Rule Chaining in MARVEL: Dynamic Binding of Parameters

George T. Heineman[†] Gail E. Kaiser[‡] Naser S. Barghouti[†]
Israel Z. Ben-Shaul[§]

CUCS-022-91
May 6, 1991

Abstract

MARVEL is a rule-based development environment (RBDE) that assists in the development of software projects. MARVEL encapsulates each software development activity in a rule that specifies the condition for invoking the activity and its effects on the components of the project under development. These components are abstracted as objects and stored in a persistent object database. Each rule applies to a specific class of objects, which is specified as the parameter of the rule. Firing a rule entails binding its formal parameter to a specific object. If the rule changes the object in such a way that the conditions of other rules become satisfied, these other rules are automatically fired. A problem arises in this forward chaining model when the classes of the objects manipulated by the rules are different. MARVEL has to determine which object to bind to the parameter of each rule in the chain, based on the object manipulated by the original rule that initiated the chain. We describe a heuristic approach for solving this problem in the current MARVEL implementation and introduce an algorithmic approach that does better.

Keywords: Programming-in-the-large, reasoning techniques, lifecycle support, experience report.

[†]Barghouti and Heineman are supported in part by the Center for Telecommunications Research.

[‡]Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM, SRA and Xerox, by the New York State Center for Advanced Technology on Computer and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

[§]Ben-Shaul is supported in part by the Center for Advanced Technology.

1 Introduction and Motivation

Software development environments (SDEs) assist developers of software projects in organizing the project's data and carrying out the development process. Every software project assumes a specific organization for its components, and a specific development process that might be different from other projects. Thus, it is not appropriate to build a single software development environment, with a hardwired assistance model. Instead, the assistance should be knowledge-based. In order to provide intelligent assistance, an SDE needs specifications of two things: the project's data model and the project's development process.

MARVEL is an instance of a particular class of SDEs, called *rule-based development environments* (RBDEs), which use a rule-based model of the development process. A project administrator writes a specification of the project's process model and loads it into MARVEL. MARVEL then tailors its functionality accordingly and presents the user with a choice of commands that correspond to the loaded rules. MARVEL provides automated assistance by applying forward and backward chaining among the rules in order to automatically invoke activities that are parts of the development process.

One distinguishing feature of MARVEL is its integration of object-oriented data modeling and rule-based process modeling [2]. Each development activity on the components of the project under development is modeled as a rule that specifies the condition for invoking the activity and the effect of the activity on the components. The project's components are modeled as objects and stored in a persistent object database. The attributes of each object and relationships to other objects are defined by its class. The rules are treated as methods of these classes; in particular, each rule has a formal parameter whose type is a specific object class[1].

When a user requests a command, the corresponding rule is invoked. The formal parameter of the rule is bound to the object selected by the user. For example, the user might wish to edit the CFILE $\mathbf{c}_1$. MARVEL invokes the EDIT rule from figure 1 and binds **?c** to $\mathbf{c}_1$. If the changes that the rule introduces to this object cause the condition of other rules to become satisfied, MARVEL automatically fires each of these rules and attempts to bind their formal parameters to actual objects. MARVEL must *infer* which objects to bind to the formal parameters of the rules in the chain. The *selection* of these objects is a difficult problem when a rule that is a method of one class chains to a rule that is a method of another class, and thus a different object must be bound to the formal parameter of the second rule. In addition, recursive data definitions cause chaining between rules that act on different objects within the same class. We call the problem of binding parameters of rules in a rule chain the *chaining problem*.

In this paper, we describe the chaining problem and explain how it is solved in the current implementation of MARVEL. We describe MARVEL, detailing only those aspects necessary to understand the chaining problem and our solutions; for a more complete description of MARVEL, see [12, 13, 11]. We next explain the chaining problem in detail with a motivating example. We describe a heuristic approach to solving this problem and discuss its implementation in the current MARVEL system. We then extend the example to show the limitations of this approach,

[1]For this paper, we restrict rules to have only one formal parameter. We are investigating chaining between multi-method rules.

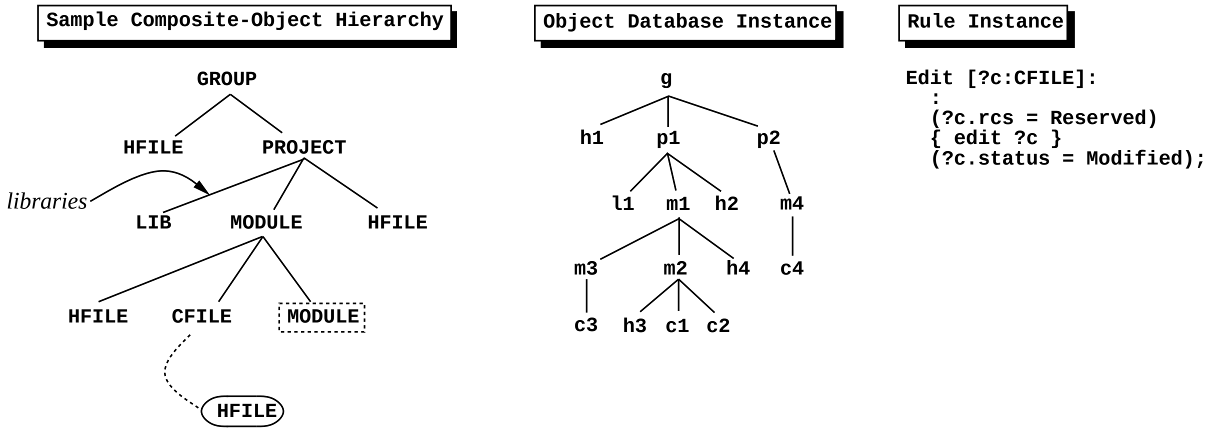


Figure 1: MARVEL data and process model

and present a better algorithmic solution. We conclude with related work on RBDEs and other kinds of rule-based systems, and summarize our contributions.

2 MARVEL Overview

MARVEL is an RBDE which allows a project administrator to create and tailor an environment by defining a *data model* and a *process model* in the MSL specification language. Software artifacts (i.e., code, documentation) are abstracted as instances of classes, which are defined in the data model, and stored in an *object database*. The attributes of each object are defined by its class; for example, a PROJECT object has a set attribute *libraries* which contains objects from the LIB class. For clarity, the names of other attributes are not shown in figure 1. The set attributes create a *composite-object hierarchy*. Figure 1 depicts a sample composite-object hierarchy and an object database instance for this hierarchy. Each object in this example belongs to the class with the same first letter. Note how the object **g** is composed of an instance of class HFILE ($\mathbf{h}_1$) and two instances of class PROJECT ($\mathbf{p}_1$ and $\mathbf{p}_2$). Essentially, the composite-object hierarchy is an "IS-PART" relationship. MARVEL also provides for *links* from one object to another. For example, in figure 1, the class CFILE has a link attribute, *includes* (dashed line), to an HFILE. This link specifies that an instance of the CFILE is associated with a particular HFILE. Links and composite-objects enhance the power of the MSL rule language by allowing queries in the rules to traverse the database structure; we term such queries as *navigational*.

The process model is defined by rules that specify a *condition* that must be satisfied for the rule to fire, and *effects* that assert changes on the object database. The collection of all the rules for a certain project is termed the *rulebase* of that project. Since rules in MARVEL are methods on objects, they have a *formal* parameter with which the rule is invoked. Rules also have *derived* parameters which are the result of queries made by the rule on the object database. The query language consists of boolean combinations of three navigational primitives, *member, linkto* and *ancestor*, and standard relational operators (i.e. $\leq$, $=$) which allow the query to navigate through the object database. *Member* finds the parent/child of an object, *ancestor* finds ancestors/descendents (in the composite-object hierarchy) for a given object, and *linkto* finds objects that are linked to a given object.

For example, the TOUCH [?M:MODULE] rule from figure 2 requires an object from the MODULE class as the formal parameter, and makes a query on the object database to determine two derived parameters. **?r** is defined as "a MODULE that is a member of the *modules* set attribute of **?m** and whose *status* attribute is `Modified`" and "a CFILE that is a member of the *cfiles* attribute of **?m** and whose *status* attribute is `Modified`." When quantified with *forall* ($\forall$), the derived parameter is bound to all the objects returned by the query, while *exists* ($\exists$) binds the parameter to the first such object found.

The condition of a MARVEL rule determines the logical state of the object database necessary for a given invocation to fire. For example, in figure 1 the intended CFILE object must be reserved (*rcs* attribute set to `Reserved`) before editing. The conditions are constructed from predicates based on the rule's parameters (formal and derived), which must be satisfied for the rule to fire. If a predicate in the condition is false, it is said to be *unsatisfied*, and backward chaining could be automatically initiated on this failed predicate in an attempt to satisfy it; this is similar to attempting to achieve a subgoal.

Each MARVEL rule has a set of (possibly) multiple, mutually exclusive effects that are asserted on the object database upon completion of the rule. For example, the EDIT rule in figure 1 asserts that the CFILE object has been modified (*status* attribute is set to `Modified`). Once an effect is asserted on the object database, forward chaining could be initiated to automatically fire other rules. A forward chain from one rule to another is initiated if the effect of the first rule makes an assertion which satisfies the condition of the second. We now present the two basic forms of chaining, *backward* and *forward*, as they operate in MARVEL.

2.1 Backward Chaining

In MARVEL, when the user invokes a rule r whose condition is not satisfied, *backward chaining* is initiated on a failed predicate p in the condition to attempt to satisfy it. The system collects together into a set S_b those rules in the rulebase that have an effect which satisfies p. The system then repeatedly removes and invokes r_i from this set until either p is satisfied, or S_b is empty. Since backward chaining is a recursive process, the system might initiate a backward chain to satisfy r_i. Because the condition is a combination of predicates, satisfying p might still not satisfy the entire condition for r, so this backward chaining process is repeated until the condition is satisfied, or all known possibilities are exhausted.

Note that the chaining problem appears when invoking r_i since it has a formal parameter that the system must automatically determine. In the current MARVEL implementation, however, we bypass this problem since we restrict effects to make assertions only on the formal parameter, not derived ones.

```
Compile [?c:CFILE]:
  :
  (?c.status = Modified)
  { compile the file }
  (?c.status = Compiled);
```

```
Link [?m:MODULE]:
  (forall CFILE ?c suchthat (member [?m.cfiles ?c])):
  (or (?c.status = Compiled)
      (?m.status = Modified))
  { Create .a file for these CFILES }
  (?m.status = UpToDate);
```

```
Touch [?m:MODULE]:
  (and (exists MODULE ?r suchthat (member [?m.modules ?r]))
       (exists CFILE  ?c suchthat (member [?m.cfiles  ?c]))):
  (or (?r.status = Modified)
      (?c.status = Modified))
  {}
  (?m.status = Modified);

Touch [?p:PROJECT]:
   (exists MODULE ?m suchthat (member [?p.modules ?m])):
   (?m.status = Modified)
   {}
   (?p.status = Modified);
```

Figure 2: Motivating Example

2.2 Forward Chaining

Forward chaining is initiated when a predicate p in the effect of a rule is asserted on the object database. MARVEL determines those rules $r_i \in S_f$ in the system whose condition might become satisfied because of this assertion. This is a tenuous connection since the condition of rule r_i might be composed of many predicates, and the system has chosen only one of these as a reason to forward chain; however, this provides MARVEL with an automatic way to logically determine the flow of control in the system.

Since the condition of a rule can involve derived parameters, the chaining problem appears and cannot be bypassed. Specifically, in rule r_i, the predicate p might be based on a derived parameter, and the system must determine the formal parameter with which to invoke r_i. In this paper we are only concerned with forward chaining, since in our current model, backward chaining exhibits no problems. The actual details of the problems arising during forward chaining will be elaborated on in the next section.

3 Chaining Problem

In a rule-based system, when the system determines that it must chain to rule r_i, it must determine which object to bind to r_i's formal parameter; this is the essence of the *chaining problem.* The reader should observe that if there were no derived parameters, this problem would not exist, for the conditions and effects of a rule would be based solely on the formal parameter.

MARVEL, like AP5 [4] and other systems, allows for rules to have *derived* parameters by using existential and universal operators to *bind* quantified parameters to certain objects based on arbitrary logical expressions. Figure 2 shows some rules in the MSL language. In this example, if the TOUCH[?M:MODULE] rule were invoked on object $\mathbf{m}_2$ from figure 1, MARVEL would bind **?c** to either $\mathbf{c}_1$ or $\mathbf{c}_2$ (whichever has a *status* attribute value of `Modified`), and **?r** would be empty since there is no child of $\mathbf{m}_2$ of the MODULE class.

The ability to make queries gives the rules more power (since their activities are not limited to formal parameters), but it complicates the chaining mechanism. Consider a forward chain to r_i based on a predicate with a derived parameter. In order for the system to fire rule r_i, it must determine the object to bind to the formal parameter. For example, if the EDIT rule from figure 1 is invoked on $\mathbf{c}_1$, then a forward chain is initiated to TOUCH[?M:MODULE], but it is not readily apparent to what object **?m** should be bound as there exists four MODULE objects. In the next section, we explore several ways to solve the chaining problem, and describe the current MARVEL implementation.

4 Heuristic Approach

There are several possible ways to determine the objects to bind to the formal parameter of a forward-chained rule.

- *Manual:* The system could ask the user for the specific object to invoke the rule with. The main drawback to this policy is that the user should not need to know the details of rules and chaining. In addition, the user might respond incorrectly, causing the system to perform unnecessary activities and fail to enact the proper activities.
- *Logical:* Since the derived parameters are logically determined, it might be possible to invert this logic to determine the proper object to use. We discuss this possibility in more detail in the algorithmic approach section.
- *Heuristics:* The system can use heuristics to search for the proper object. If the system assumes that rule chaining occurs in a localized area in the object database, then the search space can be kept small. As an alternative to searching, the system could also use a set of axioms (as put forth in GRAPPLE [10]) to determine legal choices for the parameter.

In MARVEL we have implemented a set of heuristics to search "near" an object to determine the proper objects to use during chaining. In doing so, MARVEL makes use of the composite-object hierarchy, since it assumes that objects near each other in the hierarchy are semantically related. The search path also include links (see section 2) since they also define semantic associations between objects. During a forward chain to rule r_i from a rule invoked with object O, we search for the object to bind to r_i's formal parameter in the following order:

1. The object O itself.
2. O's parent object.
3. O's immediate children.
4. The objects associated (through links) with O.
5. O's proper ancestors.

If we apply these heuristics to the example from section 3 (chaining from EDIT[$\mathbf{c}_1$] to TOUCH[$\mathbf{m}_2$]), we see that step 2 determines the correct object. To give a more involved example, if TOUCH[$\mathbf{m}_2$] is invoked, both LINK[$\mathbf{m}_2$] and TOUCH[$\mathbf{m}_1$] should be fired. Step 1 discovers the object for the LINK rule, while step 2 takes care of TOUCH. The heuristic, obviously, cannot stop at the first candidate found, but must collect all candidates together since it is possible to have multiple instantiations of the same rule with different objects as well as different rules with same or different objects.

5 Implementation

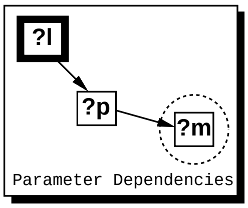
This section briefly describes the static and dynamic support currently implemented in MARVEL to provide for chaining. Since the *data* and *process* models are tailorable, MARVEL reads in their specification, as written in the MSL language by the administrator of the environment. The system compiles a *rule network*, an efficient data structure which maintains the chaining possibilities (both backward and forward) for each predicate in each rule. This rule network is stored on disk upon completion for later invocations of MARVEL. Because this is a static process, it is executed only when either model changes.

The runtime support is provided by the *Opportunist*, which is MARVEL's chaining engine. When the user issues a command, the Opportunist matches this to a rule r in the rulebase. The user selects the argument from the object database and the system invokes r. First, MARVEL generates the derived parameters by sequentially evaluating each of the quantified expressions in r's condition. Next, MARVEL evaluates the predicates that must be true for r to fire. If there is a predicate p that is not satisfied, then MARVEL collects from the rule network those rules r_i that have an effect which satisfy this predicate. MARVEL invokes these rules until the set is exhausted or the predicate is satisfied. MARVEL binds the failed object to r_i's formal parameter since effects are restricted to be based on the formal parameter.

Once r's condition is satisfied, MARVEL executes r's activity with the necessary information (see [7] for a full description of this process) and selects the proper effect to assert as determined by the result of this execution. For every predicate p in this effect, MARVEL asserts it on the object database and collects those rules from the rule network that may now be satisfied. MARVEL enters the forward chaining cycle, and automatically invokes each of these rules r_i in turn, potentially generating more rules to invoke in a recursive manner.

To invoke these rules, MARVEL must determine the object to bind to r_i's formal parameter from the predicate p. If the object o_i in p is of the proper class, then MARVEL binds the formal parameter to o_i, otherwise MARVEL performs a search from the object bound to r's formal parameter based upon the heuristics described in section 4. If MARVEL is unable to find one, then it does not invoke the rule.

```
Touch [?l:LIB]:
  (and (exists PROJECT ?p suchthat (member [?p.libraries ?l]))
       (exists MODULE  ?m suchthat (ancestor [?p ?m]))):
  (?m.status = Modified)
  {}
  (?l.status = Modified);
```



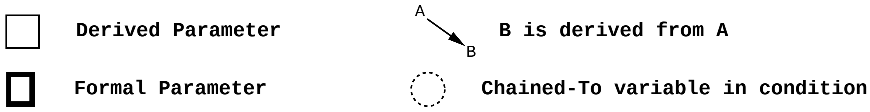


Figure 3: Extending the example from figure 2

6 Algorithmic Approach

If we extend the example from figure 2, our heuristic approach fails to generate the proper formal parameter to use for chaining; we add the TOUCH[?L:LIB] rule shown in figure 3. This rule queries the object database for the PROJECT object **?p** which contains this LIB **?l** and then finds a MODULE object **?m** which is a descendent of **?p** that is `Modified`. If **?m** is found, then **?l** is marked as `Modified`. If TOUCH[$\mathbf{m}_1$] is invoked and succeeds, `(m`$_1$ `= Modified)` is asserted and MARVEL initiates a forward chain to TOUCH[?L:LIB]. The heuristic approach outlined in section 4 will fail to determine the formal parameter since $\mathbf{l}_1$ is a sibling of $\mathbf{m}_1$ (see figure 1), and falls outside the search space; for any fixed searching heuristic, there will always be cases which fail. The algorithmic approach, however, will be able to correctly invert the derived parameters to generate the possibilities for chaining.

The intuition behind the algorithmic approach is that each navigational query (from section 2) is itself invertible. For example, in figure 3, `(exists PROJECT ?p suchthat (member [?p.libraries ?l]))` returns the parent(s) of the object(s) bound to **?l**. This can be inverted to bind to **?l** the children objects, in the *libraries* attribute, of the object(s) bound to **?p**.

We outline our algorithm in figure 4. We assume that each rule has a *dependency graph* which determines the dependencies between parameters in a given rule. In figure 3, for example, **?m** is dependent upon **?p** which is dependent upon **?l**. This information is statically determined from the derived parameters. Assume that MARVEL initiates a forward chain from rule r to rule r_i because a predicate p in r satisfies a predicate q in r_i's condition. The object O changed by p is subsequently bound to the parameter, **?V**, that q operates on. If **?V** is the formal parameter of r_i, then the system invokes r_i on O. Otherwise, **?V** is a derived parameter.

The algorithm inverts each parameter in the dependency graph, starting from **?V**, until the formal parameter is determined. The Augment procedure in figure 4 reverse-evaluates the expression of a quantified parameter to determine objects to which other parameters can be bound. For example, line 7 in Augment covers the example on *member* mentioned earlier this section. If the formal parameter remains undetermined, then the system cannot chain to r_i; in contrast, failure to find an object to bind to the formal parameter using the heuristic approach, might only mean that the heuristic is too weak.

In figure 5 we see the results of the forward chain after MARVEL has invoked TOUCH[$\mathbf{m}_1$] from figure 2. When the predicate `(?m.status = Modified)` is asserted on the object database, the system forward chains to the TOUCH[?L:LIB] rule from figure 3. To determine the object to bind to the formal parameter, the system inverts the expression which defines the parameter **?m**. Initially, **?m** is bound to $\mathbf{m}_1$. Figure 5 outlines the progress of the algorithm, giving key values at the specified lines. Note how the algorithm inverts the expressions for **?m** and **?p** in order, as it proceeds to determine **?l**. After the first call to Augment), $\mathbf{p}_1$ is added to the list of objects bound to **?p** (in line 8). The second iteration inverts the expression for **?p** and binds **?l** to $\mathbf{l}_1$. This determines the formal parameter, and TOUCH is instantiated for $\mathbf{l}_1$.

It might not be clear why this logical inversion process is preferred to searching the object database. Since the expression for a parameter can be arbitrarily complicated, any heuristic, aside from searching the entire object database, will potentially fail to find certain objects

```
      ?Par.objects is the set of objects that are bound to ?Par.
      ?T.value is the set of parameter augmentations generated by the AUGMENT procedure for a given expression.
      ?F is the formal parameter of the chained-to rule.
      S is a stack of parameters to be inverted.

ALGORITHM Invert[?V]
1.   IF (?V is the formal parameter ?F) THEN
2.      Add[?F.objects, O]
     ELSE
3.     S = {?V}; ?V.objects = {O};
4.     WHILE S not empty DO
5.       ?V = POP(S);
6.       IF (?V = ?F) THEN CONTINUE;           { Skip if this is the formal parameter}

       { The expression for the derived parameter ?V is a quantification of an arbitrary expression T. }
       { We represent T as an AND/OR tree of individual expression and reverse-evaluate for ?V         }
7.       Augment[T, ?V];
8.       Evaluate[T.value];

       { For every (derived) parameter that has had their set of possible objects augmented by the }
       { previous function, push onto the stack S.                                                 }
9.       FOR every parameter ?P in T.value DO
10.        PUSH(?P);
     END;
11. Instantiate the rule R for each object from the ?F.object list.
END;  { of ALGORITHM Invert[?V] }



    N is an arbitrary expression represented by an AND/OR tree.
   ?V is the parameter being quantified in this expression N.
   ?Par is the "other" parameter in the navigational primitive expression (primitives are binary)

PROCEDURE Augment [N,?V]
1. FOR all children C of N DO
2.   Augment[C,?V];
3. IF (N is an internal node) THEN
     CASE N OF
4.     AND:  N.value = Intersection of C.value FORALL children C of N.
5.     OR:   N.value = Union of C.value FORALL children C of N.
     END;
   ELSE
    { Note that N is a navigational primitive.  ?V is the (known) parameter in this primitive }
    { which is being quantified.  ?Par is the other (unknown) parameter in the primitive       }

    CASE N OF
     Member [?V.attr ?Par] :      { ?V is a parent of ?Par }
6.    N.value = Add[?Par.objects,objects in (?V.objects).attr];

     Member [?Par.attr ?V] :      { ?V is a child of ?Par }
7.    N.value = Add[?Par.objects,parent object of (?V.objects)];

     Linkto [?V.attr ?Par] :      { ?V links to ?Par }
8.    N.value = Add[?Par.objects,objects from (?V.objects).attr that linkto an object of the ?Par class]

     Linkto [?Par.attr ?V] :      { ?V is linked from ?Par }
9.    N.value = Add[?Par.objects,objects from the ?Par class that linkto (?V.objects) through the "attr" attribute]

     Ancestor [?V ?Par] :         { ?V is an ancestor of ?Par }
10.   N.value = Add[?Par.objects,objects from the ?Par class that are a descendant from (?V.objects)]

     Ancestor [?Par ?V] :         { ?V is a descendant of ?Par }
11.   N.value = Add[?Par.objects,objects from the ?Par class that are an ancestor of (?V.objects)]
    END;
END;  { of PROCEDURE Augment[N,?V] }
```

Figure 4: Inverting Derived Parameters Algorithm

```
Invert[?m ]  --> ?V = ?m
3. S = {?m} ; ?m.objects = {m1};
5. ?V = ?m; --> S = {}
6. ?m <> ?l
7. Augment[(ancestor [?p ?m]),?m];
8. Evaluate[Add[?p.objects, p1]]; --> ?p.objects = {p1};
10. Push(?p); --> S = {?p}

5. ?V = ?p; --> S = {}
6. ?p <> ?l
7. Augment[(member [?p.libraries ?l]),?p];
8. Evaluate[Add[?l.objects,L1]]; --> ?l.objects = {L1};
10. Push(?l); --> S = {?l}

5. ?V = ?l; --> S = {}
6. ?l = ?l

4. S is empty

11. Instantiate Touch[L1]
```

```
Augment[(ancestor [?p ?m]),?m]
1. No children
3. Leaf node
11. N.value = Add[?p.objects,p1]
```

```
Augment[(member [?p.libraries ?l]),?p]
1. No children
3. Leaf node
6. N.value = Add[?l.objects,L1]
```

Figure 5: Solving the chain for figure 3

which could be bound to the formal parameter. In Marvel, and RBDEs in general, the object database might contain several thousand, and possibly more, objects. It would be extremely inefficient to perform a search on the entire object database. We are currently investigating the implementation of the logic-based inversion method.

7 Related Work

There are many other systems that support *intelligent assistance*; we focus on those systems that execute some form of chaining to perform this assistance.

EPOS [14] and TPLAN [6] attempt to achieve subgoals (backward chaining) when the condition for an invoked rule is not satisfied. Darwin [15] carries out inferencing to determine whether or not an activity is allowed. These kind of systems perform strictly backward chaining, and are usually Prolog-based. The unification scheme in Prolog successfully determines parameters to match because the body of Prolog clauses have no quantified expressions like those in Msl. If Marvel allowed effects to be asserted on derived parameters, then backward chaining would also be affected by the chaining problem. As it stands, however, Marvel's backward chaining cycle, like Prolog unification, is straightforward. Oikos [1] extends a Darwin-like system to more complicated control structures among rules, but uses a blackboard scheme that passes all necessary information from one blackboard to another directly, thus bypassing the chaining problem.

The merlin [9] system has both backward and forward chaining capabilities, but it avoids the chaining problem; backward chaining is resolved through Prolog-style unification, while forward chaining explicitly lists the parameters (much like a blackboard scheme) that are passed from one rule to the next. alf [8] has forward chaining capabilities, but the paper does not state how it determines parameters. grapple [10] is exceptional, in that it explicitly addresses the parameter issue that arises in chaining between rules.

Odin [3], AP5 [4] and OPS5 systems have rules that are *triggers* activated when the state of the object database changes. The chaining problem does not appear under this data-driven approach, since there is no specified flow of control. Rule-based database systems like POSTGRES [16] work in a similar fashion.

The composite-object hierarchy allows MARVEL to make efficient navigational queries to the object database; in addition, the algorithmic approach outlined in section 6 is dependent on this structure, since this makes an efficient inversion process. The PROBE DBMS [5] provides for queries on object databases, and outlines some methods for optimization. If a system has relations, instead of structure, that determine the interconnections between the object entities (like in EPOS) then these navigational queries can also be implemented, but will only be as efficient as the system's query optimizer.

8 Contributions

In the current implementation of MARVEL we address the *chaining problem* by using a fixed set of heuristics to search for the proper parameter. Our heuristics seemed satisfactory for many cases. However, as we enhanced the MSL rule language and began experimenting with more intricate rules, chaining cases arose that escaped our heuristics. One possibility would be to increase the breadth of the searching heuristics, these cases can be more efficiently solved by logically inverting the derived parameters. We have given an algorithm that performs this inversion process for an arbitrarily complex MSL rule.

References

[1] V. Ambriola, P. Ciancarini, and Montanegro C. Software Process Enactment in Oikos. In *Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 183–192, Irvine CA, December 1990.

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

[3] G. M. Clemm. The Workshop System – A Practical Knowledge-Based Software Environment. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 55–64, Boston, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[4] D. Cohen. Compiling complex database transition triggers. In *1989 ACM SIGMOD International Conference on the Management of Data*, pages 225–234, Portland OR, June 1989. Special issue of *SIGMOD Record*, 18(2), June 1989.

[5] Umeshwar Dayal *et al.* Simplifying complex objects: The PROBE approach to modeling and querying them. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 390–399, Morgan Kaufmann, Inc., 1990.

[6] Deborah Frincke, Myla Archer, and Karl Levitt. A Planning System for the Intelligent Testing of Secure Software. In *Knowledge-Based Software Assistant KBSA-5*, pages 346–360, Liverpool NY, September 1990.

[7] Mark Gisi and Gail E. Kaiser. Extending A Tool Integration Language (Experience Report). Technical Report CUCS-014-91, Columbia University Department of Computer Science, April 1991. Submitted for publication.

[8] C. Godart, F. Charoy, and J.C. Derniame. Computer Assisted Software Engineering: Characterisation and Modeling. *ICCI 89 - Toronto*, 1989.

[9] H. Hünnekens and G. Junkermann and B. Peuschel and W. Schäfer, J. Vagts. A Step Towards Knowledge-based Software Process Modeling. In *1st Conference on System Development Environments and Factories (SDE&F 1)*. Pitman Publishing, London, 1990.

[10] 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. Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[11] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke. Database Support for Knowledge-Based Engineering Environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

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

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

[14] Chunnian Liu. Software Process Planning and Execution: Coupling vs. Integration. Technical report, Norwegian Institute of Technology (NTH), November 1990.

[15] 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.

[16] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In Carlo Zaniolo, editor, *1989 ACM SIGMOD International Conference on the Management of Data*, pages 340–355, Washington DC, May 1986. Special issue of *SIGMOD Record*, 18(2), June 1989.