References

[BF81] A. Barr and E. Feigenbaum. *The Handbook of Artificial Intelligence, Volume 1.* William Kaufmann, Inc., Los Altos, CA, 1981.

[CLF88] University of Southern California, Information Sciences Institute, Marina del Rey, CA. *CLF Manual*, January 1988.

[Coh86] D. Cohen. Automatic compilation of logical specifications into efficient programs. In *Proceedings of the 5th National Conference on Artificial Intelligence*, volume Science, pages 20–25, Philadelphia, PA, August 1986. AAAI.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[For81] C. L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[Gup86] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie Mellon University, Department of Computer Science, March 1986. Technical Report CMU-CS-86-122.

[HL88] K. E. Huff and V. R. Lesser. A plan-based intelligent assistant that supports the software development process. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 97–106, Boston, MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[KFP88] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[Lyn83] N. A. Lynch. Multilevel atomicity — a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[Mir86] D. P. Miranker. *TREAT: A New and Efficient Match Algrithm for AI Production Systems.* PhD thesis, Columbia University Department of Computer Science, June 1986.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.

[MR88] N. H. Minsky and D. Rozenshtein. A software development environment for law-governed systems. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65–75, Boston MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[Per89] D. Perry, editor. *5th International Software Process Workshop*, Kennebunkport, Me, October 1989. ACM Press. In press.

[RB89] C. L. Ramsey and V. R. Basili. An evaluation of expert systems for software engineering management. *IEEE Transactions on Software Engineering*, 15(6):747–759, June 1989.

[Sho76] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* North-Holland, New York, 1976.

[SKW85] D. R. Smith, G. B. Kotik, and S. J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

5.2 Consistency-Based Conflict Resolution

The conflict resolution mechanism uses the meta-rule base to decide what to do with a conflict passed to it by the conflict detection module. For example, the conflict discovered in the example of figure 4 matches the IF part of the lock-conflict meta-rule of figure 5. In this case, the conflict resolution mechanism would check (by consulting a knowledge base that stores facts about the project and personnel) if Mary and Bob are members of the same developer team, and if they are, it will ignore the conflict; otherwise, the mechanism would reject Mary's request to `test p1`. The second meta-rule in figure 5 can be used to resolve any conflict that occurs between two rules that are subtasks of two different tasks, if the order of the two rule activities does not affect the results of the activities (i.e., if the two activities are commutative).

6 Conclusions

We considered the problem of applying production systems to the domain of software development as part of rule-based development environments. In order to scale up to real applications in this domain, it is necessary to solve the problem of synchronizing the concurrent activities of multiple developers and the rule chains fired on their behalf, collectively termed multiple agents. We formulated an approach in terms of three subproblems: conflict detection, for recognizing when agents potentially interfere with each other; conflict specification, for providing knowledge about what interactions among agents are acceptable for the particular software project; and conflict resolution, for applying this knowledge within a multi-user RBDE when conflicts are detected.

We defined the concept of tasks and described the NIL protocol for detecting potential conflicts between concurrent tasks. We outlined the idea of project-specific consistency specification, to provide the consistency knowledge for each project. Finally, we explained how a conflict resolution mechanism uses the consistency specification to resolve each conflict detected by the NIL protocol.

```
Meta-Rule lock_conflict
  IF  (and (in-progress TASK ?t1 USER ?u)
           (in-progress TASK ?t2 USER ?u2)
           (has-lock ?t1 ?obj ?lock_mode_1)
           (request-lock ?t2 ?obj ?lock_mode_2)
           (not-compatible ?lock_mode_1 ?lock_mode_2))
  THEN
      (if (and (contains ?user_group ?user1)
               (contains ?user_group ?user2))
      then
         (grant-lock ?t2 ?obj ?lock_mode_2
          ``Warning: user ?user2 also has ?obj locked'')
      else
         (reject-lock ?t2 ?obj
          ``Conflict: user ?user2 has ?obj locked''))

Meta-Rule test_conflict
  IF  (and (in-progress RULE ?r1 USER ?u1 TASK ?t1)
           (in-progress RULE ?r2 USER ?u2 TASK ?t2)
           (not-equal ?t1 ?t2)
           (not-equal ?u1 ?u2)
           (conflict ?r1 ?r2))
  THEN
      (if (can-commute ?r1.activity ?r2.activity)
       then (ignore-conflict ?r1 ?r2)
       else (if (greater (timestamp ?r1) (timestamp ?r2))
             then (abort ?r1)
             else (abort ?r2)))
```

Figure 5: Examples of Consistency Meta-Rules

5.1 Knowledge-Based Consistency Specification

In RBDEs, some inconsistency can be tolerated as a price for allowing more concurrency, and thus cooperation, between concurrent tasks. The level of tolerable inconsistency depends on the operations in a particular application. The problem is finding a framework for specifying which interleavings between concurrent tasks are allowable in a particular environment. The framework should specify the *granularity* at which different database operations (performed to either evaluate the condition of a rule or assert its effects) can be interleaved. This specification framework can then be used by the conflict resolution algorithm to provide maximum concurrency while maintaining consistency.

We define project-specific consistency in terms of sets of *meta-rules* that operate on the process model rule base. A meta-rule describes a conflict situation, and how to resolve the specific conflict. A conflict situation is described in terms of either individual rules or chains of rules. The description includes when a specific event started and what activities were in progress. Figure 5 gives two meta-rules. The code in the figure is not the actual syntax of any particular RBDE language but is intended to demonstrate the idea of project-specific consistency constraints.

4.2 The NIL Protocol for Detecting Conflicts

A conflict between concurrent tasks occurs when the firing of a rule (subtask) within one of the tasks violates the serializable execution of the in-progress tasks. An execution is said to be serializable if an equivalent serial execution can be found. The conflict detection mechanism detects a violation of serializability by using a protocol called *nested incremental locking* (NIL) based on the standard two-phase locking (2PL) protocol used in most traditional DBMSs [EGLT76].

NIL associates two locks with every object in the database, a read lock and a write lock. These locks are internal to the conflict detection mechanism and invisible to developers and the rest of the system. A conflict results from two write locks or a read and a write lock on the same object. When firing a rule and before the activity of the rule is performed, read locks must be acquired by the parent task on all the objects accessed in its condition, and write locks on all the objects accessed in its effects (these are added to the set of locks previously acquired, hence the incremental in the name of the mechanism). Conflict detection requires that all tasks be *well-formed*: (1) they do not relock objects that have been locked earlier in the task, and (2) they are divided into a growing phase, in which locks are only acquired and a shrinking phase, in which locks are only released. During the shrinking phase, a task is prohibited from acquiring locks. If a rule tries to acquire a lock that has already been acquired by another sibling rule (i.e., within the same task) in an incompatible mode, an *intra-task conflict* results. If the lock has already been acquired by a rule that is a subtask of another task, this is an *inter-task conflict*.

In the example of figure 4, when Bob requests the `test ModA` command, the RBDE starts up a task T_{Bob} and fires the corresponding `test` rule as a subtask. Before the activity of the rule is invoked, T_{Bob} acquires a read lock on the test suite S and on all the procedures contained in ModA (i.e., p1 and p2) and a write lock on the object representing ModA. Similarly, when Mary requests the command `test p1`, T_{Mary} will acquire a write lock on p1. Now when Bob's task fires `modify p1`, it tries to acquire a write lock on p1, but discovers that p1 has already been locked by T_{Bob} and T_{Mary} in an incompatible mode, thus causing both an intra-task conflict (because T_{Bob} has already acquired a write lock on p1 while Bob was testing it) and an inter-task conflict.

Intra-task conflicts are not considered serious and do not cause problems since they can be resolved by either aborting one of the conflicting rules within the same task or ignoring the conflict (e.g., in the example, it does not matter that Bob had locked p1 before). Inter-task conflicts, however, are considered serious, and once detected, they must be resolved by consulting the conflict resolution mechanism, which uses the consistency specification of the particular project to decide on how to resolve the conflict (e.g., in the example, it might be OK for both Bob and Mary to edit and test p1 at the same time because Mary might have been interested in testing a feature of p1 that Bob's modification did not change; her test would still be valid in this case). We now briefly outline the conflict resolution mechanism.

5 Resolving Conflicts

Unlike in traditional database management, detecting conflicts does not automatically mean that one of the tasks involved in the conflict has to be aborted, as demonstrated by the previous example. The conflict detection mechanism presented above detects all potential conflicts and leaves it up to the conflict resolution module to consult the specific consistency requirements of the project and decide if the detected conflict violates any of them.

Figure 4: Example of the conflict detection problem

4.1 Tasks as Conflict Detection Units

To detect conflicts similar to the one presented above, each chaining cycle that is triggered in response to a developer's command is encapsulated in a transaction-like unit called *task*. Tasks are made up of a set of rules that are fired by the RBDE automatically in response to a developer's command either directly (i.e., the rule corresponding to the command) or indirectly (i.e., through chaining) during a session. Each rule in the task is a *subtask* of the task representing the whole chain. Tasks are both interactive and incremental since their behavior is made up as rules are fired and their effects are asserted (e.g., in the example in figure 4, the RBDE had no way of guessing that Bob will discover a bug that will initiate the chain).

A task is akin to a *nested transaction*, which is a composition of a set of subtransactions, each of which can be a nested transaction [Mos85]. To other transactions, the top-level nested transaction appears as a normal atomic transaction. Internally, however, subtransactions are run concurrently and their actions are synchronized by an internal mechanism. The more important point is that a subtransaction can fail and be restarted or replaced by another subtransaction without causing the whole nested transaction to fail or restart. Similarly, a subtask (rule) within a task (rule chain) can fail and be replaced by another subtask without causing the whole chain to be invalid. This is useful for automation because the RBDE might decide to execute a rule, thinking that it is ready to be executed, only to find out that its condition is not satisfied. In this case, the rule has to be abandoned and other rules explored. Certainly, we do not want the whole automation cycle to be invalidated because of this failure.

```
(rule test (MODULE ?mod; TEST ?test)
  condition: (forall PROCEDURE ?p such that (contains ?mod ?p)
              (?p.test-status = tested))
  activity: (run ?test ?mod)
  effects: (xor (?mod.test-status = tested)
                (?mod.test-status = failed))
  comment: ``We can test a module only after all of its component procedures
            have been tested.  If the module passes the test, set its
            test-status attribute to `tested'; otherwise, to `failed'.'')

(rule test (PROCEDURE ?proc; TEST ?test)
  condition: (?proc.availability = available)
  activity: (run ?test ?proc)
  effects: (xor (?proc.test-status = tested)
                (?proc.test-status = failed))
  comment:``To test a procedure, make sure that it is.'')

(rule edit (PROCEDURE ?proc; USER ?user)
  condition: (and (?proc.availability = reserved)(?proc.reserver = ?user))
  activity: (edit ?proc)
  effects: (?proc.status = changed))

(rule reserve (OBJECT ?obj; USER ?user)
  condition: (?obj.availability = available)
  activity: (reserve ?obj ?user)
  effects: (and (?obj.availability = reserved) (?obj.reserver = ?user)))
```

Figure 3: Example Rules Based on Marvel Notation

3. To resolve a detected conflict, we develop a synchronization mechanism that uses the consistency framework to achieve a reasonable compromise between two conflicting goals:

   (a) Maximize concurrent access by multiple agents to the shared project database, and thus maximize the number of operations that can proceed concurrently. This has the effect of minimizing the environment's response time to user commands, and thus improve the overall productivity of cooperating developers.

   (b) Maintain the project database in a consistent state as specified in the consistency framework, thus reducing the time wasted on fixing up inconsistent data, and thereby facilitating the successful completion of the development tasks.

The different components of the multi-agent problem and the interactions among them are depicted in figure 2. This framework is an extension of the semantics-based consistency suggested by Garcia-Molina [GM83] and Lynch [Lyn83] for traditional database management transactions. In semantics-based consistency, the behavior of each transaction must be known before hand, and it is possible to statically define the allowed interleavings between groups of transactions. In our software development domain, it is not possible to have such knowledge *a priori* because software developers figure out what activities they will perform as they go along, and even the results of rule chains cannot be known in advance when there are multiple, mutually exclusive, effects. Thus, our tasks are interactive, incrementally constructed, and non-deterministic. The result is that the synchronization problem is more complex than for database management domains.

In the rest of the paper, we precisely formulate the conflict detection problem and develop a mechanism to detect conflicts between the actions of concurrent developers. We then outline our knowledge-based consistency specification model and the conflict resolution mechanism based on it.

4 Detecting Conflicts Between Concurrent Agents

We present the conflict detection problem by means of an example. Say that Bob and Mary want to test module ModA by running a test suite, S. ModA consists of two procedures, p1 and p2. The objects (the components of the module and the test suite) are stored in a shared database. Since all user commands (e.g., `test`, `edit`, `compile`, `format`, etc.) are implemented in terms of rules, and since RBDEs provide forward and backward chaining among these rules, each command can cause the firing of a chain of rules, each of which may access objects in the database.

When Bob and Mary request commands concurrently, Bob's command might trigger a chain of rules, one or more of which might conflict with one or more of the rules in the in-progress chain that Mary's command has triggered. Alternatively, the conflict might have occurred even in a single-user context if, for example, Bob had requested two commands concurrently (e.g., in two different windows). Say that Bob requested a command `test modA` (corresponding to the test rule shown in figure 3) at time `t1`, which triggered the forward chaining cycle shown in figure 4. Mary requests a command `test p1` (corresponding to the second test rule in figure 3) at time `t2`. The condition of the rule is satisfied at that time, causing the activity of the rule to be invoked. In the meanwhile Bob discovers that p1 has a bug so he starts debugging it by initiating the command `modify p1` at time `t3`, which triggers a backward chain to `reserve p1` before calling the editor on p1. From the sequence of activities depicted in figure 4, it should be clear that a conflict results because the effects of the `reserve` rule that Bob's command chains to will negate the condition of Mary's `test` rule, which is already in-progress. This causes Mary's test to be invalid since the procedure she is testing has already been modified.

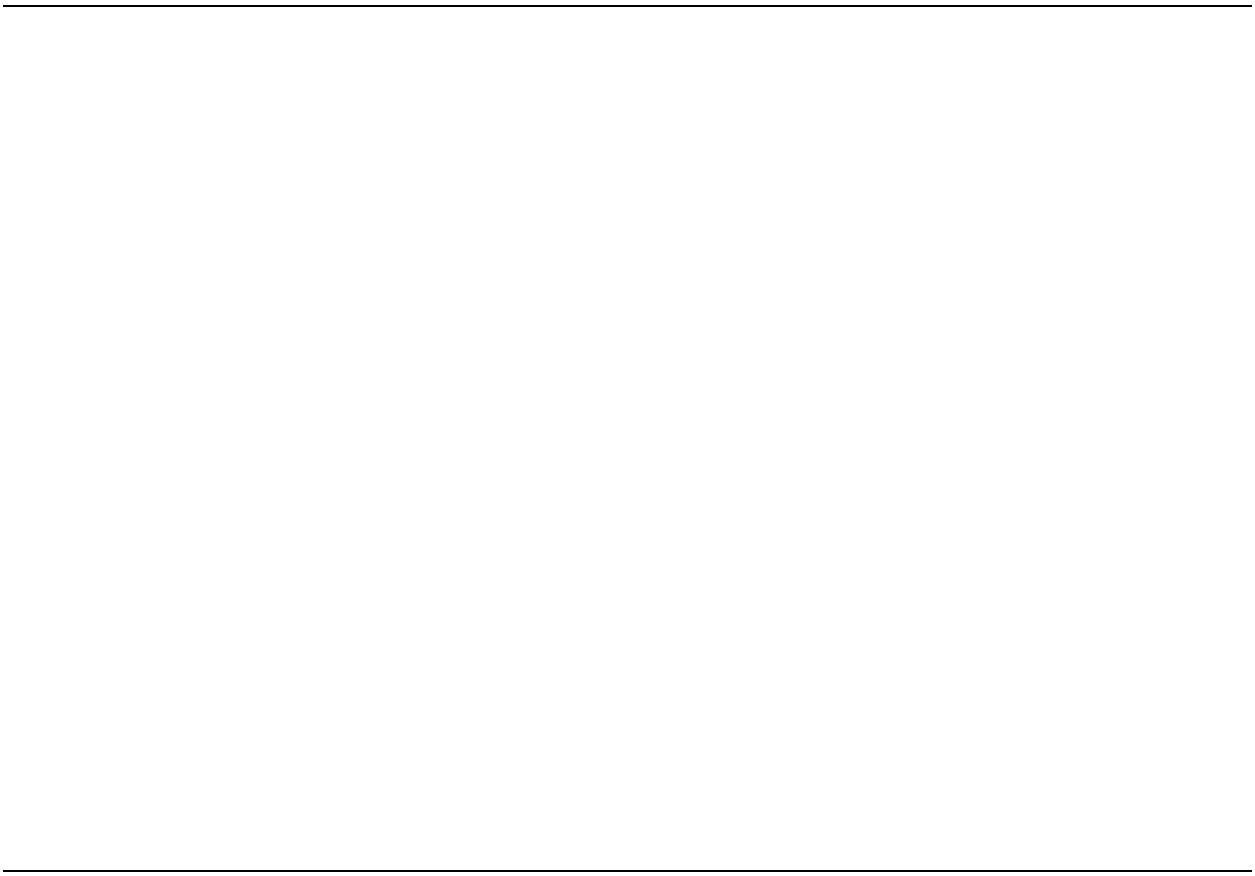

Figure 2: The Multi-Agent problem in RBDEs

Our conjecture is that an RBDE can provide a cooperative environment if it is provided with knowledge about what it means for the data of a specific project to be in a consistent state, and about the semantics of operations performed by agents on the database. These two pieces of information are different for different projects or different phases of the project. For example, the consistency specification of a small project with a couple of developers who are familiar with all the components of the project might permit those developers to access the same object at the same time. The consistency specification of a large project with hundreds of developers, however, might require strict isolation between different groups of developers, and might not allow interaction except through a strictly-defined interface.

Because of the different meaning of consistency for different projects, it is necessary for the project administrator to provide the RBDE with the consistency specification of the project rather than building-in a specific meaning of consistency that might not apply to all projects. Given a consistency specification and a description of the development process, an RBDE could (1) automate part of the development process by automatically performing some activities in response to changes made to objects in the database by other activities, and (2) maintain consistency by invoking activities that would restore consistency of the database in case it was violated.

3.1 The Multi-Agent Problem

The overall behavior of cooperating developers in an RBDE can be modeled as concurrent agents, each of which fires multiple rules that may execute concurrently to perform operations on the shared project database. Based on this, the synchronization problem can be divided into three subproblems:

- **Conflict detection**: deciding whether or not two or more rules can logically be fired within concurrent chaining cycles (i.e., making sure that the conditions and effects of these rules do not negate each other).
- **Consistency specification**: specifying what kind of consistency has to be maintained in the database of the particular project, and what kind of concurrency is allowable between agents.
- **Conflict resolution**: once a conflict has been detected, the RBDE has to decide how to resolve it. The conflict resolution strategy depends on the consistency specification in that if the conflict does violate the specified consistency, then it has to be resolved, but if the detected conflict does not violate the consistency specification, it can be ignored.

We use the following approach to formulate a solution to these problems:

1. To solve the conflict detection problem, we encapsulate rule chains (i.e., all the rules that are triggered by both the developers and the RBDE on their behalf) in logical units, called *tasks*, which resemble transactions in database systems[1]. Then, we use a conflict detection protocol based on the concept of serializability to detect conflicts between tasks.
2. To solve the consistency specification problem, we have designed a framework in which it is possible to: (1) define exactly what is meant by maintaining a specific project in a consistent state; and (2) specify allowable levels of concurrency between different tasks.

[1]The similarities and differences between tasks and transactions are clarified later.

4. **Recognize** a specific software development process, so that it can assist the developer in planning what to do next.

Some existing RBDEs distinguish between automation rules and consistency rules (e.g., CLF), some provide only consistency rules (e.g., Darwin), while others provide only automation rules (e.g., Marvel). We assume a general model in which: (1) both forward and backward chaining are supported; and (2) rules may serve the purpose of either automation or consistency maintenance. Monitoring and plan recognition are not addressed.

3 Multi-Agent RBDEs

When multiple developers cooperate on a project within an RBDE, they share a common database that contains all the objects of the project. These developers start concurrent sessions in order to complete their specific assignments. During their sessions, the developers concurrently request operations that access objects in the shared project database. There is a need to synchronize the concurrent activities because they might introduce *conflicts* that violate the consistency of the objects they access (e.g., if they concurrently change either the same attribute or dependent attributes of the same object, or of related objects).

Synchronization in an RBDE does not involve only the human developers but also rules that automatically perform operations on their behalf, since those rules also access the shared database. The term *agent* is used in this paper to refer to both human developers and rules that are fired automatically by the RBDE on their behalf. What is lacking in existing RBDEs is the ability to synchronize concurrent accesses by multiple agents to shared data while still providing an environment that supports cooperation among its users (called *cooperative environment* hereafter).

The problem of concurrent rule executions has been addressed previously, for the purpose of speeding up the execution of production systems through the use of parallelism. However, both Gupta [Gup86] and Miranker [Mir86] have concluded that the speed up that can be expected from parallelism is quite limited in the context of OPS5 production systems that are implemented using the Rete algorithm [For82]. We address a different problem that sounds superficially similar. In our model, parallelism is *intrinsic* in the application domain and not a technique used for speeding up the performance of the expert system.

The problem of concurrency has also been addressed in traditional database management domains such as banking. In these domains, there is a lack of knowledge about the application-specific semantics of database operations, and a need to design general mechanisms that cut across many potential applications. Thus, the best a database management system (DBMS) can do is to abstract all operations on a database to read and write operations. All computations are then programmed into *transactions* that consist of a sequence of read and write operations. Each transaction, if executed atomically (i.e., either all of its operations are performed in order or none are), transforms the database from one consistent state to another. When multiple transactions run concurrently, the DBMS can guarantee that the database is transformed to a consistent state with respect to reads and writes by allowing only executions of the concurrent transactions that are equivalent to a serial execution. This correctness criterion is referred to as *serializability* [EGLT76].

Existing multi-user RBDEs use this concept to synchronize concurrent agents by isolating these agents, so that one agent can see only the final results of another agent's work after it has been completed, and cannot share data and/or knowledge with other concurrent agents. Isolation guarantees serializability, and thus strict maintenance of consistency, since it makes each agent's work appear as an atomic transaction. Unfortunately, isolation between concurrent agents unnecessarily obstructs cooperation.

```
(defAutomation compile-module
    ((module) s.t. (start (and (not (mod-status module :compiled)
                               (A (proc)
                                  (implies (contains module proc)
                                  (proc-status proc :analyzed)))))))
    (lambda (module) (compile-mod module))

    documentation: ``If all the procedures contained in a module have
                     been analyzed, and if the module has not been
                     compiled, the system should go ahead and
                     run the compile-mod activity.'')
```

Figure 1: Example of CLF Automation Rule

opment activity the rule encapsulates, is executed. The condition is essentially a query (i.e., read operations) on the status of objects in the project database. The activity is modeled as a black box whose inputs and outputs are known, but in order to determine which of several possible outputs it will produce, the black box must be executed. The third part is a set of mutually exclusive *effects*, each of which changes (i.e., writes) the values of object attributes in the database. Which effect to assert depends on the results of the rule's activity.

If the effects of a rule change the objects in the database in such a way that the conditions of other rules become satisfied, those rules are fired automatically. This behavior is termed *forward chaining* and is implemented in OPS5 production systems [For81]. Alternatively, if a condition of a rule is not satisfied, *backward chaining* is performed to satisfy it. The backward chaining model is implemented in theorem provers, constraint systems, and some production systems [Sho76].

RBDEs provide either a forward chaining model or a backward chaining model or both in order to do one or more of the following:

1. **Maintain consistency** as defined by the conditions and effects of rules. Thus, no development activity will be performed until the condition of the rule encapsulating it is satisfied. Once the effects of the activity are asserted in the database, all implications (in terms of performing other activities) are taken care of.

2. **Automate** the performance of some activities that would otherwise be performed manually. Thus, 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 the RBDE on behalf of the developer who requested the first activity. Note that maintaining consistency as above is treated as mandatory: The system automatically performs some activities in order to ensure that the database is in a consistent state; if it cannot do any of the activities implied by the effects of the original rule, the original activity cannot be allowed. Automation on the other hand is opportunistic, i.e., the system automatically performs as many operations as it can so that the developer won't have to worry about them later on; not performing these operations does not affect the validity of the original operation.

3. **Monitor** the actual development process and collect information about any violations of the prescribed development process. The RBDE would only warn developers about these violations rather than enforce their prevention.

1 Introduction and Motivation

Production systems have been applied to domains such as medical diagnosis, mathematical discovery, and hardware configuration [BF81]. Recently researchers have explored the application of production systems to the domain of managing software development efforts, i.e., monitoring the development process and/or enforcing a specific model of development [RB89]. This is distinct from applying production systems techniques to automatic programming and algorithm construction, for automatically generating programs and/or algorithms from specifications.

Rule-based development environments (RBDEs) assist in software development by *enacting* the development process modeled in terms of rules. Enacting a process model involves reasoning about the rules and automatically firing rules as appropriate in order to perform development and data management chores that would otherwise be done manually. Although such integration of rule-based process modeling with database capabilities has recently gained popularity [Per89], existing RBDEs do not scale up to "real" projects.

Personnel in real projects typically work concurrently on separate components, but often share knowledge about project status and overall progress as well as data such as software libraries. This data is highly structured and stored in a common project database. It is necessary to maintain the consistency of the project database, while not obstructing cooperation and sharing of information among developers. Existing RBDEs do not scale up, since most are either single-user environments that do not allow concurrency or multi-user environments that guarantee consistency by isolating the developers, thus preventing cooperation by preventing conflicting accesses.

We investigate the scaling up of RBDEs to support cooperation among multiple developers. Our approach is to extend the RBDE model with a conflict resolution mechanism that maintains data consistency while permitting cooperative concurrent access to the data. The resulting RBDE model can provide more powerful assistance in the management of software development projects. Our mechanism consists of three modules: conflict detection, conflict resolution and consistency specification rule base. In this paper we concentrate on the conflict detection module, and only sketch the other two.

2 The Domain: The RBDE Model

Several well-known software development environments have applied rule-based technology to the domain of software development. The CommonLisp Framework (CLF) [CLF88] supports rule-based modeling of the software development process through consistency and automation rules [Coh86]; a sample CLF rule is shown in figure 1. Refine [SKW85], an automatic transformation system for the purpose of program synthesis, also provides a limited form of automation in the style of CLF. Darwin [MR88] restricts what programmers can do by treating rules as constraints and automating checking and enforcement of these constraints. Grapple [HL88] uses rules to do planning and plan recognition in order to monitor a user-specified process model. Our own system, Marvel [KFP88], enacts the process model of a specific project by providing controlled automation.

We assume a general RBDE model that enacts the development process of a particular project by automatically firing user-specified rules that encapsulate development activities. These activities manipulate the different components of the project (e.g., the source code, the test suites, the documentation, etc.), which are stored in a common project database. Each component is abstracted as an object with a set of attributes, the values of which reflect the object's status. The user-level commands either correspond to rules or are built into the environment, e.g., commands to add and delete objects. Each rule has a *condition* that must be satisfied before the second part, the devel-

Detecting Conflicts in Multi–Agent Rule-Based Development Environments

Naser S. Barghouti and Gail E. Kaiser
Department of Computer Science, Columbia University
New York, NY 10027
(212) 854-8182, naser@cs.columbia.edu

March 5, 1990

Topic: Automated Reasoning

Subtopics: Knowledge Acquisition and Expert System Design Methodologies, Knowledge Representation, Distributed Artificial Intelligence

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

Abstract

Researchers have been exploring the possibility of applying production systems to the domain of software development. One resulting approach is rule-based development environments (RBDEs), which provide expert assistance to developers working on large-scale projects. RBDEs model the development process in terms of rules, and then enact this model by automatically firing rules at the appropriate time to carry out chores that the developer would have done manually. In order to realistically model the domain of software development, RBDEs must support the cooperation among multiple developers, each of whom selects commands, causing the firing of multiple rules (either directly or via chaining) that concurrently access shared information. In this domain, rule parallelism is *intrinsic* and not *opportunistic* (i.e., not for the purpose of speeding up the production system). One of the problems resulting from executing rule chains concurrently causes is how to detect conflicts that occur between the chains. This paper presents the multi-agent RBDE model, explores the conflict detection problem, and suggests an approach to solving it. The problem of resolving conflicts without unnecessarily obstructing cooperation among developers is only briefly discussed.

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