[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 388–401, Austin TX, May 1985.

[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.

[McD82] D. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, September 1982.

[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.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Databases*, pages 26–37, Los Angeles CA, August 1988. Morgan Kaufmann.

[SGMA87] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic locking: A strategy for coping with long lived transactions. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 19.1 – 19.24, Pacific Grove CA, September 1987.

[vB89] M. van Biema. *The Constraint-Based Paradigm.* PhD thesis, Columbia University Department of Computer Science, November 1989.

[WA86] D. Wile and D. Allard. Worlds: An organizing structure for object-bases. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 16–26. ACM Press, December 1986. Published as *Sigplan Notices* in January 1987.

[Wei88] W. Weihl. Commutativity-based concurrency control for abstract data types (preliminary report). In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume II Software Track, pages 205–214, Kona HI, January 1988.

References

[AT88] A. Abbadi and S. Toueg. The group paradigm for concurrency control protocols. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 126–134, Chicago, IL, June 1988. ACM Press.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading MA, 1987.

[BKK85] F. Bancilhon, W. Kim, and H. Korth. A model of cad transactions. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 25–33, Stockholm Sweden, August 1985. Morgan Kaufmann.

[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.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. Traiger, editors, *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 249–259, San Francisco CA, May 1987. ACM Press. Special issue of @iSIGMOD Record, 16(3), December 1987.

[Gre88] I. Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings.* Morgan Kaufman, San Mateo CA, 1988.

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

[Hit88] P. Hitchcock. The process model of the aspect ipse. In *Proceedings of the 4th International Software Process Workshop*, pages 76–78, Devon UK, May 1988.

[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.

[Hon88] M. Honda. Support for parallel development in the sun network software environment. In *Proceedings of the 2nd International Workshop on Computer-Aided Software Engineering*, pages 5–5–5–7, 1988.

[Kai90] G. E. Kaiser. A flexible transaction model for software engineering. In *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles CA, February 1990. IEEE Computer Society Press.

[Kat89] T. Katamaya. A hierarchical and functional software process description and its enaction. In *Proceedings of the 11th International Conference on Software Engineering*, pages 343–352, 1989.

[KBFS88] G. E. Kaiser, N. S. Barghouti, P. H. Feiler, and R. W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 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.

[KL89] W. Kim and F. H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York NY, 1989.

Box B: Supporting Cooperative Work

Researchers from both the database community and the software development environments community have addressed the issue of cooperation among multiple developers in software projects of various sizes. In a relatively small project, in which developers work independently most of the time on the parts of the project for which they are responsible, they need to interact only at specific points to integrate their work. In a larger project, which involves a large group of developers divided into several groups, each responsible for a part of the development task, members of each group usually cooperate to complete their part. In this case, there is a need to support cooperation among members of the same group, as well as coordination between multiple groups. Thus, isolating transactions that encapsulate individual developers' tasks not only constrains concurrency, but it also prevents desired cooperation among developers and prevents transformation of the project database to a new consistent state if that transformation requires concerted actions by multiple developers. Following are some of the mechanisms and concepts that address the issue of cooperation among concurrent transactions.

Semantics-Based Consistency: Mechanisms such as altruistic locking [SGMA87], commutativity-based concurrency control [Wei88], and sagas [GMS87] use application-specific semantics to extend traditional serializability-based approach. They assume the availability of specific semantic information, such as information about access patterns of transactions, which might not be appropriate for all applications. They also assume programmed transactions and do static analysis using the semantic information. Transaction mechanisms for multi-agent RBDEs cannot assume either of these two things.

Dynamic Restructuring: One concept that has been proposed by Pu *et. al.* [PKH88] is that of dynamically restructuring transactions as more information about the usage of resources becomes available.

Coordinated Access: Another approach that has been used to coordinate access to shared data is an extension of the reserve/deposit mechanism, which works on collections of files by providing a multilevel database hierarchy capable of providing more concurrency [Hon88].

User Groups: The concept of user groups [AT88] has been used to implement concurrency control policies that support synergistic cooperation [KSUW85, Kai90].

Cooperative Transactions: The CAD group at MCC proposed a cooperative transaction mechanism that treats all the operations that are performed within the long transactions of developers in the same team as if they were requested by the same transaction [BKK85]. Thus, the mechanism would only insist that the leaf operations (not the nested transactions) be serializable.

```
build[?p:PROGRAM]:

    # if all the libraries belonging to the project (instance of the
    # PROJECT class) that contains the program have been archived,
    # and if all the C files (instances of the class CFILE)
    # that are components of the program have been successfully
    # analyzed and compiled, then the program can be built by calling
    # the BUILD tool.  Depending on the result of the tool, update the
    # status attribute of the program to be either Built or NotBuilt.

    (and (forall PROJECT ?P suchthat (member [?P.programs ?p]))
         (forall LIB ?l suchthat (member [?P.libraries ?l]))
         (forall CFILE ?c suchthat (member [?p.cfiles ?c])))
    :
    (and (?c.analyze_status = Analyzed)
         (?c.compile_status = Compiled)
         (?l.archive_status = Archived))

    { BUILD build_program ?p }    # this is the activity of the rule

    (?p.build_status = Built);    # postcondition in case of success
    (?p.build_status = NotBuilt); # postcondition in case of failure
```

Figure 9: Example of a Marvel Rule

```
(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 8: Example of CLF Automation Rule

Box A: Rule-Based Development Environments

There are several well-known RBDEs. 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 8. 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. An example Marvel rule is shown in figure 9. Aspect [Hit88] defines software development activities in terms of rules that have preconditions and postconditions similar to Marvel rules, and uses the rules to monitor and control the development process. HFSP [Kat89] provides a mechanism for enacting a process model and redoing parts of it if the original enaction led to unsatisfactory results.

All of the environments mentioned are single-user systems. One of the RBDE models that supports multiple users is the Worlds model which extends the CLF environement [WA86] by allowing multiple users to access copies of the project objectbase. These copies must be merged in order to produce one consistent copy of the objectbase.

group, who are understood to be cooperating closely, and if they are, it will ignore the conflict; otherwise, the mechanism would reject Mary's request to `test p1`. The second metarule in figure 7 can be used to resolve a conflict that occurs between a rule just being initiated by a user and the in-progress rule in another task initiated by another user. The conflict situation of the metarule depicts this situation, while the repair part checks if the two rules belong to any instance of the `COMMUTATIVITY-TROUPE`, i.e., if the order of execution of the rules does not affect the results of the tasks.

7 Conclusions

We considered the problem of applying expert systems to the domain of software development in the form 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 based on exploiting features of object-oriented programming to gain knowledge about the structure of the data that the multiple agents access and the behavior of the operations through which they access the data. Object-oriented data modeling defines the structure and behavior of data objects. Structural aspects of an objects are defined in terms of the types of its attributes, while the behavioral aspects are defined in terms of methods that prescribe how the object should behave when it receives a message. Existing RBDEs exploit only the structural aspects of object-oriented data modeling. We introduced an extension to the RBDE model that integrates both the structural and behavioral aspects of data modeling with the rule-based process modeling by unifying rules and methods.

We also introduced the concept of troupes of rules that extends the concept of collections in object-oriented programming. Membership in a troupe, unlike membership in a collection, is not decided on the basis of a single property that each member possesses, but on a property that all members collectively possess. Given the extensions to the RBDE model and to object-oriented data modeling, we defined the multi-agent problem in terms of three subproblems: conflict detection, for recognizing when agents potentially interfere with each other; consistency 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.

Acknowledgments

This article is an expansion of a paper that was submitted to AAAI-90; that paper discusses RBDEs and presents the notions of conflict detection, consistency specification and conflict resolution, but does not consider our object-oriented framework, including the treatment of rules as methods and the use of collections of objects and troupes of rules. Michael Sokolsky collaborated with us on designing and implementing Marvel and the object-oriented framework for RBDEs. Israel Ben-Shaul and George Heineman helped us in implementing polymorphism of rules and designing consistency rules for Marvel, respectively.

```
(MetaRule lock_conflict
  IF  (and (in-progress TASK ?t1 USER ?u1)
           (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 (exists USER_GROUP ?g)
               (contains ?g ?u1)
               (contains ?g ?u2))
       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'' ))

  COMMENT: ``Two users who are members of the same developer team
            (where USER_GROUP is a collection of objects, each of
            which represents a user) can share an object, while other
            users who are not members of the same team will not be
            granted permission to lock the object.'')

(MetaRule test_conflict
  IF  (and (in-progress RULE ?r1 USER ?u1 TASK ?t1)
           (start RULE ?r2 USER ?u2 TASK ?t2)
           (not-equal ?t1 ?t2)
           (not-equal ?u1 ?u2)
           (conflict ?r1 ?r2))
  THEN
      (if (exists COMMUTATIVITY-TROUPE ?t such_that
                      (and (member ?r1 ?t)
                           (member ?r2 ?t)))
       then (ignore-conflict ?r1 ?r2)
       else (abort ?r2))

  COMMENT: ``If the activity of the rule whose initiation caused the
            detection of conflict can commute with the in-progress
            rule with which it conflicted, then ignore the potential
            conflict since execution order does not matter.
            Otherwise, abort the later rule.'')
```

Figure 7: Examples of Consistency Metarules

```
COMMUTATIVITY-TROUPE:
    r1, r2, r3, r4;
    r1, r5, r6;
    . . .
```

Figure 6: Example of a Troupe of Rules

mechanism to consult the specific consistency requirements of the project and decide if any of them are violated by the detected conflict.

6.1 Troupes of Rules for Consistency Specification

In RBDEs, non-serializable access to data permits more concurrency, and thus cooperation, between tasks. The situations in which such non-serializable access violates the consistency of the project depends on the objectbase operations implied by the rules in the particular project. The problem is finding a framework for specifying which interleavings between concurrent tasks are allowable (i.e., do not violate consistency). The framework should specify the *granularity* at which different objectbase 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 *metarules* that operate on the process model rule base. A metarule has a condition that describes a conflict situation, and a repair that prescribes how to resolve the specific conflict. The conflict situation is described using two concepts: collections of objects and troupes of rules. Using collections of objects, a metarule can refer to all members of a teams of developers (each developer is an object and a team is a collection of objects that shares the same responsibility), all objects that are accessed by a task, etc.

Troupes define sets of rules that share a particular semantic property; these sets are specified in advance, along with the data model and process model. Each set is considered an instance of the troupe in which it is defined. Troupes are like classes in that they prescribe methods, or actually one specific method, in the form of a metarule. It might be interesting to further extend the concept of class to troupes by introducing attributes of troupes and inheritance among troupes, but here we consider only troupes that define maximal sets of rules sharing a mutual property. For example, the `COMMUTATIVITY-TROUPE` defined in figure 6 defines several sets of rules that can all commute with each other. In other words, it does not matter in which order we run the rules that are members of an instance of this troupe. Both collections of objects and troupes of rules provide semantic knowledge that can be used by metarules to resolve conflicts. Figure 7 gives two metarules. The notation 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.

6.2 Consistency-Based Conflict Resolution

The conflict resolution mechanism uses the metarule base to decide what to do with a conflict passed to it by the conflict detection mechanism. For example, the conflict discovered in the example of figure 4 matches the IF part of the lock-conflict metarule of figure 7. In this case, the conflict resolution mechanism would check whether Mary and Bob are members of the same user

lock in any case, but multiple rules can simultaneously hold the write lock only if this is permitted by the conflict resolution mechanism, described later. A conflict is detected even when the two rules are part of the same task, thus the protocol is *nested*.

The objects accessed by the condition of a rule are determined by the logical clause of the condition, which we call the *characteristic function*. This function describes a *collection* of objects, each of which possess the property that characterizes the collection. The collection is thus a dynamically defined set whose members change when the values of their attributes change. An example of a collection is the set of all objects that are both instances of the `PROCEDURE` class and contained in module `ModA`. This collection is used in the `test` rule of figure 5. The effects of a rule only write the values of the attributes of objects passed to the rule as actual parameters.

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. This is done by acquiring locks as the rule chain grows, and not releasing any locks until the chain terminates. 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*. The rule does not wait for a lock when there is a conflict, as might be done in a conventional database management system, but instead the conflict resolution mechanism is invoked; thus there is no possibility of deadlock, where two rule chains would be waiting for each other to release their locks.

In the example of figure 4, when Bob requests the `test ModA` command, the RBDE initiates 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} acquires 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 unless another rule that holds the lock is actually executing at the same time (as opposed to having completed previously; remember that the lock set by a subtask is held until the end of the task). In this case the first rule releases all its locks (to avoid deadlock) and waits until the second rule has finished before attempting to reacquire locks (this could result in starvation or livelock, but we ignore that here).

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 allowable for Bob and Mary to edit and test p1 at the same time because they are cooperating together closely, and Mary knows that she is testing a feature of p1 that Bob's modification will not change; her test would still be valid in this case). We now briefly outline the conflict resolution mechanism.

6 Conflict Resolution

Unlike in traditional database management, detecting violations to serializable access to objects does not automatically mean that the consistency of the database has been violated, as demonstrated by the previous example. The conflict detection mechanism presented above detects all potential conflicts (i.e., non-serializable access to objects) and leaves it to the conflict resolution

figure 5) at time `t1`, which triggers the forward chaining cycle shown in figure 4. Mary requests a command `test p1` (corresponding to the second test rule in figure 5) at time `t2`. The condition of the rule is satisfied at that time, causing the activity of the rule to be invoked. Meanwhile, Bob discovers that p1 has a bug so he issues the command `modify p1` at time `t3`, which triggers a backward chain to `reserve p1` before calling the editor on p1. The sequence of activities depicted in figure 4 causes a conflict because the effects of the `reserve` rule in Bob's chain will negate the condition of Mary's `test` rule, already in-progress. This causes Mary's test to be invalid since the procedure she is testing has already been modified.

5.1 Tasks as Conflict Detection Units

To detect conflicts similar to the one presented above, each chaining cycle triggered in response to a developer's command is encapsulated in a task. Each rule in the task is a *subtask* of the task representing the whole chain. A task is akin to a *nested transaction*, which is a composition of a set of subtransactions [Mos85]. To other transactions, the top-level nested transaction appears as a normal atomic transaction. Internally, however, subtransactions are executed concurrently and their actions are synchronized by an internal mechanism. A subtransaction can fail and be restarted or replaced by another subtransaction without causing the whole nested transaction to fail or restart. Similarly, a task consists of a partially ordered chain of rules (subtasks); a subtask 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 fire a rule, only to find out that its condition cannot be satisfied; in this case, the rule has to be abandoned and other rules explored. Certainly, we do not want the whole automation cycle to be invalidated because of such failures.

In this article, user sessions are treated as *sagas* [GMS87] in that there is no synchronization at all among sessions as a whole, only among their component tasks. This would not be acceptable for many software processes that might be supported by RBDEs, where a user session should take the project objectbase from one semantically consistent state to another and other users should not be able to view the intermediate inconsistent states of the objects manipulated during the session. We plan to apply the same approach described here for tasks to whole sessions, with metarules specifying the consistency requirements among sessions based on knowledge of the purposes of the sessions and the behavior of the tasks. Such metarules would necessarily be much more complex than those presented here. We do not discuss this issue further in this article, but discuss only the basic concurrency control among tasks rather than sessions.

5.2 Collections of Objects as a Basis 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 *nested incremental locking* (NIL) protocol based on the standard two-phase locking protocol used in most traditional DBMSs [EGLT76].

NIL associates two locks with every object in the objectbase, 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. When initiating 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 protocol is *incremental*). A conflict is detected when a rule attempts to set a read or write lock when another rule already holds the write lock; multiple rules can simultaneously hold the read

```
(rule test (MODULE ?mod; TEST ?test)
  condition: (forall PROCEDURE ?p suchthat (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:``A procedure can be tested only if it is available; i.e.,
           it is not reserved for editing by any user.'')

(rule edit (PROCEDURE ?proc; USER ?user)
  condition: (and (?proc.availability = reserved)(?proc.reserver = ?user))
  activity: (edit ?proc)
  effects: (?proc.status = changed)
  comments: ``We can edit a procedure only if it has been reserved by
             the same user who initiated the edit rule.'')

(rule reserve (OBJECT ?obj; USER ?user)
  condition: (?obj.availability = available)
  activity: (reserve ?obj ?user)
  effects: (and (?obj.availability = reserved) (?obj.reserver = ?user))
  comments: ``A user can reserve an object if it is available in the
             sense that no other user has reserved it.'')
```

Figure 5: Example Rules Based on Marvel Notation

Figure 4: Example of the Conflict Detection Problem

The different components of the multi-agent problem and the interactions among them are depicted in figure 3. This framework is an extension of the semantics-based consistency suggested by Lynch [Lyn83] for traditional database management transactions. In semantics-based consistency, the behavior of each transaction must be known beforehand, and it must be 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 decide what activities they will perform as they go along in their sessions, and even the sequence of rules in a task cannot be known in advance when rules have multiple possible effects. Thus, both sessions and tasks are incrementally constructed and non-deterministic, and our synchronization problem is more complex than for database management domains.

In the rest of this article, we formulate the conflict detection problem in terms of the underlying object-oriented data model and develop a mechanism to detect conflicts between rule chains initiated by concurrent developers. We then outline an extension to OOP for our knowledge-based consistency specification model and the corresponding conflict resolution mechanism.

5 Detecting Conflicts Between Concurrent Agents

We present the conflict detection problem by means of an example. Suppose that Bob and Mary want to test module ModA by running a test suite, S. ModA consists of two procedures, p1 and p2. 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.

Suppose that Bob requests a command `test modA` (corresponding to the test rule shown in

Figure 3: The Multi-Agent Problem in RBDEs

1. To solve the conflict detection problem, we encapsulate rule chains (i.e., all the rules that are triggered by a user command and the RBDE's consequent backward and forward chaining on the user's behalf) in logical units, called *tasks*, which resemble transactions in database systems.[3] Then, we use a conflict detection protocol based on the concept of serializability to detect potential conflicts between tasks. The protocol uses the collections of objects accessed by concurrent tasks as a basis for detecting conflicts.

2. To solve the consistency specification problem, we have designed an object-oriented framework in which it is possible to: (1) define conflict situations using the concept of troupes of rules; and (2) prescribe how the conflict situation should be resolved.

3. To resolve a detected conflict, we have developed 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 objectbase, 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 improves the overall productivity of cooperating developers.

   (b) Maintain the project objectbase in a consistent state as specified in the consistency framework, thus reducing the time wasted on repairing inconsistent data, and thereby facilitating the successful completion of the software project.

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

(late) binding and polymorphism to the rule system. Each rule has a set of typed formal parameters that are instantiated with objects of the same types (i.e., classes) as the corresponding formal parameters at the time the rule is fired. The rule's condition is a logical clause that is essentially a read-only query on the values of the attributes of the parameter objects, while the effects change the values of these objects' attributes. Thus, each rule is equivalent to a multi-method that applies to the classes that are the types of the formal parameters; like methods, rules can be inherited. If a rule has a formal parameter of class `A`, the objects that are passed as actual parameters can be instances of either `A` or any of its subclasses.

Like multi-methods, rules are identified through the unique combination of their name and the types, order and number of their parameters. Consequently, when a user requests to execute a command named `R` on a set of objects, the RBDE searches for the rule also named `R` that requires as parameters the number and types (classes) of objects indicated by the user. The same mechanisms used for method lookup in OOP are employed for this search, and the closest match is fired with objects bound as the actual parameters of the rule.

For example, assume there are four classes `A`, `B`, `C` and `D` and four rules, all of which have the name `R` but whose formal parameters are of different types, as shown in figure 2. Class `C` is a subclass of both `A` and `B`; that means that it inherits all the attributes and methods of both classes. Class `D` inherits only from class `A`. According to the inheritance hierarchy, the first rule applies directly to instances of class `A`, and indirectly (through inheritance) to instances of `C` and `D`, but since the third rule applies directly to instances of `C`, it overrides the first rule with respect to class `C`. This means that if a user requested command `R` on an instance of `C`, the third rather than the first rule will be fired. The second rule applies only to instances of `B`. The fourth rule needs two parameters, the first of which can be an instance of `A`, `C` or `D`, and the second an instance of either `B` or `C`.

4.2 The Multi-Agent Problem in Object-Oriented Terms

Given the unified model above, the overall behavior of cooperating developers in an RBDE can be modeled as concurrent requests by users to execute environment commands, each of which causes the firing of a rule chain; these rule chains then execute concurrently, performing operations on objects in the shared project objectbase. Each rule firing can be thought of as sending a message to an object, or a collection of objects for a rule with multiple arguments, resulting in reads and writes of the objects' attributes by the rule's condition and effect. Based on this, the synchronization problem can be divided into three subproblems:

- **Conflict detection**: before firing a rule within an in-progress rule chain, the RBDE must decide whether or not the rule could cause access conflicts with any other rule that has already been fired in the same chain or concurrent chains. If it does, a potential conflict is detected and must be resolved before invoking the rule's activity.

- **Consistency specification**: specifying the semantic consistency that must be maintained in the objectbase of the particular project.

- **Conflict resolution**: once a potential conflict situation has been detected, the RBDE must resolve it according to the consistency specification. If the conflict situation violates the specified consistency, then it must be prevented somehow; otherwise, it can be ignored.

We have taken an OOP approach to formulating our solution to these problems:

```
CLASS A: superclass Object;
   . . .

CLASS B: superclass Object;
   . . .

CLASS C: superclass A, B;
   . . .

CLASS D: superclass A;
   . . .

RULE R [?a:A];
   . . .

RULE R [?b:B];
   . . .

RULE R [?c:C];
   . . .

RULE R [?a:A; ?b:B];
   . . .
```

Figure 2: Inheritance and Polymorphism of Rules

reflect the object's content, status and relations to other objects. Operations on data are modeled in terms of separate rules that treat objects as passive pieces of data. Thus, rather than providing a uniform and integrated model of data structure and the operations that can be performed on data, existing RBDEs separate the two issues, foregoing the expressive power that can be gained by incorporating OOP features, such as multiple inheritance and overriding of rules and late binding between rules and their parameter objects as rules are fired. One of our primary contributions is extending the RBDE model by integrating the behavioral and structural aspects of data modeling with the rule-based process model.

4.1 Treating Rules as Methods in Single-Agent RBDEs

In OOP, behavioral aspects of each class are defined by methods that prescribe the operations that can be performed on instances of each class. Each method applies only to instances of the class in which it was defined, and is invoked by sending a message to an object indicating the name of the method. Multi-methods are sent to a collection of objects of perhaps different classes. Thus, while attributes describe the structure of objects, methods describe how they behave. Methods, like attributes, can be inherited from a class to its subclasses.

The extended RBDE model unifies the behavioral aspects of data modeling with the process model by treating the rules that manipulate objects as methods, and extending inheritance, dynamic

invoked activities might take an arbitrary amount of time and it would not be possible to cooperate by sharing knowledge during the chaining.

Concurrent rule executions has been addressed previously for the purpose of speeding up the execution of production systems through the use of parallelism. Researchers have concluded that the speedup from parallelism is quite limited in the context of OPS5 production systems that are implemented using the Rete algorithm. (For a detailed analysis of parallelism in production systems, see [Gup86].) We address a different problem that appears superficially similar. In multi-agent RBDEs, parallelism is *intrinsic* and not a technique used for improving 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, at best a database manager can abstract all operations to reads and writes. All computations are 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), is assumed to transform the database from one consistent state to another. When multiple transactions run concurrently, the system guarantees that the database is transformed to a consistent state by allowing only executions that are equivalent to a serial execution. This correctness criterion is referred to as *serializability*. (For further information on concurrency control in database systems, see [BHG87].)

Existing multi-agent RBDEs use the concept of serializability to synchronize concurrent agents by isolating them, 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. Such isolation unnecessarily obstructs cooperation.

Our conjecture is that an RBDE can support cooperation if given knowledge about the structural organization of the project's objectbase, what it means for the objectbase to be in a consistent state, and the semantics of operations performed on the objectbase. These pieces of information are different for different projects or different phases of a project. For example, the consistency specification of a small project with a few developers familiar with all the components of the project might permit those developers to access the same object at the same time, while a large project with hundreds of developers might require strict isolation between different groups of developers, restricting close cooperation to within groups. The object-oriented framework enables us to integrate the three pieces of information required by RBDEs to support cooperation.

4 An Object-Oriented Framework for RBDEs

RBDEs, like other expert systems, must understand how the domain knowledge is organized in order to provide the desired expertise. Although RBDEs typically support an underlying objectbase that models the domain knowledge, most do not exploit the features of OOP. Specifically, objectbases utilize only the structural aspects of object-oriented data modeling since they support only aggregation hierarchies (i.e., objects that have the same attributes are grouped together in one class), generalization and specialization hierarchies (inheritance), and the unique identification of objects. Existing RBDEs do not integrate behavioral modeling: the operations that can be performed on objects.

RBDEs describe structural aspects of the data model in terms of object classes, whereby each component of the project is abstracted as an object with a set of attributes, the values of which

the prescribed development process. The RBDE would only warn developers about these violations rather than enforce their prevention.

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 and consistency rules, some provide only consistency rules, while others provide only automation rules. We assume a general model in which (1) both forward and backward chaining are supported, and (2) rules may define either automation or consistency maintenance. Monitoring for other purposes and plan recognition are not addressed.

3.1 Single-Agent RBDEs

Most existing RBDEs are single-agent in the sense that they allow only one activity to be performed at any one time on any one objectbase. Some RBDEs require one developer to lock the whole project objectbase, while other RBDEs allow multiple developers to access different copies of the objectbase concurrently, forcing them to merge their changes manually later. In all single-agent RBDEs, the activities performed on the project objectbase (i.e., any single and consistent copy) are strictly serial. They do not allow two developers to access the same copy of an object in the objectbase, thus preventing cooperation.

3.2 Multi-Agent RBDEs

When multiple developers cooperate on a project within an RBDE, they share a common objectbase[2] that contains all the objects of the project. Each developer starts a *session* in order to complete his or her assignment; the sessions of multiple developers may execute concurrently and the developers may concurrently request operations that access objects in the shared project objectbase. There is a need to synchronize the concurrent operations because they might introduce *conflicts* that violate the consistency of the objects they access (e.g., if they concurrently change the same object or structurally related objects).

Synchronization in an RBDE involves not only the human developers but also rules that automatically perform operations on their behalf, since those rules also access the shared objectbase. We use the term *agent* to refer to both human developers and rules that are fired on their behalf. What is missing in existing RBDEs is the ability to synchronize concurrent accesses by multiple agents to the shared objectbase while still providing an environment that supports cooperation.

A simple example illustrates how multiple agents complicate the automation process in RBDEs. Assume that two developers Bob and Mary are working on a common chore. Bob requests a command that triggers a forward chaining cycle. Before the forward chaining resulting from Bob's command is finished, Mary requests a command that corresponds to a rule whose condition is not satisfied, and thus initiates a backward chaining cycle in order to satisfy it. One problem is that one of the rules on the chain triggered by Bob's command might conflict with one of the rules on the in-progress chain that Mary has triggered. Alternatively, a conflict might have occurred even in a single-user context if, for example, Bob had requested two commands concurrently (in two different windows). This problem arises when the RBDE allows multiple rule chains to execute concurrently. The trivial solution of simply serializing chaining cycles is not satisfactory, since the automatically

[2]This objectbase might be physically distributed among multiple client and server processes, but conceptually it is a single database.

```
rule:

   condition

   { activity }

   effect1; effect2; . . .
```

Figure 1: A Generic Rule in the RBDE Model

as a "black box" whose inputs and possible 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 status of objects in the objectbase. The results of the rule's activity determine which effects to assert. Some example rules are shown in Box A.

If the effects of a rule change the objectbase in such a way that the conditions of other rules become satisfied, those rules are fired automatically. Such forward chaining is implemented in OPS5 production systems. Alternatively, if a condition of a rule corresponding to a user command is not satisfied, backward chaining is performed to attempt to satisfy it. Backward chaining is implemented in theorem provers, constraint systems and some production systems. However, the multiple mutually exclusive effects of a rule complicates backward chaining. For example, suppose that rule `r1` has two effects `e1` and `e2` while rule `r2` has a condition `c1`, which can be satisfied if `e1` is asserted. Assume that a developer requests a command that corresponds to `r2`, which can not be immediately executed because its condition `c1` is not satisfied. In order to try to satisfy this condition, the RBDE might automatically invoke `r1`. However, it cannot be determined a priori whether invoking `r1` will satisfy the condition of `r2` because it might happen that the second effect (i.e., `e2`) is the one that will be asserted rather than `e1`.

RBDEs implement a chaining model 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 its rule is satisfied. Once the effects of the activity are asserted in the objectbase, all implications (in terms of performing other activities) are taken care of.

2. **Automate** the performance of some activities that would otherwise be done 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 RBDE automatically performs some activities in order to ensure that the objectbase 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: the RBDE automatically performs as many activities as it can so that the developer won't have to worry about them later on, however, not performing these activities does not affect the validity of the original one.

3. **Monitor** the actual development process and collect information about any violations of

the total effect of the cooperation among the developers cannot be achieved by isolating them, i.e., forcing them to take turns in developing common parts of the project.

We have investigated the problem of synergistic cooperation among multiple developers in the context of RBDEs. Our goal is to exploit the advantages of expert assistance for software development, while at the same time supporting synergistic cooperation. We have discovered that combining cooperation with RBDEs presents several new technical challenges. Specifically, a chain of rules may be fired on behalf of an individual developer in such a way that it affects other developers: the rules may modify the data seen by other developers and/or their own rule chains, and may logically negate the conditions or effects of rules executing on behalf of other developers.

This article presents our solution to these problems, which is based on employing an object-oriented framework for both the project database and the rule system. The database is defined by a *structurally* object-oriented data model (i.e., *objectbase*), in the sense of OODBs. Use of an objectbase is relatively common among approaches to synergistic cooperation, as well as environments that isolate users. Our primary innovation is to treat the rules as the "methods" of the objects, and employ *behavioral* concepts from OOP to integrate the rule system with the shared project objectbase.

Cooperation among the multiple developers is controlled by *metarules* of the expert system, which guarantee consistency in the objectbase and exploit the shared knowledge among developers and the rule chains operating on their behalf. We employ several concepts from OOP, including multiple inheritance, overriding and late binding, and also introduce two new concepts: *collections* of objects, which are not necessarily instances of the same class but are dynamically determined to share the same behavior, and *troupes* of rules, which can statically be determined to share the same metarules governing their interactions. While collections of objects and some mechanisms for aggregating rules have been considered in other work on expert systems [vB89, McD82], these ideas have not previously been applied to RBDEs or other support for cooperative work. Integrating concepts from OOP, OODBs and expert systems provides a firm basis for solving the problem of multi-agent RBDEs for cooperative software development.

In the rest of this article, we explain how the concepts of object collections and rule troupes are used to model consistent interactions among cooperating agents. We first describe the basic single-agent RBDE model and show how supporting multiple agents complicates the model. We then present an extension that uses OOP concepts to integrate the rule-based process model with the underlying object-oriented data model. The multi-agent problem is then reformulated in terms of the extended RBDE model. We then explain how we use the resulting framework to detect and resolve conflicts between concurrent rule chains based on a rule-based specification of the project's consistency that supports synergistic cooperation among human developers.

3 The Domain: The RBDE Model

RBDEs enact the development process of a particular project by automatically firing rules that encapsulate development activities (such as editing, compilation, debugging, etc.). These activities manipulate the different artifacts (designs, source code, test suites, etc.) of the project, which are abstracted as objects and stored in a common project objectbase. The user-level commands either correspond to rules or are built into the environment, e.g., commands to add and delete objects. As shown in figure 1, each rule has a condition that must be satisfied before the second part, the development activity the rule encapsulates, is executed. The condition is essentially a query (i.e., read operation) on the status of objects. Due to lack of knowledge about the internals of the tools invoked and the human decisions that will be made during the activity, the activity is modeled

1 Introduction

Rule-based expert systems and other artificial intelligence techniques have been successfully applied to several important domains, including medical diagnosis, mathematical discovery and hardware configuration. It has long been a dream of AI researchers to make it easy to develop *general* software systems, not restricted to particular domains, and there has been some success in algorithm and data structure construction and/or selection. However, in the near term the most practical approach to general software development, particularly for large scale software systems, seems to be applying expert systems to assist developers in managing software development. Rule-Based Development Environments (RBDEs) exploit expert systems via rule-based modeling of the software development process. RBDEs store the software artifacts in a project database, and define each software development activity that manipulates these artifacts as a rule. Opportunistic forward and backward chaining on the rules automates some of chores that developers would have otherwise done manually and ensures consistency in the database.

The RBDE model has become popular [Per89], and a number of environments based on this paradigm have been proposed and constructed (see Box A). Unfortunately, existing RBDEs do not scale up to real software development projects involving large teams of developers, in part because the current expert system paradigm does not handle *multiple agents* sharing the working memory of the rule system. In order to model development of real projects, RBDEs must be extended to support *cooperation* among multiple developers sharing a common project database, which is treated as the working memory of the rule system. This is difficult because conventional databases (and expert databases) guarantee data consistency by enforcing isolation among concurrent users. However, recent advances in object-oriented databases (OODBs) [KL89], extended transaction models (see box B) and computer-supported cooperative work [Gre88] provide clues as to how to support cooperation, while concepts from object-oriented programming (OOP) provide a path towards defining a semantic data model as a basis for supporting cooperation while guaranteeing database consistency.

In this article, we investigate the scaling up of RBDEs through the use of an object-oriented framework that integrates the structural and behavioral aspects of data modeling and rule-based process modeling. We use this framework to support both consistency maintenance and cooperation in *multi-agent* RBDEs. This research was motivated by our previous work on a single-user RBDE called MARVEL[1] [KBFS88]. We have since applied OOP to integrate MARVEL's data model with its rule-based process model, generalized the multi-agent problem to a range of RBDEs, and developed the proposed solution presented here. The concepts introduced in this article have not appeared in our previous papers.

2 Motivation and Related Research

There has been some successful research on cooperation among multiple expert systems, each with their own rules and working memories — typically with communication through a blackboard architecture. There has also been interest in parallelizing expert systems to improve performance. But there has been little consideration of the problem of *synchronizing* the simultaneous access of multiple rule chains to the contents of a shared working memory while multiple human developers are updating the working memory. Sharing of the working memory is necessary for cooperation among the members of a large software project. This cooperation is *synergistic* in the sense that

[1]The single-user Marvel version 2.5 is available for licensing to educational institutions and industrial sponsors; contact Mike Sokolsky, sokolsky@cs.columbia.edu, 212-854-8348 for information.

An Object-Oriented Framework for Modeling Cooperation in Multi-Agent Rule-Based Development Environments

Naser S. Barghouti* and Gail E. Kaiser†

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

Technical Report CUCS-017-90
May 1, 1990

Keywords: Conflict detection and resolution, expert systems, extended transaction models, object management systems, software development environments.

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

Abstract

Rule-Based Development Environments (RBDEs) exploit expert systems via rule-based modeling of the software development process. RBDEs store the software artifacts in a project database, and define each software development activity that manipulates these artifacts as a rule. Opportunistic forward and backward chaining on the rules automates some of chores that developers would have otherwise done manually and ensures consistency in the database. Unfortunately, Existing RBDEs do not scale up to real software development projects involving large teams of developers, in part because the current expert system paradigm does not handle multiple agents sharing the working memory of the rule system. In this article, we investigate the scaling up of RBDEs through the use of an object-oriented framework that integrates the structural and behavioral aspects of data modeling and rule-based process modeling. We use this framework to support both consistency maintenance and cooperation in multi-agent RBDEs.

Submitted for publication at IEEE Expert

*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.