Set Differentiation: A Method for the Automatic Generation of Filtering Algorithms

Fawzia Derrough Darche * and Jean-Luc Dormoy †

* Laboratoire MASI, Institut Blaise Pascal
Université P. et M. Curie, URA CNRS 818
4, place Jussieu 75252 Paris Cedex 05 France
E-mail : Fawzia.Derrough@masi.ibp.fr

† EDF R&D Center
IMA-TIEM
1, avenue du Général De Gaulle
92141 Clamart Cedex, France
E-mail : Jean-Luc.Dormoy@der.edfgdf.fr

Abstract

We present a method and its implementation, the system GAP, we developed for the automatic generation of filtering algorithms. The evaluation of left-hand sides of rules relies on the matching of condition elements with Working Memory elements. The filtering is the inference engine phase that performs this matching. Our automatic generation method is based on set differentiation which we have defined taking into account both qualitative and quantitative aspects. We present GAP's architecture and show how generic filtering algorithm skeletons are built using set differentials.

1. Introduction

The main component of a production system is its matching or filtering algorithm that computes rule instantiations. The main research in production systems focused on the performance of such algorithms and led to systems such as SNARK [12], TANGO [16] and OPS5 [3] with a compromise between execution time and space required.

The evaluation of left-hand sides of rules relies on the matching of condition elements with Working Memory (WM) elements. It has been identified by McDermott, Newel and Moore as the memory support [13]: an indexing scheme indicates precisely which subset of WM partially matches each condition element. In RETE [9] and TREAT [14], the two well known matching algorithms of OPS, this information is maintained in the alpha-memories. SNARK and BOOJUM [7] manage a set of indexes.

The problem of filtering algorithms has been studied in specific cases, particularly when condition elements in left-hand sides of rules are simple. Indeed, filtering algorithms are applicable when condition elements represent patterns of WM elements, satisfied when there exists at least one WM element that **matches** them. For instance, in OPS a WM element consists of a class name followed by a list of attribute-value pairs. In SNARK, a WM element is an object-attribute-value triplet. After the addition or the deletion of facts in the WM, updating memory supports in the previous systems is immediate and easy. That is why we qualify such condition elements as simple conditions. The fundamental feature of these conditions lies in the fact that their update depends only on the added or removed WM element and does not depend on all the WM elements.

As current production system languages allow one to use only pattern elements in left-hand sides of rules, this may lead to large production system programs. Another drawback of such programs is their illegibility. If we allow the use of complex conditions in left-hand sides of rules, we facilitate the encoding of knowledge in the expert system as the expressive power of rule language is

This work has been partially supported by Electricité de France

high. By complex conditions we mean conditions that are not pattern elements whose evaluation lies on their match with WM elements. Complex conditions are first order predicate calculus formulas with universal and existential quantifiers. Examples of complex conditions are taken from LCA rule language based on constraints and actions [2] where a complex condition is called constraint scheme. The notion of numerical constraint defined in [19] and [20] has been extended to the specification of constraints on different data types such as relations.

Classical filtering techniques referred above are not applicable to filter complex conditions. A general algorithm filtering all complex constraint schemes is not efficient. Indeed, as each constraint scheme is particular, it requires a specialized filtering algorithm. Instead of thinking about and writing a filtering algorithm each time we define a new constraint scheme, we propose a method to automatically generate filtering algorithms for constraints.

The method we use for the automatic generation of filtering algorithms is based on set differentiation which we defined taking into account both quantitative and qualitative aspects. Constraint schemes can be considered as queries addressed to the WM. We transform these queries from the predicative form into a set-based one. This representation has the advantage to be intuitive and allows us to have a "geometric" vision of constraint schemes. The qualitative and quantitative set differentials computed are then used to built the filtering algorithms. We have developed a prototype of a filtering algorithm generator, called the system GAP.

The paper is organized as follows: in section 2 we present the set-based representation. Sections 3 and 4 respectively present quantitative and qualitative set differentiation. In Section 5, we show how we use set differentiation to generate filtering algorithms and describe such algorithm skeletons. Section 6 presents GAP's architecture.

2. Constraint set-based representation

Constraint schemes can be viewed as queries addressed to the WM. We define a query language based on sets. This set-based query language has the advantage to be intuitive and allows us to have a “geometrical” vision of queries. Indeed, the expression of variations of constraint schemes solutions seems to be simpler in a set-based representation. So, we transform constraint schemes from a predicative form to a set-based representation. These variations are computed using set differentials.

2.1. Definition of set-based representation

Let F be a formula. Two cases may occur.

1- Assume F has free variables. Let $v_1, \ldots, v_n$ these variables. The set-based representation of F is a set defined as follows: $a = \{ (v_1, \ldots, v_n)$ such that $F(v_1, \ldots, v_n) \}$. Intuitively, the set-based representation associated with a formula defines its free variables. We use these variables as a superscript of the set-based representation.

2- Assume F has no free variable, the set-based representation a is a boolean proposition between sets.

So when the constraint scheme is a formula with free variables, the transformation leads to the set representing the values of the free variables for which the constraint is true. When the constraint scheme is a closed formula, the transformation results in a boolean expression involving sets. When modifications occur in the WM, we have to determine the variations produced on condition elements. We are able to characterize these variations on the set-based representations of constraint schemes. That is why we have developed set differentiation presented in next sections.

The construction of the set-based representation of a predicative formula is a recursive operation. It starts from the transformation of the atomic formulas that compose it. Given basic rules which provide set-based representations of atomic formulas, and rules for the transformation of conjunction, disjunction, negation and quantified formulas, we are able to obtain the set-based representation of any predicative formula. Intuitively, we can see the correspondence between conjunction and intersection, and between disjunction and union. The set-based representation of the negation of a given formula *f* is the complementary set of the set-based formula of *f*, if the latter is a set. More Formally, and as an example, let us give the complete transformation rule of conjunction. We can find all transformation rules in [6].

2.2. Conjunction transformation rule

Let F_1 and F_2 be two formulas, a_1 and a_2 the set-based representations respectively associated with F_1 and F_2.

1- If F_1 and F_2 have no free variable, the set-based representation associated with $F_1 \wedge F_2$ is: "$a_1 \wedge a_2$".

2- If F_1 has no free variable and F_2 is not a closed formula, the set-based representation of $F_1 \wedge F_2$ is: "if a_1 then a_2 else $\varnothing$.

3- If F_1 has free variables and F_2 is a closed formula, the set-based representation associated with $F_1 \wedge F_2$ is: "if a_2 then a_1 else $\varnothing$".

4- Assume both F_1 and F_2 have free variables. Let $u_1, \ldots, u_p, v_1, \ldots, v_q$ the free variables of formula F_1, and $u_1, \ldots, u_p, w_1, \ldots, w_r$ those appearing in F_2 such that the intersection of sets $\{v_1, \ldots, v_q\}$and $\{w_1, \ldots, w_r\}$is empty. The set-based representation of $F_1 \wedge F_2$ is:

$$\left(a_1^{u_1, \ldots, u_p, v_1, \ldots, v_q} \times (U^r)^{w_1, \ldots, w_r}\right) \cap$$
$$\left(a_2^{u_1, \ldots, u_p, w_1, \ldots, w_r} \times (U^q)^{v_1, \ldots, v_q}\right)$$

where U is the Universal set, and U^n is the Cartesian product of set U n times. To make the intersection possible between two set-based representations, each of them is "completed" with the Cartesian product with the Universal set.

Example

Let us take the example of RXN; a constraint scheme defined in LCA. Let *r* be a binary relation, (x, y) a couple of *r*. We say *y* is the successor of *x* in *r*. *x* is said to be the predecessor of *y* in *r*. RXN(r x) is true if and only if *x* has successors in relation *r*, and all successors of *x* have a unique predecessor, namely *x*. We will use this constraint scheme to illustrate all the filtering algorithm generator steps. The first order formula representing constraint RXN is:

$\exists$ y r(x , y) $\wedge \neg \exists$ t, z (r(t, z) $\wedge$ r(x, z) $\wedge \neg$ (t = x))

where *x* is a constant and *y*, *z*, and *t* are quantified variables. The transformation of the whole formula starts with the atomic formula $r(x, y)$. Its associated set-based representation is $Succ(x, r)^y = \{ y \in U$ such that $(x\ y)$ is a couple of $r\}$, which defines variable *y*. That is why we have used variable *y* as a superscript of set $Succ(x, r)$. The transformation of $\exists\, y\, r(x , y)$ results in $Succ(x, r)^y \neq \emptyset$. The set-based representation associated with $r(t, z)$ is $Set(r)^{t\ z}$ where $Set(r)$ is the set of couples of relation *r*. The transformation of $r(t, z) \wedge r(x, z)$ is $Set(r)^{t\ z} \cap (Succ(x, r)^z \times U^t)$. RXN set-based representation is:

$Succ(x, r)^y \neq \emptyset$ and not $(((Set(r)^{t\ z} \cap$
$(Succ(x, r)^z \times U^t)) \cap (Compt(\{x\}^t) \times U^z)) \neq \emptyset)$

Compt(A) is the complementary set of set A.

3. Quantitative set differentiation

High-level optimization techniques, called "formal differentiation" by Paige and Schwartz [15], generalize the classical method of "reduction in strength" [4] well known in optimization techniques of code generation [1]. The basic idea of formal differentiation is to replace repeated costly computations of set theoretic expressions whose arguments change only slightly between successive computations by computations of "incremental" or "differential" expressions that are less expensive to evaluate. An algebraic approach to formal differentiation of a class of set theoretic expressions was presented in [17] as a set of relations.

We also establish set differential relations used as rules of differentiation. The difference between our quantitative set differentiation and the work referred above lies in the fact that we distinguish two differentials associated with a set, a positive one and a negative one. In the previous work, it is possible to see what is added to a set and what is removed from it only once a set differential is computed. To avoid such a drawback we define qualitative set differentiation that allows us to determine if a given set expression increases and/or decreases. Our aim is to build a system capable of solving problems in the absence of detailed quantitative information. The idea is to do some qualitative reasoning based on the qualitative description of the studied system which permits the prediction of its future states. In the field of artificial intelligence, several qualitative models have been developed such as the confluence model defined by De Kleer and Brown [5], the qualitative process theory proposed by Forbus [8] as well as the qualitative simulation introduced by Kuipers [11]. In the case of set differentiation, we will see how qualitative reasoning may avoid the computation of quantitative set differentials by directly showing that it will be empty.

3.1. Set operations

Let U be the universal set. P(U) is the set of all the subsets of U. All the manipulated sets are structured sets, i.e. their elements are n-tuples. n is called the dimension of the set. Let us consider the following set operations : union, intersection, Cartesian product, complementation and projection. The first two operations apply when sets have the same dimension d and the resulting set is of dimension d. Cartesian product applies even if sets do not have the same dimension. Let d_2 and d_3 be the dimensions of sets S_2 and S_3 respectively. The dimension of the Cartesian product $S_1 = S_2 \times S_3$ is equal to the sum of d_2 and d_3. Each n-tuple of S_1 is the result of the concatenation of a d_2-tuple of S_2 and a d_3-tuple of S_3.

Set projection, denoted by $\prod_p$, is an unary operation defined as follows:

$$\prod_p : P(U) \times N^p \rightarrow U$$
$$(S, n_1, n_2, \ldots, n_p) \rightarrow S_1 = \prod_{n_1, n_2, \ldots, n_p}(S)$$

The idea behind this operation is that we take a set S and remove some of the components of its elements. If S is a set of dimension d, we let $\prod_{n_1, n_2, \ldots, n_p}(S)$, where n_j are distinct integers in the range 1 to d, denote the projection of S onto components $n_1, n_2, \ldots, n_p$, that is, the set of p-tuples $a_1a_2\ldots a_p$ such that there is some d-tuple $b_1b_2\ldots b_d$ in S for which $a_j = b_{n_j}$ for j = 1, 2, …p.

3.2. Positive and negative set differentials

Let S be a set. Assume that S is known at a given moment and that it changes. The modifications of S consist of additions of new elements and deletions of old ones. Thus, we distinguish set increase from set diminution. The positive set differential of a set S, denoted $d^+(S)$, is the set of elements added to S. The negative set differential of S, denoted $d^-(S)$, is the set of elements removed from S. A set differential d(S) is defined by the couple $(d^+(S), d^-(S))$. d(S) could also be defined as the union of $d^+(S)$ and $d^-(S)$, i.e., $d(S) = d^+(S) \cup d^-(S)$.

Positive and negative set differentials of a given set S verify the following properties:

- $d^+(S) = d(S) - S$ (1)
- $d^-(S) = d(S) \cap S$ (2)
- $d^-(S) \subset S$ (3)
- $d^+(S) \cap S = \emptyset$ (4)
- $d^+(S) \cap d^-(S) = \emptyset$ (5)

(5) is a consequence of (3) and (4)

Suppose that we periodically observe sets at time t_1, t_2, …, t_n, and that modifications of sets happen between these moments. We can write:

$$S^{i+1} = (S^i + d^+(S^i)) - d^-(S^i) \qquad (6)$$

or

$$S^{i+1} = S^i \,\Delta\, d(S^i) \qquad (7)$$

where S^i and S^{i+1} represent S at moments i and i+1 respectively, and Δ is the symmetric set difference. In the context of production systems, the periods correspond to inference engine cycles.

In differential calculus, given a numerical function f: $R^n \rightarrow R^p$, and the value of f at a given point x, the value of the derivative of f, denoted f', computed at this point allows one to approach the value of f at a point close to x using the following formula: $f(x + h) = f(x) + f'(x) \times h + o(h)$. (6) is the analog of the previous formula.

Two differentials are associated with a given set S, a positive one and a negative one, while a function f is associated with a unique derivative at each point, if f is continuous at this point. The main difference between set differentials and function derivatives lies in the fact that the first ones provide exact values whereas the second ones provide approximate values.

Given two numerical functions f and g, and their respective derivatives f' and g', we have a set of relations which allow us to compute the derivative of the sum of f and g, the composition of f and g, etc. …. For instance, we have $(f + g)' = f' + g'$. By analogy, we can establish relations for the computation of set differentials for the set operations described above. We can then speak about differentials of set union, set intersection, Cartesian product, complementation, and projection.

Let S_1 and S_2 be two sets and suppose both of them are modified. A positive set differential and a negative set differential are associated with S_1 and S_2, and are denoted respectively $d^+(S_1)$, $d^-(S_1)$, $d^+(S_2)$ and $d^-(S_2)$. We have developed a set of relations for the computation of differentials of set operations, and combinations of these relations can then be used to compute set differentials for a large class of set expressions. For instance, the positive differential of set union is obtained by the following relation:

$$d^+(S_1 \cup S_2) = (d^+(S_1) - S_2) \cup (d^+(S_2) - S_1) \qquad (8)$$

This relation means that the elements added to the union of two sets are those added to S_1 and that are not in S_2, as well as those added to S_2 but that are not in S_1.

The negative set differential of the union is given by the following relation:

$$\begin{aligned} d^-(S_1 \cup S_2) = {} & (d^-(S_1) \cap d^-(S_2)) \cup \\ & (d^-(S_1) - (S_2 \cup d^+(S_2))) \cup \\ & (d^-(S_2) - (S_1 \cup d^+(S_1))) \qquad (9) \end{aligned}$$

This relation means that the elements removed from the union $S_1 \cup S_2$ are:

- those removed from S_1 and S_2,
- those removed from S_1 but that are neither in S_2 nor in its increase $d^+(S_2)$,
- those removed from S_2 but that are neither in S_1 nor in its increase $d^+(S_1)$,

We have established similar relations for the other set operations in [6]. In practice, as we have both addition and deletion of WM elements, we compute positive and negative set-differentials for each operation.

4. Qualitative set differentiation

In the previous section, we established set differential rules to compute positive and negative set differentials. These rules are quantitative rules as they allow us to compute what exactly is added to or removed from a set. In this section, we introduce qualitative set differentiation and qualitative set differential denoted d_{qual}. With qualitative set differentiation we are interested in knowing the way a set is modified. In many situations, this kind of knowledge is sufficient. Moreover, it prevents us from useless computations of set differentials. Indeed, when we determine that a set increases and does not decrease, it is useless to compute its negative differential and then to discover that it is empty.

4.1. Qualitative values

To carry out qualitative reasoning on sets, four qualitative values can be associated with the qualitative set differential dqual(S) of a given set S as follows:

1- 0 when $d^+(S)$ and $d^-(S)$ are both empty, S remains the same,

2- + when $d^+(S)$ may not be empty and $d^-(S)$ is empty, in this case, we know that S does not decrease,

3- - when $d^+(S)$ is empty and $d^-(S)$ may not be empty, so S does not increase,

4- ? both $d^+(S)$ and $d^-(S)$ may not be empty, S may increase and decrease.

As for quantitative set differentials, we distinguish the qualitative set differential dqual(S) of a given set S in the case of addition or deletion of WM elements.

4.2. Qualitative set operations

In arithmetic, addition and multiplication are usually viewed as qualitative operations. For instance, we know that when we add two positive numbers, the result is a positive number. The multiplication of a positive number and a negative one results in a negative number. We can do the same thing for set operations. For instance, results of qualitative set intersection, union and cartesian product are presented in the following Table.

dqual(S_2) \ dqual(S_1)	0	+	-	?
0	0	+	-	?
+	+	+	?	?
-	-	?	-	?
?	?	?	?	?

Table 1. Qualitative set intersection, union and cartesian product

From Table 1, we can deduce that symbol ? is the absorbing element of set intersection. When S_1 or S_2 may have simultaneous increase and decrease, their intersection may simultaneously increase and decrease. When neither S_1 nor S_2 decreases, $S_1 \cap S_2$ does not decrease. So, we do not need to compute negative differential $d^-(S_1 \cap S_2)$, it is empty. Conversely, it is useless to compute $d^+(S_1 \cap S_2)$ when neither S_1 nor S_2 increases as the intersection $S_1 \cap S_2$ may not increase.

Example

Let us now compute quantitative and qualitative differentials of sets appearing in RXN set-based representation. Though we have presented quantitative set differentiation before the qualitative one, in practice qualitative set differentials are calculated before quantitative ones for the reasons given before. It is obvious that when we add a couple in relation r, Succ(x, r) may increase and cannot decrease, so $dqual_{add}(Set(r)^{t\,z}) = +$, $dqual_{add}(Succ(x, r)^y) = +$ and $dqual_{add}(Succ(x, r)^z) = +$, but $dqual_{add}(U^t) = 0$. From the previous table we can deduce that $dqual_{add}(Succ(x, r)^z \times U^t) = +$, and $dqual_{add}(Set(r)^{t\,z} \cap (Succ(x, r)^z \times U^t)) = +$, We finally have : $dqual_{add}((Set(r)^{t\,z} \cap (Succ(x, r)^z \times U^t)) \cap (Compt(\{x\}^t) \times U^z)) = +$. We have then to compute $d^+_{add}\,(Succ(x, r)^y)$ and $d^+_{add}\,((Set(r)^{t\,z} \cap (Succ(x, r)^z \times U^t)) \cap (Compt(\{x\}^t) \times U^z))$. Let suppose couple (a, b) is added in relation r. So, we have, for instance, $d^+_{add}\,(Set(r)^{t\,z}) = \{a\} \times \{b\}$ and $d^+_{add}\,(Succ(x, r)^y) =$ (if a = x then $\{b\}^y$ else $\varnothing^y$). Using quantitative set differentials of set intersection and cartesian product, we are able to determine the positive set differential of $(Set(r)^{t\,z} \cap (Succ(x, r)^z \times U^t)) \cap (Compt(\{x\}^t) \times U^z)$.

5. The generation of filtering algorithms

In this section, we show how we use qualitative and quantitative set differentials in the automatic generation of filtering algorithm process. The generated algorithms are expressed in a high level language, namely Descartes [10], a functional and set-based language. It allows one to use quantifiers and sets.

5.1 Knowledge for filtering algorithm generation

The generation of filtering algorithms is obtained from declarative specifications of the constraint schemes. Indeed, constraint schemes are specified by first order predicate calculus formulas. This initial specification does not contain any imperative information. Nevertheless, as we are interested in particular algorithms, we have implicit knowledge concerning the algorithms that have to be generated. This knowledge allows us to build skeleton of filtering algorithms as we see below.

First, we must state precisely what we expect from a filtering algorithm. A filtering algorithm associated with a constraint scheme must deduce the consequences of the modifications produced on the constraint by elementary operations of addition and deletion of WM elements. This implies that the filtering algorithm must distinguish the case where a new WM element is added from the case where a WM element is deleted. The filtering algorithm consists in a test instruction concerning the nature of the action processed. It must also have a boolean parameter indicating whether the action processed is an addition or a deletion of fact.

As the studied constraint schemes deal with binary relations, WM elements added or deleted are couples of relations. The filtering algorithms must have input parameters representing the couple of the modified relation. As said before, constraint schemes split up into two classes. The first one is the class of constraint schemes whose parameters are all constants. The second one contains constraint schemes which have at least one variable parameter. In the first case, the predicative form of the constraint scheme is a closed formula. The transformation of this kind of formula into a set-based representation leads to an expression composed of set constraints connected by "or" and "and". A constraint of this type is true or false depending on the current WM. Thus, it has a state, and this state has to be maintained when the WM is modified. In the second case, the predicative formula associated with a constraint scheme is a logical formula containing free variables. Such a constraint is true if there exist objects in the WM for which the constraint is true when the variables are instantiated by these objects. In this case, our aim is to incrementally maintain the set of the values that satisfy the constraint by computing the elements added to this set and those removed from it at each inference engine cycle. This leads us to distinguish two kinds of filtering algorithm skeletons. Due to space limitation, we present here filtering algorithm skeletons corresponding to the first category of constraint schemes. The second kind of filtering algorithm skeletons are presented in [6].

5.2 Filtering algorithm skeleton

When the constraint scheme has no variable parameter, the first order predicate formula associated with it is a closed formula, i.e., all the variables appearing in it are quantified. Moreover, the formula is composed of subformulas that are all closed, connected by the logical connectors "and" and "or". We have then a global state corresponding to the global formula and partial states corresponding to the different closed subformulas composing the global formula. The number of partial states is equal to the number of subformulas that compose the predicative representation of the constraint scheme. If we assume that the predicative form of a constraint scheme is expressed using the conjunction, the negation, and the existential quantifier, its set-based representation is of the form: $e \neq \emptyset$, where e is a set expression. Indeed, the transformation of an atomic formula into a set-based representation results in a set, the transformation of a conjunction results in set intersection, the transformation of the negation results in the complementary set, and the transformation of the existential quantifier leads to "$\neq \emptyset$".

Instead of evaluating the partial states after each modification, we maintain them. The global state, for its part, is maintained through its partial states. A filtering subfunction is then associated with each closed subformula. The goal of these filtering subfunctions is to maintain the associated partial states. Thus, the filtering algorithm associated with a constraint scheme having no variable parameter is composed of a main function and numerous subfunctions. We will first present the main function and then we describe the subfunctions and their headings.

5.2.1. Main filtering function

The main filtering function must have the following parameters in its heading:

- a boolean variable, namely *state*, representing the global state of the constraint,
- a variable, denoted *ra*, representing the modified relation,
- two variables, denoted *a* and *b*, representing the components of the added or removed couple. We suppose that all the manipulated relations are binary relations,
- variables, denoted $pc_1, \ldots, pc_m$, representing the constant parameters of the constraint scheme. The types of the parameters (constant or variable) are given in the description of the constraint scheme,
- a boolean variable denoted *add*. When *add* is true, it means that the main filtering function is called after the addition of couple (*a*, *b*) in relation *ra*, and when it is false, the call is done after the deletion of couple (*a*, *b*),
- boolean variables, denoted $state_1, \ldots, state_r$, representing the partial states of the constraint scheme.

The function returns the new values of variables $state_1, \ldots, state_r$ and *state*. The heading of the main filtering function is: Filter(*state*, *ra*, *a*, *b*, $pc_1, \ldots, pc_m$, *add*, $state_1, \ldots, state_r$).

The state of the constraint may change only when the name of the modified relation is one of those mentioned in the constraint scheme. It is, in this case, necessary to determine the consequences of this modification on the partial states because they may change. On the other hand, if the modified relation ra is different from all the relations mentioned in the constraint scheme, then none of the partial states nor the global state are modified. The old values of the partial states as well as the global state are returned.

In the first case, i.e., when one of the mentioned relations is modified, the new values of the partial states must be calculated by executing their associated functions. The calls for the different functions evaluating the partial states are done. In these calls, the name of the modified relation, the added or removed couple, the constant parameters of the constraint scheme, as well as the state associated with the called function are the arguments. The new partial states and the global one are returned. The main filtering function skeleton is presented in Figure 1.

```
Filter(state, ra, a, b, pc1, …, pcm, add, state1, …, stater)
if  (ra = r1) or (ra =r2) or … (ra = rn)
then  nstate1 = Fil1(ra, a, b, pc1, …, pcm, add, state1),
        ........
        nstater = Filr(ra, a, b, pc1, …, pcm, add, stater),
        (nstate1, …, nstater, state_tree)
else (state1, …, stater, state)
```

Figure 1 : main filtering function skeleton associated with a constraint scheme without variable parameter

State_tree is the expression of the global state by means of the partial ones (for instance, state_tree = ($state_1$ and $state_2$) or $state_3$). Fil_i, i = 1, r, represent filtering subfunctions. The main function acts as a first filter which permits to eliminate, in a straight off manner, all the operations known to have no consequence on the constraint, as they do not affect any of the relations mentioned in the constraint scheme. However, when the modification may change the state of the constraint, we must determine how it acts on the partial states.

Steps composing the generation process, namely the construction of the set-based representation, the computation of the qualitative and quantitative differentials, as well as the construction of the filtering functions, have been implemented as a module of GAP, called the generation module. GAP's architecture is presented in the next section.

Example

The main function, produced by GAP, corresponding to constraint scheme RXN is presented in Figure 2.

```
Filter_RXN(state, ra, a, b, add, r, x, state1, state2)
if (ra = r) then nstate1 = Fil1(ra, a, b, add, r, x, state1)
                 nstate2 = Fil2(ra, a, b, add, r, x, state2)
                 (nstate1, nstate2, nstate1 and not nstate2)
else (state1, state2, state)
```

Figure 2 : RXN main filtering function

$State_1$ represents the state of set constraint $Succ(x, r)^y \neq \emptyset$. $State_2$ represents the state of set constraint $((Set(r)^{t\,z} \cap (Succ(x, r)^z \times U^t)) \cap (Compt(\{x\}^t) \times U^z)) \neq \emptyset$. GAP generates two subfunctions Fil_1 and Fil_2. The first one is associated with $state_1$, and the second with $state_2$.

5.2.2. Filtering subfunctions

The heading of a filtering subfunction is of the form: Fil_i(*ra*, *a*, *b*, pc_1, …, pc_m, *add*, *state*).

As said before, functions Fil_i are associated with set constraints of the form $s \neq \emptyset$, where s is any set expression. We are interested in the following changes of the set constraint:

- when it is true, we must determine if it remains true or if it becomes false,
- when it is false, we must determine if it remains false or if it becomes true,

in both cases of addition and deletion of facts. Thus, if "state" represents the boolean value of the set constraint, the body of function Fil_i consists in the following conditional instruction:

if add then if state
    then **i_1** {i_1 determines whether the set constraint remains true or whether it becomes false after the addition of a new fact}
    else **i_2** {i_2 determines whether the set constraint remains false or whether it becomes true after the addition of a new fact}
else if state
    then **i'_1** {i'_1 determines whether the set constraint remains true or whether it becomes false after the deletion of a fact}
    else **i'_2** {i'_2 determines whether the set constraint remains false or whether it becomes true after the deletion of a fact}

What could be instructions i_1, i_2, i'_1 and i'_2? Let us take, for instance, instruction i_1. When set constraint "$s \neq \emptyset$" is true and we add a new fact, the new state to be returned in this case is the value of expression $d^-_{add}(s) \neq$

s. Indeed, if the diminution of set s is different from s, "s $\neq \emptyset$" remains true, otherwise, it becomes false. Let suppose now that the set constraint is not true before the addition. It means that s is empty. What could change the state of the set constraint is its positive differential. If it is not empty, the set constraint is false before the addition and it becomes true after it. Thus, the new state of the set constraint is given by the value of the expression $d^{+}_{add} \neq \emptyset$. Figure 3 presents filtering subfunction skeleton differentiating set constraint "s $\neq \emptyset$".

```
Fil_i(ra, a, b, pc_1, ..., pc_m, add, state)
If add   then if state
                 then (d-_add (s) ≠ s)
                 else (d+_add (s) ≠ ∅)
         else if state
                 then (d-_del(s) ≠ s)
                 else (d+_del(s) ≠ ∅)
```

Figure 3 : skeleton of subfunction differentiating set constraint "s ≠ ∅"

If we use qualitative information in this subfunction skeleton, we can immediately know the values returned by this subfunction. For instance, if the **qualitative differential** of set s, in the case of addition, equals +, i.e., s cannot decrease, we immediately know that its **negative set differential** d^{-}_{add} (s) is empty. Thus, expression (d^{-};add)(s) $\neq$ s) is true. The same reasoning applies to $d^{+}_{del}(s)$ and leads to the immediate evaluation of expression $d^{+}_{del}(s) \neq \emptyset$. Indeed, in the case of deletion, if the qualitative differential of set s equals -, i.e. s cannot increase, then its positive set differential is empty. We know that expression ($d^{+}_{del}(s) \neq \emptyset$) is false. In the two previous cases, we immediately conclude on the new state of the set constraint. We obtain an instance of the previous filtering subfunction skeleton presented in Figure 4.

```
Fil_i(ra, a, b, pc_1, ..., pc_m, add, state)
If add   then if state   then true
                         else (d+_add (s) ≠ ∅)
         else if state   then (d-_del(s) ≠ s)
                         else false
```

Figure 4 : instance of skeleton of subfunction differentiating set constraint "s ≠ ∅"

Qualitative differentials not only avoid the computation of complex quantitative set differentials that are finally empty, but they permit us, in certain cases, to immediately determine the value to be returned by filtering functions.

Example

RXN subfunction corresponding to RXN first partial state is an instantiation of subfunction skeleton of Figure 4, where s is $Succ(x, r)^{y}$. We have calculated $Succ(x, r)^{y}$ positive set differential in the previous section. This subfunction is described in next figure.

```
Fil_1(ra, a, b, r, x, add, state)
If add
    then if state  then true
                   else ( (if a = x then {b}^y else ∅^y) ≠ ∅ )
    else if state
                   then ( (if a = x then {b}^y else ∅^y) ≠ Succ(x, r)^y)
                   else false
```

Figure 5 : subfunction corresponding to the first partial state of constraint RXN

6. GAP's architecture

GAP is implemented as a rule-based system. GAP’s input is a first order calculus formula corresponding to the constraint scheme to be filtered. Its output is the associated filtering functions.

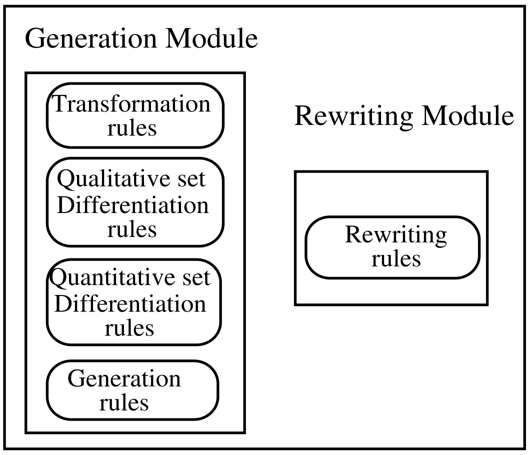

Figure 6 : GAP's Architecture

GAP is composed of five packets of rules. These packets split up into two independent modules: the generation module and the rewriting module (Figure 6). The first four packets compose the generation module, and the last one forms the rewriting module. The first packet, containing the transformation rules, builds the set-based representation of the constraint scheme. The second one is composed of rules that compute the qualitative differentials of the sets obtained in the previous step. The third packet of rules computes the quantitative set differentials. The generation rules produce the filtering functions. The last packet is composed of rewriting rules which simplify the generated functions. Indeed, these functions contain expressions that could be partially evaluated, and then could be simplified. There exist three kinds of rewriting rules. Let us explain two kinds of these rules. Some of the first kind of rewriting rules perform pure transformations of expressions. For instance, an expression of the form "(if a then b else c) op d" is rewritten "if a then b op d else c op d". In the same category of transformation rules, some ones perform local simplifications, for instance, any subexpression "a and true" is rewritten "a". GAP replaces any expression of the form "if a then b else true" by its equivalent expression "¬a or b". The second kind of rewritting rules performs contextual simplifications. Such simplifications apply on expressions taking into account the context in which these expressions appear, and outside this context, these simplifications are not applicable. To give an idea of an expression allowing contextual simplification, let us take the following instruction extracted from the filtering subfunction presented in Figure 5:

```
if add then  if state then true
                 else ( (if a = x then {b}^y else ∅^y) ≠ ∅ )
else   if state then
                 ( (If a = x then {b}^y else ∅^y) ≠ Succ(x, r) )
                 else false
```

In the case of deletion (third line), when the partial state is satisfied (i.e. state is true) before the deletion, its new value is given by the expression "If a = x then $\{b\}^y \neq$ Succ(x, r)^y else $\emptyset^y \neq$ Succ(x, r)^y ". But remember that *state* is associated with set constraint "Succ(x, r)$^y \neq \emptyset$". As this partial state is satisfied before the deletion, Succ(x, r)^y is not empty, so subexpression "Succ(x, r)$^y \neq \emptyset$" is true. GAP rewrites the previous expression "If a = x then $\{b\}^y \neq$ Succ(x, r)^y else true".

In practice, the execution of the three kinds of rewriting rules are interlaced. Figure 7 presents RXN first subfunction after the execution of the rewriting module.

```
Fil1(ra, a, b, r, x, add, state)
if add    then if state   then true
                          else (a = x)
          else if state   then ¬ (a = x) or ({b} ≠ Succ(x, r))
                          else false
```

Figure 7 : RXN first filtering subfunction after the simplifications of GAP's rewriting module

We can not consider GAP as a complete program development system like KIDS [18] for two main reasons. Firstly, the specification given to GAP is a logical formula representing the constraint scheme to be filtered. GAP produces filtering functions. KIDS derives correct and efficient programs from formal specifications which describe the problem to be solved. Secondly, algorithms produced by GAP are expressed in Descartes, a high level language, and thus are not executable programs. KIDS has data type refinement among several other components allowing it to generate executable programs. We must add to GAP modules containing knowledge concerning the choice of indexes, and the selection of adequate data structure, for instance, to produce executable programs (in C or Pascal).

8. Conclusions

In this paper, we have presented a method and its implementation in the system GAP, for the automatic generation of filtering algorithms associated with constraint schemes. Constraint schemes are first order predicate calculus formulas representing condition elements of rules. The main feature of our approach is the use of qualitative and quantitative set differentiation in the design of generic filtering function skeletons. Filtering functions corresponding to a given constraint scheme are instances of the generic filtering functions skeletons. Filtering functions corresponding to a given constraint scheme are generated once this constraint scheme is defined in the rule language. The generated functions contain boolean expressions that can be statically evaluated, i.e. expressions whose values can be known before the execution because they are independent of the data. That is the reason why we have included a rewriting module in our system. This module allows the simplifications of the generated filtering functions obtained after the generation phase. The inference engine interpreting rules containing occurrences of constraint schemes is the fusion of all generated filtering functions.

In database systems, the stored data must be consistent, i.e. in accordance with the reality it represents. An integrity constraint is in general a predicate a sub-set of the database must verify for the data to be considered coherent. Any update in the database may introduce inconsistencies and lead to controls. For performance reasons, it is necessary to reduce the cost of these controls because they involve the access to large data volume not explicitly mentioned by updates. Object oriented database management systems perform integrity constraint verifications in two steps. In the first step, the set of integrity constraints to be verified during the execution of a transaction is constructed. The filtering for the construction of this set is based on syntactic analysis of constraints and transactions. It turns out that this filtering is not selective. Indeed, constraints that would not be verified are kept in the set. In the second step, verification algorithms for each constraint of the set computed at the previous step, are automatically generated in the transaction code. To improve the filtering of integrity constraints, we are studying the application of set differentiation to this kind of problem. Indeed, it allows us to consider dynamic aspects which are not taken into account in the previous technique.

References

[[1] Aho A.V. and Ullman J.D. , "Principles of compiler design", Addison-Wesley, 1977.

[2] Beldiceanu N. "Langage de règles et moteurs d'inférences basés sur des contraintes et des actions". Application aux réseaux de Petri. PhD-Thesis, Paris 6 University . Jan 1988.

[3] Brownston L., Farrel R., Kant E., Martin N. "Programming Expert Systems in OPS5". Addison Wesley, Reading, Mass. 1985.

[4] Cocke J. and Kennedy K. , "An algorithm for reduction of operator strength", Communication of ACM, November 1977, pp 850-856.

[5] De Kleer J. and Brown J. S., "A Qualitative Physics Based on Confluence", Artificial Intelligence Volume 24, 1984, pp. 7-83.

[6] Derrough F. "Génération Automatique d'Algorithmes de Filtrage à partir de Spécifications Déclaratives : le Système GAP" PhD-Thesis, Paris 6 University. September. 1994.

[7] Dormoy J.L."Résolution qualitative : complétude, interprétation physique et contrôle. Mise en œuvre dans un langage à base de règles", PhD-Thesis, Paris 6 University , 1987.

[8] Forbus K. D., "The Qualitative Process Theory", Artificial Intelligence Volume 24, December 1984, pp. 85-167.

[9] Forgy C.L. "RETE : A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem" Artificial Intelligence. Vol. 19, pp 17-37. 1982.

[10] Ginoux B., "Génération automatique d'algorithmes par système expert, à partir de spécifications déclaratives de très haut niveau : le système COGITO", PhD-Thesis, Paris 9 University . Octobre 1991.

[11] Kuipers B., "Qualitative Simulation", Artificial Intelligence Volume 29,.1986, pp. 289-338.

[12] Laurière J.L . "SNARK, un langage déclaratif" TSI, vol.5. n°3, pp 141-172. 1986.

[13] Mc Dermott J. , Newel A. , Moore J. “The Efficiency of Certain Production System implementations”. In Pattern-Directed Inference Systems. A. Waterman, F. Hayes Roth. Academic Press. New York, 1978.

[14] Miranker D.P. "TREAT : A Better Match Algorithm for AI Production Systems". In Proceedings of AAAI. 1987.

[15] Paige B. and Schwartz J.T. "Expression continuity and the formal differentiation of algorithms", in conference record of the 4th ACM symposium of Principles of programming languages, Los Angeles, California, January 1977, pp 58-71.

[16] Rousset C. "TANGO : moteur d'inférences pour une classe de systèmes experts avec variables" Thèse de 3ème cycle. Université de Paris Sud, Octobre 1983.

[17] Sharir M., "Some observation concerning formal differentiation of set theoretic expressions", ACM Transaction on Programming Languages and systems, vol 4, n°2, April 1982, pp 196-225.

[18] Smith D., "KIDS: A Semiautomatic Program Development System", IEEE Transactions on Software Engineering, vol.16, n° 9, September 1990

[19] Steele G.L. “The definition and Implementation of a Computer Programming Language Based on constraints”. PhD-Thesis, Cambridge, 1980.

[20] Steels L. “Constraints as consultants” in Progress in Artificial Intelligence, pp.156-165, 1985.