 76]. In their model, Gray *et al.* organize data items in a tree where small items are nested within larger ones. Each non-leaf item represents the data associated with its descendants.

As in 2PL, there are two basic lock types, shared (equivalent to read) and exclusive (equivalent to write). Shared locks are denoted by `S`, while `X` is used to denote exclusive locks. In addition to `X` and `S` locks, Gray *et al.* introduced a third kind of lock mode called *intention* lock [Gray 78]. All the ancestors of a node must be locked in intention mode before an explicit lock can be put on the node. In particular, nodes can be locked in five different modes, `X`, `S`, `IX`, `IS`, and `SIX`.

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Table 4-1: Lock Compatibility Matrix for MGL protocol

A non-leaf node is locked in *intention-shared* (`IS`) mode to specify that descendant nodes will be explicitly locked in `S` mode. Similarly, an *intention-exclusive* (`IX`) lock implies that explicit locking will be done at a lower level in an `X` mode. A *shared and intention-exclusive* (`SIX`) lock on a non-leaf node implies that the whole subtree rooted at the node is being locked in `S` mode, and that explicit locking will be done at a lower level with `X` locks. `SIX` can be replaced by two explicit locks, `S` and `IX`, and is provided solely for convenience. A compatibility matrix for the five kinds of locks is shown in table 4-1. The matrix is used to determine when to grant lock requests and when to deny them.

Gray *et al.* defined the MGL protocol as follows:

1. A transaction T can lock a node in S or IS mode only if all ancestors of the node are locked in either IX or IS mode by T.
2. A transaction T can lock a node in X, SIX, or IX mode only if all the ancestors of the node are locked in either SIX or IX mode by T.
3. Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of the transaction, the transaction should not hold a lock on a node after releasing the locks on its ancestors.

The goal of the MGL protocol is to ensure that transactions never hold conflicting locks on the same object. The proof that it achieves this goal, which is not relevant here, is found in [Bernstein et al. 87]. This goal, however, is not sufficient to ensure serializability of transactions. To ensure serializability, MGL should be used in conjunction with a mechanism like 2PL. In this case, MGL replaces the flat read-write locking protocol described above for 2PL.

The granularity locking protocol employed by the LM in RBDE uses a very similar protocol as that used by MGL. We use a slightly different set of locks, and our compatibility matrix is also different from the one shown in table 4-1.

4.3.3. The NGL Protocol in RBDE

We call our protocol NGL, which stands for Nested Granularity Locking, because it detects locking conflicts on composite objects between nested transactions. We first explain the lock types supported by the LM in RBDE, and then present the NGL protocol.

4.3.3.1. Extended Lock Types

In the MGL protocol, if a transaction holds an X lock on an object, o, all other transactions are excluded from accessing any of the objects in the composite object hierarchy rooted at o. All write operations require X locks; thus, a write operation on an object essentially locks all of the object's subobjects. In RBDE, there is a distinction between write operations that change the value of a status or data attribute of an object, which do

not affect any of the subobjects, and write operations that manipulate structural attributes, which by definition affect the subobjects. We distinguish between these two kinds of operations by having a *write* lock (W), which is less exclusive than an *exclusive* (X) lock.

A W lock on an object indicates a write operation that will change only the non-structural attributes of the object. An X lock, on the other hand, indicates an operation that affects the whole subtree rooted at the object. For example, the `delete` operation on an object o requires an X lock on o, indicating that the operation will access every object in the subtree rooted at o. All write operations invoked by the execution of a rule, in contrast, only require W locks on the objects, since the database operations in the activity and effects of a rule can only change the non-structural attributes of objects.

Similarly, we distinguish between read operations that access the structural attributes and those that only access non-structural attributes. The `copy` operation, for example, copies a whole subtree rooted at an object o. An agent cannot copy an object o if any of the descendants of o is being written. It is possible, however, to copy an object if any of its descendants, or the object itself, are being read. The S lock in MGL serves exactly that purpose. The read operations in the property list and activities of rules, however, access only the non-structural attributes of objects. A read operation that accesses only the non-structural attributes of an object o should not conflict with a write operation on a subobject of o.

Thus, we need a lock that is incompatible with W and X, but compatible with S, IX, and IS. None of the locks introduced in MGL serves that purpose (note that S is incompatible with IX, as shown in the compatibility matrix in table 4-1). Consequently, we introduce another new lock to MGL locks, R. An S lock on an object, o, indicates that the whole subtree rooted at the object is being read, whereas an R lock indicates that only object o is being read.

In addition to introducing two new lock types, we have eliminated the SIX lock since it is not needed in our model (it is equivalent to setting an S and an IX lock on an object). Given the new lock types we introduced, we define the lock compatibility matrix shown in table 4-2, which is different from that of MGL.

	IS	R	IX	S	W	X
IS	yes	yes	yes	yes	yes	no
R	yes	yes	yes	yes	no	no
IX	yes	yes	yes	no	yes	no
S	yes	yes	no	yes	no	no
W	yes	no	yes	no	no	no
X	no	no	no	no	no	no

Table 4-2: Lock Compatibility Matrix for NGL Protocol

We define the lock types needed for the execution of the five built-in commands as follows. `Copy`, which makes a copy of an object (called the source) and inserts the copy as a child of another object (called the destination), requires an `S` lock on the source object and an `X` lock on the destination object. `Move` needs `X` locks on both the source and the destination objects. `Add` requires an `X` lock on the object to which we are adding a subobject; adding a top-level object (i.e., one that has no parent) does not require any locks[11]. Note, however, that this is a direct implication of the assumption we have made in chapter 2 that objects can be added only by the `add` built-in command, which is executed atomically. If this were not the case, adding a new root object would require locking the class of which the object is an instance. `Delete` requires an `X` lock on the object we are deleting. Finally, `link`, which creates a link from one object (the source) to another object (the destination), requires a `W` lock on the source object and no lock on the destination object. A read operation, $\texttt{Read}_i\texttt{[o]}$, in a rule requires an `R` lock on `o`. A write operation, $\texttt{Write}_i\texttt{[o]}$, requires a `W` lock on an object.

We define the NGL protocol as follows:

1. A transaction T can lock a node in `S`, `R` or `IS` mode only if all ancestors of the node are locked in either `IX` or `IS` mode by T.
2. A transaction T can lock a node in `X`, `W` or `IX` mode only if all the ancestors of the node are locked in `IX` mode by.

[11]This is true because our transaction mechanisms does not suffer from the phantom problem, as explained earlier in this chapter.

3. Locks are released only at the end of the transaction (in any order).

Note that the NGL algorithm is table driven: The project administrator can define an alternative lock compatibility matrix (and in fact define new lock types). In this case, the compatibility of two locks is determined according to the new table.

The complete algorithm implemented by NGL is shown in figure 4-7. NGL determines whether a transaction T_{tx_id} can set a lock of type `type` on each object in the list of objects, `obj_list`, passed to it. NGL collects all the ancestors of the objects in `obj_list`, and assigns the correct intention lock to each one. If the requested locks and the ancestor objects' locks do not conflict with any existing locks, NGL goes ahead and sets the new locks on the objects in `obj_list` and their ancestors. If a conflict is detected, then no locks are set and `CONFLICT` is returned.

The compatibility of two locks is determined by looking up the entry for the two locks types in the lock compatibility matrix shown in table 4-2. Note that if a transaction T_i requests a lock, $Nl_i[o]$ and there is already another lock, $Ml_i[o]$, held by T_i, NGL will set the new lock only if `N` is stronger than `M` and if $Nl_i[o]$ does not conflict with any other lock on `o` held by another transaction. The strength of locks in descending order is defined as follows: `X`, `W`, `S`, `R`, `IX`, `IS`. Note that these strengths can be redefined by the project administrator.

NGL guarantees that two different transactions, which do not have a parent-child relationship, will not hold incompatible locks on the same object. It also guarantees that each transaction will hold the strongest of all the locks its requested on an object, `o`, or any of the ancestors of `o`.

So far we have assumed that objects are the smallest "lockable" entities. This in fact limits concurrency because it causes NGL to detect interference between two transactions even if these two transactions are accessing different attributes of the object. This is not necessary. In order to increase concurrency, we take the granularity of locks a step further and apply it to attributes. Thus, an attribute is the smallest entity that can

```
routine NGL (tx_id: transaction identifier;
             obj_list : list of objects to lock;
             type : either S, R, W or X);
  Begin
    lock_list := empty;
    For each object, O, in obj_list Do
      Begin
        entry.obj := O;
        entry.lock := type;
        Add entry to lock_list;
      End;
    ancestor_type := intention lock corresponding to type;
    For each object, O, in obj_list Do
      Begin
        ancestors := all ancestors of O in object hierarchy;
        For each object, ancestor in ancestors Do
          If ancestor is not already on lock_list Then
            Begin
              entry.obj := ancestor;
              entry.lock := ancestor_type;
              add entry to lock_list;
            End;
      End;
    For each entry, lock, in lock_list Do
      Begin
        O := lock.obj;
        type := lock.type;
        If there are no locks held on O Then
          continue;
        If there is only one lock already set on O Then
          Begin
            cur_id := id of transaction that holds the lock;
            cur_type := type of current lock;
            If cur_id = tx_id then
              Begin
                If cur_type is the same or stronger than type Then
                  remove lock from lock_list;
                continue;
              End;
            If type and cur_type are COMPATIBLE Then
              continue;
            Else If T_cur_id is an ancestor of T_tx_id Then
              continue;
            Else return CONFLICT;
          End;
        Else if more than one lock is already set on O Then
          If type conflicts with any of these locks Then
            If the only conflict is with ancestor Then
              continue
            Else  return CONFLICT;
          Else If T_tx_id already holds more powerful lock on O
            Then remove lock from lock_list;
      End;  return OK;
  End.
```

Figure 4-7: The NGL Conflict-Detection Protocol

be locked in RBDE[12]. The NGL locking algorithm presented above applies exactly the same way if we consider each attribute of an object to be a subobject instead (conceptually)[13].. To simplify matters, we will often consider locking only at the granularity of objects.

To formalize the notion of incompatible locks and concurrency conflicts, we define a conflict situation as follows:

> **Definition 3:** A *conflict situation* (also called *interference*) occurs when a transaction T_i requests a lock `Ml`$_i$`[o]`, which is incompatible with another lock `Nl`$_j$`[o]` held by another transaction, T_j, given that T_j is not an ancestor of T_i.
>
> A conflict situation is a 2-tuple $C = (\langle T_i, T_j \rangle, \langle Nl_i[o], Ml_j[o] \rangle)$, where T_i and T_j are the two transactions that interfered, and $Ml_i[o]$ and $Nl_i[o]$ are the two locks on object o that caused the conflict.

To illustrate NGL, consider again our running example with John, Bob and Mary. Recall that the three developers are working together to complete a program, `Prog`, which contains three modules (`ModA`, `ModB` and `ModC`), three corresponding libraries (`LibA`, `LibB`, `LibC`), and several header files contained in the directory `includes`. The object code of all the C files belonging to a module are archived in the same library. Say that the project administrator has loaded the rules shown in figure 4-8.

The rules specify that editing a `CFILE` leads to outdating its object code. To update the object code, RBDE can go ahead and compile the `CFILE` automatically. Compiling a `CFILE` successfully leads to outdating the archived object code of the `CFILE`. To update the archive, RBDE can automatically invoke the `archive` rule to archive the object code in a library pointed to (via a `link`) by the module containing the `CFILE`. If all the `CFILE`s contained in a module (or any of its submodules) have been successfully archived, then the `archive_status` attribute of the module is set to "Archived".

---

[12]The actual implementation, as explained in chapter 8, considers objects to be the smallest lockable entities; individual attributes cannot be locked.

[13]Locking at the granularity of objects might already be considered expensive in many DBSs; locking at the granularity of attributes makes things even more expensive. However, we have assumed coarse-grain objects, and thus locking at the granularity of objects is not really expensive in RBDE.

```
edit [?c:CFILE]:
     :
     (and (?c.reservation_status = CheckedOut)
          (?c.locker = CurrentUser))

     { edit output: ?c.contents }

     (and no_backward (?c.status = NotCompiled)
          no_backward (?c.timestamp = CurrentTime));


compile [?f:CFILE]:
     (bind (?h to_all HFILE suchthat
                    (linkto [?f.hfiles ?h])))
     :
     no_backward (?f.status = NotCompiled)

     { compile ?f.contents ?h.contents "-g"
               output: ?f.object_code ?f.error_msg }

     (and (?f.status = Compiled)
          (?f.object_timestamp = CurrentTime));
     (?f.status = Error);


dirty[?c: CFILE]:
     :
     no_backward (?c.status = Compiled)
     { }
     (?c.status = NotArchived);


archive [?f:CFILE]:
    (bind (?m to_all MODULE suchthat (member [?m.cfiles ?f]))
          (?l to_all LIB suchthat (linkto [?m.libs ?l])))
    :
    (exists ?l):
    no_backward (?f.status = NotArchived)

    { archive ?f.object_code output: ?l.afile }

    (?f.status = Archived);
    (?f.status = Error);


archive [?m:MODULE]:
    (bind (?f to_all CFILE suchthat (member [?m.cfiles ?f]))
          (?q to_all MODULE suchthat (member [?m.modules ?q])))
    :
    (forall ?f) (forall ?q):
    (and (?f.status = Archived)
         (?q.status = Archived))
    { }
    (?m.archive_status = Archived);
```

Figure 4-8: Rules Causing a Locking Conflict at the Module Level

The chaining behavior of these rules can cause a locking conflict if two agents want to access the same library in order to archive the object code of two different CFILEs. For example, say that both John and Bob are editing the two CFILEs, f1.c and f2.c, respectively. The two transactions, T_{John} and T_{Bob}, encapsulate the executions of John's and Bob's commands, respectively. Before allowing John to access f1.c, T_{John} must have acquired a W lock on f1.c, and IX locks on all the ancestors of f1.c (i.e., ModA and Prog). Similarly, T_{Bob} would have acquired a W lock on f2.c and IX locks on both ModA and Prog. NGL would allow the two transactions to obtain their locks and proceed to invoke the edit activities.

Now say that Bob finishes editing f2.c. The RP chains to fire the compile rule on f2.c. The TM creates $T_{Bob.1}$ as a subtransaction of T_{Bob} to encapsulate the execution of this rule. To invoke the activity of this rule, $T_{Bob.1}$ must acquire a W lock on f2.c, R locks on all the objects bound to ?h, and the corresponding intention locks on the ancestors of these objects. T_{Bob}, an ancestor of $T_{Bob.1}$, has already acquired a W lock on f2.c, so $T_{Bob.1}$ can inherit this lock on f2.c. Acquiring R locks on the objects bound to ?h, and IS on all of their ancestors, does not conflict with any existing locks. Note that T_{Bob} already holds an IX lock, which is stronger than an IS lock, on Prog, and thus $T_{Bob.1}$ needs to also inherit this IX lock rather than locking Prog with an IS lock. NGL grants the requested locks to $T_{Bob.1}$, which proceeds to invoke the compiler on f2.c.

While Bob is compiling f2.c, John completes his modifications to f1.c. Again the RP will fire a forward chain to compile f1.c, which will be encapsulated in $T_{John.1}$. NGL will grant all the locks requested by $T_{John.1}$ to execute the compile activity since none of the locks conflict with existing locks. Both John's client and Bob's client are now compiling f1.c and f2.c, respectively.

Suppose both clients finish compiling at the same time. A successful compilation will cause a forward chain to fire the dirty rule followed by the archive rule. Therefore, both John's and Bob's compile rules will cause a forward chain to archive the respective C file. The server will process one of these forward chains first. Say that Bob's chain is processed first. In order to execute the archive rule, the TM creates

$T_{Bob.2}$, which must now acquire a W lock on the library LibA linked to ModA. NGL will grant this lock, and all the corresponding intention locks.

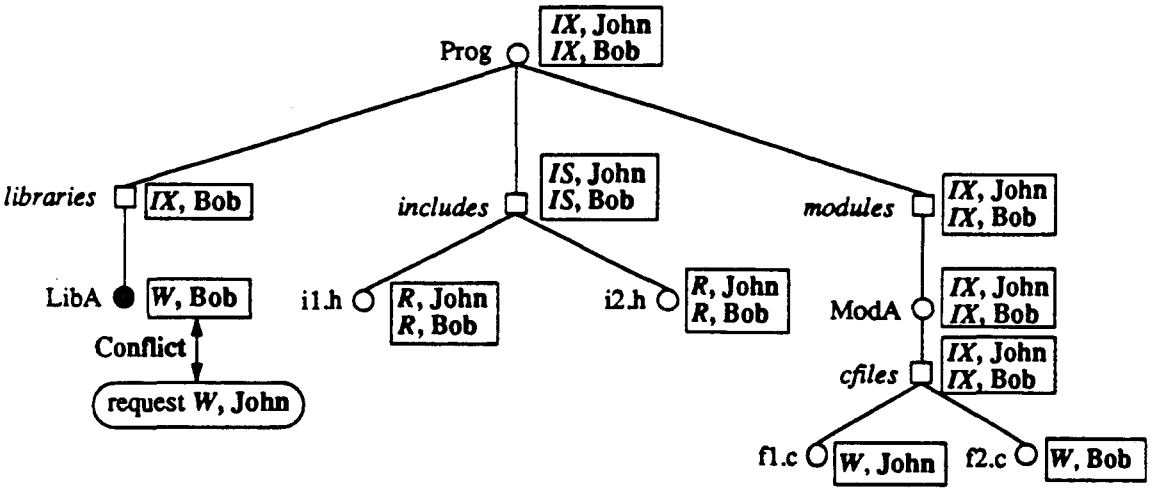

Figure 4-9: Object Hierarchy Showing Locks Held by Two Transactions

While Bob's client is busy archiving the object code of `f2.c` into `LibA`, the server continues executing John's rule chain. Before the RP can execute the `archive` (with `f1.c` as its parameter), it asks the TM to create a new subtransaction, $T_{John.2}$, to encapsulate John's `archive` rule; $T_{John.2}$ must acquire a W lock on `LibA`. NGL will reject this request because it conflicts with the W lock that $T_{Bob.2}$ already holds on `LibA`. The locks held on the objects at this point are shown in figure 4-9; the figure does not show links between objects for simplicity. The conflict on object `LibA` is shown by highlighting `LibA`. This conflict is passed on to the Scheduler, which decides how to resolve it. In chapters 5 and 6, we will present the two conflict resolution protocols that the RBDE Scheduler uses.

Say that the conflict resolution module decides that the chain containing $T_{John.2}$ should be committed at this point. T_{John} thus releases all of its locks. This leaves Bob's transactions as the only transactions running in the server. Now say that while Bob's client is archiving `f2.c`, Mary comes along and issues a command to `move Prog` to another project. As explained earlier, the `move` built-in command requires locking the source

object, `Prog`, in `X` mode. NGL will not grant this lock, because $T_{Bob.2}$ already holds an `IX` lock on `Prog` (because T_{Bob} has acquired a `W` lock on `f2.c`, so all the ancestors of `f2.c`, including `Prog`, are locked in `IX` mode), and as shown in the lock compatibility matrix, `IX` and `X` are not compatible.

4.4. Summary

In this chapter we devised a two phase locking scheme that the TM in RBDE uses to detect interference between concurrent transactions. The scheme mandates that all the locks necessary for the execution of a transaction are acquired before the transaction performs any write operation; the locks are released only after the transaction completes its execution, at which time the transaction is committed. The scheme does not suffer from deadlocks because transactions are not blocked. Our transaction scheme also guarantees that the phantom problem does not occur and minimizes the necessity for aborts.

In order to acquire locks on behalf of a transaction, the TM must request the locks from the LM. The LM verifies that the requested locks are not incompatible with existing locks held by other transactions before it grants the locks. The LM employs a nested granularity locking protocol (NGL) to detect conflict situations (interference) that arise because of lock incompatibility. We define interference as follows. If T_i holds a lock $Nl_i[o]$, and T_j requests another lock, $Ml_j[o]$, then interference occurs if `N` and `M` are incompatible according to the lock compatibility matrix. T_j is said to have *interfered* with T_i.

According to the definition of interference, whenever a conflict occurs, we have the following information:

- The two transactions, T_i and T_j, involved in the conflict,
- The phases in which the two transactions are in, and
- The two locks, $Nl_i[o]$ and $Ml_j[o]$, that caused the conflicts.

Given a transaction, T_i, we know the command (rule or built-in command) that initiated T_i, and the owner (user) of T_i (see definition 3.4.1). These pieces of information, in

addition to other information we introduce in the next chapter, are used by the Scheduler to resolve the detected conflict in a flexible way.

This completes our discussion of the NGL conflict detection protocol. We will now proceed to present the details of the Scheduler and the protocols it uses to resolve locking conflicts. We first present a default conflict resolution protocol in chapter 5 and then in chapter 6 we describe a mechanism for programming a project-specific conflict resolution protocol that overrides the default one.

Chapter 5

Resolving Concurrency Conflicts

Once interference between two concurrent transaction is detected by the NGL protocol, the lock conflict that caused the interference must be resolved before the server can continue the execution of the agents involved in the conflict. The *Scheduler* is responsible for resolving locking conflicts. Given the EMSL language and the chaining algorithms described in chapter 2, the best the Scheduler can do to resolve a locking conflict is to abort the transaction that requested the conflicting lock. The reason is that much of the semantics that the Scheduler could use to provide a more flexible concurrency control policy are only implicit in the definitions of rules rather than explicitly defined, and thus cannot be used. More specifically, rules serve two distinct purposes: (1) to express, enforce and maintain *consistency constraints*; and (2) to express and carry out opportunities for *automation*. Maintaining consistency is mandatory, whereas carrying out automation is optional. These two aspects are not distinguished in EMSL and the RBDE model, presented in chapter 2. Distinguishing between these two functions of rules can be the basis for explicitly defining and maintaining consistency constraints. These constraints would provide the semantics needed by the Scheduler to implement a flexible semantics-based concurrency control policy.

In this chapter, we first discuss the notion of semantics-based concurrency control in DBSs and describe three semantics-based mechanisms that schedulers in traditional DBSs use to provide flexible concurrency control. Next, we extend EMSL with constructs that the administrator can use to distinguish between *consistency predicates* and *automation predicates* in the conditions and effects of rules. We then present the Semantics-based Concurrency Control Protocol (SCCP), which uses the explicit definition of consistency predicates to provide the default policy for resolving concurrency conflicts. SCCP guarantees that the consistency constraints, defined by the consistency predicates in the project rule set, are maintained.

5.1. Related Work: Transaction Schedulers in Traditional DBSs

In a typical DBS, the TM (transaction manager) interacts with a *scheduler*, which controls the order of execution of concurrent transactions [Bernstein et al. 87]. When the scheduler receives a request to execute an operation from the TM, it has three options: immediately schedule the operation, delay the operation, or reject it. Typical schedulers choose only two of these three options. Based on which two options a scheduler chooses, it can be described as either *aggressive* or *conservative*.

An aggressive scheduler tends to either immediately schedule an operation or reject it. The main shortcoming of aggressive schedulers is that they process one operation at a time without looking ahead to foresee interference between concurrent transactions. Thus, an aggressive scheduler might schedule an operation from one transaction immediately, only to find out later that in order to maintain serializability it must reject an operation from the same or a different transaction. Rejecting an operation forces the TM to abort the requesting transaction. Aborting a transaction may be very costly, especially in applications like RBDE, involving long-duration interactive activities such as editing.

A conservative scheduler, in contrast, chooses to delay operations (e.g., by blocking transactions), which gives it an opportunity to re-order these operations in order to avoid rejecting any. Conservative schedulers thus avoid causing transactions to abort. Delaying operations, however, might lead to delaying the execution of a transaction unnecessarily, if these operations could have in fact been executed earlier without introducing conflicts. This may deteriorate the response time of an application like RBDE.

We combine both of these approaches in the RBDE scheduler. On the one hand, we construct an aggressive scheduler that either schedules operations immediately or rejects them. On the other hand, we make available information about the type of a transaction (as will be discussed shortly), so that the Scheduler can determine whether or not to schedule an operation belonging to the transaction. This information is made available because RBDE loads and analyzes rules, which comprise the body of a transaction in

RBDE, before executing them[14]. By doing this, we eliminate the possibility of unnecessarily delaying or aborting a transaction.

Schedulers control the ordering and execution of transaction operations in order to maintain the consistency of the database. In any DBS, consistency is maintained if every data item in the database satisfies some application-specific consistency constraints. For example, in an airline reservation system, one consistency constraint might be that each seat on a flight can be reserved by only one passenger. It is often the case, however, that the consistency constraints are not known beforehand to the designers of a general-purpose DBS. This is due to the lack of information about the computations and the semantics of database operations in potential applications.

To overcome this problem, schedulers of traditional DBSs implement general-purpose concurrency control mechanisms that do not depend on the particulars of applications. These concurrency mechanisms abstract all database operations into read and write operations. The mechanisms resolve conflicts, in general, using one of two methods: (1) locking mechanisms that resolve conflicts by forcing the transactions requesting the incompatible[15] lock to wait; and (2) optimistic mechanisms that resolve conflicts by aborting and rolling back one or more of the conflicting transactions. The mechanisms in both categories are inappropriate for advanced database applications, such as RBDE.

In RBDE, a nested transaction encapsulating a rule chain, which includes invocation of interactive activities, might last for a very long period of time. Using traditional serializability-based locking mechanisms to resolve concurrency conflicts between these long transactions causes serious performance problems if these transactions are allowed to lock resources until they commit. Other transactions wanting to access the same resources are forced to wait even though the long transaction might have finished using

---

[14]Note that although RBDE can analyze each rule, it cannot determine the exact rule chain that will be executed as a result of executing a rule. Thus, the body of a nested transaction in RBDE is not known in advance but is constructed during chaining.

[15]A read lock on an object is defines by locking mechanisms to be compatible with other read locks on the same object; a write lock on an object, in contrast, is considered incompatible with other locks on the same object.

the resources. Long transactions also increase the likelihood of automatic aborts (rollbacks), in order to avoid deadlock, or in the case of failing validation in optimistic concurrency control. When interactive software tools, like editors, are invoked as part of a rule chain, the human developers invest several hours of work into one activity. Aborting and rolling back such activities would not be appreciated by the developers.

5.1.1. Altruistic Locking: Using Information about Access Patterns

One approach to solving the problems introduced by LTs is to use semantic information about transactions to extend traditional mechanisms, such as 2PL, making them more flexible. The extended mechanism reverts back to the traditional scheme in case the additional information is not available (i.e., it might be available for some transactions but not for others).

One source of information that can be used to increase concurrency is the information about when resources are no longer needed by a transaction, so that they can be released and used by other transactions. This information can be used to allow a long transaction, that otherwise follows a serializability-based mechanism, to conditionally release some of its resources before it has acquired all the locks its needs (i.e., before entering the shrinking phase in 2PL). These resources can then be used by other transactions given that they satisfy certain requirements.

One formal mechanism that follows this approach is *altruistic locking* [Salem et al. 87], which is an extension of the basic 2PL algorithm. Altruistic locking makes use of information about access patterns of a transaction to decide which resources it can release. In particular, the technique uses two types of information: (1) negative access pattern information, which describes objects that will not be accessed by the transaction; and (2) positive access pattern information, which describes which, and in what order, objects will be accessed by the transaction.

Taken together, these two types of information allow long transactions to *release* their resources immediately after they are done with them but before the transactions have completed. The set of all data items that have been locked and then released by an LT is

called the *wake* of the transaction. If a transaction, T_i, locks a data item that is in the wake of another transaction, T_j, we say that T_i *entered* the wake of T_j. Releasing a resource is a *conditional unlock* operation because it allows other transactions to access the released resource as long as they satisfy the following two restrictions:

1. No two transactions can hold locks on the same data item simultaneously unless one of them has locked and *released* the object before the other locks it; the later lock-holder is said to be in the wake of the releasing transaction.

2. If a transaction is in the wake of another transaction, it must be completely in the wake of that transaction. This means that if John's transaction locks a data item that has been released by Mary's transaction, then any data item that is currently locked by John must have either been released by Mary before it was locked by John, or locked by John after Mary's transaction terminated (committed or aborted).

These two restrictions guarantee serializability of transactions. The protocol assumes that transactions are programmed and not incrementally (dynamically) constructed (i.e., the user cannot make up the transactions as he goes along). This assumption is necessary to determine the access patterns of transaction before a transaction starts executing. In the following example, however, we will assume an informal extension to this mechanism that will allow dynamically-constructed transactions, since these are more relevant to the RBDE model (rule chaining).

Consider again the example depicted in figure 1-1 in chapter 1, where each module in the project contains a number of C source files (subobjects). Say that Bob wants to familiarize himself with the code of all the functions of the project because he wants to write a description of the implementation. He starts a long transaction, T_{Bob}, that accesses all of the C source files, one file at a time. Bob needs to access each C source file only once to read it and add some comments about the code; as he finishes accessing each file he releases it. In the meanwhile, John starts a short transaction, T_{John}, that accesses only two files, first `f1.c` and then `f2.c`, from module `ModA`.

Suppose that T_{Bob} has already accessed `f2.c` and released it, and is currently reading

`f1.c`. T_{John} has to wait until T_{Bob} is finished with `f1.c` and releases it. At that point T_{John} can start accessing `f1.c` by entering the wake of T_{Bob}. T_{John} will be allowed to enter the wake of T_{Bob} (i.e., to be able to access `f1.c`) because all of the objects that T_{John} needs to access (`f1.c` and `f2.c`) are in the wake of T_{Bob}. After finishing with `f1.c`, T_{John} can start accessing `f2.c` without delay since it has already been released by T_{Bob}.

Now say that Mary starts another short transaction, T_{Mary}, that needs to access both `f2.c` and a third file `f3.c` that is not in the wake of T_{Bob} yet. T_{Mary} can access `f2.c` after T_{John} terminates, but then it must wait until either `f3.c` has been accessed by T_{Bob} (i.e., until T_{Bob} releases `f3.c`) or until T_{Bob} terminates. If T_{Bob} never accesses `f3.c` (Bob changes his mind about viewing `f3.c`), T_{Mary} is forced to wait until T_{Bob} terminates (which might take a long time since it is a long transaction).

To improve concurrency in this situation, Salem *et al.* introduced a mechanism for *expanding* the wake of a long transaction dynamically in order to enable short transactions that are already in the wake of a long transaction to continue running. The mechanism uses the negative access information in order to add to the wake of a long transaction objects that will not be accessed by the transaction. Following up on the example, the mechanism would add `f3.c` to the wake of T_{Bob} by issuing a release on `f3.c` even if T_{Bob} had not locked it. The addition takes place when T_{Mary} requests a lock on `f3.c`. This would allow T_{Mary} to access `f3.c` and thus continue executing without delay.

The basic advantage of altruistic locking is its ability to utilize the knowledge that a transaction no longer needs access to a data object that it has locked. If access information is not available, any transaction, at any time, can run under the conventional 2PL protocol without performing any special operations. However, because of the interactive nature of transactions in design environments, the access patterns of transactions are not predictable. In the absence of this information, altruistic locking reduces to 2PL. Altruistic locking also suffers from the problem of cascaded aborts: when a long transaction aborts, all the short transactions in its wake have to be rolled back even if they have already committed.

The SCCP protocol we present later in this chapter minimizes the probability of cascaded aborts by requiring rollback only in a few cases. SCCP uses an approach similar to that of altruistic locking, except that, in addition to access patterns, it uses semantic information about the consistency constraints of each transaction, which we will discuss later in this chapter. But before doing that, we describe two other mechanisms that use semantic information similar to the information used by the SCCP protocol.

5.1.2. Constraint-Based Schedulers

Access patterns are not the only piece of semantic information that schedulers can use to provide flexible concurrency control. In some advanced applications such as CAD, where the different parts of the design are stored in a project database, it is possible to supply semantic information in the form of integrity constraints on database entities. Design operations incrementally change those entities in order to reach the final design [Eastman 80, Eastman 81]. By definition, full integrity of the design, in the sense of satisfying its specification, exists only when it is complete. Unlike conventional domains where database integrity is maintained during all quiescent periods, the iterative design process causes the integrity of the design database to be only partially satisfied until the design is complete. There is a need to define transactions that maintain the partial integrity required by design operations. Kutay and Eastman proposed a transaction model that is based on the concept of *entity state* [Kutay and Eastman 83].

Each *entity* in the database is associated with a *state* that is defined in terms of a set of integrity constraints. Like a traditional transaction, an entity state transaction is a collection of actions that read a set of entities and potentially write a set of entities. Unlike traditional transactions, however, entity state transactions are instances of transaction classes. Each class defines: (1) the set of entities that instance transactions read; (2) the set of entities that instance transactions write; (3) the set of constraints that must be satisfied on the read and write entity sets prior to the invocation of a transaction; (4) the set of constraints that can be violated during the execution of an instance transaction; (5) the set of constraints that hold after the execution of the transaction is completed; and (6) the set of constraints that are violated after the execution of the transaction is completed. A very simple example of a transaction class is the class of transactions that have all the

entities in the database as their read set. All other sets are empty since these transactions do not transform the database in any way.

The integrity constraints associated with transaction classes define a partial ordering of these classes in the form of a precedence ordering. Transaction classes are depicted as a finite state machine; the violation or satisfaction of specific integrity constraints defines a transition from one database state to another. Based on this, Kutay and Eastman define a concurrency control protocol that detects violations to the precedence ordering defined by the application-specific integrity constraints. Violations are resolved by communication among transactions to negotiate the abortion of one or more of the conflicting transactions.

The SCCP protocol uses a subset of the semantic information used by entity-state transactions. Some of this semantic information, like the read and write sets of each agent, and the conditions and effects of rules, are already provided by the rule definitions in EMSL. Later in this chapter, we extend EMSL to provide constructs for defining the consistency constraints of each rule, which can then be used to provide a semantics-based concurrency control policy.

5.1.3. Semantics-Based Schedulers

A third piece of information, besides access patterns and integrity constraints, that can be used by schedulers is semantic information about the purpose of transactions. Garcia-Molina observed that by using semantic information about the purpose of transactions, a DBS can replace the serializability constraint by the semantic consistency constraint [Garcia-Molina 83]. The gist of this approach is that from a user's point of view, not all transactions need to be atomic. Garcia-Molina introduced the notion of *sensitive transactions* to guarantee that users see consistent data on their terminals. Sensitive transactions are those that must output only consistent data to the user, and thus must see a consistent database state in order to produce correct data. Not all transactions that output data are sensitive since some users might be satisfied with data that is only relatively consistent.

For example, suppose that Bob wants to get an idea about the progress of his programming team. He starts a transaction T_{Bob} that browses the modules and C source files of the project. Meanwhile, John and Mary have two in-progress transactions, T_{John} and T_{Mary}, respectively, that are modifying the modules and C source files of the project. Bob might be satisfied with information returned by a read-only transaction that does not take into consideration the updates being made by T_{John} and T_{Mary}. This would avoid delays that would result from having T_{Bob} wait for T_{John} and T_{Mary} to finish before reading the objects they have updated.

Given the distinction between sensitive transactions and non-sensitive transactions, Garcia-Molina defines a new semantic correctness criterion that can be used to replace serializability. A *semantically consistent schedule* is one that transforms the database from one *semantically* consistent state to another. It does so by guaranteeing that all sensitive transactions obtain a consistent view of the database. Each sensitive transaction must thus appear to be an atomic transaction with respect to all other transactions.

It is more difficult to build a general concurrency control mechanism that decides which schedules preserve semantic consistency than it is to build one that recognizes serializable schedules. Even if all the consistency constraints were given to the DBS (which is not possible in the general case), there is no way for the concurrency control mechanism to determine *a priori* which schedules maintain semantic consistency. The DBS must run the schedules and check the constraints on the resulting state of the database in order to determine if the schedules maintain semantic consistency [Garcia-Molina 83]. Doing that, however, would be equivalent to implementing an optimistic concurrency control scheme that suffers from the problem of rollback. In order to avoid rollback, the concurrency control mechanism must be provided with information about which transactions are compatible with each other.

Two transactions are said to be compatible if their operations can be interleaved at certain points without violating semantic consistency. Having the user provide this information, of course, is not feasible in the general case because it burdens the user with having to understand the details of applications. However, in some applications, as in RBDE, this kind of burden might be acceptable in order to avoid the performance

penalty of traditional general-purpose mechanisms. If this is the case, the user (the project administrator in the case of RBDE) must still be provided with a framework for supplying information about the compatibility of transactions.

Transactions are categorized into types depending on the specifics of the application, in particular, the kinds of data objects and operations on them, supported by the application. Each transaction type is divided into steps with the assumption that each step must be performed as an indivisible unit. A *compatibility set* associated with a transaction type defines allowable interleavings between steps of transactions of the particular kind with the same or other kinds of transactions. Depending on the compatibility sets of different types of transactions, various levels of concurrency can be achieved. At one extreme, if the compatibility sets of all kinds of transactions are empty, the mechanism reverts to a traditional locking mechanism that enforces serializability of the long transactions. At the other extreme, if all transaction types are compatible, the mechanism only enforces the atomicity of the small steps within each transaction, and thus the mechanism reverts to a system of short atomic transactions (i.e., the steps). In advanced applications where this kind of mechanism might be the most applicable, allowable interleavings would be between these two extremes.

The SCCP protocol categorizes transactions into five types, and uses this type information to resolve conflicts between transactions. Only one type of transaction is considered "sensitive"; SCCP guarantees that these transactions see a consistent state of the database. The "sensitivity" of the other four types of transactions depends on their type. In the rest of this chapter, we first extend EMSL with consistency predicates that can be used to define semantic consistency constraints, and then we present the SCCP protocol that uses these constraints as a basis for concurrency control.

5.2. Distinguishing Between Consistency and Automation in RBDE

Given the EMSL language and the rule execution model described in chapter 2, the administrator has no way of specifying any semantic information explicitly in a way that can be used to resolve concurrency conflicts. The best RBDE can do once a conflict is detected by the NGL protocol is to abort the transaction that caused the conflict in order

to guarantee the serializability of concurrent schedules. Aborting a transaction requires rolling it back so that every database operation performed as part of the transaction is undone. However, aborting transactions, especially those encapsulating long rule chains that involve many operations, is not always necessary in order to resolve conflicts.

For example, suppose that Bob requested to `edit` a `CFILE`, spent several hours editing the file and then exited from the editor. RBDE asserts the effect of the `edit` rule, which initiates a forward chain to the `compile` rule on the `CFILE`. Now say that while evaluating the condition of `compile`, a conflict was detected between the `compile` rule and a concurrent rule, whose activity is being executed by a client. RBDE now has no choice but to abort the execution of the `compile` rule. Since `compile` is part of a rule chain, RBDE must decide whether or not to abort the whole chain (i.e., the `edit` rule in addition to `compile`). On the one hand, aborting the chain would require undoing the editing session that lasted hours, which will not be appreciated by Bob. On the other hand, not aborting the whole chain might confuse other developers because they would assume that the `CFILE`'s object code is up to date, which is not true since it has not been compiled after the last edit.

RBDE does not have any information to help it make a decision. In order to be safe, RBDE must abort the whole chain, which, although wasteful of Bob's efforts, guarantees that the consistency of objects is maintained. If RBDE was provided with information about the consistency constraints of the project, it might have been able to determine whether not aborting a rule chain would violate any of these constraints. In particular, the administrator might want to specify that compiling a C file after it has been edited is not mandatory, while outdating the object code of the C file (e.g., by annotating it so that it is known to be out of date) after the C file has been edited is obligatory. There is no way to specify that the former reflects an opportunity for automation but the latter is a consistency constraint. The problem lies in the inability to distinguish between consistency and automation in EMSL.

5.2.1. Extending EMSL to add Consistency Predicates

We extend EMSL by adding constructs that distinguish between two kinds of predicates in the conditions and effects of rules: *consistency predicates* and *automation predicates*. Consistency predicates are used to model the consistency constraints of a project explicitly. As will be explained shortly, they constrain the chaining possibilities to only inference rules. Automation predicates define the desired but optional automation that the RBDE should attempt to carry out; they cause chaining among activation rules in order to automatically invoke development tools.

A consistency predicate is denoted either by enclosing the predicate in square brackets rather than parentheses or by attaching the prefix `consistency` to the predicate. Predicates enclosed in parentheses or preceded by the prefix `automation` are considered automation predicates.

The purpose of a rule chain depends on the kind of predicate that initiated the chain. A consistency predicate will initiate a chain to propagate changes in the database in order to transform the database to a consistent state (i.e., to maintain consistency). An automation predicate will initiate a chain to invoke software tools and perform activities automatically. Given the distinction between consistency and automation predicates, the definition of *change implication* given in chapter 2, which provided the basis for chaining, is not sufficient. We need to define two kinds of implications, corresponding to the two kinds of chaining predicates.

> **Definition 1:** If a consistency predicate, `P`, in the effect of a rule, `r1`, implies a predicate of any kind in the condition of another rule, `r2`, then if `r2` is an inference rule, it is said to be a *consistency implication* of `P`. Otherwise, if `r2` is an activation rule, then it is said to be an *automation implication* of `P`.

We restrict consistency implications to inference rules because we do not want the results of executing one development activity (e.g., edit) to be contingent upon the execution of another (e.g., compile). It is feasible to allow a consistency implication to cause chaining to an automation rule if we have information about whether or not the automation rule can be rolled back in case of failure. This kind of information, however, is not available in EMSL.

> **Definition 2:** If an automation predicate, `P`, in the effect of a rule, `r1`, implies

> a chaining predicate of any kind in the condition of another rule, `r2`, then `r2` is said to be an *automation implication* of `P`.

RBDE combines consistency and automation predicates as follows. After the client has finished executing the activity of a rule, `R`, and asserting one of the effects of `R`, the RP (rule processor) *must* carry out all the consistency implications of each predicate in the effect. If any of these implications (rules) cannot be immediately fired (i.e., because its condition is not satisfied or because of a locking conflict), then `R` (more precisely, `R`'s activity and its effect on the database) must be rolled back.

Automation assignment predicates in the effects of a rule, on the other hand, cause the RBDE to try to carry out all possible automation implications on a "best effort" basis. This means that the RBDE tries to fire all the rules (both activation and inference) whose conditions become satisfied as a result of the change to the objects' attributes caused by asserting the automation assignment predicates in one of the effects of the original rule. Not being able to fire any of these rules does not affect the validity of the original rule that caused the chaining. These automation chaining semantics differ from the chaining model presented in chapter 2.

A consistency predicate in the condition of a rule is a constraint that must be satisfied. If the value of the predicate is TRUE, the evaluation of the condition proceeds. If the value is FALSE, the evaluation stops and the value of the whole condition is UNSATISFIABLE. RBDE does not try to make the predicate satisfied but instead informs the user that the command he requested cannot be executed. In contrast, if an automation predicate in the property list of a rule is evaluated to be FALSE, RBDE initiates an automation backward chain to try to make it satisfied. Automation backward chaining causes both activation and inference rules to be fired during the chain, possibly causing the invocation of activities.

In order to control the automation behavior of RBDE, automation predicates in the effects of a rule can be preceded by the prefix `no_forward` to prevent them from initiating forward chaining. Similarly, automation predicates in the condition of a rule can be preceded by the prefix `no_backward` to prevent them from initiating backward chain-

ing. Automation predicates can be preceded by `no_chain` to prevent any kind of chaining from or into them. Both consistency and automation predicates in the effects of a rule can be preceded by `no_backward` to prevent backward chaining to them by automation predicates. Finally, both kinds of predicates in the condition of a rule can be preceded by `no_forward` to prevent forward chaining into them by automation predicates in the effects of other rules. Note that consistency forward chaining initiated by a consistency predicate in the effect of a rule cannot be "turned off". Therefore, if a consistency predicate in the effect of a rule is preceded by `no_forward` or `no_chain`, the loader will consider this an error.

To illustrate the distinction between consistency and automation predicates, consider the rules of figure 5-1. These rules are the same as the rules in figure 2-12 in chapter 2, except that some of the predicates of the `compile` and `dirty` rules have been changed to be consistency predicates (enclosed in square brackets instead of parentheses). Also, `no_chain`, `no_forward` and `no_backward` prefixes have been added to control the automation caused by the predicates of the rules.

If a user requests to `edit` a `CFILE` object (i.e., instance of the class `CFILE`) but the value of the `reservation_status` attribute of the object is not equal to "CheckedOut", the RP initiates backward chaining to fire the `reserve` rule automatically. If the first effect of `reserve` is asserted, changing the value of the `reservation_status` attribute to "CheckedOut", the editor is invoked on the corresponding file. Note that the condition of `edit` could have been not satisfied because the value of the `locker` attribute is not the same as the current user (i.e., the user who requested the edit command), even though the `reservation_status` attribute is equal to "CheckedOut". In this case, RP will not initiate a backward chain to `reserve` because the predicate `(?f.locker = CurrentUser)` is preceded by `no_chain`, which prevents backward chaining into it.

In any case, if the condition of `edit` becomes satisfied, the editor is invoked. Once the editing session is completed and the effect of `edit` has been asserted, RBDE fires the `compile` rule since the assignment predicate, `(?f.status = NotCompiled)`, in `edit`'s effect implies the predicate `[?c.status = NotCompiled]` in `compile`'s

```
reserve [?f : FILE]:
    :
    no_backward (?f.reservation_status = Available)

    { reserve output: ?f.contents ?f.version }

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

edit [?f : CFILE]:
    :
    (and (?f.reservation_status = CheckedOut)
         (?f.locker = CurrentUser))

    { edit output: ?c.contents }

    (and no_backward (?f.status = NotCompiled)
         no_backward (?f.timestamp = CurrentTime));

compile [?f : CFILE]:
    (bind (?h to_all HFILE suchthat
                    (linkto [?f.hfiles ?h])))
    :
    no_backward (?f.status = NotCompiled)

    { compile ?f.contents ?h.contents "-g"
              output: ?f.object_code ?f.error_msg }

    (and [?f.status = Compiled]
         (?f.object_timestamp = CurrentTime));
    (?f.status = Error);

dirty [?c : CFILE]:
    :
    no_backward (?c.status = Compiled)
    { }
    (?c.status = NotArchived);

archive [?f : CFILE]:
   (bind (?m to_all MODULE suchthat (member [?m.cfiles ?f]))
         (?l to_all LIB suchthat (linkto [?m.libs ?l])))
   :
   (exists ?l):
   no_backward (?f.status = NotArchived)

   { archive ?f.object_code output: ?l.afile }

   (?f.status = Archived);
   (?f.status = Error);
```

The predicates enclosed in square brackets "[...]" are consistency predicates, whereas those in parentheses "(...)" are automation predicates.

Figure 5-1: Example Rules Containing Consistency Predicates

property list. If the RP cannot execute `compile` for any reason (e.g., a concurrency conflict), the execution of the `edit` rule is not invalidated in any way. The reason is that `compile` is an automation implication of `edit`.

In contrast, the `dirty` rule is noted as a consistency implication of the assignment predicate `[?f.status = NotCompiled]` in the first effect of the `compile` rule. RBDE must be able to fire `dirty` in order for the changes made by this assignment predicate to become permanent. If it cannot do that for any reason (e.g., a concurrency conflict), the user's compilation is undone by reverting the values of the attributes `object_code`, `error_msg`, `status` and `object_timestamp` of the `CFILE` object to what they were before the `compile` rule was fired. This does not affect the changes that the editor introduced.

The `dirty` rule (which is an inference rule) has one consistency predicate in its effect. This predicate implies the predicate `(?c.status = NotArchived)` in the property list of the `archive` rule. However, since the `archive` rule is an activation rule, it is noted as an automation implication (rather than a consistency implication) of `dirty`.

5.3. Revised Rule Execution Model

The previous example is meant to give an idea about how the chaining behavior in RBDE is affected by the introduction of consistency and automation predicates. In this section, we revise the chaining algorithms of chapter 2. These revisions are necessary for the SCCP protocol, which we will present later in this chapter.

5.3.1. Compiling Forward and Backward Chains with Consistency Predicates

As explained earlier, there are three kinds of rule chains that can be initiated by the two different kinds of chaining predicates. The routine for compiling forward and backward chains, shown in figure 2-14 in chapter 2, must be revised to take into account the distinction between automation and consistency forward chains. Note that there is only automation backward chaining.

```
routine COMPILE_CHAINS();

  /* Predicate Table is global variable. */
  Begin
    For each predicate, p1, in the Predicate Table Do
      Begin
        If p1 is not a chaining predicate Then
           continue;
        For each predicate, p2, in the Predicate Table Do
          Begin

            /* Don't chain to same predicate. */

            If p1 = p2 Then
              continue;

            If p1 is an assignment predicate Then
               Begin
                 If p2 is an assignment predicate Then
                   continue;
                 Else If p1 implies p2 Then
                   Begin
                     If p2 is preceded by either no_forward
                                                 or no_chain Then
                        continue;
                     If p1 is a consistency predicate Then
                       Begin
                         r := rule containing p2;
                         If r is an inference rule Then
                          add cons. forward chain from p1 to p2;
                         Else
                          add auto. forward chain from p1 to p2;
                       End;
                     Else
                       add auto. forward chain from p1 to p2;
                   End;
               End;
            Else If p1 is a property predicate Then
               Begin
                 If p1 is a consistency predicate Then
                   continue;
                 If p2 is not an assignment predicate Then
                   continue;
                 Else If p2 implies p1 Then
                   Begin
                     If p2 is preceded by no_backward or
                        no_chain Then continue;
                     add backward chain from p1 to p2;
                   End;
               End;
          End;
      End;
  End.
```

Figure 5-2: Compiling Automation and Consistency Chains

Consistency forward chaining is triggered when a consistency assignment predicate is asserted in the effect of a rule. As defined earlier, consistency forward chains involve only inference rules. Thus, after determining that an assignment consistency predicate, `p1`, in the effect of a rule, `r1`, implies another predicate, `p2`, in the property list of another rule, `r2`, the routine must check whether `r2` is an inference or an activation rule. If `r2` is an inference rule, then the algorithm can insert a consistency forward chain from `p1` to `p2`. Otherwise, if the rule is an activation rule, the chain between `p1` and `p2` reverts to an automation forward chain. Automation predicates are treated exactly the same as in the earlier algorithm presented in chapter 2. The modified routine for compiling forward and backward chains is shown in figure 5-2. We now proceed to revise the chaining algorithms in RBDE.

5.3.2. Consistency and Automation Forward Chaining

Distinguishing between consistency and automation predicates does not affect backward chaining since consistency predicates do not initiate any backward chaining. Forward chaining, however, must be revised to distinguish between consistency implications and automation implications. After one of the effects of a rule, `r`, is asserted, the RP should attempt to execute all the rules that are consistency implications of any of the predicates in the asserted effect. Only if the RP succeeds in executing all of these rules should it proceed to carry out the automation implications of `r`'s effect.

Since the consistency implications of each predicate in the effect of a rule consist of only inference rules, the RP can execute these rules one after the other without interleaving their execution with other rules. Thus, a consistency forward chain is carried out atomically by the RP. The algorithm to carry out the consistency implications of a rule is shown in figure 5-3. The routine returns FALSE as soon as it finds out that it cannot execute one of the consistency implications because of an unsatisfied condition or a conflict. Note that we have not included any transaction operations in the algorithm for simplicity.

The forward chaining algorithm presented in chapter 2 (figure 2-16) does not distinguish between consistency implications and automation implications. The revised algorithm is

```
routine EXECUTE_CONSISTENCY_CHAIN (list: a list of predicates);

  Begin
    ret_value := TRUE;
    For each predicate, p, in list Do
      Begin
        rule := rule containing p;

        /* rule is an inference rule by definition. */

        Evaluate the condition of rule;
        If condition is not satisfied Then
           return FALSE;
         Else
           Begin
             chaining_list := ASSERT_EFFECT (0, rule);
             If chaining_list is not empty Then
                ret_value := DO_FORWARD_CHAIN (rule, chaining_list);
           End;
      End;
    return ret_value;
  End.
```

Figure 5-3: Carrying Out Consistency Implications

shown in figure 5-4. The algorithm first attempts to execute all the consistency implications of the asserted effect. Only if this succeeds, does it go ahead and insert the automation implications on the global execution stack[16]. The algorithm shows the difference between consistency forward chaining and automation forward chaining. The basic difference is that automation forward chaining is a "best-effort" activity. It is always successful by definition even if none of the rules in the chain can be executed. Consistency forward chaining, on the other hand, is successful only if all the inference rules in the chain, and all the inference rules on their consistency forward chains in turn, are executed successfully (i.e., their conditions are satisfied, and thus their effects can be asserted).

We have now completed revising all the rule execution and chaining algorithms to incorporate consistency predicates. We now proceed to describe the SCCP protocol, which implements the default concurrency control policy in RBDE. SCCP is semantics-based since it uses the distinction between consistency implications and automation implications to resolve concurrency conflicts.

[16]Recall that the server switches context to the client's execution stack, making it the global Execution_Stack, before calling the RP.

```
routine DO_FORWARD_CHAIN (rule, chaining_list: list of predicates);

   Begin
     For each predicate, p1, in chaining_list Do
       Begin
         consistency_list := consistency forward chains of p1;
         automation_list := automation forward chains of p1;
         If consistency_list <> empty Then
            ret_value := EXECUTE_CONSISTENCY_CHAIN (consistency_list);
         If ret-value = FALSE Then
            return FALSE;

         For each predicate, p2, in automation_list Do
           Begin
             rule := rule containing p2;
             If rule is not on client's stack Then
                push rule on Execution_Stack;
           End;
         If Execution_Stack is empty Then
            return TRUE;
         Else
            Begin
              rule := rule on top of Execution_Stack;
              ret_value := START_RULE_EXECUTION (rule);
              return ret_value;
            End;
       End;
   End.
```

Figure 5-4: Revised Forward Chaining Algorithm

5.4. SCCP: A Semantics-Based Concurrency Control Protocol

In the nested transaction model we presented in chapter 4, the TM (transaction manager) distinguished between top-level transactions and subtransactions. In particular, each subtransaction encapsulates a rule that has been initiated as part of a chain. This distinction is not sufficient to reflect the purpose of the agent encapsulated by the transaction. In the discussion above, we have distinguished between three kinds of chains. We need to carry over the distinction to the transaction model.

As explained in chapter 4, the TM is called from either the RP or the CE (command executor) to create a new transaction. Before the RP begins the execution of a rule, it calls the TM to create a new transaction that will encapsulate the rule's execution. When a rule is executed, the RP knows the kind of chain of which the rule is a part. For example, in the algorithm shown in figure 5-3, the RP knows that it is in the middle of a

consistency forward chain. This information is passed on to the TM. The CE calls the TM for the sole purpose of creating and terminating a transaction to encapsulate the execution of a built-in command.

5.4.1. Transaction Types in RBDE

Based on the information passed to it by either the RP or the CE, the TM determines the *type* of the transaction it is creating. In particular, the type of a transaction can be one of five possibilities: `top-level`, which represents the original rule corresponding to a user command, `built-in`, `consistency forward chaining`, `automation forward chaining`, and `backward chaining`, denoted by `tl`, `bi`, `cfc`, `afc`, and `bc`, respectively. We will sometimes use $T_{i,z}$ to denote a transaction, whose unique identifier is `i` and whose type is `z`. Thus, the type of $T_{i,tl}$ is `top-level` and the type of $T_{j,cfc}$ is `consistency forward chaining`.

The type of a transaction determines how it is nested, when it is committed, and when it must be aborted. Transactions of type `bi` do not have any subtransactions since they encapsulate the execution of built-in commands, which do not cause any chaining. These transactions are committed and aborted independently of other transactions. A transaction, $T_{i,tl}$, encapsulates the execution of an original rule (a rule that has been fired directly in response to a user's command rather than during chaining). To explain the possible nesting that transactions of this kind can have, we need to analyze forward and backward chaining a bit closer.

In chapter 4, we stated that a nested transaction cannot commit until all of its subtransactions commit; if a transaction is aborted, then all of its subtransactions must be aborted as well. Thus, the nesting of a transaction reflects three purposes: (1) a causal relationship between the parent and the child (the parent caused the child), (2) a commit dependency between the parent and the child (the parent cannot commit until all of its children have committed) , and (3) an abort dependency between the child and the parent (i.e., if the parent is aborted, then the child must be aborted), and sometimes between the parent and the child (i.e., if a child is aborted, the parent must be aborted). The first purpose is common to all three kinds of chaining. In other words, the reason for the

existence of a subtransaction, $T_{j,z}$, regardless of its type z, is the execution of the parent transaction $T_{i,y}$, regardless of its type y. The second and third purposes, however, are not common to all types of nesting. Let us analyze the three kinds of chaining to understand why.

5.4.1.1. Consistency Forward Chaining Transactions

The firing of a rule can lead to both a backward chain and a forward chain. In the case of forward chaining, there are two kinds of chains that can result: automation forward chaining and consistency forward chaining. For example, the execution of a rule `r1` can lead to an automation forward chain consisting of two rules, `r2` and `r3`, and a consistency forward chain consisting of `r4` and `r5`. The effects of `r1` cannot be made permanent except if both `r4` and `r5`, and all of their consistency implications, are executed successfully. Thus the transaction T_1, encapsulating the execution of `r1`, cannot commit until both of T_4, encapsulating the execution of `r4`, and T_5, which encapsulates the execution of `r5`, have committed. Also, if either of T_4 or T_5 aborts, T_1 must be aborted as well. This means that there is a commit and abort dependency between T_4 and T_5 on the one hand and T_1 on the other.

5.4.1.2. Automation Forward Chaining Transactions

Once T_4 and T_5 have committed, T_1 can commit and release all of its locks without waiting for T_2, which encapsulates the execution of `r2`, or T_3, which encapsulates the execution of `r3`, to commit. The reason is that both `r2` and `r3` are automation implications of `r1`. Thus, T_2 and T_3 can be considered commit-independent from T_1. Committing T_1 before starting to execute T_2 and T_3 increases concurrency because it allows other transactions to lock the objects that T_1 had locked. Otherwise, these objects would have remained locked for the duration of T_2 and T_3 even if they were not accessed by either T_2 or T_3, which would have been wasteful and overly restrictive. Thus, there is no need to have any kind of commit or abort dependency between a transaction $T_{i,z}$ and any of its subtransactions of type `afc`. To simplify matters, instead of creating a transaction $T_{i,afc}$ as a subtransaction of another transaction, the TM creates all $T_{i,afc}$ as independent top-level transactions, which might themselves be nested. Consequently, a transaction, $T_{i,z}$, where z is either `tl` or `afc`, do not have subtransactions of type `afc`. Note that

since we do not perform automation forward chaining during backward chaining, a transaction $T_{i,bc}$ has only subtransactions whose types are `bc`.

5.4.1.3. Backward Chaining Transactions

Backward chaining is more restrictive than automation forward chaining. In particular, no locks are released until the backward chaining cycle terminates and the original rule's execution is completed. Thus, a transaction $T_{i,z}$ and its subtransaction $T_{j,bc}$ have a commit dependency (i.e., $T_{i,z}$ cannot commit until $T_{j,bc}$ has either committed or aborted). Aborting $T_{i,z}$, however, does not lead to aborting $T_{j,bc}$ because the activities that $T_{j,bc}$ has executed are not in any way invalidated if $T_{i,z}$ is aborted. For example, say that in order to link a module, all of its `CFILE` objects must have been compiled. If this condition is not satisfied, backward chaining will be initiated to compile the `CFILE` objects that have not been compiled. If after compiling some of these `CFILE` objects, the transaction encapsulating the link rule is aborted, there is no reason whatsoever to roll back the compilation of the `CFILE` objects that have been already compiled.

To formalize our discussion of nesting and transaction types, we add an eighth element to the transaction notation. Consider a transaction $T = (i, U_i, S, c, u, t, l, z)$, where z is the type of the transaction and S is the set of its subtransactions. The following implications hold (we use $\rightarrow$ to denote an implication):

1. $(z = bi) \rightarrow (S = \varnothing)$; a `built-in` transaction is flat (i.e., has no subtransactions).

2. $(z = tl) \rightarrow (\forall T_{j,y} \in S \mid (y = bc)) \vee (\forall T_{j,y} \in S \mid (y = cfc))$; a transaction of type `top-level` can have subtransactions that are either all `backward chaining` transactions or all `consistency forward chaining` transactions. Note that for the purpose of our discussion we consider `afc` transactions independent.

3. $(z = bc) \rightarrow (\forall T_{j,y} \in S \mid (y = bc)) \vee (\forall T_{j,y} \in S \mid (y = cfc))$; a transaction of type `backward chaining` can have subtransactions that are either all `backward chaining` transactions or all `consistency forward chaining` transactions. Note that we do not perform automation forward chaining during backward chaining.

4. $(z = cfc) \rightarrow (\forall T_{j,y} \in S \mid (y = cfc))$; a `consistency forward chaining` transaction can have only `consistency forward chaining` transactions as its subtransactions. Again, note that any `afc` transaction initiated by a `cfc` transaction are considered independent.

5. $(z = afc) \rightarrow (\forall T_{j,y} \in S \mid (y = cfc))$; an `automation forward chaining` transaction can have only `consistency forward chaining` transactions as its subtransactions.

SCCP uses the information about the types of the two transactions that interfered to determine how to resolve the conflict. Recall that interference occurs only between two transactions that are independent of each other (i.e., they are not part of the same nested transaction). Since the execution of each nested transaction is carried out serially (i.e., no two subtransactions of the same nested transaction are executed concurrently), there can never be conflicts between two siblings or "cousins" in the same nested transaction.

Before we discuss the details of how SCCP resolves conflicts, we need to analyze the different states that transactions can be in when a conflict occurs. SCCP uses this information when deciding how to resolve a conflict.

5.4.2. States of Transactions

As far as the Scheduler is concerned, a transaction can be in one of three states when a conflict occurs: `active`, `pending`, or `inactive`. When a transaction is created, its state starts out as `active`. During its `active` state, the transaction acquires all of its locks; it acquires either `S` or `R` locks on its read set and then (possibly after evaluating a rule's condition), acquires `W` or `X` locks on its write set. The exact types of locks depends on whether the transaction encapsulates a built-in command or a rule. Note that a transaction encapsulating the execution of a built-in command remains in an `active` state for the duration of its existence. The reason is that the CE executes built-in commands atomically; thus, the transaction that encapsulates the execution of a built-in command is also performed atomically by the TM.

The TM executes one access unit from one transaction at a time. Thus, the TM could process the lock requests of only one transaction at a time. Consequently, there can be

only one transaction in an `active` state at any one time. We will often refer to this transaction as the *active* transaction. If any of the locks requested by the `active` transaction is incompatible with an existing lock, then a conflict occurs. The Scheduler, being told by the TM that a transaction's state was `active` when the conflict occurred, determines that this transaction must have been the one that caused the conflict; we call such a transaction the *interfering transaction.* There can only be one interfering transaction out of two for each conflict.

The transaction with which an interfering transaction conflicts can be in one of two states: either `pending` or `inactive`. A `pending` transaction is one that is waiting for the client to finish executing an activity. Only `tl`, `bc`, and `afc` transactions can be in a `pending` state. Such a transaction encapsulates the execution of an activation rule. Recall that a transaction encapsulating the execution of an activation rule first acquires read locks on its read set and then evaluates the property list of its condition; if the property list of the rule is satisfied, the transaction acquires write locks on its write set, and then the RP requests the client to execute the activity of the rule. Before the RP can do that, it must inform the TM, which changes the state of the transaction to `pending`; the transaction remains in that state until the execution of the activity is completed, at which point the state is changed back to `active`. Note, however, that a conflict involving a rule cannot occur after the activity of the rule has been executed because all the locks would have been acquired, and because the assertion of one of the rule's effects is performed atomically. Thus, as far as the Scheduler is concerned, a transaction in an `active` state is one that is acquiring locks, rather than one that is asserting the effect of a rule.

Finally, a transaction can be in an `inactive` state when another transaction interferes with it (i.e., the interfering transaction requests a lock incompatible with the lock that it holds). The state `inactive` indicates that a transaction is waiting for backward chaining to terminate; it also implies that one of the transactions in the backward chaining cycle is in a `pending` state. Only transactions of type `tl` or `bc` can be in an `inactive` state because other types of transactions cannot initiate backward chaining.

In addition to these three states, a transaction can be in an `ended` state. A transaction is

said to be in an ended state if its execution has been completed but it cannot be committed until its subtransactions (all of type cfc) commit. Note that a conflict can involve such a transaction only indirectly. Since a conflict cannot directly involve a transaction when it is in an ended state, we will not be concerned with this state here. In chapter 6, the PCCP protocol uses the fact that a transaction is in an ended state to execute some transaction actions that we will introduce in that chapter.

State	Type	Description
active	all types	TM is processing the transaction's request to acquire locks
pending	**tl, bc, afc**	transaction is waiting for client to finish executing activity
inactive	**tl, bc**	transaction is waiting for backward chaining to terminate
ended	all except **bi**	transaction is waiting for consistency forward chaining to complete before it can commit.

Table 5-1: Transaction States in RBDE

Table 5-1 summarizes the discussion about states of a transaction. Figure 5-5 shows a state diagram that depicts the different states that a typical transaction, encapsulating the execution of an activation rule, might pass through. Figure 5-6 shows a similar state diagram of a transaction encapsulating the execution of an inference rule. In both state diagrams, the diamond shaped box marks the point at which a conflict might happen; it is at this point that the Scheduler considers the state of the conflicting transactions to determine how to resolve the conflict. The state of the transaction at each conflict point is shown in a box next to each diamond shaped character.

One useful observation to note is that from the viewpoint of the Scheduler, a transaction whose state is active could not have performed any write operations yet. This observation is very useful when deciding whether or not to abort an active transaction because it tells us that aborting an active transaction simply involves releasing all of its locks. Only if the type of the transaction is cfc does aborting an active transaction involve more than releasing the locks (i.e., rollback and/or cascaded aborts).

In order to incorporate the type and state of a transaction in the notation we introduced in

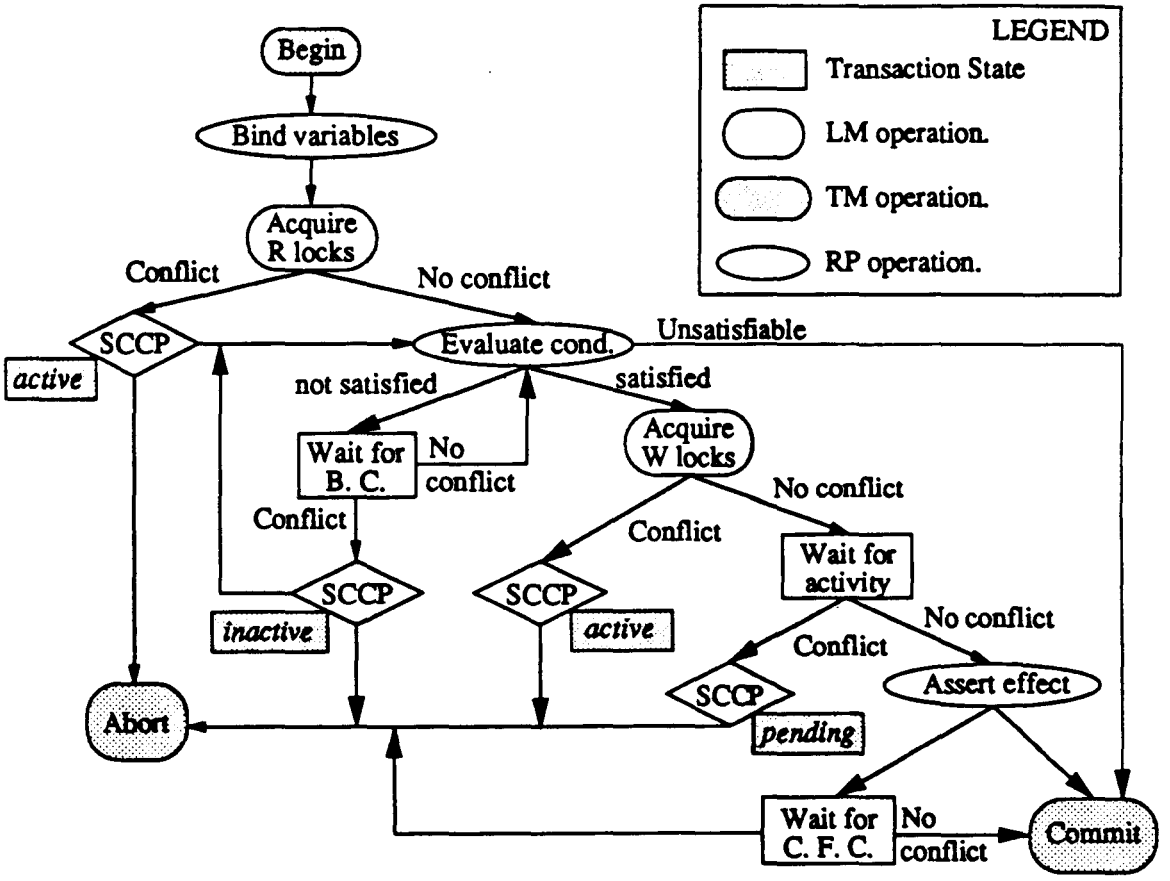

Figure 5-5: State Diagram of Transaction Encapsulating an Activation Rule

chapter 3, we extend a transaction to be a 9-tuple (rather than a 7-tuple, as in chapter 4) $T = (i, U_i, S, c, u, t, l, z, s)$, where z is the type of T and s is the state of the transaction. The rest of the elements of the tuple are as before.[17]

Now that we have described the states in which a transaction can be when a conflict occurs, we can proceed to explain the various possible kinds of interference that can occur between concurrent transactions in RBDE.

[17]To remind the reader, i is the identifier of the transaction, U_i is the set of access units comprising the body of the transaction, S is the set of subtransactions of T, c is the command whose execution is encapsulated in T, u is the owner of T, t is the timestamp of T, and l is the lock set of the transaction. We will still use $T_{i,z}$ as a shorthand notation of a transaction when we are only concerned about the identifier and the type of the transaction, or simply T_i, when we are only concerned with identifying the transaction.

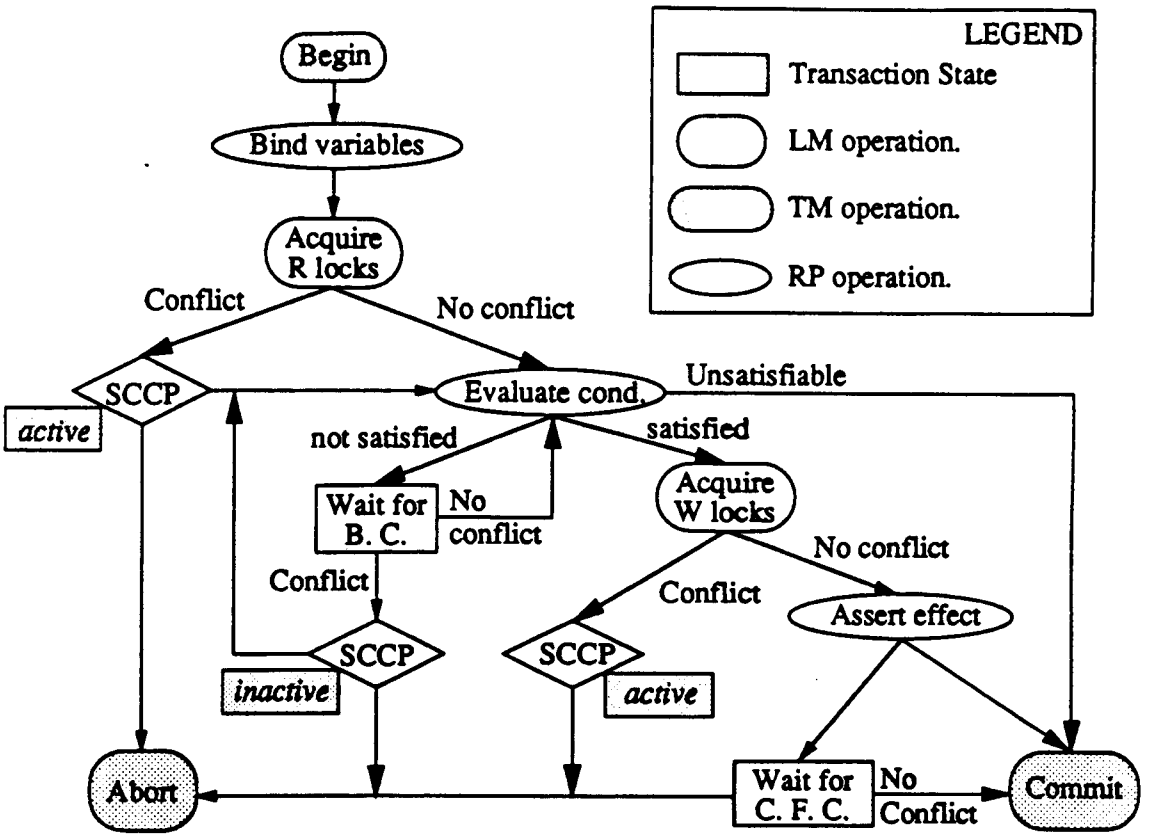

Figure 5-6: State Diagram of Transaction Encapsulating Inference Rule

5.4.3. Interference Between Two Transactions

As defined in chapter 4, a concurrency conflict (interference) occurs when a transaction T_i requests a lock, $Ml_i[o]$, that conflicts with a lock, $Nl_j[o]$, held by another transaction, T_j. From its knowledge about the types and states of transactions, the Scheduler can infer the following:

1. The state of transaction T_i is `active`, and its type can be any of the five types (`tl`, `bi`, `afc`, `cfc`, or `bc`).

2. The state of T_j must be either `pending`, in which case its type is one of three possibilities: `tl`, `bc` or `afc`, or `inactive`, in which case its type is either `tl` or `bc`.

Based on this, we can determine that there are two kinds of interference. The first kind

of interference occurs between a transaction in an `active` state and a transaction in a `pending` state. A transaction in a `pending` state does not have any subtransactions; a transaction must be either `inactive` or `ended` in order to have subtransactions. The type of the interfering transaction can be any of the five types. However, the type of the transaction that was interfered with can only be `tl`, `afc`, or `bc`. The reason is that transactions whose type is either `cfc` or `bi` cannot be interfered with.

Transactions of type `cfc` execute atomically. Thus, they cannot conflict with each other. Similarly, since `built-in` transactions are executed atomically, they never conflict with either `bi` transactions or `cfc` transactions. More precisely, if a transaction, $T_{i,z}$, where `z` is either `cfc` or `bi`, *interferes* (conflicts) with another transaction, $T_{i,y}$, then `y` must be either `afc`, `tl`, or `bc`. Similarly, no transaction can interfere with a transaction whose type is either `cfc` or `bi`. In other words, if a transaction $T_{i,z}$, where `z` is either `tl`, `afc`, or `bc`, conflicts with another transaction, $T_{j,y}$, then `y` must be either `tl`, `afc`, or `bc`.

The second kind of interference is between an `active` transaction and an `inactive` transaction. The state `inactive` implies that the transaction must have a set of subtransactions of type `bc` (backward chaining), and that one of these subtransactions is in a `pending` state (otherwise, no other transaction would have interfered with it). By the mere fact that a transaction is `inactive`, the Scheduler can infer that the transaction has not performed any write operations yet (since it is still waiting for backward chaining to make its condition satisfied).

	tl	**bc**	**cfc**	**afc**	**bi**
tl	yes	yes	no	yes	no
bc	yes	yes	no	yes	no
cfc	yes	yes	no	yes	no
afc	yes	yes	no	yes	no
bi	yes	yes	no	yes	no

Table 5-2: Interference Between Active and Pending Transactions

Tables 5-2 and 5-3 summarize our discussion about interference. The rows in both

	tl	bc	cfc	afc	bi
tl	yes	yes	no	no	no
bc	yes	yes	no	no	no
cfc	yes	yes	no	no	no
afc	yes	yes	no	no	no
bi	yes	yes	no	no	no

Table 5-3: Interference Between Active and Inactive Transactions

tables represent `active` transactions. The columns in the first table represent transactions whose state is `pending`, and in the second table they represent transactions whose state is `inactive`. A "yes" indicates that a conflict between the two types of transactions can happen and a "no" means that a conflict cannot happen. The first row in table 5-2, for example, says that a `tl` transaction whose state is `active` can interfere with `pending` transactions whose types are `tl`, `bc`, or `afc`. The first row in table 5-3 says that an `active` transaction whose type is `tl` can interfere with `tl` or `bc` transactions whose state is `inactive`.

Given its knowledge about the states and types of transactions, and its inferred knowledge about the kinds of interference, the Scheduler constructs a priority-based scheme to resolve conflicts. The objective of the scheme is to abort the "least important" transaction in order to resolve a conflict.

5.4.4. Priorities of Transactions

In order to resolve conflicts between transactions, SCCP defines a priority for each transaction running in the server. When a conflict happens between two transactions, SCCP tells the TM to abort the transaction with the lower priority. Aborting an individual transaction of any type entails rolling it back by undoing all the database write operations, if any, that were executed as part of the transaction. Aborting a transaction that has subtransactions requires that all the subtransactions be aborted first, and then the transaction can be aborted. Aborting a transaction that is a part of a consistency chain requires that the whole consistency chain, including the rule that initiated the consistency chain, be aborted. This is the only case in which cascaded aborts are necessary to maintain consistency.

5.4.4.1. Priority Based on Transaction Types

The priorities of different transactions depend on two factors: (1) the type of transaction, and (2) whether or not the transaction involves an interactive operation. With respect to the type of a transaction, the priority of different types in descending order is: `cfc` (consistency forward chaining), `tl` (top-level), `bi` (built-in), `bc` (backward chaining), and finally `afc` (automation forward chaining).

By making transactions of type `cfc` (consistency forward chaining) have the highest priority among the different types of transactions, SCCP guarantees that consistency forward chains will be allowed to continue unless they conflict with an interactive transaction (defined shortly). The reason why this is important is that `cfc` transactions are the only transactions whose abort causes cascaded aborts.

The second priority in terms of types is `top-level` transactions. A `top-level` transaction encapsulates a rule that was fired directly in response to a user command. Aborting a `top-level` transaction entails rejecting or aborting the user's command. But since the whole point of RBDE is to assist users, it should try to minimize rejection of a user's commands. Thus, a `top-level` transaction is aborted only if it conflicts with another `top-level` transaction or with a `consistency forward chaining` transaction.

Next in priority are transactions of type `built-in`, which encapsulate the execution of built-in commands. Built-in commands are requested directly by the user and thus the same consideration given to `top-level` transactions should be given to them. Unlike rules, however, built-in commands are atomic. Thus, if a `built-in` transaction conflicts with a `top-level` transaction, it must be that the rule encapsulated by the `top-level` transaction is in the middle of its execution. If we choose to abort the `top-level` transaction, we must abort the execution of the activity, in which the user might have invested many hours. On the other hand, users invest very little time and effort into executing built-in commands. If RBDE rejects a user's request to execute a built-in command (because of a conflict), the user can easily request the command at a later time. For these reasons, we have decided to make `built-in` transactions have lower priority than `top-level` transactions.

The last decision involved in assigning the priorities of transaction types was to make `backward chaining` transactions higher in priority than `automation forward chaining` transactions. The reason for this is that backward chaining is only performed when the condition of the original rule corresponding to a user command is not satisfied. Thus, the purpose of backward chaining is to try to transform objects in the database to a state that allows RBDE to execute the user's command. Automation forward chaining, in contrast, is done solely for the purpose of automatically performing optional activities. These activities were not directly requested by the user and thus not performing them will not constitute an inconvenience to the user. Aborting a backward chain, however, might imply that a command directly requested by a user will not be performed. For this reason, we chose to make `bc` transactions have higher priority than `afc` transactions.

5.4.4.2. Priority of Interactive vs. Non-Interactive Transactions

In addition to the type of transactions, SCCP considers whether the command encapsulated by each transaction is interactive or not. With regard to interactive tools, a transaction (of any type) that is either waiting for the execution of an interactive operation to complete, or that has already performed an interactive operation, has a higher priority than a transaction that does not involve interactive operations. An interactive operation is one that invokes an interactive tool. The reason for this prioritization is that we would like to decrease as much as possible the probability of aborting interactive activities, in which the human developers might have invested a lot of time and effort.

If any of the access units of the agent whose execution is encapsulated by a transaction contains an interactive operation, then the transaction is termed an *interactive transaction*. We use $IT_{i,z}$ to denote an interactive transaction. The TM determines whether a transaction is interactive or not when it first creates the transaction, based on its type and the activity of the rule whose execution is encapsulated by the transaction.

RBDE can determine whether or not a rule's activity is interactive if the rule is defined to be interactive by the administrator. A rule whose activity is interactive is denoted as such by adding the keyword `interactive` before the name of the rule. Since `interactive` is a keyword in EMSL, the loader will not accept a rule whose name is "interactive". We use `Ir` to denote an interactive rule, whose name is `r`.

To be more precise, given a transaction $T_{i,z}$, the TM denotes the transaction to be interactive if:

1. If z is not `bi` (built-in), and the rule encapsulated by $T_{i,z}$ is interactive.
2. If z = `cfc`, and the parent transaction of $T_{i,z}$ is an interactive transaction.

The second condition states that if a `cfc` transaction was initiated by an interactive transaction, then the `cfc` transaction is marked as being interactive. The reason for this is that aborting a transaction of type `cfc` will lead to aborting the transaction that initiated it, and if that transaction was interactive, then the Scheduler should take this into consideration before deciding to abort the `cfc` subtransaction.

5.4.4.3. Priority-Based Conflict Resolution

SCCP defines the following priority scheme for transactions. Given two conflicting transactions, $IT_{i,z}$ and $T_{j,y}$, $IT_{i,z}$ has a higher priority than $T_{j,y}$ if and only if one of the following conditions holds:

1. The state of $IT_{i,z}$ is `pending`, or
2. The state of $IT_{i,z}$ is either `active` or `inactive`, the state of $T_{j,y}$ is either `active` or `inactive`, and z is higher priority than (or the same priority as) y.

Condition (1) guarantees that interactive tools that are in the middle of execution will never be aborted because of a conflict with a non-interactive transaction. The rationale for the second condition is that interactive transactions whose state is `active` or `inactive` have not yet invoked interactive tools, and thus they should not cause a transaction of higher priority (type-wise) to be aborted.

Given two conflicting transactions, $IT_{i,z}$ and $IT_{j,y}$, $IT_{i,z}$ is of higher priority than $IT_{j,y}$ if and only if:

1. The state of $IT_{i,z}$ is `active` and z is `cfc` whereas y is not `cfc`, or
2. The state of $IT_{i,z}$ is `pending` and y is not `cfc`, or
3. The state of $IT_{i,z}$ is `inactive`, the state of $IT_{j,y}$ is `active`, and z is higher priority than y.

The first and second conditions establish that interactive `cfc` transactions will never be aborted, regardless of their states. The second condition states that interactive transactions that are `pending` (i.e., their interactive activity is being executed) should not be aborted because of a conflict with another interactive transaction whose activities have not been performed yet (i.e., if this other transaction is in `active` or `inactive` state). SCCP will never abort a transaction $IT_{i,cfc}$ because this kind of transaction has the highest priority, and it can never conflict with another transaction $IT_{j,cfc}$ (because consistency forward chains are carried out atomically). This guarantees that the results of interactive tools that have finished executing will never be rolled back.

The third condition states that if a conflict happens between a transaction in an `active` state and another transaction in an `inactive` state, both of which are interactive, then the priorities of the two transactions depend on their types. Neither transaction would have performed any write operations and thus there is no reason to favor one instead of the other based on their states.

Given the priorities assigned to transactions, SCCP resolves concurrency conflicts as follows. If a conflict is detected between an interactive transaction and a non-interactive transaction, SCCP uses the conditions discussed above to abort one of the transactions. If the two transactions are either both non-interactive or both interactive, then the resolution depends on the types of transactions. SCCP aborts the transaction whose type is of lower priority. If the two transactions are of the same type, then SCCP aborts the one that requested the conflicting lock (as opposed to the one that already holds the lock). Only in the case where both transactions are of type `bc` does SCCP use the timestamps of the transactions to decide which one to abort. The reason is that `bc` transactions have triggered backward chains; it makes sense to abort the more recent transaction because RBDE wouldn't have spent as much time and effort executing the backward chain it initiated. The SCCP protocol is given in figure 5-7.

```
routine DEFAULT_RESOLUTION (T_i,z, T_j,y: conflicting transactions);

  /* T_i,z is the interfering transaction. */
  /* T_j,y is the transaction interfered with. */

  Begin

    If T_i,z is interactive and T_j,y is not Then
      Begin
        If y is of higher priority then z Then
           TM_abort (i);
        Else
           TM_abort (j);
      End;

    Else If T_j,y is interactive and T_i,z is not Then
      Begin
        If state of T_j,y is ''pending'' Then
           TM_abort (i);
        Else If y is higher or same priority than z Then
           TM_abort (i);
        Else
           TM_abort (j);
      End;


    /* Either both are interactive or both are not. */

    Else If z = y Then
      If z = bc Then
        If T_i,z's timestamp is older than T_j,y's Then
          TM_abort (j);
        Else
          TM_abort (i);
      Else
        TM_abort (i);

    Else If z is of lower priority then y Then
      TM_abort (i);
    Else
      TM_abort (j);
  End.
```

Figure 5-7: Default Conflict Resolution Policy Implemented by SCCP

5.4.4.4. Details of Aborting Transactions

The only action taken by SCCP to resolve a conflict is to abort one of the conflicting transactions. We have not yet discussed the exact details of aborting a transaction. We discuss these details now to complete our discussion of SCCP.

The details involved in aborting a transaction depend on the type of the transaction, its

state, and whether or not it is interactive. Aborting any transaction involves undoing all the write operations that the transaction has performed, regardless of the type of transaction. However, as explained before, a transaction whose state is `active` would not have performed any write operations yet, so aborting an `active` transaction simply involves releasing the locks in its locks set, deleting its log file and clearing its entry from the transaction table. One complication involves `cfc` transactions.

Aborting a `cfc` transaction involves, in addition to aborting the transaction itself, aborting all the ancestors of the transaction up to and including the first ancestor whose type is not `cfc` (i.e., the transaction that initiated the consistency forward chain).

A transaction, T_i, whose state is `inactive` must be either a `tl` transaction or a `bc` transaction. As explained before, one of the subtransactions, T_j, of T_i must be in a `pending` state. Thus, aborting T_i requires finishing the execution of T_j (not aborting it, since that is not necessary), and then simply releasing the locks in the lock set of T_i. Finishing a `pending` transaction of type `bc` means that the transaction is allowed to continue executing normally (i.e., one of its effects is asserted, and all of its consistency implications are carried out), except that instead of continuing the backward chaining cycle, the cycle is terminated.

Aborting a transaction whose state is `pending` (i.e., it is in the middle of executing a development activity), requires that the server send a message to the client telling it to abort the activity in progress. This assumes asynchronous communication between the server and the clients; this kind of communication is not implemented in MARVEL, so our implementation of SCCP is limited by this fact, as will be explained in section 8.2.

5.5. Summary

In this chapter, we analyzed the EMSL and RBDE to find clues toward a solution of the concurrency control problem. We discovered that rules serve two distinct purposes, to express, enforce and maintain *consistency* and to express and carry out opportunities for *automation*. Maintaining consistency is mandatory, whereas carrying out automation is optional. We extended EMSL to distinguish between consistency predicates and

automation predicates in the conditions and effects of rules. Chaining resulting from consistency predicates is considered mandatory, whereas chaining initiated by automation predicates is optional. The distinction between these two kinds of predicates (and the three different kinds of chains that result from them) is used by the Scheduler in RBDE to provide a semantics-based concurrency control policy.

The distinction between consistency and automation rule chains is used by the TM to distinguish between five types of transactions: `tl` (top-level), `bi` (built-in), `cfc` (consistency forward chaining), `afc` (automation forward chaining), and `bc` (backward chaining). The type of each transaction is one piece of information used by the SCCP protocol to resolve a concurrency conflict.

Another piece of information that the TM makes available to the Scheduler is the states of the two transactions that interfered. When a conflict occurs one of the transactions in an `active` state (i.e., it is requesting locks) while the other is in either an `inactive` (waiting for backward chaining to terminate) or `pending` (waiting for the activity of the rule to be completed) state.

The last piece of information used by the Scheduler is whether or not the rule whose execution is encapsulated in a transaction is interactive (i.e., invokes an interactive tool). Based on this information, in addition to information about the types and states of transactions, SCCP implements a priority-based scheme that aims at aborting the "least important" of the two conflicting transactions, to resolve a conflict. The scheme avoids the possibility of cascaded aborts; it also does not suffer from deadlock, and attempts to minimize the amount of work that must be undone.

Table 5-4 summarizes SCCP's mechanism for resolving conflicts between `active` transactions and `pending` transactions. The `I` before a transaction type stands for "interactive". The columns are different transaction types whose state is `pending`, while the rows are transactions whose state is `active`; note that there are no `pending` transactions of type `cfc`, `Icfc` or `bi`. A "o" means that SCCP resolves the conflict by aborting the `pending` transaction. A blank means that SCCP resolves the conflict by aborting the `active` transaction. For example, the top leftmost box is "o"; this means

	Itl	tl	Ibc	bc	Iafc	afc
Icfc	o	o	o	o	o	o
cfc		o		o		o
Itl				o		o
tl				o		o
bi				o		o
Ibc						o
bc						o
Iafc						
afc						

Table 5-4: Resolving Conflicts Between Active and Pending Transactions

that if a conflict occurs between an `active` transaction of type `Icfc` and a `pending` transaction of type `Itl`, SCCP will ask the TM to abort the `pending` transaction. The leftmost box on the second row is blank, which means that SCCP will resolve a conflict between an `active cfc` transaction and a `pending Itl` transaction by aborting the `active cfc` transaction.

	tl	Itl	bc	Ibc
Icfc	o	o	o	o
cfc	o	o	o	o
Itl	o	o	o	o
tl	o	o	o	o
bi			o	o
Ibc				
bc				
Iafc				
afc				

Table 5-5: Resolving Conflicts Between Active and Inactive Transactions

Table 5-5 summarizes SCCP's algorithm for resolving conflicts between `active` transactions and `inactive` transactions. The columns are different transaction types whose

state is `inactive`, while the rows are transactions whose state is `active`; note that the only transactions that can be in an `inactive` state are `tl` and `bc` transactions. Again, a "o" means that SCCP resolves the conflict by aborting the `inactive` transaction; a blank means that SCCP resolves the conflict by aborting the `active` transaction. An "X" indicates that SCCP resolves the conflict by aborting the more recent (based on the timestamp attribute) of the two transactions.

This completes our discussion of the SCCP protocol. SCCP is project-specific only in the sense that the consistency constraints are. In the next chapter, we present a programmable protocol, PCCP, which can override SCCP and provide for a project-specific concurrency control policy.

Chapter 6

Programming The Concurrency Control Policy

In the previous chapter we presented the SCCP protocol, which implements the default semantics-based concurrency control policy in RBDE. When two transactions interfere with each other because of a locking conflict, SCCP resolves the conflict by aborting one of the two transactions, based on a hard-wired priority-based scheme[18]. In this chapter, we present a mechanism for overriding this default scheme. The mechanism is composed of a Control Rule Language (CRL) and a Programmable Concurrency Control Protocol (PCCP), which provides the runtime environment for CRL. The project administrator uses CRL to write *control rules*, each of which describes a specific conflict situation and prescribes a set of actions to take to resolve the conflict. If a specified conflict occurs, it is resolved by executing the set of actions prescribed by the control rule instead of the actions pre-defined by SCCP.

We first explain the limitations of the SCCP protocol and motivate the need for control rules and the PCCP protocol. We then present the details of CRL; we explain the syntax and semantics of the language, and give examples that demonstrate its constructs. Next, we present the PCCP protocol and show how it uses control rules to implement a specific concurrency control policy. Finally, we explain how the TM is extended to support the primitive transaction operations that are required by PCCP to carry out the various actions prescribed by control rules.

---

[18]Although the scheme is hard-wired, we still call the protocol a "flexible" concurrency control protocol because it is semantics-based and thus allows more flexible interactions among transactions.

6.1. Limitations of the SCCP protocol

SCCP uses the information about the types, states and timestamps of the two transactions involved in a conflict, as well as whether or not the commands whose executions are encapsulated by the transactions are interactive, to determine which of the two transactions to abort. Although SCCP employs a semantics-based policy that is less restrictive than traditional locking schemes, SCCP has three limitations:

1. The priority scheme in SCCP is hard-wired and thus cannot be changed to provide members of the same team of developers more flexibility.
2. The only action SCCP takes to resolve a conflict is `abort`.
3. SCCP does not use some of the available semantics about transactions.

The first limitation is caused by the lack of constructs in EMSL for modeling teamwork. Members of a development team that is responsible for completing a particular development task share resources and expertise. It is often desirable to make the interactions among members of the same team more flexible than those between members of one team and members of another team. Given the constructs we have presented thus far in this dissertation, there is no way for the project administrator to identify members of a development team as such. In chapter 7, we extend EMSL with two constructs, *sessions* and *domains*, that model teamwork, and integrate these two constructs into CRL.

The second limitation is inherent in all aggressive schedulers. If the project administrator wants the Scheduler to take actions other than `abort` to resolve specific conflicts, the administrator must describe the exact conflicts to the Scheduler, and prescribe the alternative actions to resolve these conflicts. To do that, we must provide a specification language and a runtime environment for this language; in addition, the TM in RBDE must be extended to provide primitive transaction operations to support any alternative actions.

Overcoming the last limitation is more complicated. When a conflict occurs, the TM knows three pieces of information: the two transactions that caused the conflict, the object over which the two transactions conflicted, and the two lock types that caused the conflict. There are several other pieces of information that are ignored.

For example, suppose that John requests to `compile` a `CFILE` object, `main.c`. The TM encapsulates the execution of `compile main.c` in a transaction T_{John}. During its execution, T_{John} obtains the lock Wl_{John}[`main.c`] before invoking the compiler on `main.c`. Suppose that while John is compiling `main.c`, Bob requests to `edit main.c`. In order to evaluate the condition of the `edit` rule, T_{Bob} requests the lock Rl_{Bob}[`main.c`]. The NGL protocol determines that Rl_{Bob}[`main.c`] is incompatible with Wl_{John}[`main.c`]. This conflict is passed to the Scheduler to determine how to proceed.

Although the SCCP protocol knows the transaction commands involved in the conflict (`edit` and `compile`), the two locks that caused the conflict, and the identity of the two users whose commands caused the conflict, it cannot use this information in a meaningful way. To be able to use these three pieces of information, the Scheduler must be provided with an "interpretation" of this information. For example, if the project administrator decides that all locking conflicts between the `edit` command and the `compile` command do not warrant aborting either, then the administrator must prescribe to the Scheduler what to do instead if such a conflict occurs (e.g., delay the `edit` until the `compile` is finished or ignore the conflict since it is not really serious, etc.).

In the rest of this chapter, we present a mechanism that overcomes the last two limitations of the SCCP protocol; the first limitation, supporting group-oriented concurrency control policies, will be addressed in chapter 7. We present a language, CRL, and a runtime environment, both of which are used (together) to enhance our concurrency control policy in three ways: (1) to prescribe actions other than `abort`; (2) to use semantic information that SCCP does not use; and (3) to tailor the concurrency control policy to different projects. We start by presenting CRL and then proceed to describe the PCCP protocol, which provides the runtime support for CRL.

6.2. The Control Rule Language

The project administrator uses CRL to write *control rules* that describe specific conflict situations, and prescribe actions that the Scheduler should execute to resolve each of the conflicts. The administrator loads the control rules for a project into RBDE whenever he wants these rules to be in effect, on the fly. The component of RBDE responsible for loading control rules is the *CRLoader*. We discuss the semantic checks that CRLoader performs on the set of control rules before loading them throughout this section and the next section.

Once the set of control rules written by the administrator of a project has been loaded, PCCP uses these control rules to provide a project-specific concurrency control policy. If no control rules are loaded into RBDE, the Scheduler employs SCCP to resolve conflicts that occur during the development of the project.

CRL should be viewed as a special-purpose language for specifying the concurrent aspects of a development process. CRL complements EMSL in this respect. CRL is thus akin to process programming languages that provide concurrency constructs, such as APPL/A [Sutton 90] and ASL [Riddle 91, Kaiser 91]. APPL/A is a programming language that extends Ada with constructs for encoding concurrent software processes. APPL/A is thus much more general-purpose than CRL, which is specifically designed to prescribe conflict resolution actions in RBDE.

The main construct provided by CRL is the control rule. Each control rule has five parts: a name, a single parameter, a selection criterion, a set of bindings, and a body composed of a set of condition-action pairs. The selection criterion and the bindings are optional; the other parts are mandatory. In this section, we present the details of each part of a control rule, explaining the syntax first and then describing the semantics.

6.2.1. Control Rule Parameters

The parameter of a control rule must be an object class — one of the classes in the project type set. A control rule, `cr`, whose parameter is `FILE`, applies only to conflicts over instances of `FILE` or any of its subclasses (e.g., `HFILE` and `CFILE`). Each control rule must have a unique name, which is used to identify it. If a conflict occurs over an instance of a class and there are no control rules that apply to this class (or any of its superclasses), the Scheduler calls SCCP to resolve the conflict. Otherwise, the Scheduler calls PCCP to fire the control rule that most closely matches the conflict situation among the control rules that apply to the class of the object over which the conflict occurred.

6.2.2. The Selection Criterion of a Control Rule

Many different conflict situations occur on instances of a single class; the project administrator might want to specify a control rule for each of these situations. The conflict situation to which a control rule applies is described in the *selection criterion* of the control rule. The selection criterion is a triple $S = (C, U, \langle N,M \rangle)$. The first part, C, is called the command specification (or command spec for short) and it specifies a set of commands. If the commands encapsulated by the two conflicting transactions are elements (members) of the set C, then we say that the command spec matches the conflict situation. The second part, U, is the user specification (user spec for short), which specifies a set of users. Again, if the owners of T_i and T_j are elements of U, we say that the user spec matches the conflict situation. The last part, the lock specification (lock spec for short), specifies the two incompatible lock types that caused the conflict.

Each of the three parts of the selection criterion can be empty, which indicates that the information for that part is immaterial to the selection of `cr`; we use $\emptyset$ to indicate an empty part. For example, the control rule whose selection criterion is $(\{c1,c2,c3\}, \emptyset, \langle M,N \rangle)$ applies to conflict situations caused by the two lock types `M` and `N`, and involving any two of the commands `c1`, `c2` and `c3`, regardless of the specific users who own the conflicting transactions. A control rule with the selection criterion $(\emptyset, \{Bob, Mary, John\}, \emptyset)$ applies to any conflict caused by transactions belonging to

Bob and Mary, Mary and John, Bob and John, Bob and Bob, etc.[19]. Such a control rule basically establishes John, Bob, and Mary as a team (as discussed in chapter 7, this is not sufficient for modeling teamwork).

A control rule, `cr`, with an empty selection criterion, called a ***generic control rule***, applies to all conflicts over instances of the class specified as the parameter. Such a control rule will be selected by the Scheduler to resolve a conflict only if a more specific control rule cannot be found.

```
compile_edit_cr [ CFILE ]

selection_criterion:
    users: Bob, Mary, John;
    commands: edit, compile;
    lock_modes: R, W;
```

Figure 6-1: The Selection Criterion of a Control Rule

An example showing the name, parameter, and the selection criterion of a control rule is given in figure 6-1. The keywords are in **bold face**. This control rule applies to conflicts over a `CFILE` involving any two of the users Bob, John and Mary; the control rule applies only when the two users concurrently request to `edit` and `compile` the same `CFILE`, leading to a locking conflict (one of them will require an `R` lock in order to evaluate its condition while the other one would have already acquired a `W` lock in order to execute the activity). The selection criterion does not specify which of the two users requested which of the two commands and in which order. Note that both `compile` and `edit` might have been fired during chaining, rather than requested directly by the users.

The selection criteria of the control rules written for a specific project determine which control rule the Scheduler will select to resolve a conflict that occurs during the development of the project. The selection is based on the notion of *closest match*. First of all, if any of the three parts of a selection criterion is not empty and it does not match the

[19]As will be discussed in the last chapter, it would be more meaningful if the user spec part specified user roles instead of user names or userids.

information in the conflict situation, we say that there is no match between that selection criterion and the conflict situation. If the Scheduler does not find any control rule that matches the conflict situation, then the Scheduler will call SCCP to resolve the conflict. If there is a match, then the match can either be full or partial. Let us first define what we mean by a *full match* more formally, and then define what we mean by a closest partial match.

Definition 1: Given a conflict situation $C = (\langle T_i, T_j \rangle, \langle Ml_i[o], Nl_j[o] \rangle)$, and a selection criterion $S = (\{c_1, \ldots, c_n\}, \{u_1, \ldots, u_k\}, \langle M', N' \rangle)$, there is a *full match* between S and C if and only if:

1. $T_i.command \in \{c_1, \ldots, c_n\} \wedge T_j.command \in \{c_1, \ldots, c_n\}$, and
2. $T_i.owner \in \{u_1, \ldots, u_k\} \wedge T_j.owner \in \{u_1, \ldots, u_k\}$, and
3. either $Ml_i[o].type = M'$ and $Nl_j[o].type = N'$, or $Ml_i[o].type = N'$ and $Nl_j[o].type = M'$.

Note that the order of the elements within a set in the selection criterion is immaterial. Thus, all of the following selection criteria are considered identical:
$S = (\{Mary, John, Bob\}, \{edit, compile\}, \langle R,W \rangle)$,
$S = (\{Bob, John, Mary\}, \{edit, compile\}, \langle R,W \rangle)$,
$S = (\{Bob, Mary, John\}, \{compile, edit\}, \langle W,R \rangle)$.

If the Scheduler does not find any control rule whose selection criterion fully matches a conflict, the Scheduler selects the rule whose selection criterion most closely matches the conflict. How closely the selection criterion of a control rule matches a conflict depends on how many, and which, of the three parts of the selection criterion match the conflict situation. Given that we have three parts in a selection criterion and that each of these three parts can be empty, there are eight possible matches to a conflict situation. We need some notation to simplify the discussion of partial matches. We use the letter *U* to denote that the user spec of the selection criterion matches the owners of the two conflicting transactions. Similarly, we use *C* and *L* to denote matches to the command spec and the lock spec of a selection criterion, respectively. Thus, a match *UCL* is a full match; *CL* denotes a partial match in which the command spec and the lock spec of the selection criterion match the conflict situation, but the user spec part is empty (as opposed to non-matching). Ø denotes an empty selection criterion, which is the weakest

match to a conflict situation on an instance of the class (or a subclass of the class) specified as the parameter of the control rule.

In order to determine how closely a partial match matches a conflict situation, we assign a *specificity* level to each kind of partial match. We have a default assignment of the specificities, but the administrator can define an alternate assignment and load it with the control rules. The default specificity levels of partial matches are as follows: $UCL = 7$ (full match), $UC = 6$, $UL = 5$, $CL = 4$, $U = 3$, $C = 2$, $L = 1$, and finally $\varnothing = 0$. This assignment is based on the rationale that the administrator would want to write user-specific control rules, command-specific control rules, and lock-specific control rule, in that order. The higher the specificity level, the closer the selection criterion matches the conflict. If the Scheduler finds two control rules whose selection criteria partially match the conflict, the Scheduler selects the control rule whose selection criterion is more specific (i.e., its specificity level is higher).

The selection criteria of two different control rules that apply to the same class cannot have the same match specificity with respect to a conflict situation. The reason for this is that if two control rules, `cr1` and `cr2`, apply to class `A`, then their selection criteria must be different; if they are not, then the two control rules should be merged into one because both of them apply to exactly the same conflicts. The CRLoader will not load the set of control rules written by the project administrator if two control rules have both identical parameters and identical selection criteria. The administrator must either merge the two control rules or remove one of them, and then attempt to reload the control rules.

Two selection criteria $S1 = (C1, U1, <M1, N1>)$ and $S2 = (C2, U2, <M2, N2>)$, where $C1 = \{c1_1, \ldots, c1_n\}$, $C2 = \{c2_1, \ldots, c2_m\}$, $U1 = \{u1_1, \ldots, u1_k\}$, and $U2 = \{u2_1, \ldots, u2_l\}$, are different if:

1. $(\exists\, c_i \in C1 \mid c_i \in C2) \rightarrow (\forall\, c_j \in C1, c_j \neq c_i)\, c_j \notin C2$, **or**
2. $(\exists\, u_l \in U1 \mid u_l \in U2) \rightarrow (\forall\, u_m \in U1, u_m \neq u_l)\, u_m \notin U2$, **or**
3. $M1 \notin <M2, N2> \vee N1 \notin <M2, N2>$.

The first condition states that at most one of the commands in the command spec of $S1$ can be also in the command spec of $S2$. To understand the reason for this condition,

suppose that in the two selection criteria $S1$ and $S2$, $U1 = U2$, $M1 = M2$, and $N1 = N2$. Say that the command spec of $S1$ is {*edit, compile, build*}, and the command spec of $S2$ is {*edit, view, archive, compile*}; both `edit` and `compile` are part of the two selection criteria. Now suppose that the conflict situation $C = (<T_i, T_j>, <Ml_i[o], Nl_j[o]>)$, where $T_i.command = edit$, $T_j.command = compile$, and both the user and lock specs of both selection criteria match the conflict situation. This situation results in ambiguity because the selection criteria of both of the two control rules fully match the conflict situation. Thus, the two control rules should be merged into one control rule whose selection criteria is ({*edit, compile, build, archive, view*}, $U1$, $<M1, N1>$).

The second condition states that at most one of the users in the user spec of $S1$ can be also in the user spec of $S2$. The justification of the second condition is similar to the reasoning above. The third condition simply states that the two lock pairs are not identical. In other words, two selection criteria are different if any one of the three parts of the first criterion is different from the same part of the second criterion. Different selection criteria do not have the same specificity with respect to a conflict; one of them must be more specific to the conflict. This realization is used by the Scheduler when searching for the control rule that is most specific to a conflict.

6.2.3. The Binding Part

The third part of a control rule is a set of binding statements, each of which binds one variable to either a transaction or to a lock. The binding statements are used to bind the information available about a conflict situation (and other information that can be derived from it) to variables; the bound variables are then used in the body of the control rule. Each binding statement consists of a variable name, beginning with "?" (to keep notation consistent with EMSL), followed by "=" and a binding function. There are five binding functions in CRL (in chapter 7 we add two more): `holds_lock()`, `requested_lock()`, `lock()`, `parent()`, and `top-level()`.

The first three functions are used to bind the two conflicting transactions, and the two locks that caused the conflict, to variables that then can be used in the body of the control rule. CRL uses the term `conflict_object` to refer to the object over which the

conflict that fired the control rule occurred. We will call this object the ***conflict object*** hereafter. `Holds_lock()` (or `requested_lock()`) returns the identifier of the transaction that holds (or requested) one of the two incompatible locks on the conflict object. These two functions can appear only once in the binding part of a control rule. `Lock(?t)` returns the lock that the transaction bound to `?t` either holds or has requested (whichever the case may be) on the conflict object. The last two functions are used to obtain additional information about the conflicting transactions. `Parent(?t)` (or `top-level(?t)`) returns the parent (or top-level) transaction of the transaction bound to the variable `?t`.

Before a variable can be used in the right hand side (i.e., after the "=") in a binding statement, or in the body of a control rule, the variable must first be bound. To simplify the discussion, we use the term "entity" to refer to either the transaction, conflict object, or lock bound to a variable.

```
compile_edit_cr [ CFILE ]

selection_criterion:
   commands: edit, compile;
   users: Mary, John, Bob;
   lock_modes: R, W;

bindings:
   ?t1 = holds_lock ();
   ?t2 = requested_lock ();
   ?l1 = lock (?t1);
   ?l2 = lock (?t2);
   ?t3 = parent(?t1);
   ?c1 = top-level (?t1);
   ?c2 = top-level (?t2);
```

Figure 6-2: The Binding Part of Control Rule

To illustrate binding statements, consider the partial control rule shown in figure 6-2. Suppose that John and Bob requested two commands that initiated the two rules `edit` and `compile`, respectively. The TM encapsulates the execution of `edit` in a transaction, T_{John}, and the execution of `compile` in another transaction, T_{Bob}. Further suppose that T_{Bob} has acquired a `W` lock on `main.c`, and is in the process of compiling `main.c`. Meanwhile, T_{John} has requested an `R` lock on `main.c`, to evaluate the con-

dition of the `edit` rule. T_{John}'s request causes a conflict; suppose that the Scheduler fires the control rule shown in figure 6-2 to resolve the conflict.

When processing the control rule, `?t1` will be bound to T_{Bob}, `?t2` to T_{John}, `?l1` to $Write_{Bob}$[`main.c`], and finally `?l2` will be bound to $Read_{John}$[`main.c`]. If T_{John} is a `top-level` transaction, then `?c2` will be assigned to T_{John} as well (since a `top-level` transaction is its own top-level). Assuming that T_{Bob} is part of a chain, `?t3` will be assigned the parent transaction of T_{Bob}, and `c1` will be bound to the top-level transaction of the nested transaction containing T_{Bob}. All of these entities (the transactions and the locks) can now be referred to in the body of the control rule.

6.2.4. The Control Rule Body

The last part of a control rule is its body, which consists of a set of condition-action pairs. The conditions are evaluated in order, stopping at the first one that evaluates to `TRUE`; the set of actions corresponding to the satisfied condition is executed. At most one set of actions, corresponding to only one satisfied condition, is executed. Each condition tests some properties of the bound variables. Note that when a variable is bound to a transaction, the condition might check the timestamp, the type, the owner, and the lock set of the transaction. The purpose of the condition is to enable the writer of the control rule to use the semantics of transactions and locks.

6.2.4.1. Conditions

The condition of a condition-action pair is a logical expression in which the connectives `and`, `or` and `not` are used to connect predicates. Each predicate is a simple atomic formula that checks the value of one of the attributes of the entity bound to a variable. And, `or` and `not` are used as connectives to form the clauses. CRL supports two kinds of predicates: comparison predicates and set predicates.

A comparison predicate compares the value of an attribute of the entity bound to a variable either to a constant or to the value of the attribute of another entity. Comparison predicates are of one of two forms: either `(?t1.att1 op value)` or `(?t1.att1 op ?t2.att2)`, where `?t1` and `?t2` are variables that have been

bound in the binding part, `att1` and `att2` are attributes of the entities bound to `?t1` and `?t2`, respectively, and `value` is a constant of the same type as `att1`. There are six comparison operations: "=" (equals), "!=" (not equals), ">" (greater than), "<" (less than), ">=" (greater than or equals), "<=" (less than or equals).

As explained earlier, a variable can be bound to either a transaction or a lock. Locks have only one attribute, which is the `type` of the lock. Thus, the only operators that can be used in predicates involving locks are "=", "!=", ">" and "<". The last two are used to compare the strengths of two lock types. Furthermore, if the predicate is of the form `(?v.type op value)`, where `?v` is a variable bound to a lock, then `value` must be one of the six lock types[20]. If it is not, a semantic error is reported by CRLoader.

Transactions, unlike locks, have nine attributes, of which we are concerned with only seven here. The values of the `owner`, `type`, `command` and `state` attributes are strings; thus, the two operations "=" and "!=" are sufficient for comparing the values of these four attributes. Consider a predicate of the form `(?t.att op value)`. If `att` is `owner` or `command`, then `value` can be any string (a user's id, such as `bob`, or the name of a command in RBDE, like `edit`). If the attribute is `state`, then `value` must be one of the three possible states of a transaction. Finally, if the attribute is `type`, then `value` must be one of the five types of transactions (`tl`, `bi`, `afc`, `cfc`, or `bc`).

The `timestamp` attribute of a transaction is a number. Thus, all six comparison predicates are needed to compare this number with other numbers (either constants or the `timestamp` attribute of another transaction). By comparing the timestamps of two transactions, we can determine if the two transactions have executed (or are still executing) concurrently. By comparing both the `timestamp` and the `state` attributes of two transactions, we can determine whether one transaction preceded (i.e., has started and ended) before the other.

---

[20]Recall that the lock types are not built-in but loaded from a file together with the compatibility matrix and the strength assignments. Thus, the administrator can define alternative lock types and a compatibility matrix, and strength assignments for lock types.

Both of the `lockset` and `subtransactions` attributes of a transaction are sets. To manipulate these attributes, we need set operations. CRL provides the two predicates `member` and `notmember` to check whether a particular transaction (or lock) is a member of a transaction's `subtransactions` (or `lockset`) attribute. In addition, CRL, like EMSL, supports existential and universal quantifiers, in CRL's case over members of either the `subtransactions` attribute or the `lockset` attribute of a transaction.

```
bindings:
   ?t1: requested_lock();
   ?t2: holds_lock();
   ?c1: top-level(?t1);

body:

   if (and (forall member [?c1.subtransactions ?t]
             suchthat (?t.timestamp > ?t2.timestamp))
           (exists member [?t1.lockset ?l]
             suchthat (and (?l.object = conflict_object)
                           (?l.type = R))))
     ...
```

Figure 6-3: Example Control Rule Condition

For example, the condition in figure 6-3 shows two clauses that manipulate set attributes of transactions. The first clause checks if the `timestamp` attribute of any of the subtransactions of the transaction bound to `?c1` is more recent (i.e., greater) than the `timestamp` attribute of the transaction bound to `?t2`. The second clause checks if the transaction bound to `?t1` has acquired an `R` lock on the object over which the conflict occurred. Note that unlike EMSL, CRL does not need to distinguish between the binding of the variable `?t` and checking its attributes; the reason is that PCCP does not support backward chaining and thus if a condition fails, PCCP does not care why it failed (i.e., which predicates caused it to fail), which is the reason for the separation in EMSL.

The evaluation of the condition of a condition-action pair is very similar to evaluating the condition of an EMSL rule (in fact, in the implementation, much of the same code is used for both evaluations). If the complex clause comprising the condition of a condition-action pair is evaluated to be TRUE, then PCCP proceeds to execute the set of actions corresponding to the condition.

6.2.4.2. Actions in CRL

The only action SCCP takes to resolve a conflict is to abort a transaction. CRL extends the possible actions the Scheduler can take to resolve a conflict by providing four primitive actions in addition to `abort`: `terminate`, `suspend`, `merge`, and `notify`. The first three, like `abort`, are actions that the TM executes on specific transactions; the fourth is a notification mechanism used to send warning or explanation messages to users. The sequence of actions corresponding to the first condition that is found to be TRUE is executed, one action at a time and in a serial order. The details of how PCCP executes `terminate` and `suspend` are discussed later in section 6.3; the details of `merge` will be discussed in chapter 7 after introducing the notion of development domains. We give a brief overview of each action here.

`Abort(?t)` tells the TM to abort the transaction bound to the variable `?t` (which can be either one of the two transactions involved in the conflict that triggered the control rule, or an ancestor of one of these two transactions). As explained in chapter 3, aborting a transaction entails rolling it back. If the transaction is part of a consistency chain, the whole chain must be rolled back. If the transaction has subtransactions, then these subtransactions must be aborted first. `Terminate(?t)`, in contrast, tells the TM to abort the transaction bound to `?t`, but not to carry out any cascaded aborts; this action is used to stop a consistency chain that has caused a conflict, without rolling it back.

Terminating a consistency chain in the middle is, by definition, a violation of the database consistency; it leaves objects in an inconsistent state. In general, this should never be allowed. It might be necessary under certain circumstances, however, to tolerate inconsistency temporarily, in order to save effort that would have been wasted if a consistency chain is aborted. We will present a technique for tolerating inconsistency later on in this chapter.

Aborting or terminating a transaction is too severe an action under certain circumstances. For example, say that Bob wanted to link module `ModA`, while John is compiling `f2.c`, which is contained in `ModA`. Instead of aborting the transaction encapsulating the execution of the `link ModA` command, the Scheduler might prefer to delay the `link`

activity until after the compilation of `f2.c`, which is known to take a very short time, has been completed.

`Suspend(?t1, ?t2)`, like blocking in traditional transaction mechanisms, involves suspending the execution of the transaction bound to the variable `?t1` until after the execution of the transaction bound to `?t2` finishes. The execution of the transaction bound to `?t1` is resumed thereafter. Suspending a transaction allows the execution of all the access units of another transaction to be done before the execution of the suspended transaction continues. Finishing the execution of the transaction bound to `?t2` entails completing the execution of the command encapsulated in the transaction, if the execution of the command's activity has already started, and carrying out all the consistency implications (i.e., inference rules) of the command; unlike normal execution, however, finishing a transaction does not initiate any automation forward chaining transactions (i.e., the automation implications of the command are not carried out).

`Merge(?t1, ?t2)` merges the two transactions bound to the variables `?t1` and `?t2` into one transaction. Merging two transactions involves transferring the set of locks and all the subtransactions of the transaction bound to the variable `?t1` to the transaction bound to `?t2`, which continues normal execution from that point on. The owner of the transaction bound to `?t1` will be informed that the execution of his command has been discontinued and that further chaining resulting from his command will be merged with another user's chain. The main purpose of this action is to carry out as much of the automation implications of a user's command as possible.

Finally, the `notify(?t, ''message'')` action is used to send a message to the owner of the transaction bound to `?t`. If the server is not currently executing that user's command, the communication line between the server and the client would not be open until the client sends a message to the server. The TM inserts the notification message in the client's context so that before the server processes the next message of the client, it sends the notification message to the client.

An example control rule that shows most of the syntax is shown in figure 6-4. This control rule is selected when a conflict, involving the two commands `edit` and

```
compile_edit_cr [ CFILE ]

selection_criterion:
   commands: edit, compile;

bindings:
   ?t1 = holds_lock ();
   ?t2 = requested_lock ();

body:

   if (?t1.command = edit) then
     {
       abort (?t2);
       notify (?t2,"User %s is updating object", ?t1.owner);
     };

   if (?t2.command = edit) then
     {
       notify (?t2, "User %s is compiling the object,
       I will let you edit it when (s)he's done.", ?t1.owner);
       suspend(?t2, ?t1);   # suspend c2 until r1 is finished.

     };
;
```

Figure 6-4: An Example Control Rule

compile, occurs over a CFILE object. The control rule specifies that if the edit is already in progress, then the compilation should not be started (i.e., it should be aborted), which is what SCCP would have done; however, if the compilation is in progress, then rather than aborting the edit, delay it until the compilation, as well as all of its consistency implications, has been completed. This guarantees that no other transaction will "sneak in" and "steal" the locks that the transaction bound to ?t1 needs between the time the compilation is completed and when the edit command is re-issued (if it were aborted).

This example is intended to give the reader a flavor of what control rules can express. We will now explain the details of how PCCP selects and executes control rules. We will give several examples that demonstrate how control rules can be used to prescribe some of the unconventional concurrency control policies that have been proposed in the literature.

6.3. PCCP: A Programmable Concurrency Control Protocol

PCCP searches for a control rule whose selection criterion matches the conflict situation as closely as possible. In order to facilitate this search, the control rules are organized into a list of hierarchies, based on their parameters and their selection criteria. Each hierarchy is composed of all the control rules that apply to a single object class. Within a single hierarchy for a class, there might be several control rules that apply to different conflict situations that can occur over instances of the class. These control rules are organized based on the specificity of their selection criteria, so that PCCP can find the most specific control rules when a conflict arises. The most general control rule for a specific class of objects, the generic control rule, has an empty selection criterion (i.e., its specificity to any conflict is 0, as explained earlier). A generic control rule for class A can be applied when PCCP could not find a more specific control rule that applies to the conflict situation.

Within the same hierarchy, control rules are organized into seven levels; each level corresponds to a specificity level. In the following discussion, we assume that all the control rules apply to object class A (i.e., these control rules have A as their parameter). All the control rules whose selection criterion has a specificity level of 6 are grouped together at the first level; these control rules form a list. The selection criterion of each of these rules contains non-empty user spec, command spec, and lock spec.

All the control rules whose selection criterion has a non-empty user spec and a non-empty command spec, but an empty lock spec, are grouped at the second level; the selection criterion of each of these rules has a specificity level of 5. The remaining five levels are organized in the same way. Note that if the project administrator provides an alternative assignment of specificity levels, then the six levels will be organized differently, based on the administrator-defined assignment.

The easiest way to explain how PCCP searches for the most specific control rule is to give an example. Suppose that PCCP is given a conflict situation $C = (\langle T_{John}, T_{Mary}\rangle, \langle Rl_{John}[main.c], Wl_{Mary}[main.c]\rangle)$, where *main.c* is an instance of class CFILE. John requested to `edit main.c` while Mary was compiling `main.c`.

T_{John} causes a conflict when it requests a read lock on `main.c` (in order to evaluate the condition of the `edit` rule). PCCP is passed information about this conflict, so it attempts to resolve it by executing a control rule.

PCCP first searches for the control rule hierarchy that applies to class `CFILE`. If PCCP could not find such a hierarchy, it informs the Scheduler that it could not resolve the conflict. If PCCP finds the specific hierarchy, it searches each of the six levels of the hierarchy in order, starting with the first level, and stopping when a control rule is found. If PCCP finds a matching control rule in the first level, then the match must have been full. In other words, the selection criterion of the control rule PCCP found must have been $S = (\{John, \ldots, Mary, \ldots\}, \{edit, \ldots, compile, \ldots\}, \langle R,W \rangle)$. If PCCP could not find a control rule that fully matches the conflict situation, it starts searching the second level to find a control rule whose user spec and command spec match the conflict situation. If PCCP finds a control rule whose selection criterion is $S = (\{John, \ldots, Mary, \ldots\}, \{edit, \ldots, compile, \ldots\}, \emptyset)$, it selects that control rule. Otherwise it goes on to the third level and so on. Recall that the order of the entries in a pair in a selection criterion is immaterial[21].

Once PCCP finds a control rule, it is guaranteed that this is the most specific control rule (otherwise it would have found a more specific one earlier at a higher level). In the worst case, PCCP might have to search all the levels of the hierarchy. If no control rule matches the conflict situation, and if there is a generic control rule (i.e., whose selection criterion is empty) for class `CFILE`, then PCCP selects that generic control rule. If there is not a generic control rule, then PCCP gives up and informs the Scheduler that it could not resolve the conflict situation.

---

[21] Although this search routine is not the most efficient one, it is sufficient for our purposes. We expect the number of control rules to be sufficiently small that searching the control rule hierarchy will never be very costly.

6.3.1. Executing a Control Rule

Once PCCP has selected the most specific control rule, it proceeds to bind the variables in the binding part of the control rule. PCCP then goes on to evaluate the condition of the first condition-action pair in the body of the selected control rule.

The evaluation of the complex clause that comprises the condition is very similar to the evaluation of the property list of a rule in EMSL (presented in section 2.4). Evaluation of the predicates is also similar except that instead of checking the values of objects' attributes, the predicates in CRL involve obtaining the values of transactions' and locks' attributes.

```
routine PCCP (conflict : a conflict situation);
  Begin
    cr := SELECT_CONTROL_RULE (conflict);
    If cr is empty Then
       return FALSE;
    BIND_CR_VARIABLES (cr, conflict);
    While not done Do
      Begin
        pair := the next condition-action pair of cr;
        cond := the complex clause of the condition of pair;
        actions := the set of actions of pair;
        status = EVALUATE_CR_CONDITION (cond);
        If status = TRUE then
          Begin
            For each action, action, in actions Do
              illegal := VERIFY_ACTION (action);
            If illegal = TRUE Then
               continue;  /* Go to the next pair. */
            For each action, action, in actions Do
               EXECUTE_CR_ACTION (action);
            done := TRUE;
          End;
      End;
    If done = TRUE Then
       return TRUE;
    else
       return FALSE;
  End.
```

Figure 6-5: The PCCP Protocol

If PCCP evaluates the condition of a condition-action pair to be TRUE, PCCP asks the TM to execute each of the actions in the corresponding set of actions, one action at a time. The complete algorithm followed by the PCCP protocol is shown in figure 6-5. The routine `EXECUTE_CR_ACTION()` calls the TM to execute the specific action. The routine `VERIFY_ACTION` (action) checks if `action` is legal; the legality of an action depends on the type and state of the transaction involved in the action. We will discuss the legality of CRL actions in detail in the next section.

6.4. Extending the TM to Handle CRL Actions

The set of transaction operations supported by the TM that we presented in chapter 3 — `begin`, `commit` and `abort` — must be extended to include the actions prescribed by CRL. In particular, the TM must support three additional transaction operations: `terminate`, `suspend`, and `merge`. We present the details of the first two of these operations in the rest of this section. The `merge` operation will be discussed in chapter 7.

6.4.1. Suspending Transactions

PCCP suspends a transaction in order to avoid aborting either of the transactions involved in a conflict. Suspending a transaction is similar to blocking a transaction in traditional two-phase locking, except that in our case, the transaction is blocked waiting for another transaction to either commit or abort, rather than for a lock to be released. Since transactions in RBDE are long-lived, the `suspend` action should be prescribed only when the administrator knows that either the execution of the transaction on whose termination the suspended transaction depends will not last for a long time or that it is OK to wait.

To be more precise, consider the conflict situation we discussed earlier, $C = (\langle T_i, T_j \rangle, \langle Ml_i[o], Nl_j[o] \rangle)$. Given such a conflict situation, SCCP would abort either T_i or T_j. To avoid this, the administrator writes a control rule specifying that instead of aborting one of the transactions, the interference is avoided (i.e., the conflict is resolved) if one of the transactions is suspended until the other finishes its execution.

When the conflict situation occurs, the Scheduler will employ PCCP, which will select the control rule that the administrator wrote and instruct the TM to perform the action `suspend` (T_i, T_j). Performing this action actually involves three steps: (1) suspending the execution of T_i; (2) *finishing* the execution of T_j; and (3) resuming the execution of T_i.

As explained earlier, the state of the interfering transaction in a conflict situation is `active`, and the state of the transaction that has been interfered with is either `pending` or `inactive`. A transaction whose state is `pending` cannot be suspended because it is in the middle of executing an activity that might be changing the contents of a data attribute; suspending such an activity halfway may corrupt the data attribute (if that attribute is accessed by any other transaction meanwhile). Thus, given a conflict involving two transactions T_i, whose state is `active`, and T_j, whose state is either `inactive` or `pending`, the control rule can prescribe suspending either T_i (with no restrictions) or T_j if the state of T_j is `inactive`.

Before PCCP starts executing any action in a condition-action pair, it first verifies that it can perform all the actions. If it cannot do that, then the condition-action pair is treated as if its action is not satisfied (i.e., it is skipped over). Thus, PCCP checks the state of the transaction that the control rule prescribes suspending to make sure that it is either `active` or `inactive`. We will introduce further checks that PCCP must carry out before executing any actions throughout the rest of this chapter.

In order to remember that it must resume a suspended transaction after finishing another, the TM maintains a *suspend queue* for each transaction. If a transaction T_i is suspended until another transaction T_j is finished, then T_i is inserted on T_j's suspend queue. Once a transaction either aborts or commits (or is terminated, as will be discussed later on), the TM resumes the execution of each transaction in its suspend queue in turn in the order in which they were inserted (i.e., the queue is first-in-first-out, FIFO).

Having explained the idea of suspending a transaction and some of the restrictions that apply, we now proceed to explain the details of how the TM suspends a transaction. For our discussion, we assume that the selected control rule that is being executed by PCCP

matches a conflict situation $C = (<T_i, T_j>, <Ml_i[o], Nl_j[o]>)$, where the state of T_i is `active` and the state of T_j is either `inactive` or `pending`. Recall that the fact that a transaction's state is `inactive` means that the type of the transaction is either `bc` or `tl`. We will first discuss how the TM can suspend T_i, and then reverse the situation and explain how the TM suspends T_j if its state is `inactive`.

6.4.1.1. Suspending an Active Transaction

The fact that a transaction is in an `active` state means that it is in the middle of acquiring locks and that it has not performed any write operations yet. In fact, the cause of the conflict situation is T_i's request to obtain the lock $Ml_i[o]$. The TM uses this information when suspending an `active` transaction. The TM follows the same procedure in suspending `active` transactions of all types except `cfc`. Transactions of type `cfc` require additional steps because the semantics of consistency forward chains require that they be executed atomically. We will first discuss how the TM suspends transactions whose types are `tl`, `bi`, `afc`, and `bc`, and then explain what is involved in suspending `cfc` transactions.

The TM suspends a transaction, T_i whose state is `active` and whose type is anything but `cfc` by delaying processing its request to obtain $Ml_i[o]$ until after the execution of T_j is finished. The TM simply releases all the locks in the lock set of T_i, including $Ml_i[o]$. The TM changes the state of T_i to `suspended` and inserts T_i on the *suspend queue* belonging to transaction T_j. Note that since T_i's locks have been released, T_i cannot be involved in any conflicts while it is suspended.

The TM then proceeds to *finish* the execution of T_j. We will discuss how the TM finishes the execution of a transaction shortly, but first we describe how the suspended transaction is resumed. When the execution of T_j is finished, the TM resumes the execution of the transactions on the suspend queue of T_j, one by one in a first-in-first-out order (the first transaction on the queue must be finished before the next one can start, and so on). Resuming the execution of a transaction whose state is `suspended` simply means restarting the transaction from scratch. Note that the only thing that needs to be redone is forming the read and write sets of T_i. This is necessary because the execution of T_j (or other transactions that executed while T_i was suspended) may have added or deleted

objects and changed the values of objects' attributes. Thus, the read and write sets of T_i may need to be changed (to add or remove objects from them).

Finishing the execution of T_j (whose state is either `inactive` or `pending`) depends on its type and its state. The type of a transaction whose state is `inactive` can be either `tl` or `bc`. Recall that the fact that a transaction, T_j, whose state is `inactive` is involved in a conflict means that T_j must have a subtransaction of type `bc` in a `pending` state. Say that this subtransaction is T_k. Finishing the execution of T_j thus entails finishing the execution of T_k, which is the same as finishing a `pending` transaction, as will be discussed later, and then committing T_j after committing all of its subtransactions (even if they have not been executed yet). In other words, finishing the execution of a transaction that is waiting for a backward chaining cycle to complete requires prematurely terminating the backward chaining cycle after finishing the execution of the rule that is currently running (and all of its consistency implications).

The type of a `pending` transaction is either `tl`, `bc`, or `afc`. The command encapsulated by such a transaction must be an activation rule (otherwise, the transaction would never be in a `pending` state). Finishing the execution of a transaction whose state is `pending` involves completing the execution of the activity that is in progress, asserting one of the effects of the activation rule, carrying out all of its consistency implications (all of which are inference rules, as defined in chapter 5), and finally committing the transaction, assuming it is not involved in any further conflicts. Note that the only difference between finishing the execution of a transaction and normally completing it is that the TM does not initiate any automation forward chaining transactions when finishing a transaction.

Having discussed the details of suspending a transaction T_i, whose state is `active` and whose type is `tl`, `bc`, `afc`, or `bi`, we now complete the discussion by considering the case where the TM suspends a transaction of type `cfc`. Suspending a transaction of type `cfc` is complicated because releasing the locks of the transaction might result in an incorrect execution of a consistency forward chain. This can happen if one of the locks acquired by the transaction was actually transferred to it from one of its ancestors. Recall that transferring locks from parent transaction to subtransaction is done to avoid

intra-transaction conflicts. Let us construct an actual scenario to provide context for our discussion.

A transaction, T_i, of type `cfc` is a subtransaction of some other transaction, by definition. Consider a transaction $T = (k, U_k, S, c, u, t, l, z, s)$, where the state `s` is `ended`, the type `z` is either `tl`, `bc` or `afc`, and $T_i \in S$. In other words, the transaction T_k has finished executing, and the assertion of one of the effects of the rule `c` encapsulated by T_k has lead to the creation of T_i. T_k cannot commit until T_i commits. Now suppose that during its execution, T_k had acquired the lock $M'\,l_k[p]$ on object `p`. Say that during its first phase, T_i (a subtransaction of T_k), requested the lock $Ml_i[p]$ on the same object `p`. As discussed earlier, the LM (lock manager) will release T_k's lock ($M'\,l_k[p]$) and set $Ml_i[p]$ (i.e., the lock is transferred from the parent to the child).

Now if T_i releases all of its locks, including $Ml_i[p]$, when it is suspended, another transaction T_l might acquire a lock on the object `p`, violating the atomicity of the consistency forward chain. Recall that unlike transactions of other types, `cfc` transactions have both a commit and an abort dependency with their parent transactions. Thus, it is necessary that the locks held by a subtransaction not be released until the whole chain is completed. In order to overcome this problem, the TM rolls back the consistency forward chain (undoes all the write operations, which involve only status attributes in the case of inference rules), releases all the locks except for the locks that are in T_k's read and write sets (that could have been transferred to any one of the children subtransactions), and inserts T_k on T_j's suspend queue.

Thus, suspending a transaction whose type is `cfc` actually results in suspending the first ancestor of the transaction whose type is not `cfc`. In other words, a consistency forward chain is never suspended in the middle; it is always rolled back and restarted. The difference between suspending a `cfc` transaction and aborting it is that the transaction which initiated the consistency forward chain (whose type is not `cfc`) will not be aborted.

However, now that the TM has transferred all the locks to T_k, the assumption we made in the previous chapter that a conflict cannot involve a transaction when it is in an

ended state is no longer valid. Both the SCCP protocol and the PCCP protocol must be revised to take this into consideration. Concerning the SCCP protocol, it should treat an ended transaction like a pending transaction in terms of priority. If SCCP decides to abort the transaction, then it must roll back its activity, which is what SCCP does in any case. Otherwise, if SCCP decides to continue the execution of T_k, T_k is left on the suspend queue of T_j and resumed only when T_j is finished.

As far as PCCP is concerned, a transaction that is in an ended state cannot be suspended; all the other actions, however, can be applied to it. If a control rule specifies suspending a transaction whose state is ended, then PCCP treats the condition-action pair containing this action as if the condition was not satisfied (i.e., it skips over that pair). Finishing a transaction that is on a suspend queue simply means keeping it on the suspend queue (until it is resumed). Resuming a transaction that is in an ended state simply requires restarting all of its cfc transactions. This completes our discussion of suspending transactions whose states are active.

Having discussed the details of how the TM suspends a transaction whose state is active, we now proceed to discuss the case when the state of the transaction is inactive. We give a complete example involving the suspend action after discussing this case.

6.4.1.2. Suspending Transactions That are Not Active

In this section, we explain how the TM suspends the inactive transaction, if that is what is prescribed in the control rule. In other words, the suspend action is of the form suspend(T_j T_i), where the state of T_i is active and the state of T_j is inactive. The first step of the suspend operation is the same as above, i.e., releasing all of the locks in the lock set of T_j and inserting the transaction on the suspend queue of T_i. Suspending T_j requires changing its state to suspended. However, when the TM resumes the execution of T_j, it will not know whether T_j's state before being suspended was active or inactive. Resuming a suspended transaction that was active before it was suspended entails restarting the transaction, whereas resuming a transaction whose state was inactive before being suspended involves only changing its state back to inactive; the transaction cannot be restarted until the backward chaining

cycle it is waiting for has been completed. In order to distinguish between these two cases, the TM changes the state of T_j to `suspended-inactive`. Now, when T_j is resumed, its state is simply changed back to `inactive`.

The second step in suspending T_j involves finishing T_i, whose state is `active` and whose type can be any one of the five possible types. If the command encapsulated by T_i is a built-in command, then finishing T_i simply means continuing the execution of the built-in command, since a `built-in` transaction does not have any consistency implications. If the type of T_i is anything but `bi`, finishing T_i is exactly the same as described above, which is simply continuing the execution of the transaction and all of its consistency implications, but none of its automation implications.

It is possible that while T_j (whose state is `inactive`) is suspended, its subtransaction, T_n, commits. In the normal case, this would mean transferring the locks held by T_n to its parent T_j. However, since T_j is suspended (its state is `suspended-inactive`), the locks are released instead, and T_j's state is changed to `suspended` to indicate that when T_j is resumed, it should be restarted. Once T_i is either committed or aborted (or terminated, as will be discussed shortly), and its locks have been released, the TM resumes the execution of T_j by restarting it.

To illustrate the `suspend` action, consider the rules shown in figure 6-6. To remind the reader of our running example, recall that `Prog` is the program that Bob, John and Mary are working on cooperatively to complete. `Prog` contains three source code modules, `ModA`, `ModB`, and `ModC`, an `includes` directory and three libraries, corresponding to the three modules. Each of the modules contains a set of `CFILE` objects (as members of the `cfiles` structural attribute), and a `link` to its corresponding library; e.g., `ModA` is linked to the library `LibA`. Every time a `CFILE` object is compiled, its object code (i.e., the contents of the file whose pathname is stored in the `object_code` attribute of the `CFILE`) is archived in the library belonging to the module containing the `CFILE`.

Suppose that Bob requested to `archive ModA`, which causes RBDE to fire the `archive` rule (the last one). The RP (rule processor) informs the TM that it wants to fire `archive`; the TM starts a transaction T_{Bob} to encapsulate the execution of the

```
edit [?h : HFILE]:
     :
     (and (?h.reservation_status = CheckedOut)
          (?h.locker = CurrentUser))
     { edit output: ?c.contents }
     no_backward [?h.timestamp = CurrentTime];

outdate_compile [?f: CFILE]:
     (bind (?h to_all HFILE suchthat
                    (linkto [?f.hfiles ?h])))
     :
     (exists ?h):
     (?h.timestamp > ?f.object_timestamp))
     {}
     (?f.status = NotCompiled);

compile [?f:CFILE]:
     (bind (?h to_all HFILE suchthat
                    (linkto [?f.hfiles ?h])))
     :
     no_backward (?f.status = NotCompiled)
     { compile ?f.contents ?h.contents "-g"
               output: ?f.object_code ?f.error_msg }
     (and [?f.status = Compiled]
          [?f.object_timestamp = CurrentTime]);
     [?f.status = Error];

dirty[?c: CFILE]:
     :
     no_backward (?c.status = Compiled)
     { }
     [?c.status = NotArchived];

archive [?f:CFILE]:
    (bind (?m to_all MODULE suchthat (member[?m.cfiles ?f]))
          (?l to_all LIB suchthat (linkto [?m.libs ?l])))
    :
    (exists ?l):
    no_backward (?f.archive_status = NotArchived)
    { archive ?f.object_code output: ?l.afile }
    (and [?f.archive_status = Archived]
         (?f.archive_timestamp = CurrentTime));
    (?f.archive_status = Error);

archive [?m:MODULE]:
    (bind (?f to_all CFILE suchthat (member [?m.cfiles ?f]))
          (?q to_all MODULE suchthat (member [?m.modules ?q])))
    :
    (forall ?f) (forall ?q):
    (and (?f.archive_status = Archived)
         (?q.archive_status = Archived))
    { }
    (?m.archive_status = Archived);
```

Figure 6-6: Example Rules to Demonstrate the Suspend Action

archive rule. The condition of the rule specifies that a module can be considered archived only if all of its submodules and all of its C source files have been archived. In order to evaluate the condition, T_{Bob} requests R locks on ModA, main.c, f1.c and f2.c (the three CFILE objects contained in ModA; ModA does not contain any submodules and so ?q will not be bound to any object). The LM does not report any locking conflicts while acquiring the locks, so the RP proceeds to evaluate the condition of the archive rule. Say that this condition evaluates to FALSE because the CFILE f2.c has not been archived. The RP initiates a backward chaining cycle by firing the archive rule (the first one in the figure) on f2.c.

Before firing this archive rule, the RP informs the TM that it is initiating a backward chain; the TM creates the transaction $T_{Bob.1}$ as a subtransaction of T_{Bob} to encapsulate the execution of this rule. $T_{Bob.1}$ first acquires R locks on f2.c, ModA and LibA to evaluate the condition, which it finds to be satisfied, and then it acquires W locks on f2.c and LibA in order to execute the activity of the archive rule.

The states and types of the two transactions belonging to Bob are as follows: the type of T_{Bob} is t1 and its state is inactive; the type of $T_{Bob.1}$ is bc and its state is pending.

Suppose that while Bob's archive rule is being executed by Bob's client, Mary had finished editing the HFILE object i2.h (encapsulated in transaction T_{Mary}), which is linked to main.c, f1.c and f2.c. The editing of i2.h causes consistency forward chaining to fire the outdate-compile rule on main.c, f1.c and f2.c. Let us assume that the RP first fires the outdate-compile rule on f1.c. The TM creates $T_{Mary.1}$ to encapsulate the execution of this rule. In order to evaluate the property list of the rule, $T_{Mary.1}$ requests an R lock on f1.c; this lock request does not conflict with T_{Bob}'s lock Rl_{Bob}[f1.c], which T_{Bob} had acquired while evaluating the condition of the archive ModA rule. Suppose the condition of the compile rule is satisfied, so $T_{Mary.1}$ proceeds to request a W lock on f1.c, in order to execute the activity. This lock request conflicts with T_{Bob}'s R lock.

The states and types of the transactions belonging to Mary so far are as follows: T_{Mary}'s

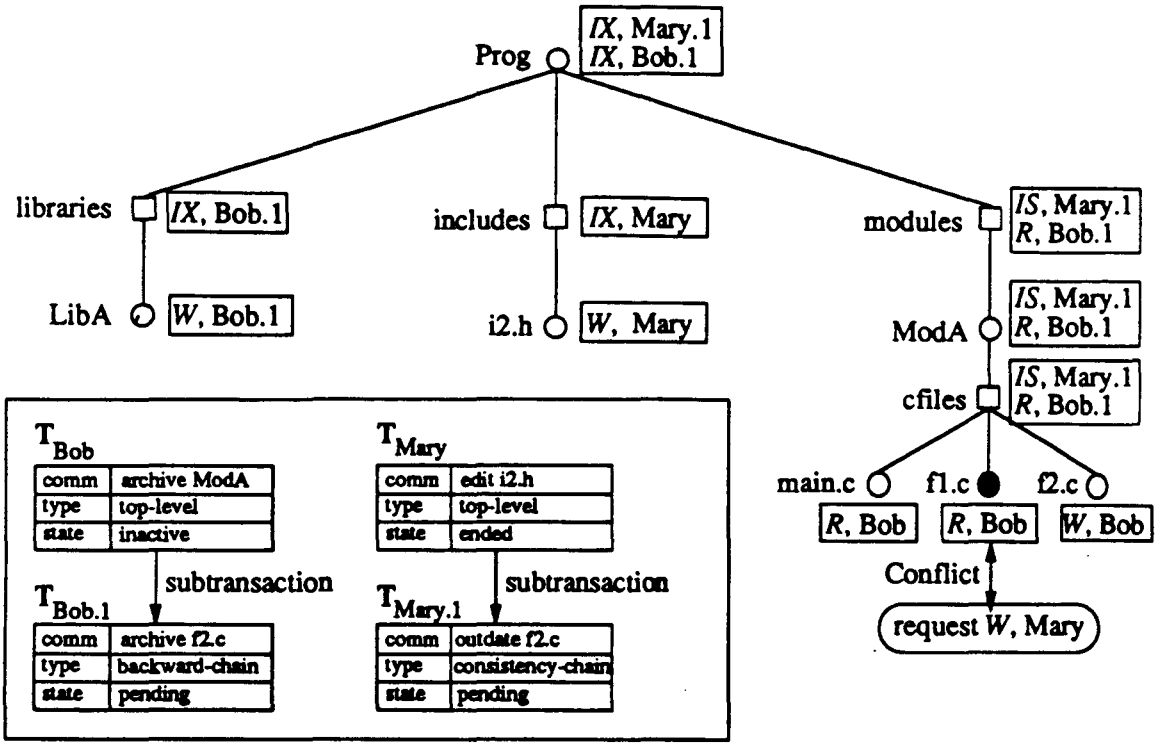

Figure 6-7: Compile-Archive Conflict Situation

type is `tl` and its state is `ended`; $T_{Mary.1}$'s type is `cfc` and its state is `active`. Figure 6-7 depicts the whole conflict situation. It shows the locks held on the objects and the states and types of the transactions.

Given this conflict, the Scheduler calls PCCP, which selects the control rule shown in figure 6-8 to resolve the conflict. The body of the control rule states that T_{Bob} (the `inactive` transaction) should be suspended until $T_{Mary.1}$ (the `active` transaction) is finished. The TM forces T_{Bob} to release all of is locks, changes its state to `suspended-inactive`, and inserts it on the suspend queue of $T_{Mary.1}$. $T_{Mary.1}$ is then finished by acquiring the `W` lock on `f1.c` and asserting the effect of the `outdate-compile` rule. Note that although the effect of `outdate-compile` has an automation implication (the `compile` rule), RBDE does not fire that rule but instead completes the execution of $T_{Mary.1}$, transferring all of its locks to its parent T_{Mary} (because the type of $T_{Mary.1}$ is `cfc`).

```
compile-archive-cr [ CFILE ]

selection_criterion:
    commands: outdate-compile, compile, archive;

bindings:
   ?t1 = requested_lock();
   ?t2 = holds_lock();

body:

   if (and (?t1.type = cfc)
           (?t1.state = active)
           (?t2.state = inactive))
   then
     {
       suspend(?t2, ?t1);
     };

   if (and (?t2.state = pending)
           (?t2.command != edit))
     {
       suspend(?t1, ?t2);
     };
;
```

Figure 6-8: Control Rule to Resolve Outdate-Archive Conflict

Recall that in addition to firing the `outdate-compile` rule on `f1.c`, Mary's original rule (`edit i2.h`) had two other consistency implications, which are to fire `outdate-compile` on both `f2.c` and `main.c`. Thus, in order to finish $T_{Mary.1}$, the TM must carry out the whole consistency forward chain, including all the consistency implications of the transaction that initiated $T_{Mary.1}$. Suppose that the RP fires `outdate-compile f2.c`; the TM starts a transaction $T_{Mary.2}$ (whose type is `cfc`) to encapsulate the execution of this rule. $T_{Mary.2}$ requests an `R` lock on `f2.c` in order to evaluate the condition of `outdate-compile f2.c`. This lock request will cause an interference between $T_{Mary.2}$ with $T_{Bob.1}$, which holds the lock $Wl_{Bob.1}$[f2.c].

The same control rule shown in figure 6-8 applies to this new conflict situation. This time, however, the condition of the second condition-action pair will be satisfied. This will cause PCCP to suspend $T_{Mary.2}$ until $T_{Bob.1}$ is finished. Since $T_{Mary.2}$'s type is `cfc`, the TM must roll back the whole consistency forward chain (which in this case does not include any other transaction since $T_{Mary.1}$ has already ended and transferred its

locks to T_{Mary}). Thus, the TM transfers the locks acquired by $T_{Mary.2}$ to T_{Mary} and then inserts T_{Mary} on $T_{Bob.1}$'s suspend queue (T_{Mary} does not release any of its locks, so it currently holds Wl_{Mary}[f1.c], Wl_{Mary}[i2.h], and R locks on i2.h and i3.h).

The TM finishes $T_{Bob.1}$ by waiting for the archive activity to complete, and carrying out all of its consistency implications. Note that the only consistency implication of archive f2.c is archive ModA (an inference rule), which is the rule encapsulated already by T_{Bob} (the parent transaction of $T_{Bob.1}$). However, since the state of T_{Bob} is suspended-inactive, the TM commits $T_{Bob.1}$, releasing all of its locks, and changes the state of T_{Bob} from suspended-inactive to suspended. This change of state indicates that when T_{Bob} is resumed, its execution can be restarted since the backward chaining cycle it was waiting for (archive f2.c) has completed.

After committing $T_{Bob.1}$, the TM resumes the execution of the suspended transaction T_{Mary}. Since the state of T_{Mary} is ended, the TM resumes its execution by carrying out its consistency implications. This leads to executing $T_{Mary.1}$ followed by $T_{Mary.2}$, which in this case will not interfere with any other transactions because no other transaction holds any locks on either f1.c or f2.c. Finally, the third consistency implication of edit i2.h, which is outdate-compile main.c, is carried out (encapsulated in $T_{Mary.3}$). After $T_{Mary.1}$, $T_{Mary.2}$ and $T_{Mary.3}$ commit, T_{Mary} is committed, releasing all of its locks.

Finally, the TM resumes T_{Bob} by restarting it. T_{Bob} re-acquires all the necessary locks, which T_{Bob} was previously forced to release when it was suspended. No conflicts will occur this time because T_{Mary} has already released all of its locks. The TM then commits T_{Bob}.

From this example, it should be clear that by suspending transactions, PCCP was able to order the execution of transactions in such a way so that it avoided aborting any of them. Mary was able to edit i2.h and take care of all the consistency implications of this activity (i.e., outdating f1.c and f2.c), and Bob was able to successfully execute the archive command. The SCCP protocol, in contrast, would have aborted both T_{Bob} and $T_{Bob.1}$, causing RBDE to reject Bob's archive ModA rule.

We have now completed the details of the `suspend` action. We proceed to discuss the `terminate` action.

6.4.2. Terminating a Transaction

The `suspend` action is very useful when it is known that the activity of the transaction that must be finished will not take a long time. In the example above, for instance, the administrator prescribed suspending $T_{Mary.2}$ when it conflicted with $T_{Bob.1}$ because the activity of $T_{Bob.1}$ (i.e., archiving a `CFILE`) takes a relatively short time. Assume, however, that the activity that $T_{Bob.1}$ was waiting to complete was an invocation of an editor. Clearly, in this case, the administrator should not suspend a transaction that will be resumed only when the editing session is over.

Instead, the administrator can prescribe terminating $T_{Mary.2}$, which, unlike `suspend`, can be applied to a transaction in any state and results in either aborting the transaction or finishing it, depending on its type and state. Terminating a transaction whose state is `inactive` is exactly the same as finishing the transaction, which basically stops the backward chaining cycle, as described in the previous section. Terminating a transaction whose state is `active` and whose type is anything but `cfc`, is exactly the same as aborting the transaction since the transaction would not have performed any write operations yet, and thus aborting it does not undo any activities. Thus, the administrator would want to prescribe the `terminate` action only in the case of a transaction whose type is `cfc` or a transaction whose state is `pending`.

In both of these cases, terminating a transaction is intended as an alternative action to `abort` in order not to waste any effort that a human developer might have invested in a development activity. In the case of `cfc` transactions, `terminate` avoids cascaded aborts. A transaction whose type is `cfc` is part of a consistency forward chain that is supposed to be atomic. Therefore, aborting a `cfc` transaction requires that all the parent transactions of $T_{i,cfc}$, up to and including the first parent transaction whose type is not `cfc`, must be aborted. Aborting a transaction whose state is `pending` entails rolling back the activity that is being executed. Aborting transactions in both cases can be very expensive, and sometimes is not really necessary.

In contrast, terminating a transaction whose type is `cfc` (which implies that its state is `active` when a conflict occurs) or whose state is `pending` (which implies that it encapsulates the execution of an activation rule) is the same as finishing the transaction except that instead of asserting one of the effects of the rule and carrying out its consistency implications, the objects' attributes whose values would have been changed either by the assignment predicates in the effect or by the effects in the consistency forward chain are *marked* as being inconsistent. Note that a consistency forward chain can be simulated because it is made up entirely of inference rules, each of which has only one effect.

Marking an attribute involves storing a pointer to the inference rule that should have been fired as part of the consistency forward chain. An object with a marked attribute is in an inconsistent state. Consistency of marked attributes can be re-established through an unmarking routine, as will be explained shortly. Thus, the `terminate` action provides a mechanism to tolerate inconsistency of objects temporarily. The fact that objects can be temporarily inconsistent has implications on the predicate evaluation algorithms we presented in chapter 2.

6.4.2.1. Revised Predicate Evaluation Algorithms

In the algorithms given in section 2.4.1.2 for evaluating a single predicate, we assumed that the truth value returned by the predicate is accurate. However, as we explained above, conflict situations might render some attributes of objects to be in an inconsistent state, causing the values of predicates over these objects to be inaccurate. In this case, the RBDE has to verify the value of any predicate over these objects before it can evaluate the condition formula in which this predicate occurs. More precisely, before using the value of a marked attribute in an evaluation, the RBDE must first *unmark* that attribute. Unmarking an attribute requires either verifying that the value currently stored in that attribute is what it should be (i.e., that the value is equal to the value stored by the TM when it terminated a `cfc` transaction involving the object), or firing the inference rule pointed to by the attribute in order to make the value of the attribute what it should be.

Only if unmarking succeeds can the evaluation proceed. The criteria for the success and

failure of a consistency backward chain will be explained shortly, but first we show how the previous algorithms should be revised.

```
Input: A predicate, P, of the form (?v1.att1 <op> ?v2.att2)
Output: True or False.

  /* Both ?v1 and ?v2 are universally quantified */

  For each object, obj1, bound to ?v1 Do
    Begin
      If obj1.att1 is "marked" Then
           Begin
              status := UNMARK (obj1, att1);
              If status = UNSUCCESSFUL Then
                  return UNSATISFIABLE;
           End;
       For each object, obj2, bound to ?v2 Do
          Begin
            If obj2.att2 is "marked" Then
                Begin
                  status := UNMARK (obj2, att2);
                  If status = UNSUCCESSFUL Then
                      return UNSATISFIABLE;
                End;

            instantiate P (?v1 := obj1 and ?v2 := obj2);
            evaluate the instantiation of P;
            If evaluation returns FALSE Then
              Begin
                P := FALSE;
                return FALSE;
              End;
          End;
     End;

  P := TRUE;
  return TRUE;
```

Figure 6-9: Evaluating a Predicate With Marking

The revised algorithm for evaluating a single predicate with two universally quantified variables is shown in figure 6-9. The modifications are shown in **bold face**. Basically, before accessing a marked attribute, the attribute must be first unmarked. The same modifications must be inserted in the other cases discussed in section 2.4.1.2.

The routine for unmarking is shown in figure 6-10. This routine is recursive. It calls EXECUTE_RULE, which in turn might call UNMARK on another attribute. The recursion is initiated by the fact that one or more of the attributes of an object that is involved in the evaluation of a predicate P are marked. The actual unmarking of attributes is carried out by the LM.

```
routine UNMARK (O: object; att: marked attribute of O);

  Begin
    rule := get the rule pointed to by att;
    ret_value := EXECUTE_RULE (rule);

    If ret_value <> UNSATISFIED or UNSATISFIABLE Then
       Begin
         unmark att;
         If all attributes of O are unmarked then
            unmark O;
         return (SUCCESSFUL);
       End
     Else
       return (UNSUCCESSFUL);
  End;
```

Figure 6-10: Unmarking Objects

To illustrate the `terminate` action, consider the `edit` rule in figure 6-6 on page 192, which applies to `HFILE` objects. The rule has one consistency predicate in its effect, setting the value of the `timestamp` attribute of the `HFILE` to the current system time. This consistency predicate caused a consistency forward chain to the `outdate-compile` rule, which applies to `CFILE` objects; whenever an `HFILE` is edited, the object code of all the `CFILE`s that are linked to the `HFILE` must be outdated.

Suppose that Mary edited an `HFILE`, `i2.h`, which is linked to three `CFILE` objects: `main.c`, `f1.c` and `f2.c` (as shown in figure 2-3 on page 30). After completing the execution of the `edit` rule, the RP attempts to outdate the compilation of the three `CFILE` objects, as in the example of the previous section. However, say that a conflict is detected halfway after carrying out `outdate-compile f1.c`, when attempting to fire `outdate-compile` on `f2.c`, because Bob is editing (rather than archiving, as before) `f3.c`. SCCP would have resolved the conflict by aborting Bob's edit. Aborting either transaction is not necessary, however, because the conflict should be ignored. The reason is that the `outdate-compile` rule leads to exactly the same effect as the `edit` rule on a `CFILE`, which is to set the `status` attribute to "NotCompiled".

There is one complication, however, and that is if the `edit` on `f2.c` was aborted for some other reason (e.g., another conflict). The result would be that the `status` at-

```
edit-outdate-cr [ FILE ]

selection_criterion:
    commands: outdate-compile, edit;

bindings:
   ?t1 = requested_lock();
   ?t2 = holds_lock();

body:

   if (and (?t1.type = cfc)
           (?t2.state = pending))
   then
     {
       terminate (?t1);
     };
;
```

Figure 6-11: Control Rule to Resolve Edit-Outdate Conflict

tribute of `f2.c` would not be assigned the value "NotCompiled". This would give the false implication that `f2.c`'s object code is up to date, which it is not because `i2.h` was edited. To avoid this possibility, instead of ignoring the conflict between `outdate-compile f2.c` and `edit f2.c` (we did not provide any mechanisms for doing that so far), the administrator can prescribe the termination of the transaction encapsulating the execution of the `outdate-compile` rule, as shown in the control rule in figure 6-11. The Scheduler carries out the prescribed actions by terminating the execution of `outdate-compile f2.c` and marking `f2.c`. Marking `f2.c` involves recording that the object `f2.c` will be in a consistent state only if its `status` attribute is assigned the value "NotCompiled". In addition, the RP will store a pointer to the rule `outdate-compile`. To re-establish the consistency of `f2.c`, all the RP needs to do is fire the rule `outdate-compile` on `f2.c`.

Terminating the `cfc` transaction encapsulating `outdate-compile f2.c` does not involve only marking `f2.c`, but also simulating any consistency forward chaining that the `outdate-compile` rule would have caused, and marking all the objects that would have been changed by this consistency forward chain. In our example, the `outdate-compile` rule initiates a consistency forward chain to outdate the `LIB` object that includes the object code of the `CFILE`. Then, in addition to marking `f2.c`, the

TM must ask the LM to mark `LibA` and indicate that its value should be "NotArchived".

The TM then continues the rest of the consistency forward chain resulting from the `edit i2.h` command (i.e., firing `outdate-compile` on `main.c`) as if the conflict did not occur.

6.5. Summary

In this chapter, we presented a mechanism for overriding the SCCP default concurrency control protocol presented in chapter 5. The main reason for introducing this mechanism is to be able to tailor the concurrency control policy to the needs of a specific project.

The mechanism is composed of a language, CRL, in which the administrator writes control rules. Each control rule applies to a specific conflict situation (interference), and prescribes actions that the TM should take in order to resolve the conflict situation. CRL provides constructs for describing a spectrum of conflict situations, ranging from very general conflicts (e.g., any conflict that occurs over an instance of class `FILE`) to very specific conflicts (e.g., involving two particular users and two specific commands). CRL also provides constructs for using most of the attributes (semantic information) of the two conflicting transactions: their subtransactions, the commands whose execution they encapsulate, their owners, timestamps, lock sets, types and states. By using all of this information, the administrator can prescribe very specialized conflict resolution strategies.

The control rules provide semantic information that can be used to allow relaxation of the default policy. If a conflict occurs and the conflict situation matches the situation described in a control rule, then the actions prescribed by the control rule are carried out instead of the default actions. CRL supports three actions in addition to `abort`: `suspend`, `terminate` and `notify`.

`Suspend(`T_i`, `T_j`)` involves suspending the execution of T_i until after the completion of T_j, at which point the execution of T_i is resumed. Finishing the execution of T_j requires completing the execution of the command encapsulated in the transaction, if the

execution of the command's activity has already started, and all the consistency implications of the command. Unlike normal execution, however, it does not initiate any automation forward chaining transactions (i.e., the automation implications of the command are not carried out). Suspending a transaction allows the execution of all of the access units of another transaction to be done before the execution of the suspended transaction continues.

The `terminate` action, unlike `abort`, avoids cascaded aborts in the case of consistency forward chains. In the case of transactions of type `cfc`, aborting the transaction requires that all of the parent transactions of $T_{i,cfc}$, up to and including the first parent transaction whose type is not `cfc`, must be aborted. This can be very expensive, and sometimes not really necessary. Instead, PCCP might fire a control rule that prescribes terminating the consistency forward chain prematurely, rather than aborting and rolling it back. This enables RBDE to tolerate inconsistency temporarily (until the inconsistent objects are accessed again). In this case, the RBDE must simulate the rest of the consistency forward chain, but instead of changing the values of the objects' attributes, it only *marks* the objects' attributes involved in the rest of the chain, and stores the value that the consistency forward chain would have assigned to these attributes. Consistency is re-established by an unmarking routine that is called whenever an inconsistent object is accessed.

Finally, the `notify(?t, ''message'')` action is used to send a message to the user whose command initiated the transaction bound to `?t1`.

Chapter 7

User Sessions and Development Domains: Supporting Teamwork

The CRL language and the PCCP protocol presented in the previous chapter have two significant limitations: (1) although the `terminate` action allows for the temporary tolerance of inconsistency of an object, there is no way to assign the responsibility of re-establishing consistency to one of the users whose concurrent actions caused the object to be inconsistent; and (2) CRL does not provide constructs for writing team-specific control rules (i.e., control rules that resolve conflicts among members of a specific development team). In this chapter, we add three constructs: ***user sessions***, ***obligations*** and ***development domains***, which form the basis for overcoming these two limitations. We integrate the three constructs into CRL, with the result of being able to tailor the concurrency control policy to different development teams.

We first present user sessions and obligations, and show how they are used to enhance the process modeling abilities of EMSL and the rule execution model in RBDE. From the point of view of the database and the TM, user sessions are similar to sagas; we describe the concept of sagas and explain the similarities between sagas and user sessions. We then integrate sessions and obligations into CRL to remove the first limitation mentioned above. Next, we introduce the notion of development domains. This notion is based on concepts from group-oriented transaction mechanisms, which we briefly overview. Finally, we integrate development domains into CRL and show how the project administrator can use the information about domains to write team-specific control rules.

7.1. User Sessions and Obligations

In the previous chapters, we have assumed that the commands requested by the same user are independent of each other. A user requests one command, and after the command and all of its implications have been executed by RBDE (assuming no conflicts), the user requests another command, and so on. In practice, however, a software developer performs several commands in order to achieve a single development task. For example, fixing a bug often requires an iterative process of editing the source file, compiling it, testing the executable, etc. After fixing a bug, the developer might proceed to modify the documentation to reflect the bug fix. It is desirable to group together in a single unit all the commands that the developer requests to achieve such a development task as fixing a specific bug. The provision of such a unit enables developers (and RBDE) to reason about development tasks and not just individual commands.

A unit representing a development task is also advantageous from the point of view of concurrency control. Although CRL provides the `terminate` action, which can be used to tolerate inconsistency, CRL does not provide any mechanism for assigning the task of restoring consistency to specific developers. The main reason for not being able to support such a mechanism is the lack of a higher-level unit (on top of user commands) that can act as a context for restoring consistency.

To illustrate, consider again the example we introduced in chapter 6 to demonstrate the `terminate` action. In that example, Mary edited an `HFILE` object, `i2.h`, which is linked to three `CFILE` objects: `main.c`, `f1.c` and `f2.c`. After completing the execution of the `edit i2.h` rule, the RP (rule processor) attempts to outdate the compilation of the three `CFILE` objects. However, a conflict is detected after carrying out `outdate-compile f1.c`, when attempting to fire `outdate-compile` on `f2.c`, because Bob is editing `f3.c`. The control rule that PCCP fires to resolve this conflict (shown in figure 6-11 on page 201) prescribes terminating the transaction encapsulating Mary's `outdate-compile f2.c` rule. This leads to leaving `f2.c` in an inconsistent state temporarily.

PCCP tolerates inconsistency by marking the inconsistent objects, `f2.c` in the example,

and indicating what the correct values of their attributes should be. Before any other command can access the object, the object must be unmarked to re-establish its consistency. If no other command accesses `f2.c`, however, it remains inconsistent. It would make more sense if RBDE somehow made sure that either Bob or Mary, both of whose actions caused `f2.c` to be inconsistent, re-establishes the consistency of `f2.c` before they complete their current development tasks. The consistency of `f2.c` can be re-established by either outdating the object code of `f2.c`, changing the value of its `status` attribute to be equal to "NotCompiled", or re-compiling `f2.c`.

7.1.1. Adding User Sessions to RBDE

We extend RBDE with *user sessions* (or sessions, for short), whose purpose is to provide a unit on top of rule chains that can serve both as a context for re-establishing consistency of objects that are left in an inconsistent state because of concurrency conflicts, and as a unit for grouping together user commands that achieve a particular development task assigned to an individual developer. We concentrate on the role of user sessions with respect to concurrency control; the other purpose of sessions is outside the scope of this dissertation and thus is only briefly addressed. The issue should be explored further, as we discuss in chapter 8.

We add two built-in commands to RBDE: `begin_session` and `end_session`. A developer starts a new session by issuing the command `begin_session`, and ends a session by issuing the command `end_session`. The commands that are requested between these two built-in commands by the same developer in the same client comprise the body of the session. Note that sessions are persistent across logouts and failures. Note also that a single developer might have multiple simultaneous sessions in different client processes. RBDE maintains information about the session to which each user command belongs. Also, a session is not associated with a single process since the owner of the session might log out of the system and continue his session in another process. Commands that are requested outside a session are considered singleton sessions. We do not consider nested sessions in this dissertation.

7.2. Obligations: Enhancing the Rule Execution Model

Given a session that groups a set of commands, RBDE can now reason about the set of commands as a unit with respect to automated assistance. In the rule execution model presented in chapter 2, one of the effects of a rule is asserted immediately after the rule's activity, if it is not empty, is executed. Each assignment predicate in the effects of a rule changes the value of one of the status attributes of an object to either reflect the changes that the rule's activity has introduced (in the case of an activation rule) or to propagate changes to the values of other attributes (in the case of inference rules). The conditions of rules are sufficient for modeling what users can and cannot do in terms of development activities; rule effects provide a construct for defining the immediate effects of tools and development activities. Neither construct, however, can be used to express the responsibilities of developers in terms of completing their work and avoiding obstructing other developers' work.

For example, the reserve rule (figure 2-13, page 53) states that a developer can reserve a `FILE` object only if the value of the `reservation_status` attribute of the object is "Available". The rule further prescribes that once the reserve activity is completed successfully (by invoking the `rcs` tool), the values of the `reservation_status` and `locker` attributes of the object should be changed to reflect that. The rule does not define the responsibilities of the developer who reserved the object. It would enhance the expressive power of EMSL if the rule could specify that reserving an object obliges the reserver to deposit the object in the same session either immediately after he is done with it or before ending the session. This is exactly the purpose of *obligations*.

An obligation is exactly of the same form as a predicate in the property list of a rule. However, instead of being part of the condition, obligations are added to the effects of the rule. Each obligation specifies what the value of an attribute of an object (which is bound to a variable in the binding part of the rule) should be before ending the session in which the rule was fired. Figure 7-1 shows a revised EMSL rule template that includes obligations. After each effect, the rule can prescribe a list of obligations that correspond to the assignment predicates in the effect.

```
     # Rule name and parameter list
1.   rulename [?param1 : CLASS1; ?param2 : CLASS2; ... ]:

     # Binding part of condition.
2.   (bind (?var1 to_all CLASS3 suchthat characteristic clause)
3.         (?var2 to_all CLASS4 suchthat characteristic clause)
           ( ... ))
4.   :

     # Property list part of condition.
5.   (property list)

     # Activity
6.   { <Envelope> argument1 ... argumenti;
                  output: argumenti+1 ... argumentn  }

     # Effects
7.   (effect1 obligations1);
8.   (effect2 obligations2);
     (...);
```

Figure 7-1: A Template for EMSL Rules With Obligations

Since the only way for a user to change the value of an object's attribute is to request a command that corresponds to a rule, each obligation must be implied (in the sense of the definition we gave in chapter 2) by at least one assignment predicate in the effects of one or more of the rules in the project rule set. The Loader builds an Obligation Table in which it inserts all the obligations of the rules in the rule set. While loading the project rule set, the Loader verifies that each obligation in the Obligation Table is implied by at least one assignment predicate in the Predicate Table. If an assignment predicate, `P`, in the effects of a rule, `r1`, implies an obligation, `O`, in the effects of another rule, `r2`, then a backward chain is established from `O` to `P`. The backward chain means that if `O` is not satisfied, the RP should fire `r1` to try to make it satisfied. If an obligation is not implied by any assignment predicate, the Loader will refuse to load the rule set. The project administrator must either remove the obligation or make sure that one of the rules in the project rule set can satisfy it.

RBDE maintains a list of obligations for each session. This list is empty when the session is first started. Whenever a rule is fired in the context of a session, the RP adds the obligations in the asserted effect of the rule to the list of obligations of the session. In

addition, the RP checks if asserting any of the assignment predicates in the effect of the rule satisfied any of the obligations already on the lists of current sessions (i.e., that have not ended yet). If it did, the satisfied obligations are removed from the sessions' lists of obligations. Note that the same obligation can be inserted at a later time on the list of obligations, in which case it must again be satisfied before the session can be ended. An empty list of obligations indicates that the user can end the session. Note also that one user's command might satisfy the obligation in another user's session.

If a user attempts to end a session and the list of obligations for the session is not empty, RBDE will attempt to satisfy each of the obligations on the list automatically by initiating backward chaining. If the RP cannot make an obligation satisfied, RBDE will inform the user that he cannot end the session at this time because an obligation is not satisfied. The user should then attempt to take actions (such as executing built-in commands, firing rules, or re-attempting to end the session at a later time) that will make the condition satisfied. Note that like predicates in the property list of a rule, obligations can initiate a backward chaining cycle. The reason why backward chaining sometimes cannot satisfy an obligation is that the condition of the rule that can satisfy the obligation might be UNSATISFIABLE because of a consistency (or a no_backward) predicate or because of a locking conflict; the user, however, can go ahead and request commands that can make the condition satisfied or delay ending the session until after the locks that caused the conflict during the earlier attempt to end the session have been released.

Figure 7-2 shows a revised version of the `reserve` rule in which an obligation has been added. The obligation states that it is the responsibility of the developer whose command invoked the `reserve` rule to make sure that the `reservation_status` attribute of the reserved object is set to "Available". This obligation must be satisfied before the user can end the session in which the `reserve` rule was fired. The assignment predicate in the effect of the `deposit` rule satisfies this obligation of the `reserve` rule, and thus a backward chain is established between the obligation and the assignment predicate. Note that this backward chain overrides the `no_chain` prefix that precedes the assignment predicate in the effect of the `deposit` rule but only for the purpose of satisfying the obligation and not in general; further backward chaining (i.e., if

```
deposit [?f : FILE]:
     :
     (and [?f.reservation_status = CheckedOut]
          [?f.locker = CurrentUser])

     { deposit output: ?f.contents ?f.version }

     no_chain (?f.reservation_status = Available);


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

     { reserve output: ?f.contents ?f.version }

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

Figure 7-2: Reserve Rule Showing Obligation

`deposit` initiated further backward chaining because its condition is not satisfied) is performed normally, as discussed before.

If none of the user's commands within the session cause the `deposit` rule to be fired, RBDE will fire the rule automatically when the user issues the `end_session` command. If RBDE does not succeed in making the obligation satisfied (e.g., because the condition of the `deposit` rule is not satisfied or because of a locking conflict), it aborts the `end_session` operation and informs the user that he cannot end his session before satisfying all of his obligations.

7.2.1. Related Work: Other Notions of Obligations

Our notion of obligations is based on two previous work done by other researchers, which we briefly describe in this subsection.

7.2.1.1. Obligations in Inscape

The Inscape environment [Perry 89b] is an SDE that provides assistance in the development and evolution of large-scale software projects. Inscape uses a formal module interface specification to provide the user with information about how a particular object in the project is meant to be used and how the object is actually being used by developers.

The module interface specification language is based on Hoare's input/output predicates [Hoare 69]. Each component of the project in Inscape has preconditions, which resemble input predicates in Hoare's model, and postconditions, which resemble Hoare's output predicates. The predicates, however, are used not for validation purposes but for program construction. More specifically, the preconditions of an operation on a component are assumptions that must be satisfied before the operation can be performed; the postconditions are the results produced by the operations.

Inscape extends Hoare's model by adding obligations to operations [Perry 87]. Obligations in Inscape, like our obligations, are conditions that must eventually be satisfied. Obligations are incurred as side effects to operations. Unlike postconditions, obligations are not guaranteed to be true immediately after the performance of an operation; Inscape only checks whether they are guaranteed to be satisfied eventually. Operations in Inscape are defined in units (components) that can be encompassed within other units; an interface is defined between a unit and its encompassing unit. Obligations defined for an operation in a unit can either be satisfied by the postconditions of other operations in the same unit or propagated to the encompassing interface. There is no single unit, like sessions in our model, that provides a scope for satisfying obligations. Instead, obligations can be propagated from one unit to another until they are eventually satisfied.

Our obligations serve the same purpose as Perry's obligations; the two constructs are also similar in that an obligation is of the same form as a condition.

7.2.1.2. Extending Permissions with Obligations

Minsky and Lockman presented obligations [Minsky and Lockman 85] that serve a purpose slightly different from ours and Perry's. Minsky and Lockman's obligations, which we will call ML obligations hereafter, are intended as an extension to authorization mechanisms. Most authorization mechanisms specify what actors (an abstraction for active components) can do (i.e., permissions), but they do not specify the obligations that these actors incur as a result of executing a permitted action. ML obligations provide authorization mechanisms with the capability to attach obligations to permissions.

One difference between our notion of obligations and ML obligations is that of the re-

quirement as to when the obligations must be satisfied. In our notion, obligations must be satisfied before the end of the session in which they were incurred; there is not explicit timing requirement. Similarly, Perry's notion simply states that obligations must be satisfied eventually. Minsky and Lockman, in contrast, add a time dimension to obligations. Each obligation is attached with a timing requirement that specifies the *deadline* for satisfying the obligation. This deadline can be either a specific time or relative to (i.e., before or after) the occurrence of a specific event (e.g., before the termination of a session).

Another difference between our obligations and ML obligations is in the specification of what to do if an obligation is not satisfied. In our case, it is implicit; the only thing RBDE does is prevent the user from ending the session in which the unsatisfied obligation was incurred. An ML obligation, on the other hand, explicitly prescribes a *sanction*, which must be carried out if the obligation is not satisfied by the specified deadline. The sanction is typically an operation that "gets around" the fact that the obligation was not satisfied on time.

ML obligations are more powerful than our obligations and provide better control over when the obligations must be satisfied. It was not clear to us, however, how the timing dimension of ML obligations can be implemented without incurring a significant overhead from the enforcement of the time requirements; in their paper, Minsky and Lockman do not suggest an implementation strategy. But since one of the requirements of this thesis work was practicality in the implementation sense, we preferred to implement a simpler notion of obligations that still provides a significant extension to EMSL and RBDE.

7.2.2. Using Sessions and Obligations in Control Rules

In order to be able to use the notions of sessions and obligations in control rules, we must extend the TM to handle sessions. A session is treated like a transaction in the sense that the TM creates an entry for each session in the transaction table, and maintains several attributes for each session, similar to the transactions' attributes. The TM maintains information about which transactions belong to which sessions. A transaction,

T_i, belongs to a session, S, if T_i is created to encapsulate the execution of a command that was requested within S.

Unlike a transaction, however, a session is a non-atomic unit that is nonetheless a logical unit of database operations. As such, sessions are similar to sagas [Garcia-Molina and Salem 87], which are long transactions that can be broken up into a collection of sub-transactions; the subtransactions of a saga can be interleaved in any way with other transactions.

7.2.2.1. Related Work: Sagas

A saga is not just a collection of unrelated transactions because it guarantees that all its subtransactions will be completed or they will be *compensated* (explained shortly). A saga thus satisfies the definition of a transaction as a logical unit; a saga is similar to Moss's nested transactions and Lynch's multilevel transactions in that respect. Sagas are different from nested transactions, however, in that, in addition to there being only two levels of nesting, they are not atomicity units since sagas or other transactions may view the partial results of other sagas at the granularity of steps within the sagas (each step is atomic and its partial results cannot be viewed by other sagas or transactions). By structuring long transactions in this way, non-serializable schedules that allow more concurrency can be produced. Mechanisms based on nested transactions as presented earlier produce only serializable schedules.

In traditional concurrency control, when a transaction is aborted for some reason, all the changes that its write operations introduced are undone. This procedure is called rollback, as we explained before. Aborting a transaction, T_i, requires rolling back the objects that were written by T_i to the state in which they were before T_i started. If the scheduling policy allowed other transactions to access the same objects that T_i has changed, cascaded aborts are required to restore the database to its state before T_i started. Cascaded aborts would be required very often in the case of sagas since, by definition, sagas allow other transactions to view their partial results. In order to avoid costly cascaded aborts, user-supplied *compensation functions* are executed to compensate for each transaction that was committed at the time of failure or automatic abort.

A compensation function undoes the actions performed by a transaction from a semantic point of view. For example, if a transaction reserves a seat on a flight, its compensation function would cancel the reservation. We cannot say, however, that the database was returned to the state that existed before the transaction started, because in the meantime, another transaction could have reserved another seat and thus the number of seats that are reserved would not be the same as it was before the transaction.

Although sagas were introduced to solve the problem of long transactions in traditional applications, their basic idea of relaxing serializability is applicable to design environments. For example, a long transaction to fix a bug in a design environment can be naturally modeled as a saga that consists of subtransactions to edit a file, compile source code, and run the debugger. These subtransactions can usually be interleaved with subtransactions of other long transactions. Using compensation functions instead of cascaded aborts is also suitable for advanced applications. For example, if one decided to abort the modifications introduced to a file, one can revert to an older version of the file and delete the updated version.

Our notion of user sessions is similar to sagas in several respects. First, a user session, like a saga, is non-atomic. Instead, every step in the session (i.e., a user command and all the chaining resulting from it) is atomic. The steps of one user session can be interleaved with the steps of other sessions. Second, sessions, like sagas, are user-controlled. The user must specifically start and end a session. Third, once a step in a session is committed, it cannot be rolled back. It can only be compensated by one or more steps.

7.2.2.2. Extending CRL to Handle Sessions and Obligations

Given that user commands are grouped in sessions, a control rule can prescribe obligations that guarantee the resolution of a conflict to one or both of the sessions involved in the conflict. The control rule in figure 7-3 applies to conflict situations over instances of the class `FILE` that involve the two commands `outdate-compile` and `edit`. The interfering transaction is bound to variable `?t1` and the other transaction involved in the conflict is bound to `?t2`; the session containing the interfering transaction is bound to `?s1`. We have added the function `session(?t1)`, which returns the identifier of the session containing the transaction bound to `?t1`. The control rule prescribes two ac-

```
edit-outdate-cr [ FILE ]

selection_criterion:
    commands: outdate-compile, edit;

bindings:
   ?t1 = requested_lock();    # the interfering transaction
   ?t2 = holds_lock();        # the other conflicting trans.
   ?s1 = session(?t1);        # ?t1's session.

body:

   if (and (?t1.type = cfc)
           (?t2.state = pending))
   then
   {
     terminate (?t1);
     add_obligation(?s1,(conflict_object.status=NotCompiled));
   };
;
```

Figure 7-3: Control Rule to Resolve Edit-Outdate Conflict with Obligation

tions: terminating the transaction bound to the variable `?t1`, and adding an obligation to the session containing the transaction bound to `?t1`. We have added the action `add_obligation`, which is handled by the TM. Given this action, the TM will instruct the RP to add the specified obligation to the session whose identifier is bound to `?s1`. Thus, this control rule not only tolerates inconsistency, but also adds an obligation to the session of one of the users involved in the conflict to re-establish consistency. Note that the user to whose session the obligation was added can satisfy the obligation by firing the `outdate-compile` rule. Alternatively, if the same user or any other developer requested the `edit` command on the inconsistent object, the obligation will be satisfied by the effect of the `edit` rule.

This completes our discussion of user sessions and obligations. We now present the concept of development domains and show how it can be used to write project-specific control rules.

7.3. Development Domains: Modeling Teamwork

The task of completing the development of a large-scale software project is typically broken down into several subtasks. Typically, a development team is assigned the job of completing a subtask. The concept of modularity dictates that the interaction between the subtasks, and thus the development teams, be minimized. Members of a development team that is responsible for completing a particular development task share resources and expertise. We model the shared resources that the task requires through the concept of *development domains*, or just domains. In order to model cooperation among members of a development team, the sessions of each member in the team must belong to the domain modeling the resources of the development task. In fact, each user session must belong to a development domain, which gives the session a context for its operation. By default, a user session belongs to an empty domain that does not include any resources.

Domains provide a grouping mechanism for both sessions and objects involved in a development task achievable by a team of developers. Domains are thus akin to group-oriented transaction models, which we overview next.

7.3.1. Related Work: Group-Oriented Transaction Models

Researchers in the area of advanced database applications have studied the mechanisms needed to support cooperation and teamwork. Several of the mechanisms proposed are based on the concept of a *group* of cooperating transactions. The members of a group usually work on the same task (or at least related tasks), and thus need to cooperate among themselves much more than with members of other groups. We overview some of the mechanisms that have been proposed and explain the differences between them and our notion of domains.

7.3.1.1. The Group Paradigm

Since developers of a large project often work in small teams, there is a need to formally define the kinds of interactions that can happen among members of the same team as opposed to interactions between teams. El-Abbadi and Toueg defined the concept of a *group* as a set of transactions that when executed transforms the database from one consistent state to another [El Abbadi and Toueg 89]. They presented the group paradigm to deal with consistency of replicated data in an unreliable distributed system. They hierarchically divide the problem of achieving serializability into two simpler ones: (1) a local policy that ensures a total ordering of all transactions within a group; and (2) a global policy that ensures correct serialization of all groups.

A group, like a nested transaction, is an aggregation of a set of transactions. There are significant differences, however, between groups and nested transactions. A nested transaction is designed *a priori* in a structured manner as a single entity that may invoke subtransactions, which may themselves invoke other subtransactions. Groups do not have any *a priori* assigned structure and no predetermined ordering imposed on the execution of concurrent transactions within a group. Another difference is that the same concurrency control policy is used to ensure synchronization among nested transactions at the root level and within each nested transaction. Groups, however, could use different local and global policies (an optimistic local policy, for example, and a 2PL global policy). The existence of more than one policy might lead to increased concurrency.

The group paradigm was introduced to model inter-site consistency in a distributed database system. It can be used, however, to model teams of developers, where each team is modeled as a group with a local concurrency control policy that supports cooperation. A global policy could then be implemented to coordinate the efforts of the various groups. Of course, the local policies and the global policy must be compatible in the sense that they do not contradict each other's correctness criteria. El-Abbadi and Toueg do not sketch the compatibility requirements between global and local policies.

Dowson and Nejmeh have applied the group concept to model the resources shared by a team of programmers. They introduced the notion of *visibility domains*, which model

groups of programmers executing nested transactions on versions of objects [Dowson and Nejmeh 89]. A visibility domain is a set of data items that can be shared by a group of users. Each transaction has a particular visibility domain associated with it. Any member of a visibility domain of a transaction may start a subtransaction on a new version of the copy of data that belongs to the transaction. The only criterion for data consistency is that the visibility domain of a transaction be a subset of the visibility domain of its parent.

7.3.1.2. Transaction Groups

In order to allow members of the same group to cooperate and to monitor changes in the database, there is a need to provide concurrency control mechanisms with a range of lock modes of varying exclusiveness. The *transaction groups* model proposed for the ObServer system replaces classical locks with <lock mode, communication mode> pairs to support the implementation of a nested framework for cooperating transactions [Skarra 91, Skarra and Zdonik 89, Fernandez and Zdonik 89]. The lock modes provided by ObServer indicate whether the transaction intends to read or write the object and whether it permits reading while another transaction writes, writing while other transactions read, and multiple writers of the same objects. The communication modes specify whether or not a transaction must be notified if another transaction either requests to access or update an object on which the first transaction holds a lock.

A transaction group (TG) is defined as a process that controls the access of a set of cooperating transactions (members of the transaction group) to objects from the object server. Since a TG can include other TGs, a tree of TGs is composed. Within each TG, member transactions and subgroups are synchronized according to an *input protocol* that defines some semantic correctness criteria appropriate for the application. The criteria are specified by semantic patterns, and enforced by a recognizer and a conflict detector. Examples of semantic patterns include lock filters, which specify the set of locks that the group may grant to its members, and operation filters, which specify the set of permissible operations that a member may request from its group. A typical lock filter for a cooperative transaction group might be one that includes only locks with non-NULL communication modes (e.g., all members must request locks that include

notification) [Fernandez and Zdonik 89]. The recognizer ensures that a lock request from a member transaction matches an element in the lock filter of the input protocol of the transaction's group. The conflict detector ensures that a request to lock an object in a certain mode does not conflict with the locks already held on the object.

If a transaction group member requests an object that is not currently locked by the group, the group has to request a lock on the object from its parent. The input protocol of the parent group, which controls access to objects, might be different from that of the child group. Therefore, the child group might have to transform its request lock into a different lock mode accepted by the parent's input protocol. The transformation is carried out by an *output protocol*, which consults a lock translation table to determine how to transform a lock request into one that is acceptable by the parent group.

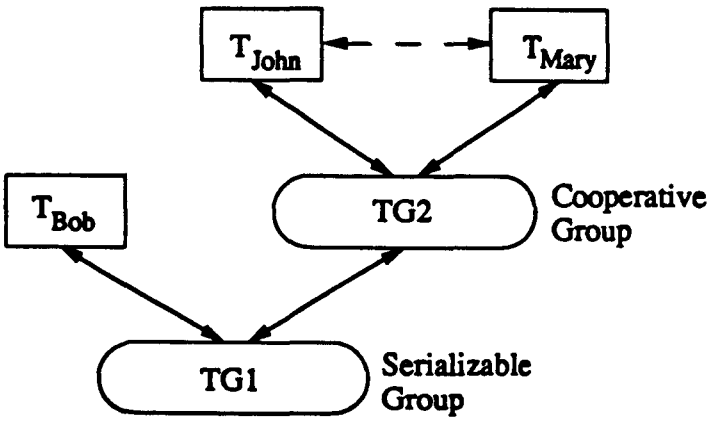

Figure 7-4: Transaction Groups

Transaction groups and the associated locking mechanism provide suitable low-level primitives for implementing a variety of concurrency control policies. To illustrate, consider the following example. Mary and John are assigned the task of updating the three `CFILE` objects in module `ModA`, while Bob is assigned responsibility for updating module `ModB`, which is fairly independent of `ModA`. Mary and John need to cooperate while updating the modules whereas Bob only needs to access the final result of the

modification of ModA in order to integrate the modified ModB with ModA. Two transaction groups are defined, TG1 and TG2. TG1 has T_{Bob} and TG2 as its members, and TG2 has T_{John} and T_{Mary} as its members. The output protocol of TG2 states that changes made by the transactions within TG2 are committed to TG1 only when all the transactions of TG2 have either committed or aborted. The input protocol of TG2 accepts lock modes that allow T_{Mary} and T_{John} to cooperate (e.g., see partial results of their updates to the modules) while isolation is maintained within TG1 (to prevent T_{Bob} from accessing the partial results of the transactions in TG2). This arrangement is depicted in figure 7-4.

7.3.1.3. Participant Transactions

The transaction groups mechanism defines groups in terms of their access to database objects in the context of a nested transaction system. Another approach is to define a group of transactions as *participants* in a specific *domain*[22] [Kaiser 90]. Participant transactions in a domain need not appear (from the point of view of participants) to have been performed in some serial order with respect to each other. The set of transactions that are not participants in a domain are considered *observers* of the domain. The set of observer transactions of a domain must be serialized with respect to the domain and should not be able to detect the non-serializable interleaving of participant transactions. A particular transaction may be a participant in some domain and an observer for others whose transactions access the same objects.

A user can initiate a transaction that nests subtransactions to carry out subtasks or to consider alternatives. Each subtransaction may be part of an implicit domain, with itself as the sole participant. Alternatively, one or more explicit domains may be created for subsets of the subtransactions. In the case of an implicit domain, there is no requirement for serializability among the subtransactions since there is only one participant in the implicit domain; however, each subtransaction must appear atomic with respect to any participants, other than the parent, in the parent transaction's domain.

---

[22]The word domain means different things in participant transactions, visibility domains, and development domains.

The domain in which a transaction participates would typically be the set of transactions associated with the members of a cooperating group of users working towards a common goal. However, unlike transaction groups, there is no implication that all the transactions in the domain commit together, or even that all of them commit (some may abort). Thus it is misleading to think of the domain as a top-level transaction, with each user's transaction as a subtransaction, although this is likely to be a frequent case in practice. The transaction groups mechanism described above is thus a special case of participant transactions.

Each transaction is associated with zero or one particular domains at the time it is initiated. A transaction that does not participate in any domain is the same as a classical (but interactive) transaction. Such a transaction must be serializable with respect to all other transactions in the system. A transaction is placed in a domain in order to non-serializably share partial results with other transactions in the same domain, but it must be serializable with respect to all transactions not in the domain.

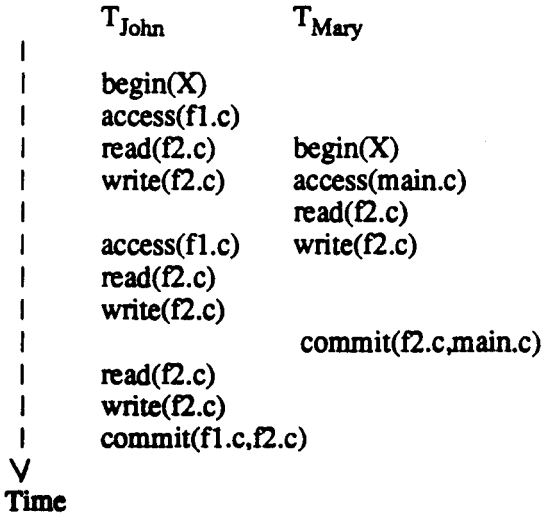

Figure 7-5: Example of a Participation Schedule

To illustrate, say a domain X is defined to respond to a particular modification request, and programmers Mary and John start transactions T_{Mary} and T_{John} that participate in X. Assume that an *access* operation is either a read or a write operation. The schedule

shown in figure 7-5 is not serializable. T_{Mary} reads the updates that T_{John} made to the CFILE object `f2.c` that are written but are not yet committed by T_{John}, modifies parts of `f2.c`, and then commits. T_{John} continues to modify `f1.c` and `f2.c` after T_{Mary} has committed. Since T_{Mary} and T_{John} participate in the same domain X, the schedule is legal according to the participation transactions mechanism.

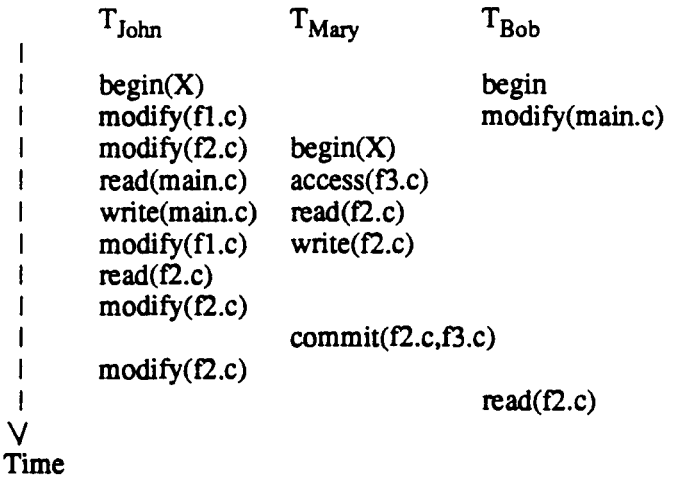

Figure 7-6: Example of a Participation Conflict

Now say that Bob starts a transaction T_{Bob} that is an observer of domain X. Assume that the sequence of events shown in Figure 7-6 happens. Bob first modifies the CFILE object `main.c`. This by itself would be legal, since T_{Bob} thus far could be serialized before T_{John}(but not after). But then T_{Bob} attempts to read `f2.c`, which has been modified and committed by T_{Mary}. This would be illegal even though T_{Mary} was committed. T_{Mary} cannot be serialized before T_{Bob}, and thus before T_{John}, because T_{Mary} reads the uncommitted changes to `f2.c` written by T_{John}. In fact, T_{Mary} cannot be serialized either before or after T_{John}. This would not be a problem if it was not necessary to serialize T_{Mary} with any transactions outside the domain. Mary's update to `f2.c` would be irrelevant if John committed his final update to `f2.c` before any transactions outside the domain accessed `f2.c`. Thus the serializability of transactions within a participation domain need be enforced only with respect to what is actually observed by the users who are not participants in the domain.

Our notion of development domains combines the notions of the previous group-oriented mechanisms. The group paradigm, transaction groups, and participant transactions apply only to transactions. Visibility domains group together objects accessed by a group of users. Development domains group together sessions (substitute for transactions), users and objects in one unit. The motivation behind this is to be able to specify cooperative concurrency control policies that specify not only which sessions (belonging to which users) can cooperate, but also on which objects they can cooperate, if that is required. Thus, it is possible to limit the cooperation to only a subset of objects in the database. With transaction groups and participant domains, it is an "all or none" policy; e.g., transactions can either cooperate, in which case they can access all objects cooperatively, or they do not, in which case they cannot access any objects cooperatively.

7.3.2. Adding Domains to EMSL and RBDE

EMSL provides three operations that the administrator can request to create and manipulate domains: `add_domain`, `delete_domain`, and `link_object`. The project administrator uses `add_domain` to create a new domain at any point during the development of a project. The administrator must assign a unique name to the new domain. The administrator can then use the `link_object` command to link a set of existing objects in the project database to the domain. By linking an object to a domain, the object is considered a part of the shared resources that the domain contains. A single object can be linked to more than one domain. Linking of objects to domains can be done on the fly by the administrator. In practice, however, the administrator will typically set up several development domains, which reflect the organization of development teams, before the developers actually start working on the project.

Given a set of domains, a developer can start a session that belongs to a specific domain by issuing the `start_session` command. Thus, the user can issue the command `start_session d1` to start a session in domain `d1`. If no domain is specified in the `start_session` operation, the session is assumed to belong to a singleton domain that includes only the session. In order to delete a domain, all the sessions contained in the domain must be ended first.

7.3.3. Integrating Development Domains into CRL

Domains can be used to write group-oriented control rules. In order to integrate domains in CRL, we add another function to the binding part of a control rule: `domain(?v)`, where `?v` is bound to either a transaction or a session. This function returns the domain to which the transaction or session belongs. A transaction belongs to the same domain to which the session containing the transaction belongs.

Since a domain is linked to a set of objects, we must also add a predicate that can determine if the object over which the conflict occurred is part of the shared resources of the domain. We add the predicate `linkto[?o,?d]`, which returns TRUE if the object bound to `?o` (typically, the `conflict_object`), is linked to the domain bound to `?d1`. Note that it is possible for the project administrator to ignore the set of objects that are linked to a domain; the administrator can write a control rule that allows two transactions that belong to the same domain to cooperatively (i.e., in a non-serializable manner) access objects which have not been linked to the domain. This would reduce development domains to domains in the participant transactions sense.

```
lock_conflict_cr [ FILE ]

selection_criterion:

bindings:
  ?t1 = holds_lock();
  ?t2 = requested_lock();
  ?s1 = session(?t1);
  ?s2 = session(?t2);
  ?d1 = domain(?s1);

body:

   if (and (?t1.type = cfc)
           (?t2.state = pending)
           (?d1 = ?d2)
           (linkto [conflict_object, ?d1]))
   then
   {
     terminate (?t1);
     add_obligation(?s1,(conflict_object.status=NotCompiled));
   };
;
```

Figure 7-7: Control Rule Using Development Domains

To illustrate how domains can be used in control rules, consider the control rule shown in figure 7-3 on page 215, which prescribes terminating a consistency forward chain in order to resolve a conflict. It might be desirable for the administrator to prescribe such an action only if the two sessions to which the conflicting transactions belong are in the same development domain. A revised version of that control rule, shown in figure 7-7, does exactly that. The control rule prescribes the `terminate` action only if two conditions are met: (1) the two conflicting transactions belong to the same domain, and (2) the object over which the conflict occurred is part of the shared resources of that domain.

Given the fact that we can now model development teams, we add an action to CRL that makes sense only in the context of developers who are closely cooperating with each other within a development team. The action, `merge`, involves merging two conflicting transactions by making one transaction a child of the other.

7.3.4. Merging Two Transactions

Merging two transactions is intended as an alternative to `abort`, `terminate` or `suspend`, when two transactions conflict because the agents whose execution they encapsulate are closely cooperating on the same set of objects. The main reason for merging transactions, in addition to avoid aborting either of them, is to complete as much of the work as the two transactions would have performed had they not conflicted. Merging two transactions entails making one transaction a subtransaction of the other, which implies that the two transactions will be allowed to carry out all of the automation and consistency implications of the agents they encapsulate. Both `suspend` and `terminate`, in contrast, prevent automation implications of one of the transactions (the one that was finished or terminated) from being carried out. Thus, the only reason to use the `merge` action is to have RBDE carry out the automation implications of a transaction that was involved in a conflict. The `merge` action was motivated by the work done on restructuring transactions by Pu *et al.* [Pu et al. 88]. We briefly describe this work here and then present the details of the `merge` action.

7.3.4.1. Related Work: Dynamic Restructuring of Transactions

In many advanced database applications, such as design environments, operations are interactive. The operations a user performs within a transaction might be of uncertain development, i.e., it cannot be predicted which operations the user will invoke a priori. Traditional transaction models do not allow transactions, especially long transactions, to be restructured dynamically to reflect a change in the needs of the users. To solve this problem, Pu *et al.* introduced two new operations, *split-transaction* and *join-transaction*, which are used to reconfigure long transactions while in progress.

The basic idea is that all sets of database actions that are included in a set of concurrent transactions are performed in a schedule that is serializable when the actions are committed. The schedule, however, may include new transactions that result from splitting and joining the original transactions. Thus, the committed set of transactions may not correspond in a simple way to the originally initiated set. A split-transaction divides an ongoing transaction into two or more serializable transactions by dividing the actions and the resources between the new transactions. The resulting transactions can proceed independently from that point on. More important, the resulting transactions behave as if they had been independent all along, and the original transaction disappears entirely, as if it had never existed. Thus, the split-transaction operation can be applied only when it is possible to generate two serializable transactions. Join-transaction does the reverse operation of merging the results of two or more separate transactions, as if these transactions had always been a single transaction, and releasing their resources atomically.

To clarify this technique, suppose that both Mary and John start two long transactions T_{Mary} and T_{John} to modify the two modules `ModA` and `ModB`, respectively. After a while, John finds out that he needs to access module `ModA`. Being notified that T_{John} needs to access `ModA`, Mary decides that she can "give up" the module since she has finished her changes to it, so she splits up T_{Mary} into T_{Mary} and T_{MaryA}. Mary then commits T_{MaryA}, thus committing her changes to `ModA` while continuing to retain `ModB`. Mary can do that only if the changes committed to `ModA` do not depend in any way on the previous or planned changes to `ModB`, which might later be aborted. T_{John} can now read `ModA` and use it for testing code. Mary commits T_{Mary} independently,

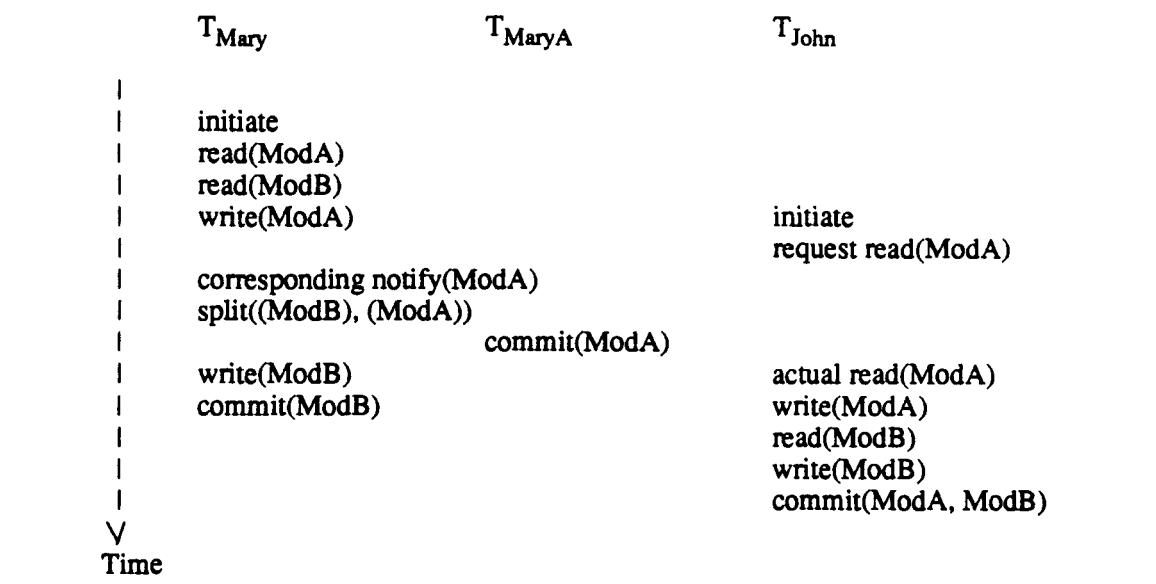

Figure 7-8: Example of Split-Transaction

thus releasing ModB. T_{John} can then access ModB and finally commit changes to both ModA and ModB. The schedule of T_{Mary}, T_{MaryA} and T_{John} is shown in figure 7-8.

The split-transaction and join-transaction operations relax the traditional concept of serializability by allowing transactions to be dynamically restructured. Eventually, the restructuring produces a set of transactions that are serialized. Unlike all the other approaches described earlier in this section, this approach addresses the issue of user control over transactions since it allows users to dynamically restructure their long transactions. This is most useful when opportunities for limited exchange of data among transactions arise while they are in progress. The split and join operations can be combined with the other techniques.

One interesting issue that arises in merging transactions (actually in restructuring transactions in general) is the user interface issue. One valid assumption for implementing joins is a multi-threaded transaction model: when transactions are joined, multiple threads of the same transaction can continue executing; the owners of these threads (i.e., the users who see the results of the threads) do not have to be the same user. This assumption is not supported in our model. Basically, when two transactions are merged

into one, the resulting transaction will have only one owner. Only one of the transactions continues to execute whereas the other ''disappears''. From the point of view of the owners of the two transactions, assuming that the two owners are different, one of them will be informed that his transaction has been merged with another transaction. As far as he is concerned, his transaction has ended and he can request a new command. The behavior seen by the other owner will not be different from the behavior he would have seen if the two transactions had not been merged. This is similar to the assumption made in [Kaiser 92].

In addition to the user interface issue, there are other complications that arise from the fact that transactions in our model have different types, and thus different purposes, and that they can be in different states. To be more formal, consider the conflict situation $C = (\langle T_i, T_j \rangle, \langle Ml_i[o], Nl_j[o] \rangle)$. The state of T_i is `active`, whereas the state of T_j is either `pending` or `inactive`. The action `merge` (T_i, T_j) involves making T_i a subtransaction of T_j. PCCP performs the merging only under two conditions which have to be met by the two transactions:

1. The type of T_i can be only `afc` or `cfc`.
2. If the type of T_i is `afc`, then the type of T_j must be either `tl` or `afc`.

The reason for the first condition is that it would not make sense to merge transactions of other types. A transaction whose type is either `tl` or `bi` corresponds directly to a user command. Making such a transaction a subtransaction of another user's transaction will cause confusion among the users. Suppose, for example, that John requested to `edit` an object. The TM creates T_{John} to encapsulate the command. Now suppose that T_{John} interferes with the execution of another transaction T_{Mary} that belongs to Mary. The fact that T_{John} interfered with T_{Mary} implies that T_{John} is in an `active` state, which means that the editor has not been invoked. Making T_{John} a subtransaction of T_{Mary} means that an editor will be invoked on Mary's screen; since Mary did not request any edit command, she will be surprised and confused about what happened. John, who expected the editor to be invoked on his screen, will be surprised that it was not. To avoid this kind of confusion, PCCP refuses merging a transaction whose type is either `tl` or `bi` with another transaction.

A transaction whose type is `bc` does not have any automation forward chains, by definition. Merging such a transaction with another transaction would defeat the whole purpose of the `merge` action. Thus, PCCP does not allow merging a transaction whose type is `bc` with another transaction.

The second condition basically states that PCCP will not merge an automation forward chain with a backward chain (recall that the type of T_j can only be `tl`, `afc` or `bc`). The reason for this is that the rule execution and chaining model does not perform automation forward chaining while in the middle of backward chaining. Backward chaining is performed only to try to make the condition of a rule that corresponds to the user command satisfied. Thus, RBDE tries to minimize the number of rules it fires during backward chaining. Whereas it is mandatory that consistency forward chaining be performed after a rule has been fired in a backward chaining cycle, performing automation forward chaining is only optional. Therefore, we have chosen not to perform automation forward chaining during backward chaining. Merging an `afc` transaction with a `bc` transaction would contradict this model and thus we chose not to allow it.

Given the two conditions above and a conflict situation involving T_i and T_j, where T_i is the interfering transaction (i.e., it is in an `active` state), there are four possibilities: (1) the type of T_i is `cfc` and T_j is in a `pending` state, (2) the type of T_i is `cfc` and T_j is in an `inactive` state, (3) the type of T_i is `afc` and T_j is in a `pending` state, and (4) the type of T_i is `afc`, the type of T_j is `tl` and T_j is in an `inactive` state. Note that a transaction whose type is `afc` cannot be in an `inactive` state. We briefly discuss the details of these four possibilities, giving an example of each.

The last two possibilities are straightforward. Consider a conflict situation $C = (\langle T_i, T_j \rangle, \langle Ml_i[o], Ml_j[o] \rangle)$. If the type of the interfering transaction is `afc`, then merging it with T_j simply requires adding it as if it were an automation implication of the rule encapsulated by T_j. T_i will be executed after the execution of T_j is completed. For example, say that John requested editing the `CFILE` object `f1.c`, which causes an automation forward chain leading to firing the `archive` rule on `f1.c` in an attempt to archive the object code of the new version of `f1.c` in one of the libraries (`LibA` in this case). Suppose that the transaction T_{John}, which encapsulates the execution of this

`archive` rule, interferes with another transaction T_{Bob}, which encapsulates the execution of the rule `archive f2.c` (i.e., John's `archive` rule wants to access `LibA` which is being accessed in write mode by Bob).

This conflict can be resolved by aborting T_{John} or suspending it. Neither action is necessary, however, because T_{John} can simply be merged with T_{Bob}. Merging the two transactions leads to executing `archive f1.c` after Bob's `archive` rule is completed. Meanwhile John can go ahead and request other commands since his transaction's execution has been completed as far as he is concerned; archiving `f1.c` and all the implications that result from that will be taken care of by Bob. This will avoid aborting John's `archive` rule and also avoids making John wait until Bob's `archive` rule is completed.

Merging in the case when the type of the interfering transaction T_i is `afc` is slightly more complicated. Assume that the ancestor of T_i that initiated the consistency forward chain of which T_i is a part is T_k (i.e., the type of T_k is not `cfc`). In order to merge T_i with T_j, the TM makes T_k a subtransaction of the parent of T_j. If such a parent does not exist (i.e, if T_j is of type `tl`), the TM creates a new hypothetical transaction, $T_{J.0}$, and makes both T_j and T_k subtransactions of $T_{J.0}$. Thus, T_j and T_k become siblings in the same nested transaction. The problem now reduces to ordering the execution of T_j and T_i within the same nested transaction (recall that the subtransactions of a nested transaction are executed in a serial order). This ordering depends on the state of T_j.

If T_j is in a pending state, then the TM rolls back the consistency forward chain (not T_k though), completes the execution of T_j, but instead of carrying out the consistency implications of T_j, it restarts the consistency forward chain of T_k. Only after this chain is completed does the TM carry out the consistency forward chain of T_j. The automation implications of all the transactions are pushed on the execution stack of T_j's client; these are executed after the consistency implications of both T_k and T_j have been completed.

To illustrate the `merge` action, consider again the example we presented in chapter 6 to demonstrate the `suspend` action. The example involved the set of rules shown in figure 6-6 on page 192. A conflict situation was detected because the transaction

```
compile-archive-cr [ CFILE ]

selection_criterion:
    commands: outdate-compile, compile, archive;

bindings:
   ?t1 = requested_lock();
   ?t2 = holds_lock();

body:

   if (and (?t1.type = cfc)
           (?t1.state = active)
           (?t2.state = inactive))
   then
     {
       merge(?t1, ?t2);
     };
;
```

Figure 7-9: Control Rule to Resolve Outdate-Archive Conflict by Merging

$T_{Mary.1}$ encapsulating Mary's `outdate-compile f1.c` rule (fired as a result of editing `i2.h`) interfered with the transaction T_{Bob} encapsulating Bob's `archive ModA` rule, which requires locking all `CFILE` objects contained in `ModA` (including `f1.c`) with an `R` lock. The conflict situation is depicted in figure 6-7 on page 194. Suppose that instead of firing the control rule shown in figure 6-8, which suspends T_{Bob}, PCCP attempts to resolve this conflict by firing the control rule shown in figure 7-9.

Unlike the control rule in chapter 6, the control rule in figure 7-9 will merge T_{Mary} and T_{Bob} into one transaction (owned by Bob). Both $T_{Mary.1}$ and T_{Mary} are actually committed (Mary is informed that her command has been executed and that further implications will be taken care of by Bob). The TM then creates the hypothetical transaction $T_{Bob.0}$ and makes T_{Bob} its subtransaction; the TM also creates $T_{Bob'}$ and $T_{Bob'.1}$ to replace T_{Mary} and $T_{Mary.1}$, respectively. $T_{Bob'}$ is created as a subtransaction of $T_{Bob.0}$. This will allow $T_{Bob'.1}$, which encapsulate the `outdate-compile f1.c` rule that caused the conflict, and the two other consistency implications of T_{Mary} (i.e., `outdate-compile main.c` and `outdate-compile f2.c`), to be carried out. Following that, T_{Bob} will be completed after its backward chaining cycle ($T_{Bob.1}$) is finished. Then after T_{Bob} commits, the automation implications of Mary's rule (and the automation implications of its consistency implications) will be carried out. The actual

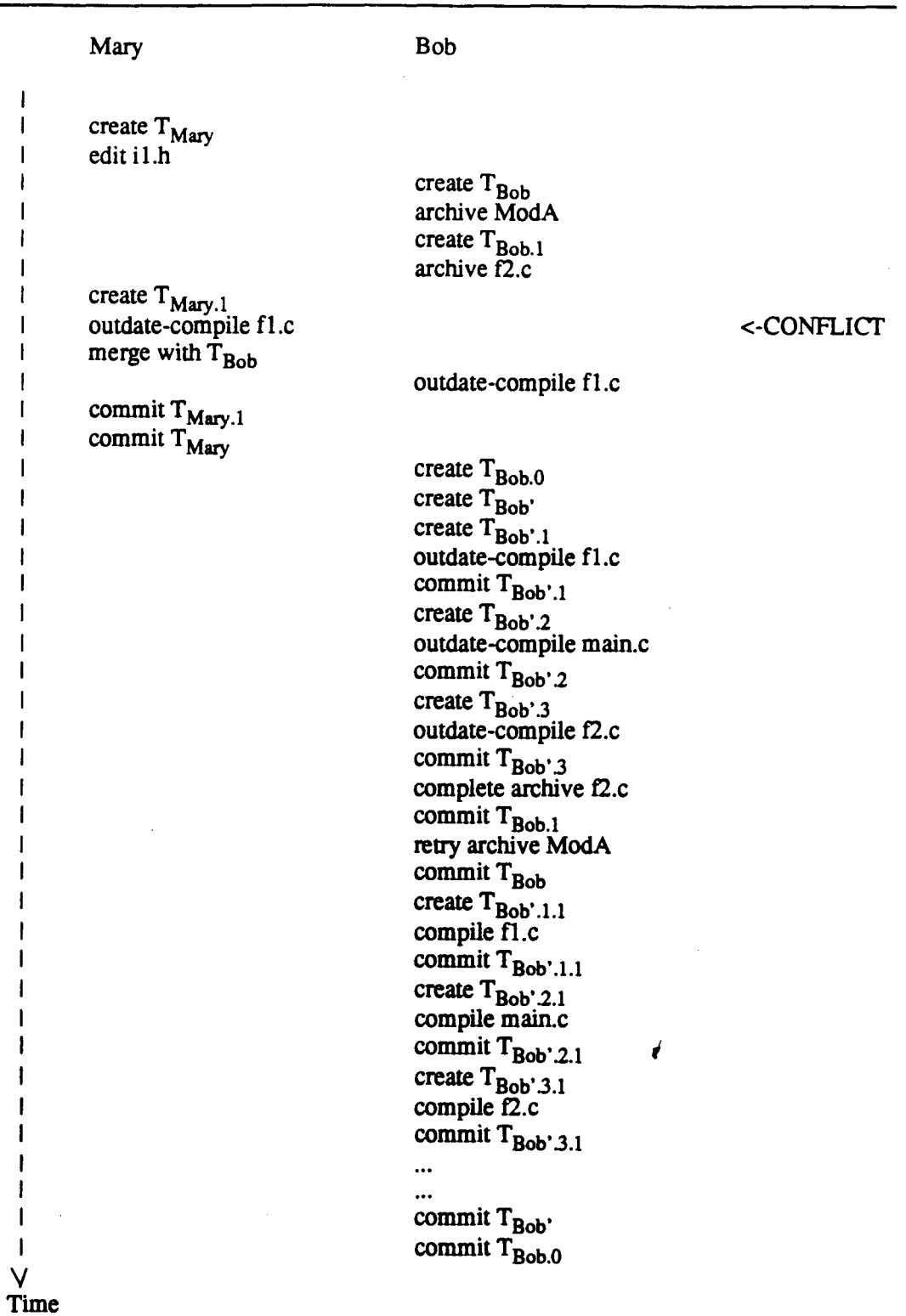

Figure 7-10: Example of Merging Two Transactions

schedule of transactions that gets executed (assuming no other conflicts) is shown in figure 7-10.

This completes our discussion of development domains and the `merge` action.

Chapter 8

Summary, Evaluation and Future Directions

In this final chapter, we summarize the results of this dissertation, overview the implementation status of our thesis work, evaluate our contributions, and finally discuss directions for future work.

8.1. Summary

This dissertation addressed the concurrency control problem in process-centered SDEs. We have constructed a database model for process-centered SDEs, identified the semantic information that is available in that model, and devised a concurrency control mechanism that uses this semantic information to support two protocols: a semantics-based concurrency control protocol that provides the default policy and a programmable protocol that provides a mechanism for specifying a project-specific policy. We have implemented our approach in a specific SDE architecture in order to prove its applicability and practicality.

8.1.1. The Architecture

In chapter 2, we constructed an architecture based on the MARVEL rule-based SDE. The ideas behind MARVEL and its implementation are not part of this thesis work; we have only formalized the MARVEL model and abstracted it in terms of database operations. Our architecture is composed of a specification language, EMSL, and a kernel, RBDE. RBDE provides process-centered assistance by loading specifications of both the data model and the software development process of a project, written in EMSL by the project administrator. EMSL models the development process in terms of rules that operate on the components of a software project. Each rule prescribes a development activity by defining the condition that must be satisfied before the activity can be carried

out, and the effects of executing the activity on the project components. These components are abstracted as objects and stored in a database. Both the condition and the effects are written in a subset of first-order logic.

RBDE presents the users (developers of the project) with commands, each of which is either a built-in command or correspond to a rule. If a user requests a command that is not built-in, RBDE executes the command by firing the corresponding rule. Automated assistance is provided by a chaining engine that fires a chain of rules in response to user commands, automatically performing some development activities that the users would have otherwise done manually. A client/server model enables RBDE to support multiple users, who can request commands concurrently.

8.1.2. The Concurrency Control Problem

When multiple developers cooperate on a project, they share a common database that contains all the components (source code, documentation, test suites, etc.) of the project. The developers may request commands that access objects in the shared database concurrently. RBDE participates in enacting the development process by initiating concurrent rule chains on behalf of the developers. The concurrent rule chains may *interfere*, violating the consistency of the objects accessed by the chains.

Interference (also called concurrency conflict) occurs when objects that have been accessed by a rule, `r1`, are updated by another rule, `r2`, in a concurrent chain. This overlapping access might, for example, lead to changing the values of attributes of objects that were read during the evaluation of the condition of `r1`. The validity of `r1`'s execution depends on the assumption that these objects will not be changed by other rules except after the chain containing `r1` is completed. Thus, if any of the objects is actually changed, `r1`'s execution might have to be invalidated. In addition, all the rules that were fired after `r1` in `r1`'s chain might have to be invalidated since the reason for firing them (i.e., the successful firing of `r1`) has now been reversed.

Interference cannot occur if rule chains are executed in a serial fashion (i.e., a rule chain starts and executes to completion before another one can be started), or in a serializable

fashion (i.e., equivalent to a serial execution). To formalize the notion of interference, we constructed a database model of RBDE in chapter 3. The execution of an individual command (built-in command or a rule) is abstracted as a set of access units, each of which consists of a set of database read and write operations that are performed atomically. The access units are grouped in units called agents. To guarantee the atomicity of agents, we encapsulate their execution in atomic transactions.

The TM (transaction manager) creates a transaction, with a unique identifier and a timestamp, to encapsulate the execution of an agent by a `begin` operation and a `commit` (or `abort`) operation. A *transaction* is a sextuple $T = (i, U_i, S, c, u, t)$, where i is the unique identifier of the transaction, U_i is the set of access units belonging to T, S is the set of subtransactions of T, c is the command whose execution is encapsulated by T, u is the owner of T, and t is the unique timestamp that the TM assigned to T.

In the multi-agent RBDE model the RP interleaves the execution of multiple agents at the granularity of access units. Corresponding to this interleaved execution of agents is a transaction schedule (or execution) in which the execution of the access units of multiple concurrent transactions (encapsulating the agents) are interleaved. The only kind of concurrent schedules that can cause interference are non-serializable schedules, i.e., schedules that interleave the execution of database operations of multiple transactions in a fashion that cannot be done in a serial schedule.

Based on this, the concurrency control problem can be divided into three subproblems: (1) finding a mechanism to detect non-serializable executions of concurrent transactions, (2) devising a concurrency control protocol that provides a default policy for resolving detected conflicts, and (3) developing a mechanism for overriding the default policy in specific situations in order to support cooperative non-serializable interactions whenever required.

8.1.3. The Solution

To solve the problem, we have developed a concurrency control mechanism composed of two modules: a conflict detection module and a conflict resolution module. The conflict detection module, which we presented in chapter 4, consists of a two-phase locking (2PL) mechanism employed by the TM and a nested granularity locking protocol, NGL, employed by the lock manager (LM) to detect conflicts. Each individual rule is encapsulated in an atomic transaction to guarantee that its three parts (condition, activity, and effects) execute as a unit. A rule can trigger a rule chain composed of several other rules; we group all the rules of one chain in a nested transaction, where each rule becomes a subtransaction. Before a transaction can execute an access unit, it must acquire appropriate locks on all the objects accessed by the database operations in the access unit. The TM acquires the locks on behalf of the transaction by requesting them from the LM. The NGL protocol employed by the LM detects locking conflicts (which indicate interference) between concurrent nested transactions.

The conflict resolution module is a Scheduler that employs two concurrency control protocols. The first, presented in chapter 5, is a semantics-based concurrency control protocol, SCCP, which provides the default concurrency control policy in RBDE. Given a concurrency conflict detected by NGL, SCCP decides which of the two conflicting transactions to abort. SCCP uses information about the consistency constraints of a project in order to construct a priority-based conflict resolution scheme. To make the consistency constraints of a project explicit, EMSL provides constructs that can be used to distinguish between *consistency predicates* and *automation predicates* in both the conditions and effects of rules; a single rule might contain both kinds of predicates. Consistency predicates constrain both forward and backward chaining. They prescribe to RBDE the mandatory steps that RBDE must perform in response to firing a rule.

The default resolution of serialization conflicts is based on the consistency requirements of the particular project. By default, RBDE should maintain the consistency constraints defined by the rules. SCCP uses the distinction between consistency predicates and automation predicates to do two things: (1) to construct a priority scheme based on the types and states of transactions, and (2) to define the abort and commit dependencies

between subtransactions and their parent transactions. The priority scheme is used to determine which of the transactions involved in a conflict should be aborted. Aborting a subtransaction sometimes (but not always) necessitates aborting other subtransactions in the same nested transaction. If the chain encapsulated by the transaction was initiated by a consistency predicate then the whole chain (i.e., all the subtransactions that represent the chain) must be aborted and rolled back. In contrast, if the rule chain was initiated by an automation chain, then only one subtransaction needs to be aborted while the whole chain is simply terminated.

The conflict resolution module provides a second protocol, PCCP, which can be used to override the default concurrency control policy. We presented PCCP in chapter 6. Unlike SCCP, PCCP does not have a hard-wired policy to determine how to resolve a conflict. Instead, PCCP matches the conflict to a control rule, which is written by the project administrator, and executes the actions prescribed by the control rule. Each control rule describes a conflict situation and prescribes actions to resolve the conflict. The control rules are written in a language called CRL, which provides a mechanism for defining the legal concurrent interactions between developers of the same project.

The actions prescribed by a control rule might lead to the relaxation of the default policy. These actions include aborting one of the conflicting transactions (the one that SCCP would not have aborted, for example), suspending one of the transactions while the other is finished, and terminating one of the transactions. Terminating a transaction encapsulating a consistency forward chain results in violating the consistency constraints of the project, as defined by the consistency predicates of the rule. RBDE tolerates this violation of consistency temporarily by marking the objects that are left in an inconsistent state; the consistency of these objects is established later on by completing the consistency forward chain that was terminated prematurely and unmarking the object.

In order to enhance the expressive power of CRL and EMSL, we extended both with three constructs that provide a context for tolerating inconsistency and modeling teamwork (chapter 7). The three constructs are user sessions, obligations and development domains. Both user sessions and obligations provide a mechanism for specifying when and by whom the consistency of objects should be re-established after prematurely ter-

minating a consistency forward chain. Development domains model teams of programmers by specifying which of the developers' sessions belong to which domain and the objects on which the sessions within each domain can cooperate. The fact the two transactions belong to sessions that are in the same domain is used in control rules to prescribe flexible resolution mechanisms such as merging two rule chains into one chain, so that their execution continues as one chain from that point on. This allows members of the same development team more flexibility in terms of accessing objects.

8.2. Implementation of the Thesis Work in MARVEL

Much of the work described in this dissertation has been implemented. In general, all of the work described in chapters 2, 3, and 4 has been implemented. Most of the work described in chapter 5 has been implemented. An initial implementation of CRL has been completed and work is in progress to implement all the constructs presented in chapter 6. The only two parts that have not been implemented at all yet are user sessions and domains, which were presented in chapter 7. In the rest of this section, we discuss in more detail the implementation of the various parts of this dissertation.

8.2.1. EMSL and RBDE

The extensions that both EMSL and RBDE added to the original implementation of MARVEL (MARVEL 2.6) have been fully incorporated in the first multi-user implementation, MARVEL 3.0. These extensions include the distinction between consistency and automation predicates, described in chapter 5, and the client/server model, as described in chapter 2. The `output` keywords has not been implemented. Basically, in the current implementation, all the arguments of a rule's activity are considered "output" arguments; thus, any attribute that is passed as an argument to an activity is included in the write set of the rule.

In addition, the syntax of the specification language used in MARVEL, MSL, is slightly different from the EMSL syntax we presented in chapter 2. The main difference is in the condition of a rule. In MSL, the quantifiers are included in the binding part of the condition, rather than in the property list. The rules given in appendix B, which work in MARVEL, show the MSL syntax.

8.2.2. The Transaction Model

The nested transaction model presented in chapters 4 and 5 has been implemented except for a few details. As described in these chapters, we have implemented a transaction model composed of a TM (transaction manager), an LM (lock manager), and a Scheduler. The TM supports nested transactions and categorizes transactions according to the type of chain they encapsulate.

The LM supports locking only at the granularity of objects and not attributes. Thus, if an attribute of an object is written, the whole object must be locked with a `W` (or `X`) lock. Also, instead of maintaining a lock table, the LM maintains the locks on the objects themselves. It is actually the OMS that sets and releases locks on objects, at the request of the LM. Unlike objects, however, locks are not persistent. Thus, if a crash occurs, the information about locks is lost. This is obviously not sufficient and needs to be replaced with a persistent lock table.

As described in the dissertation, the Scheduler employs two concurrency control protocols. The priority scheme described in chapter 5 has been implemented except for the distinction between interactive and non-interactive transaction (which is being implemented now). However, this scheme is currently not being used as the default policy because there are a few necessary pieces of code that must be written first. The main obstacle to fully implementing SCCP is that asynchronous communication between the clients and the server has not been implemented yet. This results in not being able to abort transactions whose state is `pending`, which is required by SCCP.

Currently, the default policy dictates that the active transaction, regardless of its type, be aborted if it interferes with another transaction. Work is in progress to implement SCCP in the new version of MARVEL.

Finally, a prototype implementation of CRL and PCCP has been done (not included in MARVEL 3.0 because it is still not robust enough). This implementation is limited in several respects: only the underlying mechanisms needed to implement all the actions and constructs presented in chapter 6 have been implemented but not the actual actions; we are currently implementing the `terminate`, `suspend` and `merge` actions. We

have completed the implementation of a parser for CRL, a search algorithm to find the control rule that most closely matches a conflict situation, an evaluator for conditions of control rules, and the marking and unmarking routines needed for the `terminate` action.

8.3. Contributions and Evaluation

We evaluate our thesis work by answering three questions: *why is solving the concurrency control problem in RBDE important? what are the contributions of this dissertation over previous work?* and *what are the limitations of our work and how can they be overcome?*.

8.3.1. The Importance of Solving the Problem

The concurrency control problem arises in all multi-user applications. Although the problem has been extensively studied for some applications, such as banking and airline reservations, which are similar in nature to each other, it has not been solved for advanced applications, such as SDEs. SDEs must be able to support multiple concurrent users in order to scale up to large-scale software development. Thus, the concurrency control problem will arise in any realistic SDE.

8.3.2. The Contributions

We have formalized and solved the concurrency control problem in the context of a specific SDE architecture. By doing that, we have analyzed the specific characteristics of advanced applications like SDEs, we have delineated some of the different pieces of semantic information that can be used to provide flexible concurrency control, and we have devised a transaction model that uses semantic information to prioritize transactions and re-order their executions to avoid conflicts.

The contributions of our work are:

1. The database model of RBDE. Although the database model we presented in chapter 3 was constructed in the context of RBDE, it is general enough to apply to other rule-based process-centered SDEs. In fact, the idea of

abstracting each activity in terms of database operations, and grouping together sets of operations into atomic units (access units) is applicable to all process-centered SDEs. The idea of having transactions encapsulate activities and of nesting transactions to reflect causality relationships between activities is also applicable to other process-centered SDEs. In short, our work is among the first to clearly identify the database issues in process-centered SDEs in a formal manner and to pinpoint the concurrency control problem in multi-user process-centered SDEs. Incidentally, the same database-centered approach might be useful when discussing the concurrency control problem of parallel production systems and inference engines. The working memory can be modeled as a database and rules can be abstracted in terms of read and write access to the working memory.

2. The characterization of some important aspects of the semantic information that can be extracted from the definition of the development process in process-centered SDEs. The development process of a project consists of several development steps (activities), which are explicitly modeled in a process-centered SDE (e.g., in terms of rules in RBDE). We have identified seven pieces of semantic information that can be extracted from the process model and the runtime environment: (1) the distinction between the two purposes of "enacting" (e.g., executing) the development process, maintaining consistency and providing automated assistance; (2) defining each development step as either interactive (i.e., involving human effort) or non-interactive; (3) the timestamp of each step; (4) the identity of each step (e.g., compiling, editing, sending mail, etc.); (5) the human developer who initiated the step; (6) the development task of which the step is a part; and (7) the development team to which each developer belongs. We believe that these seven pieces of information can be made available in every process-centered SDE. This semantic information is the basis for supporting concurrency and cooperation.

3. The SCCP protocol and the priority-based scheme. We have devised a mechanism for using three of the semantic information items listed above in order to provide a hard-wired default concurrency control policy, SCCP. Unlike traditional concurrency control protocols, SCCP ranks transactions according to their importance and aborts the less important of the two con-

flicting transactions when a conflict occurs. We would like to define the importance of a transaction in terms of how much effort and time will be wasted if the transaction is aborted and rolled back. This can be estimated by using the first three pieces of semantic information listed above. For example, an interactive activity, such as editing, involves more human effort than a non-interactive activity; thus, a transaction encapsulating an interactive activity is more important than one that encapsulates a non-interactive activity. A scheme similar to the priority scheme used in SCCP can be generated for any process-centered SDE that makes available the first three pieces of semantic information listed above.

4. CRL and the PCCP protocol. We have developed a special-purpose language that enables the project administrator to use the seven pieces of information listed above to determine how to resolve a conflict. In addition, we have provided four actions, `terminate`, `suspend`, `merge`, and `notify`, that can be used to implement a wide range of conflict resolution strategies. The `terminate` action, together with marking and unmarking, enables RBDE to tolerate inconsistency temporarily; obligations provide a mechanism for ensuring the consistency is re-established. The `merge` action allows the TM to interleave the steps of two nested transactions in a non-serializable fashion by merging the two transactions into one transaction. The `suspend` action uses a restricted form of blocking to avoid aborting any of the conflicting transactions. Finally, the `notify` action is a simple notification mechanism to inform the users of the progress of their transactions.

8.3.3. Evaluation

We have enumerated a list of requirements in chapter 3. These requirements, as explained earlier, were discussed widely in the literature. In this section, we evaluate how well our solution meets these requirements.

Supporting long-duration operations: Activation rules in RBDE invoke tools whose execution might last an arbitrarily long period of time. This causes the transaction that encapsulates the activation rule to be a long transaction. In addition, the rule chains

result in long nested transactions. We have handled long transactions in two ways: (1) by distinguishing between automation and consistency transactions, we are able to break up automation chains into independent transactions; (2) ...

User control: interactive vs. non-interactive transactions; pending transactions. SCCP attempts to have the least impact on the user.

8.3.4. Comparison to Related Work

The thesis of this dissertation is quite similar to two other dissertations: Skarra's [Skarra 91] and Sutton's [Sutton 90]. There are significant differences, however, between our approach and the approaches that these two dissertations took. Although Skarra addresses the problem of devising a programmable concurrency control policy, she does not solve the problem in the context of any specific environment. Instead, Skarra constructs an abstract cooperative transaction model based on transaction groups, which we described in chapter 7, and devises a finite state machine-based mechanism as a basis for concurrency control for her cooperative transactions model. Skarra implemented her work in a toy system.

There are two main differences between our work and Skarra's: (1) we construct a complete environment in which we implement the cooperative transaction model; and (2) our concurrency control model does not require monitoring of all operations, which is required in the case of finite state machines. In Skarra's model even serializable interactions between transactions have to be programmed in terms of finite state machines. Every operation must match at least one transition in one of the state machines. If an operation leads to a dead state, then it is illegal; the finite state machines must be in a final state for the database to be in a consistent state. In our model, in contrast, all serializable interactions are legal and do not require any additional processing.

Sutton's thesis, like ours, is directly applicable to SDEs. Sutton developed a flexible consistency model (FCM) in which explicit consistency constraints are defined on relations between objects. Whenever a relation is accessed, the constraints defined on the relation are checked to verify that they have not been violated. Sutton provides a

mechanism for tolerating inconsistency by "turning off" the enforcement of constraints temporarily. There are two major differences between our work and Sutton's: (1) Sutton's FCM mechanism is constructed a special-purpose programming language, APPL/A, rather than within a transaction mechanism; and (2) Sutton's mechanism for tolerating inconsistency requires checking all objects in the database when constraint enforcement is "turned on" because inconsistent objects (which became inconsistent when enforcement was turned off) are not marked. The first difference is significant because Sutton's mechanism is more akin to concurrent programming, where the exact concurrent interactions must be specified. Our mechanism, in contrast, is database-oriented, where all serializable concurrent interactions are allowed by default, and explicit specification of concurrent interactions is required only in the case of non-serializable interactions. Sutton's mechanism, like Skarra's, requires programming even serializable interactions, and thus suffers from the same overhead.

A third piece of work that is related to this dissertation is the CLF project [CLF Project 88]. CLF supports multiple user through a replicated database mechanism, in which each developer has a private copy of (parts of) the project database. Like RBDE, CLF distinguishes between automation and consistency aspects of the software process, but, unlike our thesis work, CLF does not use this distinction in the concurrency control mechanism. Instead, CLF implements an optimistic concurrency control mechanism that allows multiple developers to change the project database concurrently. Changes to the same objects in the database are then merged to produce a consistent version of the database. Our notion of tolerating inconsistency is based on Balzer's notion of tolerating inconsistency [Balzer 91], which, we assume, will be integrated with CLF at some point.

We have partially implemented our thesis work in MARVEL; we have implemented enough of the work to be able to identify some subtle points that have been ignored so far by the process-centered SDE community, such as the exact nature of the semantic information that must be extracted in order to support cooperation and how to extract this information. Most of the discussion about concurrency control and database support for process-centered SDEs, in contrast, is either theoretical or at the design level. For example, many of the process modeling formalisms (AP5 [Cohen 86] being one clear

exception) do not distinguish between consistency and automation in SDEs; we have shown how this distinction is crucial to providing a cooperative but yet consistency preserving environment. The user interface issue, largely ignored in advanced transaction mechanisms, is another area that the implementation forced us to deal with.

8.3.5. Limitations and Future Directions

We discuss three areas that credit further research and development. The first is extending EMSL and RBDE, and their corresponding implementation in MARVEL. The second area concerns the nested transaction model we presented in this dissertation. Finally, the last area suggests further development of the CRL language and the PCCP protocol to provide more expressiveness and capabilities. All the future work described in this section is intended to overcome the limitations and shortcomings of our thesis work.

8.3.5.1. Extensions to RBDE

As explained in chapter 2, built-in commands can be encapsulated in rules. However, these rules currently do not have any effects. They only add a logical condition that must be satisfied before the built-in command can be executed. It would be more meaningful if the rules encapsulating built-in commands can also have effects, with both consistency and automation predicates. These rules must be integrated in the chaining networks produced for normal rules. This would enable the administrator to specify implications for built-in commands. For example, adding a `CFILE` object to a `MODULE` should outdate the compilation of the module.

The tool integration model we briefly described in chapter 2 is based on envelopes, which support "black box" tool integration: tools are passed inputs and RBDE obtains the outputs of the tools; no further communication between RBDE and tools is supported. This is not sufficient for several reasons. One of the reasons that are relevant to this dissertation is that currently the TM cannot abort a tool in the middle of execution because RBDE has no way of communicating with the tool while it is being executed. Another reason is that we cannot support tools that access objects incrementally, i.e., whose input set cannot be determined before the tool is executed. An example of such a tool is `emacs`, which allows its user to access objects on the fly. It would be nice if

there was a way for `emacs` to tell RBDE that it needs to access a new object, so that the TM can go ahead and obtain the appropriate lock on the object on the fly. This of course complicates our 2PL transaction mechanism.

User sessions, as presented in chapter 7, do not provide enough support as a work unit. Our main concern with sessions was to be able to use them in control rules as a context for delayed actions. However, if the features and capabilities of sessions are extended, they can enhance the RBDE model. One addition is to support partial and complete aborts of sessions. By this we mean the ability to undo all or part of the work that has been done during a session. Supporting such a feature would complicate the entire transaction mechanism. In this case, the TM would not be able to commit the transactions that are part of a session until the session itself or part of it has been committed. Another addition to sessions would be to support nested sessions in order to provide more modularity of development tasks and the ability to define subtasks. A third extension to sessions is to support multi-user sessions, i.e., sessions that belong to more than one user at the same time. This would be a natural way to model tasks that require close cooperation among multiple developers.

In chapter 7, we described how obligations can be used to enhance the rule execution model of RBDE. However, we only described dynamic obligations that are added due to firing rules or control rules. It would greatly enhance the process modeling abilities of RBDE if the administrator could define user sessions with static obligations. Such obligations would reflect the purpose of the session. For example, the administrator, knowing exactly the development tasks that need to be completed, can define the obligations that guarantee the completion of these tasks. A development task to release a program, for instance, can be defined in terms of a session that has a static obligation specifying that all the modules of the program must be archived and tested (i.e., their `status` attributes must have the value "Archived" and their `test_status` attributes must have the value "Tested"). The developer who is assigned the task must ensure that this obligation is fulfilled before he can end the session that represents the task. Unlike dynamic obligations, static obligations are not removed except at the end of the session. In other words, they have to be satisfied at the end of the session.

8.3.5.2. Extensions to the Transactions Manager

One major shortcoming of our thesis work is that the mechanisms and protocols we presented are not directly apply to all SDEs. Although the notions of semantics-based concurrency control and programmable concurrency control are general enough to be applicable to many applications, we have not presented any general mechanism for realizing these notions in applications that differ significantly from our architecture. If our ideas were to be applied to other applications, we expect that some work will be required to devise application-specific concurrency control protocols equivalent to SCCP and PCCP. In some respects, it would have been more useful if we had designed a transaction manager that is completely separate from the process server (the rule processor in RBDE). Such a transaction manager can be "plugged" into any SDE, given that the SDE is able to provide the transaction manager with the semantics needed to implement protocols similar to SCCP and PCCP.

Another limitation of our transaction model is that the only ordering of subtransactions within a nested transaction is the serial ordering. In other words, each nested transaction is executed serially: the execution of one subtransaction cannot begin until the execution of the currently-active subtransaction of the same nested transaction has completed. This model is overly restrictive. Performance can be improved by executing automation forward chaining subtransactions of the same nested transaction in parallel if they do not conflict. The TM can determine if two subtransactions will conflict by collecting the read and write sets of the two subtransactions and checking whether the write set of one subtransaction overlaps with the read or write set of the other.

The current implementation of the lock manager supports locks only at the granularity of objects. This causes conflicts to be detected even if two transactions access different attributes of the same object. Supporting locking at the granularity of attributes will decrease the number of detected conflicts and thus decrease the overhead of resolving these conflicts (which are really not serious conflicts and should be ignored).

8.3.5.3. Enhancing the Expressive Power of CRL

The user specification part of the selection criterion of control rules currently supports user names. Thus, a control rule can be specific to two specific users. However, since specific users can be assigned different roles during the lifetime of a project, it would make more sense to specify user roles instead of user names (or userids). This would require that RBDE supports user roles. For example, there can be several user roles, such as manager, programmer, member_team1, secretary, etc. It would be more expressive if the user spec part of a control rule specifies two user role (e.g., manager and programmer), and prescribes what actions to take if a certain conflict involves a manager and a programmer. Then, the identity of the specific users who happen to be assigned the manager and the programmer roles becomes immaterial.

The logical expressions that comprise the conditions of control rules are limited in that one cannot write an expression that examines the complete histories of the two conflicting transactions. The expressive power of the condition construct can be enhanced if we used data path expressions (DPEs) or a similar concurrent formalism, such as the patterns in Skarra's dissertation [Skarra 91], instead of simple logical predicates. Such a formalism would enable us to compare the parallel execution of two histories.

8.4. Conclusions

By implementing out thesis work in MARVEL, we have shown in this dissertation that semantics-based concurrency control is a feasible approach to concurrency control in advanced database applications that require cooperation. The concurrency control mechanisms that were designed for traditional applications like banking and airline reservations are too restrictive for advanced applications. The problem has been, however, that the theoretical concurrency control mechanisms that were proposed for advanced applications were mostly abstract and impractical. Although we have developed a mechanism for only one class of advanced applications, it is possible that our approach of extracting semantics and using them to support cooperation is applicable to many other classes of advanced applications. It is our approach, rather than the exact details of our mechanism, that we expect to be useful for other researchers in the field.

Bibliography

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

[Balzer 87]
Balzer, R. M.
Living in the Next Generation Operating System.
IEEE Software 4(6):77-85, November, 1987.

[Balzer 91]
Balzer, R.
Tolerating Inconsistency.
In *13th International Conference on Software Engineering*, pages 158-165. IEEE Computer Society Press, Austin, TX, May, 1991.

[Bancilhon et al. 85]
Bancilhon, F., Kim, W., and Korth, H.
A Model of CAD Transactions.
In *11th International Conference on Very Large Data Bases*, pages 25-33. Morgan Kaufmann, Stockholm, Sweden, August, 1985.

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

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

[Barghouti and Kaiser 91a]
Barghouti, N. S. and Kaiser, G. E.
Concurrency Control in Advanced Database Applications.
ACM Computing Surveys 23(3):269-317, September, 1991.

[Barghouti and Kaiser 91b]
Barghouti, N. S., and Kaiser, G. E.
Scaling Up Rule-Based Development Environments.
In *Third European Software Engineering Conference, ESEC '91*, pages 380-395. Springer-Verlag, Milan, Italy, October, 1991.
Published as *Lecture Notes in Computer Science* no. 550.

[Bates and Wileden 83]
Bates, P. and Wileden, J. C.
An Approach to High-Level Debugging of Distributed System.
In *ACM SIGSoft/SIGPlan Software Engineering Symposium on High-Level Debugging*, pages 107-111. Pacific Grove, CA, March, 1983.
Special issue of *Software Engineering Notes*, 8(4), August 1983.

[Beeri et al. 88] Beeri, C., Schek, H. -J., and Weikum, G.
Multilevel Transaction Management, Theoretical Art or Practical Need?
In *International Conference on Extending Database Technology: Advances in Database Technology, EDBT '88*, pages 134-154. Springer-Verlag, Venice, Italy, March, 1988.

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

[Benali et al. 89] Benali, K. et al.
Presentation of the ALF Project.
In *Ninth International Conference on System Development Environments and Factories*. Berlin, Germany, May, 1989.

[Bernstein 87] Bernstein, P.
Database System Support for Software Engineering -- An Extended Abstract.
In *Ninth International Conference on Software Engineering*, pages 166-178. IEEE Computer Society Press, Monterey, CA, March, 1987.

[Bernstein and Goodman 81]
Bernstein, P., and Goodman, N.
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[Bernstein et al. 87]
Bernstein, P. A., Hadzilacos, V. and Goodman, N.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley, Reading, MA, 1987.

[Bjork 73] Bjork, L. A.
Recovery Scenario for a DB/DC System.
In *28th ACM National Conference*, pages 142-146. ACM Press, Atlanta, GA, August, 1973.

[Bobrow 86] Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik M. and Zdybel, F.
CommonLoops: Merging Common Lisp and Object-Oriented Programming.
In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17-29. ACM Press, Portland, OR, September, 1986.

[Boehm 88] Boehm, B. W.
A Spiral Model of Software Development and Enhancement.
Computer 21(5):61-72, May, 1988.

[Chang and Lee 73]
Chang, C-L., and Lee, R. C.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, NY, 1973.

[Cheatham 90] Cheatham, T. E.
The E-L System Support for Process Programs.
In Katayama, T. (editor), *Sixth International Software Process Workshop*. IEEE Computer Society Press, Hakodate, Hokkaido, Japan, October, 1990.
In press.

[Chrysanthis and Ramamritham 90]
Chrysanthis. P. K., and Ramamritham, K.
ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior.
In *ACM SIGMOD International Conference on the Management of Data*, pages 194-203. ACM Press, Atlantic City, NJ, May, 1990.

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

[CLF Project 88] *CLF Manual*
University of Southern California, Information Sciences Institute, Marina del Rey, CA, 1988.

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

[Cohen 89] Cohen, D.
Compiling Complex Database Transition Triggers.
In *ACM SIGMOD International Conference on the Management of Data*, pages 225-234. ACM Press, New York, NY, 1989.
Published as a special issue of *SIGMOD Record*, 18(2).

[Dart et al. 87] Dart, S. A., Ellison, R. J., Feiler, P. H., and Habermann, A. N.
Software Development Environments.
Computer 20(11):18-28, November, 1987.

[Davies 73] Davies, C. T.
Recovery Semantics for a DB/DC System.
In *28th ACM National Conference*, pages 136-141. ACM Press, Atlanta, GA, August, 1973.

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

[Dowson and Nejmeh 89]
Dowson, M., and Nejmeh, B.
Nested Transactions and Visibility Domains.
In *ACM SIGMOD Workshop on Software CAD Databases*, pages 36-38. ACM Press, Napa, CA, February, 1989.
Position paper.

[Eastman 80] Eastman, C.
System Facilities for CAD Databases.
In *17th ACM Design Automation Conference*, pages 50-56. ACM Press, June, 1980.

[Eastman 81] Eastman, C.
Database Facilities for Engineering Design.
In *Proceedings of the IEEE Computer Society*, pages 1249-1263. IEEE Computer Society Press, October, 1981.

[El Abbadi and Toueg 89]
El Abbadi, A. and Toueg, S.
The Group Paradigm for Concurrency Control Protocols.
IEEE Transactions on Knowledge and Data Engineering 1(3):376-386, September, 1989.

[Eswaran et al. 76]
Eswaran, K., Gray, J., Lorie, R. and Traiger, I.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[Feldman 79] Feldman, S. I.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[Fernandez and Zdonik 89]
Fernandez, M. F., and Zdonik, S. B.
Transaction Groups: A Model for Controlling Cooperative Work.
In *Third International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128-138. Queensland, Australia, January, 1989.

[Garcia-Molina 83]
Garcia-Molina, H.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
ACM Transactions on Database Systems 8(2):186-213, June, 1983.

[Garcia-Molina and Salem 87]
Garcia-Molina, H., and Salem, K.
SAGAS.
In Dayal, U., and Traiger, I. (editor), *ACM SIGMOD Annual Conference*, pages 249-259. ACM Press, San Francisco, CA, May, 1987.

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

[Gray 78] Gray, J.
Notes On Database Operating Systems.
IBM Research Report RJ2188, IBM Research Laboratory, San Jose, CA, 1978.

[Gray et al. 76] Gray, J., Lorie R., Putzolu, G. and Traiger, I.
Granularity of Locks and Degrees of Consistency in a Shared Database.
Modeling in Data Base Management Systems.
North Holland, Amsterdam, Holland, 1976, pages 365-395.

[Heineman et al. 91]
Heineman, G. T., Kaiser, G. E., Barghouti, N. S. and Ben-Shaul, I. Z.
Rule Chaining in MARVEL: Dynamic Binding of Parameters.
In *Sixth Annual Knowledge-Based Software Engineering Conference*, pages 276-287. Syracuse, NY, September, 1991.

[Hoare 69] Hoare, C.A.R.
An Axiomatic Approach to Computer Programming.
Communications of the ACM 12(10):576-580, 583, October, 1969.

[Huff 89] Huff, K. E.
Plan-Based Intelligent Assistance: An Approach to Supporting the Software Development Process.
PhD thesis, Computer and Information Science Department, University of Massachusetts at Amherst, September, 1989.

[Ishida and Stolfo 85] Ishida, T. and Stolfo, S. J.
Toward the Parallel Execution of Rules in Production System Programs.
In *International Conference on Parallel Processing*, pages 568-575. IEEE Computer Society Press, 1985.

[Kaiser 90] Kaiser, G. E.
A Flexible Transaction Model for Software Engineering.
In *Sixth International Conference on Data Engineering*, pages 560-567. IEEE Computer Society Press, Los Angeles, CA, February, 1990.

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

[Kaiser 92] Kaiser, G. E. and Pu, C.
Dynamic Restructuring of Transactions.
Database Transaction Models for Advanced Applications.
Morgan Kaufmann, San Mateo, CA, 1992, Chapter 8.
In press. Available as Columbia University Department of Computer Science, CUCS-012-91, August 1991.

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

[Kaiser et al. 88b] Kaiser, G. E., Barghouti, N. S., Feiler, P. H., and Schwanke, R. W.
Database Support for Knowledge-Based Engineering Environments.
IEEE Expert 3(2):18-32, Summer, 1988.

[Kaiser et al. 90] Kaiser, G. E., Barghouti, N. S., and Sokolsky, M. H.
Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel.
In *23rd Annual Hawaii International Conference on System Sciences*, pages 131-140. IEEE Computer Society Press, Kona, HI, January, 1990.

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

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

[Kim et al. 87] Kim, W., Banerjee, J., and Chou, H.
Composite Object Support in an Object-Oriented Database System.
In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 118-125. ACM Press, Orlando, FL, October, 1987.

[Klahold et al. 85] Klahold, P., Schlageter, G., Unland, R., and Wilkes, W.
A Transaction Model Supporting Complex Applications in Integrated Information Systems.
In *ACM SIGMOD International Conference on the Management of Data*, pages 388-401. ACM Press, Austin, TX, May, 1985.

[Korth and Silberschatz 86]
Korth, H., and Silberschatz, A.
Database System Concepts.
McGraw-Hill Book Company, New York, NY, 1986.

[Korth and Speegle 90]
Korth, H., and Speegle, G.
Long-Duration Transactions in Software Design Projects.
In *Sixth International Conference on Data Engineering*, pages 568-574. IEEE Computer Society Press, Los Angeles, CA, February, 1990.

[Kuo et al. 90] Kuo, S., Moldovan, D., and Cha, S.
Control in Production Systems with Multiple Rule Firings.
In *International Conference on Parallel Processing*, pages II-243 - II-246. The Pennsylvania State University Press, August, 1990.

[Kutay and Eastman 83]
Kutay, A., and Eastman, C.
Transaction Management in Engineering Databases.
In *Annual Meeting of Database Week; Engineering Design Applications*, pages 73-80. IEEE Computer Society Press, San Jose, CA, May, 1983.

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

[Linton 81] Linton, M.
A Debugger for the Berkeley Pascal System.
Master's thesis, University of California at Berkeley, June, 1981.

[Lynch 83] Lynch, N. A.
Multilevel Atomicity — A New Correctness Criterion for Database Concurrency Control.
ACM Transactions on Database Systems 8(4):484-502, December, 1983.

[Marvel 91] Programming Systems Laboratory.
Marvel 3.0 Administrator's Manual
Columbia University Department of Computer Science, 1991.
Technial Report # CUCS-032-91.

[Minsky and Lockman 85]
Minsky, N. H. and Lockman, A. D.
Ensuring Integrity by Adding Obligations to Privileges.
In *Eighth International Conference on Software Engineering*, pages 92-102. IEEE Computer Society Press, London, UK, August, 1985.

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

[Minsky and Rozenshtein 90]
Minsky, N. H., and Rozenshtein, D.
Configuration Management by Consensus: An Application of Law-Governed Systems.
In Taylor, R. N. (editor), *Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 183-192. ACM Press, Irvine, CA, December, 1990.
Special issue of *Software Engineering Notes*, 15(6), December 1990.

[Miranker et al. 90]
Miranker, D. P., Kuo, C. M., and Browne, J. C.
Parallelizing Compilation of Rule-Based Programs.
In *International Conference on Parallel Processing*, pages II-247 - II-251. The Pennsylvania State University Press, August, 1990.

[Moss 85] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, Cambridge, MA, 1985.

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

[Papadimitriou 86] Papadimitriou, C.
The Theory of Database Concurrency Control.
Computer Science Press, Rockville, MD, 1986.

[Pasik 89] Pasik, A. J.
A Methodology for Programming Production Systems and its Implications on Parallelism.
PhD thesis, Columbia University Department of Computer Science, 1989.

[Perry 87] Perry, D. E.
Software Interconnection Models.
In *Ninth International Conference on Software Engineering*, pages 61-69. IEEE Computer Society Press, Monterey, CA, March, 1987.

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

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

[Pu et al. 88] Pu, C., Kaiser, G. E., and Hutchinson, N.
Split Transactions for Open-Ended Activities.
In *14th International Conference on Very Large Databases*, pages 26-37. Morgan Kaufmann, Los Angeles, CA, August, 1988.

[Reed 78] Reed, D.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT Laboratory of Computer Science, September, 1978.
MIT LCS Technical Report 205.

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

[Rochkind 75] Rochkind, M. J.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1(4):364-370, December, 1975.

[Royce 87] Royce, W. W.
Managing the Development of Large Software Systems: Concepts and Techniques.
In *Ninth International Conference on Software Engineering*. IEEE Computer Society Press, Monterey, CA, March, 1987.

[Salem et al. 87] Salem, K., Garcia-Molina, H., and Alonso, R.
Altruistic Locking: A Strategy for Coping with Long Lived Transactions.
In *Second International Workshop on High Performance Transaction Systems*, pages 19.1 - 19.24. Pacific Grove, CA, September, 1987.

[Schmolze 89] Schmolze, J. G.
Guaranteeing Serializable Results in Synchronous Parallel Production Systems.
Technical Report 89-5, Tufts University Department of Computer Science, October, 1989.

[Skarra 91] Skarra, A. H.
A Model of Concurrency Control for Cooperating Transactions.
PhD thesis, Department of Computer Science at Brown University, May, 1991.

[Skarra and Zdonik 89]
Skarra, A. H. and Zdonik, S. B.
Concurrency Control and Object-Oriented Databases.
Object-Oriented Concepts, Databases, and Applications.
ACM Press, New York, NY, 1989, pages 395-421.

[Stallman 84] Stallman, R. M.
EMACS: The Extensible, Customizable, Self-Documenting Display Editor.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984, pages 300-325.

[Stolfo 84] Stolfo, S. J.
Five Parallel Algorithms For Production System Execution on the DADO Machine.
In *National Conference on Artificial Intelligence, AAAI-84*, pages 300-307. August, 1984.

[Sutton 90] Sutton, S. M. Jr.
APPLIA: A Prototype Language for Software Process Programming.
PhD thesis, Department of Computer Science, University of Colorado at Boulder, July, 1990.

[Taylor et al. 88] Taylor, R. N., Selby, R. W., Young, M., Belz, F. C., Clarke, L. A., Wileden, J. C., Osterweil, L. and Wolf, A. L.
Foundations of the Arcadia Environment Architecture.
In Henderson, P. (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1-13. ACM Press, Boston, MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[Tichy 85] Tichy, W. F.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[Yeh et al. 89] Yeh, S., Ellis, C., Ege, A., and Korth, H.
Performance Analysis of Two Concurrency Control Schemes for Design Environments.
Information Sciences 49:3-33, 1989.

Appendix A

Glossary

A.1. Acronyms

2PL Two-Phase Locking: the most popular concurrency control protocol employed by traditional schedulers in DBSs. 2PL depends on well-formed transactions that have two phases, one in which they acquire all the locks and a second phase in which all the locks are released.

CRL Control Rule Language: the language in which control rules are written.

CE Command Executor: the RBDE component responsible for executing built-in commands such as `add`, `delete`, `copy`, `move` and `link`.

DBS Data Base System: a general term that denotes any system that uses a database, including a simple file system with a transaction management facility

EMSL
Extended MARVEL Strategy Language: the language in which the data and process models of a project are encoded. EMSL is based on the MARVEL Strategy Language (MSL) but differs slightly in its syntax.

LM Lock Manager: the component of RBDE responsible for setting and releasing locks on objects. The LM employs a locking protocol (NGL) to detect incompatible locks.

MGL Multiple Granularity Locking: a locking protocol suggested by Gray et al. to minimize the number of locks necessary in a composite object hierarchy.

NGL Nested Granularity Locking: the protocol used by the LM to detect locking conflicts. NGL employs a compatibility matrix that is coded by the project administrator. NGL is also provided with the lock types and the relative strength of locks; none of the lock types is built-in.

PCCP
Programmable Concurrency Control Protocol; the protocol employed by the Scheduler to select and execute control rules.

RBDE
Rule-Based Development Environment: the kernel of our architecture; RBDE consists of the RP, the OMS, the TM, the Scheduler, and the LM.

RP Rule Processor: the component of RBDE responsible for executing rules and providing assistance through rule chaining.

SCCP
Semantics-based Concurrency Control Protocol; the default protocol employed by the Scheduler to resolve conflicts.

SDE Software Development Environment: a software system that assists developers of a software project. SDEs typically provide a collection of tools, an object management system, and a uniform interface to the objects and the tools.

TM Transaction Manager: the component of RBDE responsible for encapsulating the execution of rules in transactions, and implementing a 2PL locking policy.

A.2. General Terms

Interference
occurs between two transaction T_i and T_j, when the interfering transaction T_i requests a lock that is incompatible with a lock that T_j holds; also referred to as a *concurrency conflict* or *conflict situation.*

Process-centered SDE
an SDE that provides specialized assistance in carrying out the software development process of a project. Such an SDE is based on a formalism for encoding the software process.

Nested Transaction
a composition of a set of subtransactions; each subtransaction can itself be a nested transaction. To other transactions, only the top-level nested transaction is visible and appears as a normal atomic transaction.

Rule-based process modeling
using a rule language as a formalism for encoding the software process. Such an encoding is used to provide automated assistance in a process-centered SDE.

Transaction
a unit that groups a set of database operations and guarantees that this set will be executed as if they were executed atomically. A transaction transforms the database from one consistent state to another.

Serializability
a correctness criterion that establishes that an execution of concurrent transactions is correct only if it is either serial or if it is equivalent to a serial execution.

A.3. RBDE Terms

Access Unit
a set of database operations (either read or write) that are executed as one unit by the server.

Agent
a sequence of one or two database access units that must appear to have been performed atomically. An agent abstracts the execution of an individual rule or a built-in command. See **Nested Agent**.

Attribute
a typed field in an object. An attribute can be of four kinds: status, data, structural, or link attribute. The four kinds of attribute store information about the status, contents, structure, or relationships of an object, respectively.

Backward chaining
the process of firing rules in order to attempt to make the condition of a rule satisfied.

Client
a process with which the developer interacts. The client presents the developer with the RBDE built-in commands, and commands corresponding to the rules that the administrator loaded into RBDE.

Class a template that defines attributes. Each project's components are abstracted as an instance of a class.

Cooperation
exchanging objects in a non-serializable manner.

Envelope
an executable program that is invoked by RBDE (MARVEL) to initiate an external tool resident on the operating system. Envelopes serve as intermediaries between RBDE and external tools.

Forward chaining
the process of firing rules as a result of asserting the effect of a rule.

Nested Agent
a composition of a set of agents, each of which can itself be a nested agent. A nested agent abstracts the execution of a rule chain.

Original rule
a rule which is fired in direct response to a user command and not through chaining.

Object
an abstraction for a project component. An object is an instance of one of the classes defined in the project type set; it inherits attributes from the class.

Project database
the database where RBDE stores objects belonging to a single project.

Read operation
a database operation that returns the value stored in an attribute of an object in the project database; we use `Read[o.att]` to denote a read operation.

Read Set
is the set of all objects bound to the variables of the rule, including the parameters.

Rule a construct that prescribes the condition, the activity and the effects of a development activity. There are two kinds of rules: inference rules, in which the activity is empty, and activation rules, which invoke external tools.

Server
a process that controls all access to the project database. There is a single server per project database. A server can service several clients concurrently by interleaving the execution of their requests.

State of a transaction
a field in the transaction's entry in the transaction table that indicates the state of execution of a transaction. There are five states:

1. **Active**: the transaction is acquiring locks.

2. **Inactive**: the transaction is waiting for backward chaining to complete before it can continue its execution.

3. **Pending**: the transaction is waiting until the client completes the execution of the activation rule whose execution the transaction encapsulates.

4. **Ended**: the transaction has finished executing all of its operations, but it has not committed yet because some of its subtransactions have not terminated yet.

5. **Suspended**: the transaction's execution has been suspended by the TM pending the completion of another transaction.

Transaction

an entity that the TM creates to encapsulate the execution of an individual rule or a built-in command. A transaction has nine attributes: its identifier, a set of access units, a set of subtransactions, the command it encapsulates, its owner, its timestamp, its lock set, its type and its state.

Type of a transaction

a categorization of transactions that indicates the purpose of initiating the transaction. There are five types of transactions:

1. **Top-level (tl)**: the transaction encapsulating the execution of an original rule.

2. **Consistency forward chaining (cfc)**: a transaction that encapsulates the execution of a rule that is fired within a consistency forward chain.

3. **Automation forward chaining (afc)**: a transaction that encapsulates the execution of a rule that is fired within an automation forward chain.

4. **Backward chaining (bc)**: a transaction that encapsulates the execution of a rule that is fired within a backward chain.

5. **Built-in (bi)**: a transaction encapsulating the execution of a built-in command.

Write operation

a database operation that changes the value of a single attribute of an object in the project database; we use `Write[o.att, val]` to denote a write operation.

Write Set

the set of all objects bound to the variables used either as output arguments in the activity or in the left hand side of any assignment predicate in the effects of the rule.

Appendix B

Implementation of Example in MARVEL

In this appendix we give the complete set of class definitions, rule definitions and control rules used throughout the dissertation. These defitions were loaded into a MARVEL environment and used to test the validity of our examples. The syntax used in this appendix is that of MSL, which differs slightly from EMSL. MSL groups definitions into modules called *strategies*.

B.1. The Project Type Set

The following classes model the organization and structure of the example project in chapter 1.

```
strategy data_model

imports none;
exports all;

objectbase

# Top-level program instances.
PROGRAM :: superclass ENTITY;
    modules : set_of MODULE;
    libraries : set_of LIB;
    includes : INCLUDE;
end

# Superclass of any class whose instances can be reserved.
RESERVABLE :: superclass ENTITY;
    locker : user;
    purpose: string;
    reservation_status : (CheckedOut,Available,None) = None;
end

# All files share have a content and a timestamp
FILE :: superclass RESERVABLE;
    timestamp : time;
    contents : text;
end

# The extension of include files is ''.h'' by convention.
HFILE :: superclass FILE;
    contents : text = ".h";
end
```

```
# Group include files that are related into one object.
INCLUDE :: superclass ENTITY;
    hfiles : set_of HFILE;
    archive_status : (Archived,NotArchived,INotArchived)
                          = NotArchived;
end


# A C source file has many more attributes than other files,
# specifically an object code and some status attributes.
CFILE :: superclass FILE;
    contents : text = ".c";
    error_msg : text = ".err";
    includes : set_of link HFILE;
    status : (NotCompiled, Compiled, NotArchived,
              Archived, Error, Initial) = Initial;
    object_code : binary = ".o";
    test_status : (Tested, NotTested, Failed) = NotTested;
    object_timestamp : time;
    archive_timestamp : time;
    libs : set_of link LIB;
end


# Each library object has a binary file whose extension is ".a"
# in which the object code of C source files are archived.
LIB :: superclass ENTITY;
    afile : binary = ".a";
    archive_status : (Archived,NotArchived,INotArchived)
                          = NotArchived;
    timestamp : time;
end


# A Module object groups together several C source files.
MODULE :: superclass RESERVABLE;
    archive_status : (Archived,NotArchived,None)
                          = NotArchived;
    test_status : (Tested, NotTested, Failed) = NotTested;
    cfiles : set_of CFILE;
    modules : set_of MODULE;
    lib : link LIB;
end


TEST_SUITE :: superclass FILE;
   test : string;
end

end_objectbase
```

B.2. The Tool Definitions

The following are the tool definitions of all the tools we used in our examples. Each definition models a tool with possibly several operations. Each operation is handled by invoking the envelope whose name is given after the "string" type keyword.

```
strategy tools

imports data_model;
exports all;

objectbase

# The editor tool has two operations; the first invokes the
# editor and supplies it with a TAGS file while the other
# invokes the editor without a TAGS file.  The two envelopes
# specified invoke emacs.

EDITOR :: superclass TOOL;
    edit : string = editor;
    editnt : string = editor_no_tags;
end


# The various operations for the compiler are used to handle
# Yacc and Lex files.
COMPILER :: superclass TOOL;
    compile : string = compile;
    lex_compile : string = lex_compile;
    yacc_compile : string = yacc_compile;
end


ARCHIVER :: superclass TOOL;
    archive        : string = stuff;
    list_archive : string = list_archive;
    randomize    : string = randomize;
    update       : string = update;
end


# The RCS tool si used to either reserve or deposit a file.
RCS :: superclass TOOL;
    reserve : string = reserve;
    deposit : string = deposit;
end


TESTER :: superclass TOOL;
    run_test : string = test;
end

end_objectbase
```

B.3. The Project Rule Set

The following are the MSL (MARVEL Strategy Language) definitions of all the rules used as examples in this dissertation. All of these rules work in MARVEL and have been tested. Note that the syntax of the condition part is different from EMSL (although the underlying semantics are the same); also, the keywork "output" is not implemented yet, and thus all arguments in the activity part are considered both readable and writeable.

```
strategy ruleset

imports data_model, tools;
exports all;

rules

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

    { RCS reserve ?f.contents ?f.version }

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


edit [?h:HFILE]:
    :
    (and no_forward (?h.reservation_status = CheckedOut)
         (?h.locker = CurrentUser))

    { EDITOR edit ?h.contents }

    no_backward [?h.timestamp = CurrentTime];


edit [?c:CFILE]:
    :
    (and no_forward (?c.reservation_status = CheckedOut)
         (?c.locker = CurrentUser))

    { EDITOR edit ?c.contents }

    (and (?c.status = NotCompiled)
         no_backward (?c.timestamp = CurrentTime));


outdate_compile [?f:CFILE]:
    (exists HFILE ?h suchthat (linkto [?f.includes ?h]))
    :
    (?h.timestamp > ?f.object_timestamp)

    { }

    (?f.status = NotCompiled);
```

```
compile [?f:CFILE]:
    (forall HFILE ?h suchthat (linkto [?f.includes ?h]))
    :
    no_backward (?f.status = NotCompiled)

    { COMPILER compile ?f.contents ?h.contents "-g"
               ?f.object_code ?f.error_msg }

    (and [?f.status = Compiled]
         [?f.object_timestamp = CurrentTime]);
    [?f.status = Error];

dirty[?c:CFILE]:
    :
    no_backward (?c.status = Compiled)
    { }
    [?c.status = NotArchived];

archive [?f:CFILE]:
   (and (forall MODULE ?m suchthat (member [?m.cfiles ?f]))
        (exists LIB ?l suchthat (linkto [?m.lib ?l])))
   :
   no_backward (?f.status = NotArchived)

   { ARCHIVER archive ?f.object_code ?l.afile }

   (and [?f.status = Archived]
        (?f.timestamp = CurrentTime));
   (?f.status = Error);

archive [?m:MODULE]:
    (and (forall CFILE ?f suchthat (member [?m.cfiles ?f]))
         (forall MODULE ?q suchthat (member [?m.modules ?q])))
    :
    (and (?f.status = Archived)
         (?q.archive_status = Archived))
    { }
    (?m.archive_status = Archived);

test [?f:CFILE , ?t:TEST_SUITE]:
    :
    (?f.reservation_status = Available)

    { TESTER run_test ?t.contents ?f.object_code }

    (?f.test_status = Tested);
    (?f.test_status = Failed);


test [?mod:MODULE , ?t:TEST_SUITE]:
    (forall CFILE ?f suchthat (member [?mod.cfiles ?f]))
    :
    (?f.test_status = Tested)

    { TESTER run_test ?t.contents ?mod }

    (?mod.test_status = Tested);
    (?mod.test_status = Failed);
```

B.4. The Project Coordination Model

Following are all the control rules that were given in the dissertation. All of these control rules are parsed correctly by the CRL parser. Only the first one, however, can be executed by the current implementation; the `suspend` and `merge` actions have not been fully implemented yet.

```
edit-outdate-cr [ FILE ]

selection_criterion:
    commands: outdate-compile, edit;

bindings:
   ?t1 = requested_lock();
   ?t2 = holds_lock();

body:

   if (and (?t1.type = cfc)
           (?t2.state = pending))
   then
     {
       terminate (?t1);
     };
;


# The same control rule as above with obligations added.

edit-outdate-cr [ FILE ]

selection_criterion:
    commands: outdate-compile, edit;

bindings:
   ?t1 = requested_lock();    # the interfering transaction
   ?t2 = holds_lock();        # the other conflicting trans.
   ?s1 = session(?t1);        # ?t1's session.

body:

   if (and (?t1.type = cfc)
           (?t2.state = pending))
   then
   {
     terminate (?t1);
     add_obligation(?s1,(conflict_object.status=NotCompiled));
   };
;
```

```
lock_conflict_cr [ FILE ]

selection_criterion:

bindings:
  ?t1 = holds_lock();
  ?t2 = requested_lock();
  ?s1 = session(?t1);
  ?s2 = session(?t2);
  ?d1 = domain(?s1);

body:

   if (and (?t1.type = cfc)
           (?t2.state = pending)
           (?d1 = ?d2)
           (linkto [conflict_object, ?d1]))
   then
   {
     terminate (?t1);
     add_obligation(?s1,(conflict_object.status=NotCompiled));
   };
;


compile_edit_cr [ CFILE ]

selection_criterion:
   commands: edit, compile;

bindings:
   ?t1 = holds_lock ();
   ?t2 = requested_lock ();

body:

   if (?t1.command = edit) then
     {
       abort (?t2);
       notify (?t2,"Another user is updating the object");
     };

   if (?t2.command = edit) then
     {
       notify (?t2, "Another user is compiling the object,
                  I will let you edit it when he's done.");
       suspend(?t2, ?t1);   # suspend c2 until r1 is finished.

     };
;
```

```
# Control rule that resolves conflict between outdate-compile
# and archive rules by suspending the archive rule, which
# is waiting for backward chainig to complete, until after the
# execution of outdate-compile is completed.

compile-archive-cr [ CFILE ]

selection_criterion:
    commands: outdate-compile, archive;

bindings:
   ?t1 = requested_lock();
   ?t2 = holds_lock();

body:

   if (and (?t1.type = cfc)
           (?t1.state = active)
           (?t2.state = inactive))
   then
     {
       suspend(?t2, ?t1);
     };

   if (and (?t2.state = pending)
           (?t2.command != edit))
     {
       suspend(?t1, ?t2);
     };

;


# Control rule that resolves conflict between outdate-compile
# and archive rules by merging the two chains, if they belong
# to sessions in the same domain.

compile-archive-cr [ CFILE ]

selection_criterion:
    commands: outdate-compile, archive;

bindings:
   ?t1 = requested_lock();
   ?t2 = holds_lock();
   ?s1 = session(?t1);
   ?s2 = session(?t2)

body:

   if (and (?t1.type = cfc)
           (?t2.state = inactive)
           (?s1.domain = ?s2.domain))
   then
     {
       merge(?t1, ?t2);
     };
;
```