ECAI 94

QUALITATIVE ALGORITHMICS USING ORDER OF GROWTH REASONING

Antoine Missier[1], Spyros Xanthakis[2], and Louise Travé-Massuyès[1]

1 INTRODUCTION

Software programming and software analysis [1] are among the most complex activities that humans can undertake. The programmer must follow a creative and constructive process in order to translate specifications into a low-level language and reconstitute and understand the software according to its observed behaviour.

The current techniques allowing one to analyse (or test) a piece of software can be grouped in two classes : (1) *dynamic techniques* which rely on executing the software and (2) *static analysis techniques* which, examining the software independently of the detailed execution, tend towards a global comprehension of algorithm semantics.

Symbolic execution, which is indisputably very powerful [5, 6], falls in the second category. Unfortunately, it suffers from several intrinsic limitations due to formal manipulations and it does not permit one to identify which output is influenced (in which way and direction) by which input.

As a matter of fact, all analysis techniques (dynamic and static) have limitations. When we reconstruct a program behaviour from a subset of test cases, we obtain a dynamic view of the program which is unfortunately limited, for practical reasons. On the other hand, when we try to obtain a more global view of the program by means of static analysis, conclusions are formally more sound but, either too coarse (i.e. data flow analysis [17, 18]) or too precise (i.e. symbolic execution). In this paper, we address the following question: *is it possible to reason in a generic manner (as static analysis does) while maintaining a dynamic aspect (i.e. which input influences which output and in which direction) of the program behaviour (as dynamic techniques do)?* In other words, is it possible to associate internal dynamics at each program statement while maintaining the interpretation capability?

We provide an answer to this question by proposing an essentially different formalism directly inspired from the work in Qualitative Reasoning [16]. Although QR has mainly been concerned with reasoning about physical systems so far, it provides general knowledge representation formalisms and inference algorithms like *qualitative algebras and qualitative calculus techniques, order of magnitude models, qualitative differential equations, causal influences*, etc. [13] which can easily be apply to other domains. Our idea is that the same type of qualitative reasoning which can be performed upon the behaviour of physical systems, can also be performed upon software behaviour. On the other hand, we have naturally taken inspiration from the type of qualitative reasoning commonly performed by a human programmer. The formalism that we propose in this paper, called *Qualitative Algorithmics (QA)* relies on associating to each variable a ***Valued Order of Growth (Vog)***. *Vogs* are an entirely new concept that we devised to cope with limit behaviours of functions of time (see section 3). They allow us to interpret each statement of a program in a *dynamic* manner and to perform an exaggeration reasoning which exhibits marginal behaviours.

[1] LAAS-CNRS, 7, av. du Colonel-Roche, 31077 Toulouse Cedex, France, *email: missier, louise@laas.fr.*

[2] OPL, Futuropolis, 6, rue Maryse-Hilsz, 31502 Toulouse Cedex, France.

2 THE IDEAS UNDERLYING QUALITATIVE ALGORITHMICS

Qualitative Algorithmics conveys two main ideas :

- To consider *dynamic input data* by executing a program with *Valued Orders of Growth* including a *value (v)* and an *order of growth (og)*. Both *v* and *og* are given qualitative quantity spaces for exhibiting the interesting qualitative properties and to benefit from qualitative calculus techniques, allowing the coverage of a whole set of numeric executions but avoiding the awkwardness of symbolic execution.
- To perform a type of *exaggeration reasoning* which allows us to capture the limit behaviour of a program.

If we consider a given program *P* as a dynamic system with a set of input variables and a set of output variables, then our qualitative execution method is equivalent to considering a qualitative program *[P]* in which the input variables have been provided with their own dynamics, i.e. we make the input variables vary according to temporal laws. Hence, the vector of input variables is given by $S=(f_1(t), f_2(t),..., f_n(t))$, where $f_i(t)$ are functions of time (see Section 3). Our formalism allows us to characterise qualitatively the limit behaviour of these functions, given as input, and to obtain the qualitative characterisation of the temporal functions followed by the output variables by propagating the qualitative features.

The characterisation of temporal functions is based on clustering their behaviour at infinity with respect to gauge functions of the form βt^{α}, that we represent by the couple $(v=\beta, og=\alpha)$. In our formalism, *v* and *og* take qualitative values so that we capture both the *qualitative value* and the *qualitative order of growth* of the function at infinity. The couple *(v, og)* characterising the temporal function associated with a variable *x* is called the *Valued Order of Growth* of the variable and denoted by *Vog(x)*.

A specific qualitative calculus presented in Section 3 allows us to combine Vogs such that they can be propagated through the program. Propagation is performed *in a local manner*, by interpreting every statement of the program step by step. This is referred to as a *Vog-execution*. The propagation mechanisms take into account and combine the two components *v* and *og* of the *Vogs*, making *Vog* qualitative calculus sensibly more powerful than the qualitative calculus associated with other formalisms (see sections 3 and 4).

By capturing the limit behaviour of variables and performing a type of exaggeration reasoning, *Vogs* allow us to determine invariable properties. The qualitative program *[P]* can be viewed as an equivalence class, including all programs P_i providing the same qualitative outputs, given the same qualitative inputs. For example, (`P1 : read(x); y:=x+7;`) and (`P2 : read(x); y:=x+3;`) are qualitatively equivalent, i.e. *[P1]=[P2]*, in the sense that the output y has the same limit behaviour given the same input *x*.

The paper is organised as follows : in Section 3, we present the formal model underlying QA and its algebraic properties followed, in Section 4, by the semantics of the qualitative interpretation process (showing how algorithms can be interpreted qualitatively). In Section 5, we illustrate the process of analysing software

behaviour with a detailed example. Finally, we discuss the salient features of QA.

3 FORMALIZATION OF THE ORDER OF GROWTH CONCEPT

3.1 Hardy fields

In this section we provide a mathematical structure to model the behaviour of a function of time at infinity, i.e. in a neighbourhood of infinity. As was explained in the previous section, every program input variable, is not associated to a single value but to a function of time. The idea is to deduce the asymptotic behaviour of the sequence of outputs given the asymptotic behaviour of the sequence of inputs. In the next section we show how this permits us to qualitatively analyse the program behaviour.

Hardy fields [2] constitute a natural framework for asymptotic analysis at infinity. Early works of G. H. Hardy [7] have recently motivated new interest [3], [4]. A Hardy field is a set of real valued functions (defined in a neighbourhood of infinity) which is closed under differentiation and form a field under ordinary addition and multiplication. A Hardy field contains germs of functions which are such that any algebraic differential combination[753] is of constant sign at infinity [3]. This excludes oscillatory behaviours. Examples of Hardy fields are : any subfield of $\mathbb{R}$, the field $\mathbb{R}(t)$ of rational functions with real coefficients in the variable t, the set of rational fractions formed by the monoms of the form $(ln^{\alpha}t.t^{\beta}.e^{\gamma t})$, etc.

A function belonging to a Hardy field is called **D-consistent** [3]. Most usual functions are D-consistent, except those including an explicit oscillating term like *sint* or *cost*. Another relevant property is that a Hardy field is always totally ordered, which permits us the classification of D-consistent functions relatively to their growth. Moreover any D-consistent function is always *comparable*[754] to any function of the class $(t^{\alpha})_{\alpha \in \mathbb{R}}$.

3.2 Order of growth

The notion of order of growth comes from a comparison between the behaviour of a function at infinity and a class of reference functions[755] . A natural reference is the class $(t^{\alpha})_{\alpha \in \mathbb{R}}$, for which the comparison is possible with respect to any D-consistent function.

Intuitively the **simple order of growth** [10] of a D-consistent function f is the limit of the coefficients α for which f is negligible with respect to t^{α}, and the α for which t^{α} is negligible with respect to f. It is the lower bound α such that $\lim_{t \to +\infty} \frac{f(t)}{t^{\alpha}} = 0$. For example the orders of growth of $\frac{t^4 - t^3}{t^2 - 1}$, e^t, $-\sqrt{t}$, 0, 3, arctan(t), ln(t) , 1/ln(t) are 2, $+\infty$, 1/2, $-\infty$, 0, 0, 0, 0 respectively.

The concept of simple order of growth does not allow us to distinguish the constant functions (or those having a finite limit) from *log* or *1/log*, which have nevertheless very different asymptotic behaviours. We introduce the concept of **extended order of growth** to formalise those differences. ε and ω are objects we use to represent respectively infinitesimal and infinitely big quantities. Concerning usual operations, ε and ω are supposed to be such that : $\varepsilon + \varepsilon = \varepsilon$, $\varepsilon \times \varepsilon = \varepsilon$, $\forall a \in \mathbb{R}^{+*}$, $\varepsilon \times a = \varepsilon$, and equally for ω ; we have also $\forall a \in \mathbb{R}$, $\omega + a = \omega$, $1/\varepsilon = \omega$.

Although the notations are initially inspired from non-standard analysis [8], here our symbols have not the same meaning. The object ε does not represent a single non-standard real but a halo of non-standard reals. It is a qualitative view of the set of non-standard reals. Like in classical qualitative algebras [14], the operations may lead to undetermination represented by ? : $\omega - \omega =$? represents all the possible values and $\varepsilon - \varepsilon = ?\varepsilon$ represents the set $\{-\varepsilon, 0, +\varepsilon\}$, that is, all infinitesimals.

The extended order of growth of a D-consistent function f, noted $og(f)$, is defined as the sum of the simple order of growth (α) and an infinitesimal $(-\varepsilon, 0, +\varepsilon)$, depending on whether f is negligible with respect to t^{α} ($og(f) = \alpha - \varepsilon$), equivalent to βt^{α} ($og(f) = \alpha$) or t^{α} is negligible with respect to f ($og(f) = \alpha + \varepsilon$). When the simple order of growth is infinite the extended order of growth is represented by the symbol ω (for consistency with the notations of non-standard analysis). Examples : $og(-e^t) = +\omega$, $og(t - 1/t) = 1$, $og(t \ln t) = 1 + \varepsilon$, $og(t^2/\ln t) = 2 - \varepsilon$, $og(0) = -\infty$.

The quantity $-\infty$ allows us to distinguish the order of growth of the zero function from $-\omega$. This is motivated by the fact that the order of growth of the zero function is lower than the order of growth of any function that tends "infinitely fast" towards zero (like e^{-t}).

3.3 Valued Order of Growth (Vog)

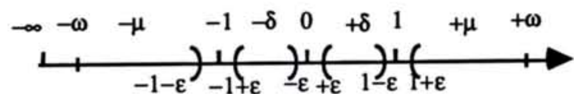
Despite the fact that extended order of growth is very powerful, it suffers from a lack of important information such as the sign of a function. We therefore enrich the concept of order of growth by adding a ponderation coefficient, called the *relative magnitude* of f and noted $v(f)$. When $f \sim \beta t^{\alpha}$, we have $og(f) = \alpha$ and $v(f) = \beta$; otherwise, we define $v(f) = sgn(f) \in \{-, 0, +\}$, since no precise information can be added to the order of growth which represents, in this case, a whole class of functions. We call the couple $(v(f), og(f))$ the **valued order of growth** of f, noted Vog(f).

Examples of Vogs : $Vog(0) = (0, -\infty)$, $Vog(-2e^{t^2}) = (-, +\omega)$ $Vog(7) = (7, 0)$, $Vog(-2t^4 + t) = (-2, 4)$, $Vog(\ln t) = (+, +\varepsilon)$, $Vog(1/\ln t) = (+, -\varepsilon)$, $Vog(\arctan t) = (1, 0)$. Oscillating functions do not admit a Vog. However, they can be bounded within an *interval of Vogs*. Vogs are provided with good properties relatively to algebraic manipulation [756][10] In particular:

- $(v_1, og_1) \otimes (v_2, og_2) = (v_1.v_2, og_1 + og_2)$
- if $og_1 > og_2$ we have: $(v_1, og_1) \oplus (v_2, og_2) = (v_1, og_1)$
- for the derivative: $\partial f = sgn(f') = sgn(f).sgn(og(f))$

3.4 Qualitative Vogs

In the formal mathematical model of Vogs presented above, v takes its values in $\mathbb{R}^*$ or $\{-, 0, +\}$ and og in $\mathbb{R} \cup \{-\infty, +\infty\} + \{-\varepsilon, 0, +\varepsilon\}$. In order to obtain a qualitative characterisation which only retains the interesting properties of functions with respect to algorithmics (like divergence, convergence, sign, convexity), we perform a discretisation which is at the basis of our qualitative model of Vogs.

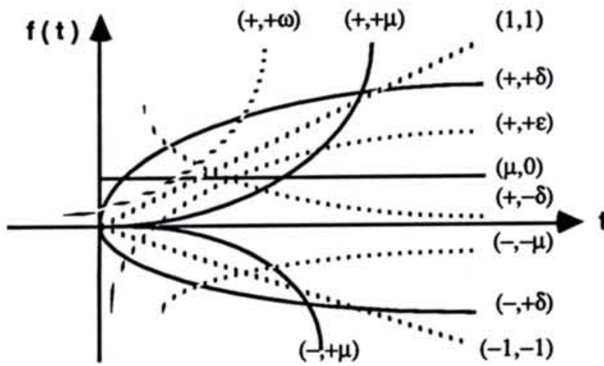


Qualitative operations are defined like in any standard qualitative algebra [14]. This qualitative representation allows us to distinguish functions of the following types : zero, all the constants, log, square, linear functions, power, exponential, and their inverse or their opposite (see figure below).

---

753 An algebraic differential combination of f is any polynom $p(f, f', \ldots, f^{(n)})$.
754 f and g are comparable if $f = o(g)$ or $g = o(f)$ or $f \sim \beta g$, $\beta \in \mathbb{R}^*$.
755 This idea of gauge functions has still been used in [19] but the general framework (partial differential equations) is not similar.

756 and also relatively to differentiation : if $og(f) \neq 0$ then $og(f') = og(f) - 1$.



QUALITATIVE INTERPRETATION OF PROGRAM STATEMENTS

A computer program can be considered as a sequence of statements, which transform the initial program state into a final state. Each state contains a list of associations: *{variable 1 = value 1; ...; variable n = value n}*. In our case, values are Vogs and states are qualitative states (*qstates*). For instance the qstate $S = \{a = [(-1, 0), (1, 0)]; b = (+, \delta)\}$ means that variable a has a Vog within the interval $[(-1, 0), (1, 0)]$ and b has a Vog equal to $(+, \delta)$. Without loss of generality, we will consider in this section, that all variables follow D-consistent functions.

Intuitively, the program initial state represents the order of growth of the inputs. Input dynamics are therefore implicitly expressed by their Vogs, directly manipulated by program statements. If, during those subsequent executions an input remains constant and another input decreases towards zero their Vogs will be respectively represented by a zero and a negative order of growth. The result of the execution permits one to determine not only which outputs are influenced by the input dynamics (their order of growth is non zero) but also the order of growth of the resulting output function.

In that way qualitative execution captures two features that are essential to the software programming task. The first one concerns data flow relationships among variables: which variable is used to evaluate which other. In our case those relationships are not static but dynamic since they are obtained according to an effective qualitative execution based on variables with a dynamic ontology. The second feature resides in the fact that the qualitative execution captures the limit behaviour of the program. Indeed it performs an exaggeration reasoning by capturing the dynamic behaviour of variables at infinity. In this sense it can be viewed as a formalization of some operations of limit testing. Limit testing [12] is considered as one of the most widespread and efficient black box testing technique since it uncovers many frequent and serious programming errors: abnormal cases that are not treated, missing code for handling extreme values (i.e. very small values), conditions that are supposed to be always true, etc. This is due to the exaggeration capability of the limit testing which permits us to sensitise rare combinations of program paths. Let us now see how qualitative execution is performed.

In the QA approach, execution is *not deterministic*. A state may therefore be a *union* of qstates. In most programs (like C, Pascal, etc.) this propagation is made by compounding three fundamental syntactic structures: assignment statements, decision and iteration statements. We briefly describe the manner those structures are qualitatively interpreted (for a more detailed description see [11]).

4.1. Assignment statements

An assignment statement has the following form: `x := E;` where `x` is a program variable and E is an arithmetic expression. The execution of the assignment replaces the value of x (in each qstate of the last state) by the result of the (qualitative) evaluation of the expression E according to the Vog's operation laws. The properties of Hardy fields ensure that the result is always a D-consistent function.

4.2. Decision statements

Decision statements have the following form: `if Cnd then p1 else p2;` where *Cnd* is a condition (e.g. $a < b$) and $p1, p2$ are statements to be executed according to the truth value of the condition.

Assume that at some point of a program we meet the condition $(a > b)$, then that a has a greater Vog than b does **not** necessarily mean that all possible numeric executions should satisfy that condition. It rather means that if the numeric values of the program entries are chosen to evolve so that their propagation makes a and b evolve according to the $Vog(a)$ and $Vog(b)$ at hand, then there exists a range from which a is definitively greater than b, thus satisfying the condition. Conditions are hence interpreted dynamically. This interpretation can be viewed as a reasoning based on exaggeration [15].

In the QA approach, conditions may not have mutually exclusive truth values because equal Vogs lead to undetermined situations. This indeterminism can be handled by associating three *(qualitative)* truth values to each condition ('+' for *True*, '-' for *False* and '?' for *Indeterminate*), thus permitting a very natural and powerful algebraic manipulation [14]. Compound conditions are evaluated by means of the two following tables summarising the qualitative logic operations *OR* and *AND*:

Or	+	-	?
+	+	+	+
-	+	-	?
?	+	?	?

And	+	-	?
+	+	-	?
-	-	-	-
?	?	-	?

The execution of a decision statement is performed using two operations, which split the current state into two partial states that, when interpreted by the condition, yield truth values qualitatively equal to '+' and '-'. The computation of the splitting functions is, in its generality, NP-complete. Nevertheless, the qualitative discretisation of the Vogs makes it straightforward.

4.3. Iterative statements

An iterative statement has the following form: `while Cnd do p;` which means that the statements represented by `p` must be executed while the condition (dynamically evaluated) is true.

Intuitively, an iteration has the same semantics as a sequence of decision statements. We therefore have to perform the splitting operation. Loop execution is represented by a function that successively transforms an initial state until a fixed point is eventually reached. QA permits a straightforward *induction process* that estimates the final Vog values of the variables that evolve during the iteration.

5 AN ILLUSTRATIVE EXAMPLE

This section illustrates, with a concrete example based on a widely used algorithm, the different possibilities of QA. More precisely we will show that QA is able to:

- *Simulate and verify some programmers' algorithmic reasoning concerning dynamic relationships among variables during execution.*
- *Propose and verify qualitative algorithmic models.*
- *Perform a wide range of efficient error detection techniques.*

The specifications of the example are the following: *Write an algorithm that examines the first N elements of an array A and calculates the mean value of the elements that are greater than or equal to a threshold T. The threshold and the elements of the array are strictly positive. The total size of the array is greater than N.*

We chose this example for two reasons. The first one is that it contains all the elementary syntactic structures. The second is that this example, although very simple, allows us to outline the usefulness and subtlety of QA reasoning. Let us consider the following (proper) implementation of the algorithm:

```
(1) i := 1;                  { Program Inputs = N, A, T }
(2) sum := 0;                { Initialisations }
(3) k := 0;
(4) M := 0;                  { Initial Mean Value }
(5) while (i <= N) do        { Loop condition      }
     begin
(6)   if (A[i] >= T) then
        begin                    { A good element is found. }
(7)     sum := sum + A[i];   { Element is accumulated     }
(8)     k := k + 1;              { and counted                }
        end;
(9)   i := i + 1;                { Next element in the array  }
     end;

(10) if (k > 0) then
(11)     M := sum / k;       { Result = Mean value = M}
```

Program statements (`PS`) are numbered from `1` to `11`. Let us consider that all the elements of the array vary at the same time and are symbolised by the variable A.

In the following, we present some forms of reasoning that a programmer may have when trying to understand the previous algorithm. Most of them [9] consist in checking in what way different variations performed on the entries (for example N, A and T) influence program outputs (for example M). For instance: *if this entry increases by following that law, which are the affected outputs and how are they affected?, which entry must I vary and in what direction to make these outputs vary in a desired direction?, etc.* When this form of reasoning is oddly used, it commonly generates current programming errors [9, 12].

• **Case 1**: Let's start with a very simple form of reasoning that comes out from the following question: *How does a linear increase of the number of terms to be summed (namely N) influence the final mean value M?*

A naive answer, based on the fact that we sum more elements, might conclude that M also increases. Of course, this is not always true, and QA provides directly the right answer in the following manner. As N increases linearly, we choose an order of growth equal to 1 (linearity), and a positive valuation, say, +. The Vog of N is therefore: $(+, 1)$. In our case, N is effectively increasing since $\partial N = [+]\oplus[1] = +\oplus+ = +$. The Vog associated to the other two entries (A and T), which remain static, are chosen as $(+, 0)$. The execution of the iteration yields a steady state:

$$S = \{\, i = (\mu, 0);\ sum = [(0, -\infty), (+, 1)];\ \kappa = [(0, -\infty), (\mu, 0)];\ N = (+, 1);\ A = (+, 0);\ T = (+, 0);\ M = (0, -\infty)\}$$

After the induction step we obtain a state where: $sum = [(0, -\infty), (+, \mu)]$; $k = [(0, -\infty), (+, 1)]$. The last condition (`k>0`) (`PS n°10`) gives two partial states. The execution of the assignment yields: $M = sum / k = [(0, -\infty), (+, \mu)]$. This convex Vog interval contains several Vogs, like: $(0, -\infty)$, $(+, -\delta)$, $(+, 1)$, etc. We conclude that M is positive or zero (which is a trivial result), and that $\partial M = ?$. The functions followed by M may be either a zero function (where $\partial M = 0$), an inverse root function (where $\partial M = -$), a linear function (where $\partial M = +$), or any other function provided that its value (at infinity) remains positive or zero.

• **Case 2**: *How does the exponential increase of the threshold T (when the other inputs are constant) influence M?*

A hasty response could conclude that M decreases since we sum fewer elements. On the other hand, another naive explanation could be that, since bigger elements are added, then the mean value must increase. Hence, we could tend to think that we are in the same case as *Case 1* in which the result is unpredictable. QA can demonstrate than none of the previous explanations are correct and that the mean value keeps its initial value.

In fact, if we give to T the Vog, $(+, \delta)$ for expressing that it exponentially increases but still remains positive, and if we give to the other parameters N and A which remain constant, the Vogs $(+, 0)$, the condition inside the loop (`PS n°6`) is never satisfied, then k keeps its initial value $(0, -\infty)$ implying that the last condition (`PS n°10`) is not satisfied and that M keeps its initial value $(0, -\infty)$.

• **Case 3**: Let's now take a more subtle problem: *What are the initial directions of change of the entries which guarantee that M decreases.*

A first solution could consist in decreasing the threshold only, which is not good. Indeed, such a solution causes all the elements A to become greater than T, and the *sum* to accumulate all the elements of the array. N and A being fixed, then the *sum* remains fixed and therefore M has zero order of growth. Another solution could consist in decreasing the elements of A. But, in that case, no element will be summed and M will remain unchanged.

We conclude that the solution is to decrease *simultaneously* the entries A and T. Let us give the Vogs $(+, -\delta)$ and $(+, -1)$ respectively, and the Vog $(\mu, 0)$ to N. Of course, there exists many other possibilities. During the qualitative execution, the condition (`A[i] >= T`) is always verified and we reach the following final state: $S = \{i = (\mu, 0);\ sum = (+, -\delta);\ k = (\mu, 0);\ N = (\mu, 0);\ A = (+, -\delta);\ T = (+, -1);\ M = (0, -\infty)\}$

The last condition (`PS n°10`) is satisfied and we obtain: $M = sum / k = (+, -\delta) / (\mu, 0) = (+, -\delta)$ that is, M decreases which is the original requirement. Note that QA also permits us to conclude that M is decreasing following the <u>same</u> order of growth than A.

In conclusion, we claim that QA allows the programmer to verify that the information concerning the limit behaviour of its *implementation*, is in accordance with the way he *mentally* conceives his algorithm. The conclusions in the example could of course be obtained by a simple reading of the algorithm because the algorithm is simple (although it is already easy to make wrong conclusions!). However, we believe that for more complex algorithms such results, directly obtained by a simple automatic qualitative execution, may be invaluable.

Let us now consider a key point of QA, that is, the possibility to propose and verify algorithmic models. This is unthinkable using classic algorithmic reasoning, because of the hugeness of the quantitative space of inputs/outputs. **Qualitative Algorithmic Models** (*QAM*) may be built from the qualitative values of inputs (Test Data) associated with their expected corresponding output. These tables may be built (of course manually) by the algorithm's designer (or specifier).

Let us give an illustration of QAM for the previous algorithm. This model is uniquely based on the expected direction of change of the output, ∂M, according to the directions of change of the inputs ∂N, ∂A, ∂T:

Specified QAM for Mean-Value	∂N	∂A	∂T	*Expected* ∂M
1.	+	*0*	*0*	*?*
2.	*0*	*0*	+	*0*
3.	*0*	-	-	-
4.	*0*	+	*0*	+
5.	-	*?*	+	*0*
6.	+	*0*	+	*0*

We have here a very poor example of QAM. The Vogs model permits us to build much <u>more precise</u> models in which, by using different orders of growth, we may vary two entries at a different rate.

QAM can be naturally used for black box testing (in which TD are specification driven). It is enough to take the qualitative TD

included in the specified QAM and to compare the expected results with the results given by qualitative execution. This kind of comparison can be easily automated. Three illustrations of this validation process follow.

• We can verify that lines **1.**, **2.** and **3.** of our QAM are in accordance with the corresponding qualitative executions (*Cases* 1, 2, and 3.)

• Suppose that the programmer has inadvertently typed the `if` condition (`A[i] <= T`) instead of (`A[i] >= T`). By trying to execute qualitatively the algorithm according to the inputs of line **4.**, the tester would find $\partial M = 0$ which of course does not correspond to the specifications ($\partial M = +$).

• Suppose now that the programmer has made an error by typing the while condition, (`k <= N`) instead of (`i <= N`). This error is extremely difficult to detect by classic testing techniques (static or dynamic) because, in a conventional execution, the result M often remains close to the correct one (i.e. they have the same order of magnitude). By using TD of line **6.** in the QAM, qualitative execution exaggerates the program behaviour and immediately detects the possibility of an infinite loop, since k (which is now a control variable) is never incremented.

6 PERSPECTIVES FOR QUALITATIVE ALGORITHMICS

The ideas of QA, i.e. the formal model of Valued Orders of Growth and the associated qualitative calculus as presented in Section 3 as well as the qualitative interpretation of program statements presented in Section 4, have been implemented in a qualitative executor prototype for Pascal called QUALEX. QUALEX is written in Prolog and runs on a Macintosh.

We believe that this paper is the first step towards a whole new research domain, namely Qualitative Software, which may be as promising and fruitful as Qualitative Physics in the next few years.

We do not claim that QA will solve the whole problem of software analysis which is extremely difficult. We do believe that it introduces a novel and powerful aspect in software programming which may help not only the testing process (as a "dynamic" enrichment of static analysis) but also domains like the general behavioral analysis of software components.

Structural testing consists of the production of Test Data by executing (sensitising) some program logic paths. The main problem in structural testing is to find the right combination of entries that give the desired truth values to the conditions appearing in the path to be covered. Automation of this Test Data generation process is nowadays essentially based on random generation. We understand that a Vog-execution can provide a guidance to identify the evolution law to be given to the entries to guarantee that a path within the source code is sensitised. It may therefore help to detect some forms of infinite loops.

In specification verification, QA can obviously be used, as shown in our example of Section 5, to perform tendency analysis, i.e. for answering questions of the kind *"What is the direction of change of the output variables given the direction of change of the input variables?"*, as well as for building qualitative models.

QA appears also to be an elegant formalization of a very efficient and widespread black box testing technique, called *limit testing*, for which various formalization attempts have failed. QA may equally provide a qualitative formal framework for assisting programmers during the debugging stage.

In qualitative simulation of software components which may be at the basis of *model-based diagnosis systems*, it is enough to perform the simplest Vog-execution of a program; i.e. by propagating the value part (v) of the Vogs only while maintaining the order of growth part (og) to zero. If the qualitative values of the input variables are acquired across time, the Vog-execution provides, by propagation, the trajectories of the output variables, i.e. their qualitative values across time. This has considerable impact considering that most control and monitoring systems are nowadays software intensive, leaving a very small part to hardware components.

Last but not least, we want to emphasise that the formal model of Valued Orders of Growth by itself is a powerful mathematical tool in the analysis domain. In particular, very important results are foreseen in an application domain that is not considered at all in this paper. Indeed, we are concurrently investigating the application of the Vog model to the asymptotic analysis of differential equations and the results are very promising [10].

7 REFERENCES

[1] B. Beizer (1992), *Software Testing Techniques*, Van Hostrand Reinhold.

[2] N. Bourbaki (1961), *Fonctions d'une Variable Réelle*, Hermann.

[3] M. Boshernitzan (1981), *An Extension of Hardy's Class L of "Orders of Infinity"*, Journal of Mathematical Analysis, *39*, 235-255.

[4] M. Boshernitzan (1982), *New Orders of Infinity*, Journal of Mathematical Analysis, *41*, 130-167.

[5] L.A. Clarke and D.J. Richardson (1983). *Symbolic Evaluation - an aid to Testing and Verification*, University Of Massachussets, Technical Report 83-41.

[6] P. D. Coward (1988), *Symbolic Execution systems-a review*, Software Engineering Journal, 3 (6) 229-239.

[7] G.H. Hardy (1910), *Orders of Infinity*, Cambridge University Press.

[8] R.F. Hoskins (1990), *Standard and nonstandard Analysis*, Ellis Horwood Ltd.

[9] W. L. Johnson (1986), *Intention Based Diagnosis of Novice Programming Errors*, Pitman, Morgan Kauffman.

[10] A. Missier, S. Xanthakis, L. Travé-Massuyès (1994), *Order of Growth Concepts*, LAAS-CNRS Internal Report n°94002, Toulouse (France).

[11] A. Missier, S. Xanthakis and L. Travé-Massuyès (1994), *Qualitative Algorithmics using Order of Growth Reasoning (Full Version)*, LAAS-CNRS Internal Report n°94003, Toulouse (France).

[12] G. J. Myers (1979), *The art of Software Testing*, J. Wiley.

[13] L. Travé-Massuyès (1992), *Qualitative Reasoning Over Time: History and Current Prospects*, The Knowledge Engineering Review, *Vol.7:1*, 1-18.

[14] L.Travé-Massuyès, N. Piera, A. Missier (1989), *What can we do with qualitative calculus today* ?, IFAC/IMACS/IFORS International Symposium on Advanced Information Processing in Automatic Control, Nancy (France).

[15] D.S. Weld (1988), *Exaggeration, AAAI'88.*

[16] D.S. Weld & J. de Kleer editors (1989),, *Readings in Qualitative Reasoning about Physical Systems*, Morgan-Kaufman.

[17] C. L.Wilson & L. J. Osterweil (1985),, *OMEGA: a Data Flow Analysis tool for the C programming language*, IEEE TSE, Vol 11, No 9.

[18] S. Xanthakis, C. Skourlas (1992), *A Data Flow Algebra Model for Static Analysis of Structured Programs*, Int. Conf. on Software Quality, North Carolina.

[19] K.M. Yip (1993), *Model Simplification by Asymptotic Order of Magnitude Reasoning*, AAAI'93, Washington DC (USA).