Optimization of the Binding Mechanism of the Characteristic Function in MARVEL

Toni A. Bünter

Technical Report CUCS-029-93
COLUMBIA UNIVERSITY

Abstract

The applied binding mechanism in MARVEL 3.1 for the characteristic function checks all instances of a given class against the binding formula, regardless of the actual structure of the formula and its predicates. This can cause unnecessary computation overhead while executing a rule. This report displays a more advanced mechanism considering relational information between objects, the structure of the binding formula and optimizing rewriting of the binding formula.

Contents

1 Introduction **1**
1.1 MARVEL Rules 1
1.2 The Rest of this Paper 1

2 Query Section **1**
2.1 Syntax 1
2.2 Semantics of the Characteristic Function 2

3 Query Processing in MARVEL 3.1 **3**
3.1 The Algorithm 3
3.2 Performance 5

4 Walkthrough Binding Mechanism **6**
4.1 The Algorithm 6
4.2 AND Optimization 9
4.3 First Results 11

5 Future Work **12**

6 Appendix **12**

References **14**

Figures **15**

1 Introduction

MARVEL is a rule-based development environment [2]. 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 which build the development process. Their activation can be triggered off by a user or by forward/backward chaining of another rule [1].

1.1 MARVEL Rules

A rule consists of five sections: the *parameter list*, the *characteristic function*, the *property list*, the *activity* and the *assertions*. The parameter list and the characteristic function shape the *query section* of a rule. The property section checks for possible backward chaining. The activity calls a tool with bound objects as arguments. The assertions update the objectbase depending on the outcome of the tool execution. (More detailed information is available in the above mentioned literature references.)

The concern of this work is the query section. After a precise definition of the syntax and semantics of the characteristic function, we discuss the evaluation mechanism of the characteristic function and display an optimized mechanism using the actual structure of the binding formula.

1.2 The Rest of this Paper

Section 2 defines the query sections' syntax and semantics precisely. Section 3 displays the algorithm used in the MARVEL 3.1 version. Section 4 introduces the walkthrough mechanism and the AND-optimization. Performance considerations and results are included. The appendix displays an example of a rule with an objectbase definition and an object hierarchy.

2 Query Section

2.1 Syntax

The query section of a rule consists of the parameter list and the characteristic function. By the activation of a rule, objects are bound to the parameters. In addition, the characteristic function binds objects which are used by the property list, the activity and/or the assertions. The characteristic function consists of a ordered list of bindings. A *binding* is basically a pair of a *binding variable* and a

binding formula. The binding variable belongs to a class, the *binding class.* The MSL syntax of a binding is:
(*quantor binding class* ?*variable* `SUCHTHAT` (*formula*))
The quantor has no meaning during the evaluation of the binding.

The binding formula consists of predicates and logic operators. It is defined recursively:

- `member(?a.att, ?b)`, `ancestor(?a, ?b)`, `linkto(?a.att, ?b)` and `linkto(?a.att, nil)` are *navigational* predicates. A formula of the form `(?a.att` *operator* `?b.att)` or `(?a.att` *operator const*) is an *associative* predicate, whereas *operator* is a comparison operator and *const* a constant. `?a.att` refers to the attribute `att` of objects in `?a`.
- If f_i for $i = 1...n$ are predicates or binding formulae, `(AND` $f_1...f_n$`)` and `(OR` $f_1...f_n$`)` are binding formulae.
- If f is a binding formula, `(NOT` f`)` is a binding formula.

2.2 Semantics of the Characteristic Function

The bindings in the characteristic function are evaluated in sequence. Each binding binds objects to the binding variable which belongs to the class or a subclass of the binding variable, and hold against the binding formula. A variable which appears in the binding formula is either the binding variable itself or an already bound binding variable or parameter, also *bound operand* called. The meaning of a bound operand is a set of objects.

Let us look now at the evaluation of a binding B_j with B_j= (*Q C V* `SUCHTHAT` *f*) belonging to a list of n bindings. Except for the binding variable f, all occurrences of the variables in the binding formula f are substituted with the sets of objects which are bound to them (formally: $\mathcal{S}(f)$). After this substitution, the binding formula represents a boolean function with the single variable V.
Every object o in the objectbase which is of the same class or a subclass of the class of C, is bound to the binding variable V if $\mathcal{S}(f(o)$ is true. The value of $\mathcal{S}(f(o)$ is defined by the following definitions:

- `member(?a.att,` o`)` is true $\Longleftrightarrow \exists s(s \in ?\mathtt{a} \wedge o \in s.att)$.
- `member(`o`.att, ?b)` is true $\Longleftrightarrow o.att \cap ?\mathtt{b} \neq \emptyset$.
- `ancestor(?a,` o`)` is true $\Longleftrightarrow ?\mathtt{a} \cap anc(\{o\}) \neq \emptyset$.
- `ancestor(`o`, ?b)` is true $\Longleftrightarrow \exists s(s \in ?\mathtt{b} \wedge o \in anc(s))$.
- `linkto(?a.att,` o`)` is true $\Longleftrightarrow \exists s(s \in ?\mathtt{a} \wedge o \in s.att)$.

- `linkto(`o`.att, ?b)` is true $\Longleftrightarrow o.att \cap ?\mathtt{b} \neq \emptyset$.
- `linkto(`o`.att, nil)` is true $\Longleftrightarrow o.att = \emptyset$.
- `(`o`.att` *operator const*`)` is true $\Longleftrightarrow \exists s(s \in o.att \wedge (s\ operator\ const))$.
- `(`o`.att1` *operator* `?b.att2)` is true $\Longleftrightarrow \exists s(s \in ?\mathtt{b} \wedge (o.att1\ operator\ s.att2))$.
- `(?a.att1` *operator* o`.att2)` is true $\Longleftrightarrow \exists s(s \in ?\mathtt{a} \wedge (s.att1\ operator\ o.att2))$.
- `NOT` f`(o)` is true $\Longleftrightarrow \neg f(o)$.
- `(AND` f_1`(o)` ... f_n`(o))` is true $\Longleftrightarrow f_1(o) \wedge ... \wedge f_n(o)$.
- `(OR` f_1`(o)` ... f_n`(o))` is true $\Longleftrightarrow f_1(o) \vee ... \vee f_n(o)$.

Notes: By the evaluation of the navigational predicates, the $o.att$ and $s.att$ represent the so called *large attributes*, which are sets of objects. Therefore, set operation can be applied.
The associative predicates use so called small attributes referenced by $o.att$ and $s.att$.
The not further specified function $anc(o)$ produces the set of ancestor objects in the actual objectbase.

3 Query Processing in MARVEL 3.1

In this section we discuss the binding mechanism running in the MARVEL 3.1 version. First, we present the algorithm and then take a closer look at the performance characteristics.

3.1 The Algorithm

After the binding of the parameter the list of bindings in the characteristic function is evaluated in sequence. We display hier a reduced and simplified fragment of the query processing function `get_all_bound_objects` with the recursive function `check_obj_against_formula` which evaluates an single object against the binding formula.

Input:

```
CLASS class; /* binding class */
FORMULA formula;
```

The set variables which are referred in the binding formula, are accessible through the binding formula itself.

Output:

```
OBJECT_LIST bound_objects;
```

The binding variable overtakes the objects bound to the set variable `bound_objects`, after the final return of the `get_all_bound_objects` function.

Algorithm:

```
OBJECT_LIST bound_objects;

get_all_bound_objects (class)
     CLASS class;

{
  OBJECT_LIST obj_list;
  OBJECT obj;
  CLASS subclass;

  obj_list = get_obj_list (class);
  for (obj = get_next_obj (obj_list); obj != NULL;
       obj = get_next_obj(obj_list))
    if (check_obj_against_formula (formula, obj) == TRUE)
      add_obj_to_list (obj, bound_objects);
  for (subclass = get_subclass (class); subslass != NULL;
       get_next_class (subclass))
    get_all_bound_objects (subclass);
}


BOOLEAN check_obj_against_formula (formula, object)
     FORMULA formula;
     OBJECT object;
{
  FORMULA subformula;
  switch (formula->type) {
  case AND:
    for (subfomula = formula->child; subformula != NULL;
 subformula = subformula->next) {
      if (check_obj_against_formula (formula, object) == FALSE)
```

```
return (FALSE);
    }
  case OR:
    for (subfomula = formula->child; subformula != NULL;
 subformula = subformula->next) {
      if (check_obj_against_formula (formula, object) == TRUE)
return (TRUE);
    }
  case NOT:
    if  (check_obj_against_formula (formula->child, object) == TRUE)
      return (TRUE);
    if (check_obj_against_formula (formula->child, object) == FALSE)
      return (FALSE);
  PREDICATE:
    switch (formula->operator) {
    case MEMBER:
      return (check_member (formula, object));
    case ANCESTOR:
      return (check_ancestor (formula, object));
    case  LINKTO:
      return (check_linkto (formula, object));
    case  ASSOCIATIVE_PREDICATE:
      return (check_associative_predicate  (formula, object));
    }
  }
}
```

The main idea of the algorithm is the sequential checking of all objects belonging to the binding class or to one of its subclass. The function `check_obj_against_formula` recursively checks these objects along the structure of the binding formula. The recursion ends with predicate check functions as `check_member`. The successfully checked objects are finally added to the `bound_objects` list.
The called function `get_obj_list` returns the list of objects of the given class. (Every class can have more then one subclass. We assume, for every class exists a list of all direct subclasses. These subclasses are reached through the function `get_next_class`. `get_subclass` returns the first subclass of a class.)

3.2 Performance

To investigate the performance of the above algorithm, we show first an example (see appendix 1 and figure 1). Supposing the compile rule is triggered with c1

in a environment with 30 objects of the class `MINIPROJECT`, 600 `C-files` and 400 `H-files`. The first binding touches and checks 30 objects while the second binding touches 400 h-files.
In general, we can calculate the number of touched objects with:

$$\sum_{i=0}^{i=number\ of\ bindings} |objects\ of\ binding\ class_i|$$

If we further assume that the number of objects of each class is growing equally, the performance time is linearly growing with the number of class instances.
To be able to compare this mechanism to our approach we define an `efficiency factor` for $binding_i$ as:

$$\frac{bound\ objects}{touched\ objects}$$

In our example the finally bound objects are h1, h2, h3 and h4. That means, an efficiency factor of the second binding is $\frac{4}{400} = 0.01$.

4 Walkthrough Binding Mechanism

In this section we display our approach of the binding mechanism.
The feasibility of this approach is based on the fact, that the implementation of objects, as it is done for MARVEL 3.1, provides a direct access to the parent object, the linked objects and the children objects.

4.1 The Algorithm

The former defined mechanism 'walks' through the objectbase and compares all the objects of a certain classes against the formula. Instead of traversing the objectbase, we propose a *formula walkthrough*. Two set variables, *history variables* called, keep track of the walkthrough history. The set variable *universe* represents the current search range, whereas the set variable *bound_objects* collects the valid objects. At the beginning of a binding, the universe comprises all objects which belong to the binding class or a subclass while the set variable *bound_objects* is empty. During the formula walkthrough, *bound_objects* is assigned as follows;

- Predicate p: $bound_objects := \{o | o \in universe \wedge p(o)\}$
- `(AND` $f_1 ... f_n$`)` : $bound_objects := universe \cap (\bigcap_{i=1}^{n} bound_objects_{f_i})$
- `(OR` $f_1 ... f_n$`)` : $bound_objects := universe \cap (\bigcup_{i=1}^{n} bound_objects_{f_i})$
- `(NOT` f`)` : $bound_objects := \{o | o \in universe \wedge o \notin bound_objects_f\}$

Input:

```
OBJECT_LIST universe = /* objects of class or subclass of binding class */;
CLASS class; /* binding class */
FORMULA formula;
```

The set variables which are referred in the binding formula, are accessible through the binding formula itself.

Output:

```
OBJECT_LIST bound_objects;
```

Algorithm:

```
 /* global variables */
CLASS class;
OBJECT_LIST bound_objects;

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 (subfomula = 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 (subfomula = 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);
    bound_objects = set_complement (universe, bound_objects);
    return;

  PREDICATE:
    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:
      return (get_associative_predicate (universe, formula));
    }
  }
}
```

All defined and mentioned functions are working on the global variable *bound_objects*. The set variable *universe* is referred by value. The algorithm is depth first and recursive.

Performance

The computation of the second binding of the compile rule in the same example environment touches four objects, all of them are valid. The efficiency factor for the second binding is therefore 100 times bigger than the factor of the current mechanism.

The formal performance analysis of the walkthrough algorithm is difficult and is given here by approximation. And the following assumptions reduce the general

case: the average number of objects per member attribute is am, per link attribute al, and the maximum height of the actual objectbase is mh.

- $P(member(a.att, b)) = am * |a|$ if b is the binding variable.
- $P(member(a.att, b)) = |b|$ if a is the binding variable.
- $P(ancestor(a, b)) \leq am^h * |a|$ if b is the binding variable.
- $P(ancestor(a, b)) \leq h * |b|$ if a is the binding variable.
- $P(linkto(a.att, b)) = al * |a|$ if b is the binding variable.
- $P(linkto(a.att, b)) = al * |b|$ if a is the binding variable.
- $P((a.att\ operator\ const)) = |universe|$.
- $P((a.att1\ operator\ b.att2)) = |universe| \times |a|$ if b is the binding variable.
- $P((a.att1\ operator\ b.att2)) = |universe| \times |b|$ if a is the binding variable.
- $P(NOTf) = |objects\ of\ class/subclass\ of\ the\ binding\ class| + P(f)$.
- $P((ANDf_1...f_n)) = \sum_{i=1}^{n} P(f_i)$.
- $P((ORf_1...f_n)) = \sum_{i=1}^{n} P(f_i)$.

The formula can not be given explicitly because it depends on the binding variables of the former evaluated bindings and/or the parameter variables. Nevertheless, the predicate computation shows a linear dependency between performance and of the average size of large attributes and, a linear dependency between performance and the size of the *universe* for associative predicates. The next section makes use of later mentioned fact.

4.2 AND Optimization

The commutativity of the binding formula's `AND` operator offers a performance optimization through reordering of arguments.

The `AND` section in the walkthrough algorithm shows that after the evaluation of the first argument f_1 the set *bound_objects* is used as the universe for the next subformula. Therefore, the size of the universe depends on the former evaluated `AND`-subformula. This consideration and the commutativity of the `AND` operator leads to the conclusion that the subformula, which produces a small `bound_objects` set, should be evaluated as the first subformula. We assume the navigational predicates produces a very small number of bound objects, compared with the whole objectbase.

Upon this assumption we build a formula definition: the *navigational formula*.

- Navigational predicates are navigational formulae.
- The formula (AND $f_1 ... f_n$) is navigational if f_1 is navigational.
- The formula (OR $f_1 ... f_n$) is navigational if all f_i are navigational.

Among the not navigational formulae we can define the class of potentially navigational formulae, which are transformable into navigational formulae through interchange of AND subformula:

- Navigational predicates are potentially navigational formulae.
- The formula (AND $f_1 ... f_n$) is potentially navigational if at least one argument f_i exists which is potentially navigational.
- The formula (OR $f_1 ... f_n$) is potentially navigational if all f_i are potentially navigational.

The interchange is recursive and moves the navigational formulae in the AND section at the first place. The algorithm is as follows:

```
optimize_AND (formula)
     FORMULA formula;

{
  OBJECT_LIST obj_list;
  OBJECT obj;

  switch (formula->type){
  case AND:
    for (subfomula = formula->child; subformula != NULL;
 subformula = subformula->next) {
      if (optimize_AND (subformula) == NAVIGATIONAL) {
        put_at_front (formula, subformula);
        return (NAVIGATIONAL);
      }
    }
    return (NON_NAVIGATIONAL);

  case OR:
    for (subfomula = formula->child; subformula != NULL;
        subformula = subformula->next) {
      if (optimize_AND (subformula) == NOT_NAVIGATIONAL) {
        return (NOT_NAVIGATIONAL);
      }
    }
```

```
    return (NAVIGATIONAL);
  case NOT:
    return (NOT_NAVIGATIONAL);
  PREDICATE:
    return (NON_NAVIGATIONAL);
  }
}
```

Implementation

The implementation of the walkthrough mechanism emerged to the following tasks:

- The history variables had to be installed into the `get_all_bound_object` and `check_obj_against_formula` which collapsed to the `get_all_bound_object` function.
- The `get_member`, `get_ancestor` and `get_linkto` are new functions. Instead of comparing object, they bind objects which are directly accessible through the already bound variable to *bound_objects*.
- The function `get_associative_predicate`, similar as in the former `get_all_bound_object` implementation, sequences through the universe and uses `check_associative_predicate` to filter the appropriate objects.
- The `AND` optimization function `optimize_AND` is called in the `AND` section of `get_all_bound_objects`. Because the reordering of subformulae has only to be done once, it can be moved into the *loader* part.

AVL Trees as Data Structure for Sets

In the former implementation, the structure of a variable which keeps track of the bound objects is list structure. For operations as union and difference of sets which are highly used in the new implementation, the list structure is inappropriate. Therefore, we designed a structure, called *stacktree*, which combines an AVL tree with a stack. The stack keeps track of an actual position in the tree for a sequential read process of the tree elements. The algorithms for the AVL trees are taken from [3].

4.3 First Results

First results on very small objectbases (less than 50 objects) were of random nature. No significant improvement could be shown. Responsibility for this disappointing results is mainly the enhanced complexity of the algorithm and the more sophisticated data structure.

Significant improvements showed up by running the new algorithm on a medium sized objectbase (more than 1000 objects). In our sample, we let run 26 rules out of the rule set {compile, deposit, mk_flatsrc, build, arch}. The performance is measured in time and in touched objects, including retrivial of objects from the stacktree structure.

MARVEL 3.1 algorithm :	4992 obj	244 ms
Walkthrough algorithm :	2219 obj	143 ms

5 Future Work

Next step in the Marvel/Oz project is the geographically distribution of a development environment. Every site, called subenvironment, supports one Marvel object servers. It is assumed, that object hierarchies belong to the same server. However, links between objects can be drawn beyond the range of a subenvironment. The walkthrough algorithm is directly applicable for computing member and ancestor predicates, in contrast to the current MARVEL 3.1 implementation. Encouraged through this results further investigation for the case of link and associative predicates will be done.

6 Appendix

Throughout this work we used the compile rule of the C-Marvel environment specification as an example.

```
* parameter list *
compile [?c:CFILE]:
* characteristic function *
  (and (exists MINIPROJECT ?mp suchthat (member [?mp.files ?c]))
       (forall HFILE ?h suchthat (linkto [?c.hfiles ?h])))
   :
* property list *
   (and (?c.analyze_status = Analyzed)
     no_chain (?c.compile_status = NotCompiled)
     no_forward (?c.reference_status = Referenced))
```

```
* activity *
  { LOCAL cfile_compile ?c.contents ?c.object_code
        ?h.contents ?c.history ?c.options }
* assertions *
(and (?c.compile_status = Compiled)
  no_chain (?c.object_time_stamp = Current Time));
  no_chain (?c.compile_status = ErrorCompiled);
```

The objectbase definition is a simplified part out of the C-Marvel environment specification:

```
MINIPROJECT :: superclass BUILT, PROTECTED_ENTITY;
      files            : set_of FILE;
      config           : string;
end

CFILE :: superclass FILE;
      analyze_status   : (NotAnalyzed, ErrorAnalyzed, Analyzed);
      contents         : text = ``.c'';
      hfiles           : set_of link HFILE;
      config           : string = ``MSL'';
end

HFILE :: superclass FILE;
      contents   : text = ``.h'';
end
```

Figure 1 shows an instantiated objectbase (object hierarchy) of the above defined data model.

References

[1] 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.

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

[3] Niklaus Wirth. *Algorithmen und Datenstrukturen*. Teubner, 1983.

Figures

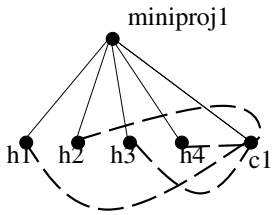

Figure 1: Example objectbase.