An Approach for Distributed Query Processing in MARVEL: Concepts and Implementation

Toni A. Bünter

Technical Report CUCS-030-93
COLUMBIA UNIVERSITY

Abstract

This work displays an approach for the query processing of MARVEL rules upon a distributed MARVEL objectbase. Rules and rest rules run simultaneously on different subenvironments, synchronized by a coordinating subenvironment. Instead of transmitting objects, the showed method transmits images. The concept of lazy calling is introduced.

Contents

1 Introduction

Distributed query processing in databases is well explored for relational databases. In this work we elaborate a technique for distributed query processing in a MARVEL objectbase, distributed over subenvironments. Results from the research of distributed relational databases are applied in form of semijoint optimization [2].
The entire work is done inside the scope of the MARVEL project. In spite of this restriction, the displayed technique is believed to be applicable for a wide range of objectbases.

1.1 MARVEL

MARVEL is a rule-based development environment [4]. An objectbase and rules can be tailored with the MARVEL Strategic Language (MSL). The objectbase keeps track of the process and production data. MARVEL rules are the atomic elements building the development process. Their activation can be triggered off by a user or by forward/backward chaining of another rule [3].

The most recent release, MARVEL 3.1, runs a single server, administering one single objectbase. One or more clients can access the objectbase.
The next step in the MARVEL project will include the *geographical distribution* of a development project. This means, a single development project can be distributed for different servers running on different computing sites. Further, it has to be assumed that the communication resources among the different servers are scarce. Because query and rule processing can not be restricted to one local server, the redesign the rule/query processing is one consequence.

The scope of this work is the query section of a MARVEL rule. The mechanism we propose, starts the rule on one subenvironment, coordinating subenvironment (cse) called, which synchronizes rest rules on remote subenvironments. Information exchange among the subenvironments is basically reduced to synchronization signals and images of objects.
A priori it is not given to involve all subenvironments for a rule processing. Therefore, we show an mechanism, called *lazy calling* which involves a subenvironment from the point on it is necessary.

1.2 Rest of this Paper

Section 1 introduces the main concepts of our approach of the distributed query processing. Section 2 discusses the algorithm in detail. Section 3 illustrates the mechanism with a example. Section 4 gives an outlook over the remaining parts of the rule processing.

2 Concepts

2.1 Marvel Rules and the Query Section

A Marvel rule consists of five sections:

- The *parameter list*: Actual parameters are bound to the formal parameter while firing a rule.
- The *characteristic function*: Through evaluation of the binding formulae, more objects are bound to the binding variables, accessed by the remaining parts of the rule processing.
- The *property list*: Analysis of the bound objects and possible backward chaining.
- The *activity*: Activation of tools.
- The *assertions*: Depending upon the results of the tool execution, assertion are made to the bound objects, and forward chaining is activated.

The first two section of a rule, the parameter list and the characteristic function make up the *query section* of a rule. The query section consists of a list of parameter variables and a list of bindings. A binding is basically a pair of a *binding variable* and a *binding formula*.

The list of bindings is executed sequentially. Each binding binds objects which are valid against the binding formula and belong to the same class or a subclass of the binding variable. Immediately before the evaluation of a binding, all variables except the binding variable which occur in the binding formula, are bound to sets of objects. We refer to them as *bound operands*.
A thorough definition of its syntax and semantics are given in [1].

2.2 History Variables

Two variables, the *history variables* called, keep track of the evaluation state during the formula walkthrough . The *universe variable* represents at each point of the formula the actually search space, whereas the *bound object variable* collects the valid objects, and assigns them to the binding variable. In case of more than one subenvironments, the history variables are represented by the totality of instances bound to the local universe variable or bound object variable, respectively.
The default binding for the universe are the objects of the objectbase belonging to the class or a subclass of the binding variables.

Geographical Distribution

The following project of MARVEL 3.1, the MARVEL/OZ project, is committed to the geographical distribution of the objectbase. It is assumed that more than one computing site is involved in the same development project.

On each computing site an MARVEL/OZ server, called *subenvironment (se)*, is active. We call the set of subenvironments belonging to the same development project, the project's *domain* . Each subenvironment maintains its own objectbase and its own hidden file system. The different subenvironments are connected by a low end network (e.g. a WAN).

The distribution is understood, that the hierarchical closure of a objectbase is fully placed on the same subenvironment, but linked objects do not have to be local. Further, we assume, that a link to a remote object is represented locally by its *image*, which is represented as the pair:

(subenvironment_id, object_id).

2.3 Rest Rules

Our approach for distributed rule processing proposes the firing of a rule on a *coordinating subenvironment (cse)*, which synchronizes *rest rules* on remote subenvironments. A rest rule of level j is a rule which starts first at a binding j. More formally:

- A *rest rule* of level j is a rule R without the first j bindings. The parameter and the binding variables $b_1 ... b_{j-1}$ are initialized with empty sets of objects.

Before each binding, the rule running on the cse checks for additional subenvironments in the project's domain which have to be addressed for a rest rule activation.

2.4 Image Processing

A straight forward approach for distributed query processing would request all necessary objects to the cse, check them for validity and send them back. This can cause an immense load for the network resources.
Instead of convey objects, we propose to notify remote subenvironments, by transmission of images, to bind objects locally.
The only exception is the case of an associative predicate with two variables. This is equivalent to the computation of a semijoint in distributed relational databases. Instead of transmitting images, sets of values of involved attributes has to be transmitted (see [2]).

Throughout this work we mostly use the term 'object' for instances as well for images. If necessary, we use *real* for objects which are exclusively instances.

Basically, there are two different needs to transmit images. On one hand images are 'sent back' to the subenvironments, where they belong to, for binding their (real) objects to local binding variables. On the other hand we send images to remote subenvironments to complete the range of a local search space for enabling local computation.

Lazy Calling and Rest Rule Firing Criteria

Our approach approximates the minimum of subenvironments which have to be involved in the processing of a rule.
A subenvironment is addressed for a rest rule execution after checking the *rest rule firing criteria*. Due to this mechanism we call the rule processing *lazy calling*.

The criteria for firing a rest rule on a not yet involved subenvironment *se*, depends on the characteristic of the binding formula and the objects bound to the bound operands. Therefore, we define the property *restricted* of a binding formula:

- `member()` and `ancestor()` are restricted.
- `(AND` $f_1 ... f_n$`)` is restricted if f_1 is restricted.
- `(OR` $f_1 ... f_n$`)` is restricted if all $f_1 ... f_n$ are restricted.

This definition designates formulae which require only subenvironments which have objects or images bound to the bound operands. This means, that the cse has to address for rest rule activation exactly the subenvironments which are not yet involved, but have at least one image bound remotly to one bound operand.

In case of a non restricted binding variable, on all not yet involved subenvironment of the domain a rest rule is started.

2.5 Communication

For every subenvironment and rule or rest rule, a readable and writable *port* is provided. Messages are written as triples:

(receiver address, message body, sender address)

Arriving messages are buffered in a queue and the order in which messages are written on a port is preserved.

We assume, that all subenvironments can write asynchronously to their port and messages are delivered savely.

In the given algorithm we use the two primitives `write_port (message)` and `read_port (filter)`. `read_port` waits or overreads messages which do not suit the filter until an acceptable message arrives.

Subenvironments are able to send messages to every other subenvironment. The cse takes care of the synchronization.
In the most used synchronization, the cse waits for `write_end_signal`s from all subenvironments and releases afterward a `continue_signal` to all subenvironments which are waiting to resume the computation.

3 Algorithm

This section discusses how the above shown concepts can be implemented. It also provides a more profound understanding the concepts.

Appendix 1 displays the binding algorithm for the cse, appendix 2 for a subenvironment running the rest rule. They are written closely to the actual implementation of the query processing in MARVEL 3.1 and the programming language C.

Each algorithm consists of the two main functions `build_characterized_binding`, which takes care of the rest rule firing, and the `get_all_bound_objects`, which computes the binding.

Before the evaluation of a binding, the `build_characterized_binding_CSE` function on the cse site calls the `check_new_subenvironment_criteria` to check for new subenvironments to be addressed. In case of a restricted formula all remote subenvironments notify the cse about subenvironment candidates. The cse adds them to the `list_of_new_se`, along with candidates found on the cse itself. Then, the cse notifies the new subenvironments to run the rest rule.
The `build_characterized_binding_SE` on the called subenvironment initializes the first *level* bindings with empty sets and notifies the cse about the end of initialization. The cse waits for the acknowledgement from each new subenvironments. After the rest rule initializations, the cse sends a `continue_signal` to all subenvironments.

All subenvironments begin now with the `get_all_bound_objects` function. As described in [1], `get_all_bound_objects` walks through the binding formula. The history variables, `universe` and `bound_objects`, keep track of the objects remaining in the search space and the objects already bound.

`get_all_bound_objects` requires two kinds of synchronization.

- During the evaluation, real objects as well as images are bound to the history variables. But, if e.g., a `NOT` or a predicate has to be computed, images have to be accessed locally and are therefore 'sent back' to their subenvironment where their real objects are retained. Then, the real objects of the arriving images are bound locally to history variables. Globally seen, this process does not change the content of the history variables. After this replacement, only real objects are bound the history variables.

  The replacement mechanism enables the predicates and the `NOT` operator to run locally. If still images were bound to history variable, incorrect results would occur. For example, to bind an object to `bound_objects`, we have also to check if this object is still in the universe. If we allowed, having images bound to the history variables, it would be possible that the object's image is bound to a remote universe variable, and therefore it would be dismissed in any case as a valid object.

  In the implementation, the function `replace_images` sends images to their subenvironments and binds the objects of incoming images to the corresponding variable. On the subenvironments site `replace_images` sends `write_end_signal` and waits for a `continue_signal` from the cse site. On the cse site `replace_images` waits for all `write_end_signal`s and releases the `continue_signal`.

- The `linkto(?`$A.att$`, ?`X`)` predicate, in which A is a set of objects and X is the binding variable, is a special form of a semijoint. Instead of computing the entire carthesian product between A and the universe over the whole domain, this predicate can be computed by finding out for every o in A that there is at least one linked object l in the attribute $o.att$, and l is bound to a universe variable in one of the subenvironment. There are basically two ways to get this linked objects: (1) Send images of all object bound to the universe variables to all subenvironments, or (2) send all images of objects in the bound variable A to all subenvironments. Assuming that A is generally smaller than the universe variable, we chose (2) for our implementation. Fortunately, the semijoint is not 'pure'. The image of an object o in A has only to be sent to the subenvironments for which an image exists in $o.att$.
  This mechanism is implemented with the function `complete_bound_operand`.

The `AND` and `OR` case can be evaluated without remote interference.

In the `NOT` f case, first, the subformula f is computed and objects are bound to `bound_objects`. Applying the `NOT` operator to the subformula means to bind all objects in the universe which are not in `bound_objects`. Therefore images in the history variables are replaced. Afterwards, the `set_complement` can be computed locally.

A predicate takes two operands: the binding variable and a bound operand or a constant. Because images only consist of the object id and subenvironment id, the images in the bound operands are replaced by objects, next to the replacement of the images in the universe. The predicates `member()` and `ancestor()` can be evaluated locally. Local evaluation of associative predicates is only possible if one operand is a constant. If so, the function `get_associative_predicate` acts locally.

If both operands are variables, the computation of the predicate is a semijoint between the bound operand and the universe. For the computation of vertical distributed relational databases, optimizing solutions already exists (e.g. [2]). For completeness, we display a simple, non trivial, but suitable strategy.

Be a the binding variable and b the bound operand. Be $a.att1$ and $b.att2$ the attributes over which the semijoint is computed, and op the conditional operator. If op is $<$, $<=$, $>$ or $>=$ the optimal solution is straightforward:
Supposing, the operator is $<$. Each subenvironment computes the maximal value of all $b.att2$s. These maximal values are sent to the cse which looks for the maximum among them. This maximum is a global maximum among all values in all $b.att2$. It is sent back to all subenvironments. Then, the $b.att2$ symbol in the formula is replaced by the global maximum, and `get_associative_predicate` can be computed locally.

In case of the $==$ or the $!=$ operator, we have to compute the carthesian product between the tuples $\{(o, o.att2)|o \in b\}$ and $\{(o, o.att1)|o \in universe\}$, and to extract the pairs $\{((o_1, o_1.att2), (o_2, o_2.att1))|att1 = att2\}$. This can be accomplished by handling only the occurrences of attribute values in b. We send all attribute values $att2$ occurring in $b.att2$ to the cse. The cse builds up a list of all values. This list is sent back to all subenvironments. With help of this value list the valid objects o in the universe of corresponding class – $o.att1$ is element of the value list – are bound to the binding variable a.
The `get_assoc_semijoint_predicate` function implements this mechanism.

4 An Example

To illustrate the algorithm and the concepts, we display a non-trivial – even if made up – example of query:

```
example\_query [?m1:MODULE ?m2:MODULE]:
  (forall CFILE ?c suchthat (OR member [?m1 ?c] member [?m2 ?c] ))
```

```
(forall HFILE ?h suchthat (AND ancestor (?m2, ?h) linkto [?c.include ?h]))
(forall MODULE ?s suchthat (?h.date == ?s.date))
    :
```

The classes in the objectbase are:

```
MODULE :: superclass ... ;
files   : set_of FILE;
include : set_of HFILE;
end

CFILE :: superclass FILE ... ;
contents : text;
include : set_of link HFILE;
end

HFILE :: superclass FILE ... ;
date  : integer ;
end
```

The `example_query` will run on a domain with three subenvironment. *SE1* is the cse. Figure 1 displays the actual objectbases on the three sites.

We discuss the execution of the query by watching at the major steps in the execution trace.

The query is issued on the cse with the following arguments:

example_query (mod1, (2, mod2))

Because the second argument is an image and the first binding formula is restricted, the cse addresses the subenvironment SE2 to run a rest rule. Before calling `get_all_bound_objects` the state is:

	SE1	SE2	SE3
?m1	{mod1}	{}	n.a.
?m2	{(2, mod1)}	{}	n.a.

The first binding formula begins with an `OR` of two member predicates. The second replaces the image $(2, mod2)$ before calling `get_member`. The state after `get_all_bound_objects` is:

	SE1	SE2	SE3
?m1	{mod1}	{}	n.a.
?m2	{}	{mod2}	n.a.
?c	{c1, c2}	{c3, c4}	n.a.

The second binding formula is also restricted. The bound operands in the binding variables ?m1 and ?c are free from images and therefore, SE3 is not yet addressed. After the execution of the first predicate in the `AND` list, the state of the universe variable is:

	SE1	SE2	SE3
universe	{}	{h2, h3}	n.a.

Before the link predicate can be computed the images of the c-files which have a link to an remote h-file, are sent to the subenvironments of the corresponding h-file. (This is done by the function `complete_bound_operand`.) The image of c2 is sent to SE2 and the image of c3 to SE1. Just before `get_linkto` is called, the 'completed' operands ?c are:

	SE1	SE2	SE3
?c	{c1, c2, (2, c3)}	{(1, c2), c3, c4}	n.a.

After the local execution of the function `get_linkto` the binding variable is:

	SE1	SE2	SE3
?h	{}	{h2, h3}	n.a.

The third binding formula is not restricted and therefore a rest rule is fired on the third subenvironment. The formula consists of one associative predicate with two variables. The meaning of the formula is binding all h-files which have the same date as at least one in ?h.
The function `get_assoc_semijoint_predicate` is called. All subenvironments send all occurrences of date values of the h-files in ?h to the cse. The cse builds a list of all occurrences, which is in our example {930829, 930721}. (This is a simple integer codification of 8/29/93 and 7/21/93.) The list is then sent to all involved subenvironment for extracting h-files with a date value in the list. We finally get:

	SE1	SE2	SE3
?s	{}	{h2,h3}	{h5}

5 Final Words

5.1 The remaining sections of the rule

The shown approach of the query section ends with locally bound objects. Thorough investigation about distributing the remaining sections in the rule processing goes beyond this work. Although some suggestion should be added.
The property section is similar or often simpler than the evaluation of binding formulas, a similar mechanism may be applicable. Firing of rules by backward chaining is possible because our mechanism allows images as arguments. Not yet solved is the coordination of a distributed backward chain itself.
The activity section has similarity to a semijoint. But in addition to the number of objects in the objectbase that have to be transmitted, the size of the medium attribute, the size of the file in the hidden file system has to be considered.
After images are replaced, the assertions can be done directly, without additional interference from remote subenvironments. A coordination mechanism for the forward chaining, probably similar to the backward chaining mechanism, has to be developed.

5.2 Performance

Because the proposed approach is not yet implemented, no performance results can be shown at this time.

6 Appendix 1

Implementation of the query processing on the coordinating subenvironment site:

```
/* ---------------------------------------------------------------- */
/* QUERY PROCESSOR FUNCTION OF THE COORDINATING SUBENVIRONMENT (CSE) */

/* Global objects:
   domain   - list of the subenvironments belonging to the same development
              project
   cse_id   - id of this (coordinating) subenvironment
   se_list  - current list of subenvironment involved in the rule processing
   local_ob - objectbase of this (the cse) subenvironment
*/


/* Signals:
   write_end_signal
```

```
   continue_signal
   end_init_rest_rule
*/

build_characterized_binding_CSE (bindings)
     BINDING bindings;
{
  LIST_OF_SE list_of_new_se, se;

  for (binding = bindings, level = 0;
       binding != NULL;
       binding = binding->next, level++)
    {
      check_new_subenvironment_criteria (list_of_new_se, binding->formula);
      while ((se = get_next_se (list_of_new_se)) != NULL) {
initiate_rest_rule (se, level);
insert_list (se_list, se);
read_port (cse_id, end_init_rest_rule, se);
      }
      /* Send continue_signal to all subenvironments.
       */
      get_all_bound_objects (binding->formula, local_objectbase, formula);
    }
}
/* ---------------------------------------------------------- */

check_new_subenvironment_criteria (list_of_new_se, formula)
     LIST_OF_SE list_of_new_se;
     FORMULA formula;
{
  BINDING_VARIABLE b_var;
  LIST_OF_SE se;
  if (is_restricted (formula)) {
    /* Read, until write_end_signal is read, from all subenvironments in
       se_list,  the incoming messages of the subenvironments to be addressed
       and add them to 'list_of_new_se'.
    */

    /* Add all subenvironment, which have an image bound to a already
       bound binding variable in the formula and are not in se_list, to
       'list_of_new_se'.
    */
```

```
    while (b_var = get_next_bound_variable (formula) != NULL) {
      while (obj = get_next_object (b_var->object_list))
if (is_image (obj))
  if (! is_element (se_list, obj->se_id))
    add_to_list (list_of_new_se, obj->se_id);
    }
  }
  else {
    /* Assing all subenvironments  which belong to the
       project's domain, but are not yet element of 'se_list' to
       'list_of_new_se'.
    */
    while ((se = get_next_se (domain)) != NULL)
      if (! is_element (se_list, se))
add_to_list (list_of_new_se, se);
  }
  /* Notify the subenvironments about new subenvironment.*/
}
/* ---------------------------------------------------------- */

get_all_bound_objects (universe, formula)
     OBJECT_LIST universe ;
     FORMULA formula;

{
  OBJECT_LIST obj_list;
  OBJECT obj;
  FORMULA subformula;

  switch (formula->type){
  case AND:
    for (subformula = formula->child; subformula != NULL;
 subformula = subformula->next) {
      get_all_bound_objects (universe, subformula);
      universe = bound_objects;
      bound_objects = EMPTY;
    }
    bound_objects =  universe;
    return;

  case OR:
    for (subformula = formula->child; subformula != NULL;
```

```
 subformula = subformula->next) {
      get_all_bound_objects (universe, subformula);
      obj_list = union (obj_list, bound_objects);
      bound_objects = EMPTY;
    }
    bound_objects =  obj_list;
    return;

  case NOT:
    get_all_bound_objects (universe, formula->child);
    replace_images_cse (universe);
    replace_images_cse (bound_objects);
    bound_objects = set_complement (universe, bound_objects);
    return;

  PREDICATE:
    replace_images_cse (universe);
    if (has_bound_operand (formula))
      replace_images_cse (bound_operand (formula));
    switch (formula->operator) {
    case MEMBER:
      return (get_member (universe, formula));
    case ANCESTOR:
      return (get_ancestor (universe, formula));
    case  LINKTO:
      bound_var = bound_operand (formula);
      complete_bound_operand (bound_var, attribute (formula, bound_var));
      return (get_linkto (universe, formula));
    case  ASSOCIATIVE_PREDICATE:
      if (semi_joint (formula)){
return (get_assoc_semijoint_predicate (universe, formula, se_list));
}
      else {
return (get_associative_predicate (universe, formula));
      }
    }
  }
}
/* --------------------------------------------------------- */

complete_bound_operand_cse (bound_var, att)
     OBJ_LIST bound_var;
```

```
     ATTRIBUTE att;
{
  LIST_OF_SE se, se_log_list;
  int done = FALSE;
  OBJECT obj1, obj2;
  MESSAGE message;

  /* (1) Send the images of all objects 'obj1' in bound_var to the
         subenvironment 'se', if there is an object 'obj2' in 'obj1->att'
 with 'obj2->se_id == se'. The list 'se_log_list' keeps track that
 an object is not sent more than once to the same se.
  */
  while ((obj1 = get_next_obj (bound_var)) != NULL) {
    se_log_list = NULL;
    while ((obj2 = get_next_obj (obj->att)) != NULL)
      if (is_image (obj2) && in_list (se_list, obj2->se_id) &&
  ! in_list (se_log_list, obj2->se_id)) {
write_port (obj2->se_id, obj1->obj_id, cse_id);
add_to_list (se_log_list, obj2->se_id);
      }
  }

  /* (2) Read incoming messages until all subenvironments have sent a
 write_end_signal. If message
 body is an image, bind it to 'bound_var'.
  */
  while (! done) {
    message = read_port;
    if (message->body ==  write_end_signal) {
      add_to_list (se, message->sender);
      if (se == se_list)
done = TRUE;
    }
    else if (is_image (message->body))
      add_to_list (bound_var, message->body);
  }

  /* (3) Send continue message to all subenvironments.
   */
}
/* ---------------------------------------------------------- */
```

```
replace_images_cse (obj_list)
{
  LIST_OF_SE se;
  int done = FALSE;
  MESSAGE message;
  /* (1) Write all images in obj_list to the port and remove them
         afterwards. (see replace_images_se)
  */

  /* (2) Read incoming messages until from all se a
 write_end_signal message is received. If message
 body is an image, bind its local object to
 the object list.
  */
  while (! done) {
    message = read_port;
    if (message->body ==  write_end_signal) {
      add_to_list (se, message->sender);
      if (se == se_list)
done = TRUE;
    }
    else if (is_image (message->body))
      add_to_list (obj_list, object_of_image (message->body));
  }

  /* (3) Send continue message to all subenvironments.
   */
}
/* --------------------------------------------------------- */

get_assoc_semijoint_predicate (universe, formula)
{
  integer extremal;
  LIST_OF_INTEGER integer_list;

  /* We only treat the case (a.att1 cond_operator b.att2)
     where a.att1 is the binding variable and b is already bound.
  */
  obj_list = object_list (formula->right_symb);
  switch (cond_operator (formula)) {
  case "<":
    /* (1) Assign the maximum value of the att2 values of the objects in b
```

```
          to the integer 'extremal'.
      (2) Read incoming messages until write_end_signal from every se
          has arrived.
  Message bodies contain integers. If the message body's
  integer is smaller than extremal replace it.
   */
   write_port (se, extremal, CSE); /* for all se in se_list */

   /* (3) Replace formula->right_symb with the extremal value and call the
          get_assoc_semijoint_predicate function.
   */
   replace (formula->right_symb, extremal);
   return (get_associative_predicate (universe, formula));
   break;

 case "<=":
 case ">":
 case ">=":
   break;

   /* Case "<=", ">" and ">=" are equivalently executed as case "<"
      with regard to the operator semantics.
   */
 case "==":
   /* (1) Add all occurring values in b.att2 to the global_list.
      (2) Read incoming values and add them to the global_list
          until an write_end_signal from all se in se_list has been read.
      (3) Send the global_list to all se in se_list.
      (4) Bind all object 'o' of the corresponding class and with 'o.att1'
          in global_list to a.
   */
   break;
 case "!=":
   /* (1,2,3) as in the "==" case.
      (4) Bind all object 'o' of the corresponding class in the universe
      and with 'o.att1' not in global_list to a.
    */
   break;
 }
}
```

/* ———————————————————————————— */

```
/* ---------------------------------------------------------------- */
/* QUERY PROCESSOR FUNCTION OF THE COORDINATING SUBENVIRONMENT (CSE) */

/* Global objects:
   domain   - list of the subenvironments belonging to the same development
              project
   cse_id   - id of this (coordinating) subenvironment
   se_list  - current list of subenvironment involved in the rule processing
   local_ob - objectbase of this (the cse) subenvironment
*/


/* Signals:
   write_end_signal
   continue_signal
   end_init_rest_rule
*/

build_characterized_binding_CSE (bindings)
     BINDING bindings;
{
  LIST_OF_SE list_of_new_se, se;

  for (binding = bindings, level = 0;
       binding != NULL;
       binding = binding->next, level++)
    {
      check_new_subenvironment_criteria (list_of_new_se, binding->formula);
      while ((se = get_next_se (list_of_new_se)) != NULL) {
initiate_rest_rule (se, level);
insert_list (se_list, se);
read_port (cse_id, end_init_rest_rule, se);
      }
      /* Send continue_signal to all subenvironments.
       */
      get_all_bound_objects (binding->formula, local_objectbase, formula);
    }
}
/* --------------------------------------------------------- */

check_new_subenvironment_criteria (list_of_new_se, formula)
     LIST_OF_SE list_of_new_se;
```

```
     FORMULA formula;
{
  BINDING_VARIABLE b_var;
  LIST_OF_SE se;
  if (is_restricted (formula)) {
    /* Read, until write_end_signal is read, from all subenvironments in
       se_list,  the incoming messages of the subenvironments to be addressed
       and add them to 'list_of_new_se'.
    */

    /* Add all subenvironment, which have an image bound to a already
       bound binding variable in the formula and are not in se_list, to
       'list_of_new_se'.
    */
    while (b_var = get_next_bound_variable (formula) != NULL) {
      while (obj = get_next_object (b_var->object_list))
if (is_image (obj))
  if (! is_element (se_list, obj->se_id))
    add_to_list (list_of_new_se, obj->se_id);
    }
  }
  else {
    /* Assing all subenvironments  which belong to the
       project's domain, but are not yet element of 'se_list' to
       'list_of_new_se'.
    */
    while ((se = get_next_se (domain)) != NULL)
      if (! is_element (se_list, se))
add_to_list (list_of_new_se, se);
  }
  /* Notify the subenvironments about new subenvironment.*/
}
/* -------------------------------------------------------- */

get_all_bound_objects (universe, formula)
     OBJECT_LIST universe ;
     FORMULA formula;

{
  OBJECT_LIST obj_list;
  OBJECT obj;
  FORMULA subformula;
```

```
 switch (formula->type){
 case AND:
   for (subformula = formula->child; subformula != NULL;
subformula = subformula->next) {
     get_all_bound_objects (universe, subformula);
     universe = bound_objects;
     bound_objects = EMPTY;
   }
   bound_objects =  universe;
   return;

 case OR:
   for (subformula = formula->child; subformula != NULL;
subformula = subformula->next) {
     get_all_bound_objects (universe, subformula);
     obj_list = union (obj_list, bound_objects);
     bound_objects = EMPTY;
   }
   bound_objects =  obj_list;
   return;

 case NOT:
   get_all_bound_objects (universe, formula->child);
   replace_images_cse (universe);
   replace_images_cse (bound_objects);
   bound_objects = set_complement (universe, bound_objects);
   return;

 PREDICATE:
   replace_images_cse (universe);
   if (has_bound_operand (formula))
     replace_images_cse (bound_operand (formula));
   switch (formula->operator) {
   case MEMBER:
     return (get_member (universe, formula));
   case ANCESTOR:
     return (get_ancestor (universe, formula));
   case  LINKTO:
     bound_var = bound_operand (formula);
     complete_bound_operand (bound_var, attribute (formula, bound_var));
     return (get_linkto (universe, formula));
```

```
    case  ASSOCIATIVE_PREDICATE:
      if (semi_joint (formula)){
return (get_assoc_semijoint_predicate (universe, formula, se_list));
}
      else {
return (get_associative_predicate (universe, formula));
      }
    }
  }
}
/* ---------------------------------------------------------- */

complete_bound_operand_cse (bound_var, att)
     OBJ_LIST bound_var;
     ATTRIBUTE att;
{
  LIST_OF_SE se, se_log_list;
  int done = FALSE;
  OBJECT obj1, obj2;
  MESSAGE message;

  /* (1) Send the images of all objects 'obj1' in bound_var to the
         subenvironment 'se', if there is an object 'obj2' in 'obj1->att'
 with 'obj2->se_id == se'. The list 'se_log_list' keeps track that
 an object is not sent more than once to the same se.
  */
  while ((obj1 = get_next_obj (bound_var)) != NULL) {
    se_log_list = NULL;
    while ((obj2 = get_next_obj (obj->att)) != NULL)
      if (is_image (obj2) && in_list (se_list, obj2->se_id) &&
  ! in_list (se_log_list, obj2->se_id)) {
write_port (obj2->se_id, obj1->obj_id, cse_id);
add_to_list (se_log_list, obj2->se_id);
      }
  }

  /* (2) Read incoming messages until all subenvironments have sent a
 write_end_signal. If message
 body is an image, bind it to 'bound_var'.
  */
  while (! done) {
    message = read_port;
```

```
    if (message->body ==  write_end_signal) {
      add_to_list (se, message->sender);
      if (se == se_list)
done = TRUE;
    }
    else if (is_image (message->body))
      add_to_list (bound_var, message->body);
  }

  /* (3) Send continue message to all subenvironments.
   */
}
/* -------------------------------------------------------- */

replace_images_cse (obj_list)
{
  LIST_OF_SE se;
  int done = FALSE;
  MESSAGE message;
  /* (1) Write all images in obj_list to the port and remove them
         afterwards. (see replace_images_se)
  */

  /* (2) Read incoming messages until from all se a
 write_end_signal message is received. If message
 body is an image, bind its local object to
 the object list.
  */
  while (! done) {
    message = read_port;
    if (message->body ==  write_end_signal) {
      add_to_list (se, message->sender);
      if (se == se_list)
done = TRUE;
    }
    else if (is_image (message->body))
      add_to_list (obj_list, object_of_image (message->body));
  }

  /* (3) Send continue message to all subenvironments.
   */
}
```

```
/* ------------------------------------------------------------ */

get_assoc_semijoint_predicate (universe, formula)
{
  integer extremal;
  LIST_OF_INTEGER integer_list;

  /* We only treat the case (a.att1 cond_operator b.att2)
     where a.att1 is the binding variable and b is already bound.
  */
  obj_list = object_list (formula->right_symb);
  switch (cond_operator (formula)) {
  case "<":
    /* (1) Assign the maximum value of the att2 values of the objects in b
           to the integer 'extremal'.
       (2) Read incoming messages until write_end_signal from every se
           has arrived.
   Message bodies contain integers. If the message body's
   integer is smaller than extremal replace it.
    */
    write_port (se, extremal, CSE); /* for all se in se_list */

    /* (3) Replace formula->right_symb with the extremal value and call the
           get_assoc_semijoint_predicate function.
    */
    replace (formula->right_symb, extremal);
    return (get_associative_predicate (universe, formula));
    break;

  case "<=":
  case ">":
  case ">=":
    break;

    /* Case "<=", ">" and ">=" are equivalently executed as case "<"
       with regard to the operator semantics.
    */
  case "==":
    /* (1) Add all occurring values in b.att2 to the global_list.
       (2) Read incoming values and add them to the global_list
           until an write_end_signal from all se in se_list has been read.
       (3) Send the global_list to all se in se_list.
```

```
       (4) Bind all object 'o' of the corresponding class and with 'o.att1'
           in global_list to a.
    */
    break;
  case "!=":
    /* (1,2,3) as in the "==" case.
       (4) Bind all object 'o' of the corresponding class in the universe
       and with 'o.att1' not in global_list to a.
     */
    break;
  }
}
```

/* ———————————————————————— */

7 Appendix 2

Implementation of the query processing on the subenvironments running a rest rule.

```
/* -------------------------------------------------------------- */
/* QUERY PROCESSOR FUNCTION OF THE SUBENVIRONMENT (SE) */

/* Global objects:
   domain   -  list of the subenvironments belonging to the same development
               project
   cse_id   - coordinating subenvironment id
   se_id    - id of this subenvironment
   local_ob - objectbase of this (se_id) subenvironment
   se_list  - current list of subenvironment involved in the rule processing
*/

/* Signals:
   write_end_signal
   continue_signal
   end_init_rest_rule
*/

build_characterized_binding_SE (bindings, level)
     BINDING bindings;
     int level;
{
  for (binding = bindings, i = 0;
```

```
       binding != NULL;
       binding = binding->next, i++)
    {
      if (i <= level) {
binding->variable = NULL;
if (i == level) {
  /* Next increment of i will be the level of the processing.
     Therefore, notify the cse about the end of initialization.
  */
  write_port (cse_id, end_init_rest_rule, se_id);}
else {
  if (i > (level + 1)) {
    /* Send subenvironments ids to the cse, which have images
       in already bound binding variables in binding->formula
       and are not in se_list.
       Send write_end_signal to cse.
       Read update for se_list from cse.
     */
  }
  read_port (cse_id, continue_signal, se_id);
  get_all_bound_objects (binding, local_objectbase, formula);
}
/* -------------------------------------------------------- */

get_all_bound_objects (universe, formula)
     OBJECT_LIST universe ;
     FORMULA formula;

{
  OBJECT_LIST obj_list;
  OBJECT obj;
  FORMULA subformula;

  switch (formula->type){
  case AND:
    for (subformula = formula->child; subformula != NULL;
 subformula = subformula->next) {
      get_all_bound_objects (universe, subformula);
      universe = bound_objects;
      bound_objects = NULL;
    }
    bound_objects =  universe;
```

```
    return;

  case OR:
    for (subformula = formula->child; subformula != NULL;
 subformula = subformula->next) {
      get_all_bound_objects (universe, subformula);
      obj_list = union (obj_list,  bound_objects);
      bound_objects = EMPTY;
    }
    bound_objects =  obj_list;
    return;

  case NOT:
    get_all_bound_objects (universe, formula->child);
    replace_images_se (universe);
    replace_images_se (bound_objects);
    bound_objects = set_complement (universe, bound_objects);
    return;

  PREDICATE:
    replace_images_se (universe);
    replace_images_se (bound_operand (formula));
    switch (formula->operator) {
    case MEMBER:
      return (get_member (universe, formula));
    case ANCESTOR:
      return (get_ancestor (universe, formula));
    case  LINKTO:
      return (get_linkto (universe, formula));
    case  ASSOCIATIVE_PREDICATE:
      if (is_semi_joint (formula)){
return (get_assoc_semijoint_predicate (universe, formula));
}
      else {
return (get_associative_predicate (universe, formula));
      }
    }
  }
}
/* ------------------------------------------------------- */

complete_bound_operand_se (bound_var, att)
```

```
     OBJ_LIST bound_var;
     ATTRIBUTE att;
{
  LIST_OF_SE se;
  int done = FALSE;
  OBJECT obj1, obj2;
  MESSAGE message;
  /* (1) Write images in bound_var->att on the port.
         'not_yet_sent' should prevent that an object is sent twice.
         It probably need a list who keeps track.
  */
  while ((obj1 = get_next_obj (bound_var)) != NULL)
    while ((obj2 = get_next_obj (obj->att)) != NULL)
      if (is_image (obj2) && not_yet_sent (obj1))
write_port (obj2->se_id, obj1->obj_id, cse);

  /* (2) Notify cse of end of broadcasting
   */
  write_port (cse_id, write_end_signal, se_id);

  /* (3) Read the incoming message. If message
 body is an image, bind it to bound_var.
     (4) Wait for continue_signal from cse then return.
  */
  while ( (message = read_port) != (se_id, continue_signal, cse))
    if (is_image (message->body))
      add_to_list (bound_var, (message->sender, message->body));
  return;
}
/* -------------------------------------------------------- */

replace_images_se (obj_list)
{
  /* (1) Write all images in obj_list to the port and remove them
         afterwards.
     (2) Notify cse of end of broadcasting
     (3) Read the incoming message.  If message
 body is an image, bind its local object to
 the object list.
     (4) Wait for continue_signal from cse then return.
         image, bind its local object to the object list.
  /* (1) */
```

```
  while ( (obj = get_next_obj (obj_list)) != NULL) {
    if (is_image (obj)) {
      write_port (obj->se_id, obj->obj_id, se_id);
      remove(obj_list, obj);
    }
  }
  /* (2) */
  write_port (cse_id, write_end_signal, se_id);
  /* (3,4) */
  while ( (message = read_port) != (se_id, continue_signal, cse))
    if (is_image (message->body))
      add_to_list (obj_list, object_of_image (message->body));
  return;
}
/* ---------------------------------------------------------- */

complete_bound_operand_se (bound_var, att)
     OBJ_LIST bound_var;
     ATTRIBUTE att;
{
  LIST_OF_SE se, se_log_list;
  int done = FALSE;
  OBJECT obj1, obj2;
  MESSAGE message;

  /* (1) Send the images of all objects 'obj1' in bound_var to the
         subenvironment 'se', if there is an object 'obj2' in 'obj1->att'
 with 'obj2->se_id == se'. The list 'se_log_list' keeps track that
 an object is not sent more than once to the same se.
  */

  while ((obj1 = get_next_obj (bound_var)) != NULL) {
    se_log_list = NULL;
    while ((obj2 = get_next_obj (obj->att)) != NULL)
      if (is_image (obj2) && in_list (se_list, obj2->se_id) &&
  ! in_list (se_log_list, obj2->se_id)) {
write_port (obj2->se_id, obj1->obj_id, se_id);
add_to_list (se_log_list, obj2->se_id);
      }
  }
  write_port (cse_id, write_end_signal, se_id);
  /* (2) Read incoming messages until continue_signal arrives. If message
```

```
 body is an image, bind it to 'bound_var'.
  */
  message = read_port;
  while (message->body !=  write_end_signal) {
    add_to_list (bound_var, message->body);
    message = read_port;
  }
}
/* -------------------------------------------------------- */

get_assoc_semijoint_predicate (universe, formula)
{
  integer extremal;
  LIST_OF_INTEGER integer_list;
  /* We only treat the case (a.att1 cond_operator b.att2)
     where a.att1 is the binding variable and b is already bound.
  */
  obj_list = object_list (formula->right_symb);
  switch (cond_operator (formula)) {
  case "<":

    /* (1) Assign the maximum value of the att2 values of the objects in b
           to the integer 'extremal'.
       (2) write_port (cse_id, extremal, se_id);
       (3) extremal = read_port (se_id, integer, cse_id)
       (3) Replace formula->right_symb with the extremal value and call the
           get_assoc_semijoint_predicate function.
    */
    replace (formula->right_symb, extremal);
    return (get_associative_predicate (universe, formula));
    break;

  case "<=":
  case ">":
  case ">=":
    break;
    /* Case "<=", ">" and ">=" are equivalently executed as case "<"
       with regard to the operator semantics.
    */

  case "==":
    /* (1) Send all occurring values in b.att2 to cse.
```

```
       (2) Wait for global list from the cse.
    */
    list_of_values = read_port (se_id, list of integer, cse);
    /* (3) Bind all object 'o' of the corresponding class and with 'o.att1'
           in global_list to a.
    */
    break;

  case "!=":
    /* (1,2) as in the "==" case.
       (3) Bind all object 'o' of the corresponding class in the universe
           and with 'o.att1' not in global_list to a.
     */
  }
}
/* -------------------------------------------------------- */
```

References

[1] Toni A. Bünter. Optimisation of the characterisic function in MARVEL rules. Technical report: 1993.

[2] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases.* MacGraw-Hill computer science series. MacGraw-Hill, 1984.

[3] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in marvel: Dynamic binding of parameters. *IEEE Expert*, 7:26–32, 1992.

[4] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, pages 40–49, May 1988.

Figures

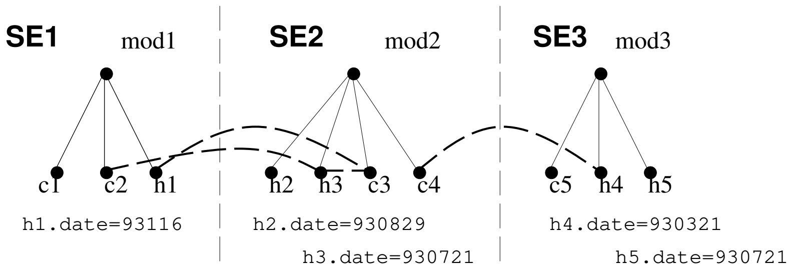

Figure 1: Example objectbase.