The V3F Project

Benjamin Blanc[1], Fabrice Bouquet[2], Arnaud Gotlieb[3], Bertrand Jeannet[3], Thierry Jéron[3], Bruno Legeard[2], Bruno Marre[1], Claude Michel[4], and Michel Rueher[4]

[1] {Benjamin.Blanc, Bruno.Marre}@cea.fr
CEA LIST LSL
91191 Gif-sur-Yvette cedex, France
[2] {bouquet, legeard}@lifc.univ-fcomte.fr
Laboratoire d'Informatique
Université de Franche-Comté
16, route de Gray - 25030 Besançon cedex 13, France
[3] {Arnaud.Gotlieb, Bertrand.Jeannet, Thierry.Jeron}@irisa.fr
IRISA-INRIA,
Campus de Beaulieu,
35042 Rennes cedex, France
[4] {cpjm, rueher}@essi.fr
Université de Nice–Sophia Antipolis, I3S–CNRS,
930 route des Colles, B.P. 145,
06903 Sophia Antipolis Cedex, France

Abstract. This paper describes the main results of the V3F project (which stands for "Validation and verification of software handling floating-point numbers")[1]. The goal of this project was to provide tools to support the verification and validation process of programs with floating-point numbers. We did investigate two directions: structural testing of a program with floating-point numbers and verification of the conformity of a program handling floating-point numbers, with its specification. Practically, a constraint solver over the floats was developed for the generation of test sets in structural testing framework. Different techniques have been developed to evaluate the distance between the semantics of a program over the real numbers and its semantics over the floating-point numbers.
Key words : verification and validation of programs, constraint programming, floating-point numbers, structural testing, reactive systems.

1 Introduction

Computations with floating-point numbers are a major source of failures of critical software systems. It is well known that the result of the evaluation of an arithmetic expression over the floats may be very different from the exact value

[1] This project is a joint project research supported by the ACI "sécurité et informatique" (security and computer sciences), an MNRT initiative conducted by CNRS, INRIA and DGA.

of this expression over the real numbers. Formal specification languages, model-checking techniques, and testing are currently the main issues to improve the reliability of critical systems. Over the past years, a significant effort was directed towards the development and the optimization of these techniques, but few work has been done to tackle applications with floating-point numbers. A correct handling of a floating-point representation of the real numbers is very difficult because of the extremely poor mathematical properties of floating-point numbers; moreover the results of computations with floating-point numbers may depends on the hardware, even if the processor complies with the IEEE 754 standard.

The goal of the V3F project was to provide tools to support the verification and validation process of programs with floating-point numbers. In other words, project V3F did investigate techniques to check that a program satisfies the calculation hypothesis on the real numbers that have been done during the modeling step. The underlying technology is based on constraint programming. Constraint solving techniques have been successfully used during the last years for automatic test data generation, model-checking and static analysis. However in all these applications, the domains of the constraints were restricted either to finite subsets of the integers, rational numbers or intervals of real numbers. Hence, the investigation of solving techniques for constraint systems over the floating-point numbers is an essential issue for handling problems over the floats.

Actually, the problem comes from the fact that the set of real numbers is not finite and thus cannot be store in a computer. Moreover, a numerical representation of a rational number like $1/3$ or an irrational numbers like π would require an infinite number of digits. Thus, due to memory limitations, and to keep computation efficient, computers mainly rely on the two following finite subsets of the real numbers :

- *fixed-point numbers* which use a computer integer m to represent the number $m \cdot 2^{-N}$, where the magnitude $N > 0$ is fixed.
- *floating-point numbers* which use two integers, the mantissa m and the exponent e to represent the number $m \cdot 2^e$.

The first representation requires a careful choice of N which takes into account the targeted computations, the magnitude of the numbers to be represented, as well as the chosen computation precision. However, arithmetic of fixed-point numbers has some nice properties: for example, adding two fixed-point numbers always gives another fixed-point number (as long as the result stays in the range of the represented set of fixed-point numbers).

The second representation benefits from a wide spread standardization, i.e., the well known IEEE 754 standard [ANS85]. This standard has been made available on almost all modern computers and is usually implemented using hardware. As a consequence, computing using floating-point numbers is fast. However, floating-point numbers have poor arithmetic properties. The result of the addition of two floating-point numbers is almost never a floating-point number. Thus, arithmetic over the floating-point numbers requires a rounding operation

to choose the most appropriate representation of the result of an operation in the set of floating-point numbers.

Critical software systems that accumulate results of computations on a long period of time have, up to now, used fixed-point numbers. However, for different reasons (like the ease of use), industrials developing critical software have shown an increasing interest for floating-point numbers. As a result :

- the validation process (verification as well as test) needs to be adapted to floating-point numbers : validation tools must take into account the specific semantics of floating-point numbers.
- the "unpredictable" behavior of floating-point number computations make the validation process much more complex.

That's why the goal of the V3F was to provide tools to support the verification and validation process of programs with floating-point numbers. To do so, we did investigate two directions. First, we did address the problem of structural testing of program with floating-point numbers. In particular, we did develop a constraint solver over the floating-point numbers for the generation of test sets. Second, we did investigate the problem of the verification of the conformity of a program using floating-point numbers with its specification based on real numbers. Different techniques have been developed to evaluate the distance between the semantics of a program over the real numbers and its semantics over the floating-point numbers.

Outline of the paper. We will first recall some classical problems which occur when floating-point numbers are used. Then, we will detail the test set generation problem when the program contains floating-point numbers operations. The main features of the constraint solver over the floats we have developed will be detailed. Next, we study the problem of the verification of the conformity of a program handling floating-point numbers with its specification. Two techniques to handle this problem will be detailed.

2 Basic problems with floating-point numbers

Floating-point numbers have very poor arithmetic properties [Gol91]. Indeed, most of the usual properties of arithmetic over the reals are no more true over the floating-point numbers. Addition and multiplication over the floating-point numbers are neither associative nor distributive: $((a + b) + b) \neq a + (b + b)$ and $a * (b + c) \neq (a * b) + (a * b)$ for numerous floating-point values of a, b and c. However, these two operations are commutative, i.e., $a * b = b * a$. The same kind of limitations are true for the subtraction and the division.

Most of these drawbacks indirectly due to the use of a finite subset of the real and the requirement for a rounding operation. This limited representation of the real numbers has also some more direct and annoying effects: the absorption phenomena and the cancellation phenomena. Absorption occurs when a small floating-point number is added to a much bigger one. In such a case, the addition acts as a null operation, i.e., $a+b = b$. For example, adding 10^-9 to 10^9 gives 10^9.

Cancellation results from the subtraction of two nearby quantities. In this case, this operation may produce some significant round off error whose propagation by other operations could be catastrophic. For example[2], the computation of $(10.00000000000004 - 10.0)/(10.000000000000004 - 10.0)$ yields[3] 11.5 while the result should be 10.0

Floating-point arithmetics is highly context sensitive. Any modification or choice in the rounding value, the floating-point unit, the mathematical library or the evaluation order of the expression may produce a different result. Therefore, a solution of a problem over the floating-point number is always related to a given environment.

The IEEE 754 standard does not even insure that all processors compliant to the standard will provide the same result. The standard is ambiguous enough to let both the Sun Sparc processors and the Pentium Intel be compliant. These two family of processors may produce very different results: the Sparc processor implement a correctly rounded operation for each available type of floating-point numbers (simple, double and long double) while the Intel Pentium only implement correctly rounded operations for the long double. Moreover, a Sparc processor uses 128 bits to represent a long double while an Intel processor uses only 80 bits.

The specific properties of the floating-point numbers do not allow a solver over the reals to correctly tackle problems over the floating-point numbers. For example, consider equation $16.1 = 16.1 + x$. This equation has 0 as the unique solution over the reals, and a solver over the reals like Prolog IV correctly answers that the only possible solution is 0. However, all the floating-point numbers contained in the interval $[-1.77635683940025046e-15, 1.77635683940025046e-15]$ are solution to $16.1 = 16.1 + x$ (on a Sparc processor, with doubles and a rounding mode set to "near"). Thus, to tackle problem over the floating-point numbers, a dedicated solver is required.

Next section addresses test set generation issues for programs with floating-point numbers.

3 Structural testing of programs with floating-point numbers

Structural testing aims at exercising execution paths of a program to fulfill some coverage criteria (e.g. statement or branch coverage). It improves the program reliability by ensuring that, for example, all the statements of a given program have been executed once. A major challenge in structural testing consists in generating automatically test data, i.e., finding some input values so that a given point in a program is actually reached. Constraint based structural testing attempts to tackle this issue using a constraint programming model of the problem. This problem is then solved by means of an adapted constraint solver to find test sets. The key points are loop unfolding and constraint solving.

[2] This example comes from http://www.cs.princeton.edu/introcs/91float/.
[3] With gcc, on an Intel Pentium platform running under Linux.

3.1 Structural testing

Symbolic execution is a classical structural testing technique which evaluates a selected control flow path with symbolic input data. A constraint solver can be used to enforce the satisfiability of the extracted path conditions as well as to derive test data. Automatic test case generation can be handled efficiently by translating a non-trivial imperative program into a CSP over finite domains. However, when these programs contain arithmetic operations that involve floating-point numbers, the challenge is to compute test data that are valid even when the arithmetic operations performed by the program to be tested are unsafe.

Expressions in programming languages are more ruled than constraints. A directed acyclic graph (DAG) is often used to represent them. An expression like $x_1 = x_2 + x_3$ in C states the computation of x_1 given x_2 and x_3. However, especially with floating-point numbers, it does not allows to directly compute x_2 or x_3 given the two other values.

Whenever path conditions contain floating-point computations, a naive strategy would consist in using a constraint solver over the rationals or the reals. Unfortunately, even in a fully IEEE-754 compliant environment, this leads not only to approximations but also can compromise correctness: a path can be labelled as infeasible although there exists floating-point input data that satisfy it, or labeled as feasible when no floating-point input data can satisfy it.

For example, consider the C program given in Fig.1 and the symbolic execution of path 1→2→3→4. The associated path conditions can be written as $\{x > 0.0, x + 1.0e12 = 1.0e12\}$. It is trivial to verify that these constraints do not have any solution over the reals and a solver like the IC library of the Eclipse Prolog system or Prolog IV will immediately detect it. However, any IEEE-754 single-format floating-point numbers of the closed interval $[1.401298464324817e-45, 32767.9990234]$ is a solution of these path conditions. Hence, a symbolic execution tool working over the reals or the rationals would declare this path as being infeasible.

Conversely, consider the path conditions $\{x < 10000.0, x + 1.0e12 > 1.0e12\}$ which could easily be extracted by the symbolic execution of path 1→2→3→4 of program foo2 of Fig.2. All the reals of the open interval (0, 10000) are solutions of these path conditions. However, there is no single floating-point value capable to activate the path 1→2→3→4. Indeed, for any single floating-point number x_f in (0, 10000), we have $x_f + 1.0e12 = 1.0e12$. Hence the path 1→2→3→4 is actually infeasible although a symbolic execution tool over the reals or the rationals would have declared it as feasible.

In the V3F project, we addressed the peculiarities of the symbolic execution of program with floating-point numbers. Issues in the symbolic execution of this kind of programs were carefully examined and practical details on how to build correct and efficient projection functions over floating-point intervals have been described in [BGM06]. First, an approach called normalization was defined to preserve the evaluation order and the precedence of expression operators of floating-point computations as specified by the programming language. The key idea was to take advantage of the expression's shape of the abstract syntax tree

```
float foo1(float x) {
    float y = 1.0e12, z ;
1.  if (x > 0.0)
2.      z = x + y ;
3.  if (z == y)
4.      ...
```

Fig. 1. Program foo1

```
float foo2(float x) {
    float y = 1.0e12, z ;
1.  if (x < 10000.0)
2.      z = x + y ;
3.  if (z > y)
4.      ...
```

Fig. 2. Program foo2

built by the compiler of the program (without any rearrangement nor any simplification due to optimizations). Next, a constraint-based propagation engine over floating-point intervals was build by defining efficient floating-point variable projection functions implementation. Our work covered not only arithmetic operators but also comparison and format-conversion operators for numeric and some symbolic floating-point values. FPSE, a symbolic execution tool for ANSI C floating-point computations [BG05], was developed and experimented on several C programs extracted from the literature. These experiments demonstrated that the proposed approach was suitable to deal efficiently with small-sized C floating-point computations.

Next section detailed the essential features of the FPCS solver.

4 FPCS: a constraint solver over the float

Solving constraint over the floating-point numbers is a two step process: a propagation step and an enumeration step. While the latter might rely on usual approaches, the first step requires more attention.

Roughly speaking, the propagation step consists in a fixpoint algorithm which attempts to reduce the size of the domain of the variables taking advantage of the constraints between the variables. For example, consider the simple expression $z = x + y$ where $x \in [1, 3]$ and $y \in [2, 3]$. According to the constraint, the domain of $z \in [3, 6]$. Thus, if the initial domain of z is $[0, 10]$, it can be reduced to $[3, 6]$. This simple process is usually extended to the computation of the domain of x (resp. y) according to the domains of y (resp. x) and z. However, due to the poor mathematical properties of the floating-point numbers, the computation of the so called "inverse projections" requires special attention in this case.

FPCS relies on two key properties:

- Interval computation [Moo66] is conservative of the solutions over the floating-point numbers. More precisely, interval arithmetic, when computed with floating-point numbers and with outward rounding, preserves the solution over the floats provided the computation follows the same operation order than the one specified by the expression over floats[4]. Moreover, when the rounding mode is known, interval computation can be done more precisely by using the current rounding mode instead of an outward rounding.

[4] Such an order depends on the programming language.

– A computation of the inverse projection is possible, provided the FPU conforms to the IEEE 754 standard for floating-point numbers. In such a case, basic arithmetic operation are correctly rounded. Correctly rounded means that the computation of the result over the floating-point numbers is the same as if the computation were done over the reals before being rounded. More formally, let $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ be a binary operator over the floating-point numbers, $. \in \{+, -, *, /\}$ be a binary operator over the real numbers, x and y, two floating-point numbers, and *Round* a rounding function, then, if $\odot$ is correctly rounded: $x \odot y =_{def} Round(x \ . \ y)$. This property allows us to devise a mean to compute the inverse projection.

4.1 Local filtering of floating-point numbers

Two approaches have been explored. The first one extends the concept of *box*-consistency [BMH94] to floating-point numbers [MRL01]. A basic property that a filtering algorithm must satisfy is the conservation of all the solutions. So, to reduce interval $\mathbf{x} = [\underline{x}, \overline{x}]$ to $\mathbf{x} = [x_m, \overline{x}]$ we must check that there exists no solution for some constraint $f_j(x, x_1, ..., x_n) \diamond 0$ when $\mathbf{x}$ is set to $[\underline{x}, x_m]$. This job can be done by using interval analysis techniques to evaluate $f_j(x, x_1, ..., x_n)$ over $[\underline{x}, x_m]$ when all computations comply with the IEEE 754 recommendations. *box*-consistency algorithms attempt to reduce the size of the domains of the variable using this property by means of a shaving strategy. Though this approach is effective, it needs a lot of computations to achieve its task.

The second approach extends the concept of 2*b*-consistency [Lho93] to floating-point numbers. 2*b*-consistency algorithms first decompose complex expressions into simple basic operations for which projections functions are available. For example, to reduce the size of $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, the domains of the variables x, y and z, according to constraint $z = x + y$, 2*b*-consistency uses 3 projection functions:

$$\begin{cases} \mathbf{x} \leftarrow \mathbf{x} \cap \Pi_x(\mathbf{y}, \mathbf{z}) \\ \mathbf{y} \leftarrow \mathbf{y} \cap \Pi_y(\mathbf{x}, \mathbf{z}) \\ \mathbf{z} \leftarrow \mathbf{z} \cap \Pi_z(\mathbf{x}, \mathbf{y}) \end{cases}$$

Over the floating-point numbers, the last projection function is easily computed by means of interval arithmetic. Usually, the current rounding mode r being known, we have : $\Pi_z(\mathbf{x}, \mathbf{y}) == [\underline{\mathbf{x}} +_r \underline{\mathbf{y}}, \overline{\mathbf{x}} +_r \overline{\mathbf{y}}]$ where $+_r$ is the addition of two floating-point numbers with a rounding mode set to r. Thus, the issue is now the computation of Π_x and Π_y. For the sake of simplicity, let $r = -\infty$. Then, Π_x is given by:

$$\Pi_x(\mathbf{y}, \mathbf{z}) = [\underline{\mathbf{z}}^- -_+ \overline{\mathbf{y}}, \overline{\mathbf{z}} -_{-\infty} \underline{\mathbf{y}}]$$

where $\underline{\mathbf{z}}^-$ is the predecessor of $\underline{\mathbf{z}}$, $-_{-\infty}$ is the subtraction computed with a rounding mode set to $-\infty$, and $\underline{\mathbf{z}}^- -_+ \overline{\mathbf{y}}$ is equal to $(\underline{\mathbf{z}}^- -_{+\infty} \overline{\mathbf{y}})+$ if $\underline{\mathbf{z}}^- -_{+\infty} \overline{\mathbf{y}} = \underline{\mathbf{z}}^- - \overline{\mathbf{y}}$ (i.e., the successor of the result of the subtraction computed with a rounding mode set to $+\infty$), or to $\underline{\mathbf{z}}^- -_{+\infty} \overline{\mathbf{y}}$ otherwise. IEEE 754 compliant units provide a flag which is raised in the second case, and thus, allow the implementation of

the inverse projection function. Note that to compute Π_y projection function, we just need to change the parameters. Each rounding mode requires a slightly different formulae and each formulae assumes that the underlying operation is correctly rounded, i.e., that the floating-point unit is an IEEE 754 compliant unit. These results are more detailed in [Mic02].

4.2 FPCS implementation

FPCS implements the second strategy as a callable C++ library. The targeted platform is an Intel Pentium running linux while constraints are analyzed as C language expressions.

The Intel Pentium floating-point unit has one distinctive feature: all floating-point computations are done using the unique available format, i.e., with 80 bits floating-point numbers. Thus, any double or simple format is converted in an 80 bits extended floating-point number before any computations. Computations are then done using operations over extended floating-point numbers. The result is then converted into the floating-point type of the targeted variable. However, all the arithmetic operations stick to the IEEE 754 standard. Therefore, special attention is required when translating a C expression in a set of basic constraints.

FPCS supports all the basic arithmetic operations, i.e., $+$, $-$, $*$ and $/$. It also supports the square roots which is a correctly rounded operation. Some other functions have also been implemented (sin, cos, ...). However, these functions are not correctly rounded and their precision is not the one of correctly rounded functions. Projection functions for theses functions do take this into account. Unfortunately, it is not possible to guarantee that no solution is lost.

4.3 Illustrative examples

The solver directly handles C expressions. For example, to know whether or not there is a double `x` such that its square is equal to 2, we simply write

```
x*x == 2.0
```

and the solver answers that there is no solution ! As a matter of fact, there is no double which fulfill such equation when the rounding mode is set to near, the default rounding mode. If the rounding mode is set to down, then the solver displays the two solutions. -1.4142135623730951 and 1.4142135623730951.

Now consider the computation of the cubic root:

```
int gsl_poly_solve_cubic (double a, double b, double c,
                          double *x0, double *x1, double *x2) {
  double q = (a * a - 3 * b);
  double r = (2 * a * a * a - 9 * a * b + 27 * c);

  double Q = q / 9;
  double R = r / 54;
```

```
  double Q3 = Q * Q * Q;
  double R2 = R * R;

  double CR2 = 729 * r * r;
  double CQ3 = 2916 * q * q * q;

  if (R == 0 && Q == 0) {
    ...  /* Point to reach */
  } else if (CR2 == CQ3) {
    ...
  } else if (CR2 < CQ3) {
    ...
  } else {
    ...
  }
}
```

This code has been extracted from the Gnu Scientific Library. Assume that our aim is to get some input values for `a`, `b` and `c` such that the very first `if` of the program will be reached. This problem is equivalent to solving the following set of constraints:

```
q = (a * a - 3.0 * b)
r = (2.0 * a * a * a - 9.0 * a * b + 27.0 * c)
Q = q / 9.0
R = r / 54.0
Q3 = Q * Q * Q
R2 = R * R
CR2 = 729.0 * r * r
CQ3 = 2916.0 * q * q * q
R == 0.0
Q == 0.0
```

Note that the solver distinguishes the affectation `=` from the equivalence `==` which handle signed zero in different ways. If the value of `a` is set to `15.0`, then a simple filtering process reduces the domain of all other variables to a single double:

```
a = [ 1.50000000000000000000e+01,  1.50000000000000000000e+01]d
b = [ 7.50000000000000000000e+01,  7.50000000000000000000e+01]d
c = [ 1.25000000000000000000e+02,  1.25000000000000000000e+02]d
q = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
r = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
Q = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
R = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
Q3 = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
R2 = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
CQ3 = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
CR2 = [ 0.00000000000000000000e+00,  0.00000000000000000000e+00]d
```

This last example shows how effective a $2b$-filtering over the floating-point numbers is. FPCS has been tested on a variety of programs.

Next section addresses another key issue of the validation process: the conformity of a program using floating-point numbers with its specification based on real numbers.

5 Conformity of a program using floating-point numbers with its specification based on real numbers

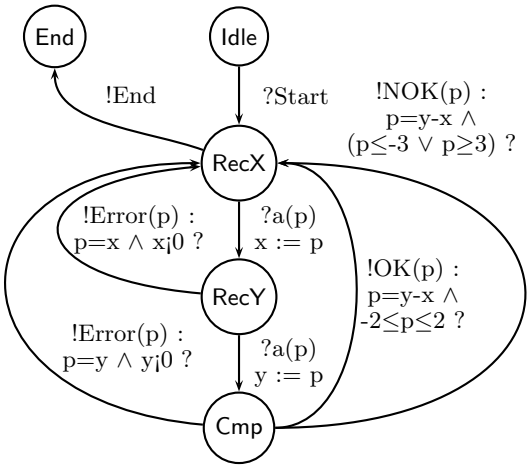
The aim is here to check whether or not a program is conform (with a given meaning) to a formal specification. To take floating-point numbers into account, a tolerance should be introduced in the conformance relation: floating-point values are allowed to differ to some degree from the real values of the specification. The main issue is the capability to test such a relaxed conformance relationship. The values allowed by the specification are defined by means of an automata. The automata execution provides some real values for the tests. If the conformance relation is relaxed, then, the interaction between the automata and the program under test might diverge.

Two approaches have been investigated:

- the conformance testing of asynchronous reactive systems. The issue of conformance testing [BJK+05] is to check whether an implementation (a program) is conform in some precise sense to a formal specification. As in structural testing, one also considers *test purposes*, which typically consists in maintaining the executions of the implementation in some set during the execution of the test. The aim of such test purposes is to orient the test toward specific functionalities of the implementation, or to simulate particular environments in which the implementation should run correctly.
- the generation of functional test sequences for Lustre descriptions with numerical values. GATeL is a functional testing tool based on Lustre descriptions: a formal modeling language belonging to the synchronous data-flow family. Given a control program and a partial description of its behavior as a Lustre model, the main role of GATeL is to automatically generate test sequences (representing an evolution of input data flows over time) according to test objectives. When numerical values have been taken into account into Lustre descriptions, we decided as a first step to give a real interpretation to the computations made. However, this pure interval semantics has a too poor decision power in our context. To avoid this problem, called reification in the testing community, we chose to force input data flows to be instantiated only to double (seen as particular reals) during the resolution process.

5.1 Conformance testing of asynchronous reactive systems

Background. STG (Symbolic Test Generator) is a tool for generating test cases for asynchronous reactive systems [RdBJ00,CJRZ02], based on the principles



receive Start, read 2 integers x, y on the channel a
check that they are positive and that $|x - y| \leq 2$ (otherwise emit resp. ERROR or NOK)
emit OK($x - y$) and start again reading values

Fig. 3. Example of a IOSTS specification S

first developed by [Tre96]. The observable points of the implementation under test $\mathcal{I}$ are supposed to be the input (resp. output) messages it receives from (resp. emits to) its environment. The implementation is thus a black box whose observable behavior is traces of input/output messages.

The aim of STG is to test the conformance of an implementation $\mathcal{I}$ w.r.t. a specification S modeled as a input/output symbolic automaton (IOSTS), which is basically a finite automaton extended with variables, guards and assignments, and where actions carry values[5]. Fig. 5.1 gives an example of an IOSTS. Such a specification defines a set of conformant traces $traces(S)$. The implementation exhibits a conformance error during its execution if its observable behavior at some point does not belong any more to $traces(S)$.

In this context, the tester is a program that will be run in parallel with the implementation $\mathcal{I}$, and whose role is to provide suitable inputs to $\mathcal{I}$ and to check whether the outputs of $\mathcal{I}$ are correct w.r.t. the semantics of S.

For the sake of simplicity, we do not describe here how to perform test selection by the mean of test purposes, and we focus only on checking conformance.

Taking into account the behavior of floating-point numbers. As explained in the introduction, the semantics of floating-point operations is non-deterministic and suffers from non-intuitive properties.

[5] Alternatively, an IOSTS corresponds to a bounded-memory, non-recursive program emitting and receiving valued messages from its environment.

As the semantics of floating-point numbers is thus hardly understandable and/or predictable by an human being, who writes the specification, we make the following fundamental choice:

> We assume that the reference semantics of the specification S is based on its (deterministic) real number semantics.

This choice has also the benefit to enable easy semantics-preserving program transformations (which is much more difficult with floating-point numbers).

In this context, we take into account the non-deterministic semantics of floating-point numbers (that are still used in the implementation) by relaxing the conformance relation: we allow a limited slew between the observed values of the implementation $\mathcal{I}$ and the values authorized by the reference semantics of S, without letting them diverge as the execution proceeds. This can be roughly formalized by defining, for some $\epsilon > 0$:

$$\mathit{traces}^\epsilon(S) = \{\sigma_0 \dots \sigma_n \mid \sigma'_0 \dots \sigma'_n \in \mathit{traces}(S) \wedge \forall i \leq n : d(\sigma_i, \sigma'_i) \leq \epsilon\}$$

where $d(\cdot, \cdot)$ is a suitable distance between messages. $\mathit{traces}^\epsilon(S)$ defines a relaxed semantics of S.

The main difficulty now is to check the inclusionship of the observable behavior of $\mathcal{I}$ in $\mathit{traces}^\epsilon(S)$. In the standard case, without floating-point numbers, this inclusionship is incrementally tested by executing the IOSTS S in parallel with $\mathcal{I}$. However, accepting a slack between the "ideal" values expected by the IOSTS S and the actual values emitted by $\mathcal{I}$ raises a problem: since this values may be used in the next execution step of S, an increasing divergence may arise as the execution proceeds between the values that such a relaxed IOSTS S accepts and those defined by $\mathit{traces}^\epsilon(S)$.

The solution we developed uses an orthogonal projection of values that are conformant "upto ϵ" onto the set of strictly conformant values. Such a projection may be easily be implemented if we restrict numerical conditions in the IOSTS S to be defined with logical combinations of linear constraints.

We proved that this technique allows to effectively check the inclusionship in *a subset of* $\mathit{traces}^\epsilon(S)$. Although we do not decide the inclusion in the exact set $\mathit{traces}^\epsilon(S)$, this is still satisfactory as we intuitively check that the acceptable slack between the strict semantics of S and $\mathcal{I}$ does not induce a divergence in the long term.

This approach has some requirements:

- The IOSTS S should be executed with a real-number semantics. This may be done by using multi-precision rational numbers, but this restricts the arithmetic operations that may be used in a specification, hence the kind of properties that can be tested.
- The previous point implies that conversions from and to floating-point numbers are required. It is easy, although potentially costly, to convert precisely a floating-point number to a multi-precision rational number. However, the opposite conversion may imply an approximation. This is valid as long as the distance is less than ϵ.

A weakness of this approach is that it uses a absolute tolerance ϵ, instead of a relative tolerance taking into account the magnitude of the floating-point values. We conjecture that the same approach could be developed using a norm instead of a distance and a relative tolerance for relaxation, but we have not yet worked out this direction.

We still have to implement this method in STG and to experiment its relevance. The implementation consists mainly in adding the conversion operations and the orthogonal projection onto an union of convex polyhedra (induced by the logical combination of linear constraints allowed in guards). The orthogonal projection onto a single convex polyhedron can easily be performed using a linear programming solver, and then generalized to union of convex polyhedra. The orthogonal projection also allows to compute the distance between a vertex and such polyhedral sets, which is needed for checking the relaxed conformance.

5.2 The generation of functional test sequences for Lustre descriptions with numerical values

GATeL is a functional testing tool based on Lustre descriptions: a formal modeling language belonging to the synchronous data-flow family. Given a control program and a partial description of its behavior as a Lustre model, the main role of GATeL is to automatically generate test sequences (representing an evolution of input data flows over time) according to test objectives. These test objectives are characterized by a property to be reached in order to exercise meaningful situations. It can be the raise of an alarm, or the execution of a predetermined scenario, or a general property on inputs and outputs. To build a sequence reaching the test objective, according to the Lustre model of the program and its environment - also described in Lustre, these three elements are automatically translated into a constraint system. A resolution procedure then solves this system.

The obtained test sequences can then be submitted to the program under test. When only dealing with boolean and integer data-flows, there are two level of conformity. The first one is to ensure that the test objective pointed by the test sequence is actually reached *at the same cycle* by the program. The second also checks that the outputs guessed by the Lustre model and those computed by the program are equal at every cycle.

When numerical values have been taken into account into Lustre descriptions, we decided as a first step to give a real interpretation to the computations made. Indeed, the Lustre descriptions manipulated by GATeL users are mainly considered as a model of the control program with no direct link between them. Designers are supposed to be more familiar with real interpretations than floating-point ones. This also allows to generate sequences as independently as possible of the underlying hardware.[6]

[6] A second step would be to give a floating-point semantics to the computations using FPCS to deal with cases where the Lustre model is also considered as an executable

We did it the usual way, by widening the computations into the largest interval of double precision floating-point values. Intervals are created when parsing the Lustre description, while reading a numerical string not exactly representable by a double. This happens very frequently, for it concerns very usual constants, e.g. 0.1 or 0.02. This semantics allows to ensure that no real solution is lost in the resolution process. However, this pure interval semantics has a too poor decision power in our context.

Consider the following example, where the `reach` directive states to reach a sequence where `r` is true at the final cycle.

```
node Const(x:real)
returns(r:bool);
let
  r = (x + 0.06 = 0.08);
  (*! reach r !*)
tel;
```

According to this semantics, constants 0.06 and 0.08 are interpreted as intervals since they are not exactly representable by doubles: [0.0599999999999999978, 0.0600000000000000047] and [0.0799999999999999878, 0.0800000000000000017] respectively. After building the initial constraints system and propagating the equation for `r`, the domain allowed for `x` is [0.019999999999999983, 0.020000000000000004]. This domain is the smallest satisfying the three projections of this equation (direct addition and two indirect subtractions). Since the resolution procedure cannot choose any value within the interval representing the constants, none of these values could be further discriminated by the resolution procedure. To submit this sequence to the program under test, a sequence build on this result should then pick any double in the domain for `x`. However, some values for `x` (e.g. 0.019999999999999983) leads to a contradictory value for `r`, when evaluated by a compiled program with a "to-the-nearest" rounding mode (and this would be also the case for any other rounding mode). This difference is interpreted by the conformance relation as a bug, while it only is a consequence of extra widening.

To avoid this problem, called reification in the testing community, we chose to force input data flows to be instantiated only to double (seen as particular reals) during the resolution process. Similarly, constant values are interpreted as the nearest double. In our example, constants are interpreted as 0.06 (slightly above its real value) and 0.08 (resp. slightly under its real value). Consequently, the new domain for `x` is the successor of 0.02 : 0.020000000000000004. A floating-point evaluation of the computation in a round-to-nearest mode confirms the correct result. Any other value given to `x` would lead to a wrong result.

program. As usual, exact informations about the compilation of Lustre programs should then be needed.

Another feature of our solver improves the decision power in order to be able to deal with exact equalities/inequalities test objectives. It allows to distinguish between reducible and unreducible intervals, according to a special status. Unreducible interval are interval created during an operation between constants. These intervals are treated as *open* intervals. However, in order to avoid mixing closed and open intervals during the resolution process, any operation involving open intervals lead to a closed reducible one. A modification of our previous example shows the benefits of this feature:

```
node Const(x:real)
returns(r:bool);
let
  r = (x + 0.01 = 0.06);
  (*! reach r !*)
tel;
```

The initial propagation of the corresponding constraints system imposes `x` to belong to an interval coming from the three projections altogether. The difference with the previous example lies in the fact that there exists no double value at the intersection of the three domains of the projections. Since input should be completely instantiated, there is then no test sequence leading to this objective according to our semantics.

Using this feature also leads us to increase the decision power when coping with equation/inequality. For instance, if x is a double and y an unreducible interval, then when always have that $x \neq y$, and $max(x) \leq min(y) \wedge x < y$. Moreover if only y status is known and if $x = y \wedge max(x) = min(y)$, then the resolution fails.

Of course, this semantics does not contain all real solutions of a problem, but only those which are representable by double values. However, it is conservative for representable ones. Extra floating-point solutions could be introduced by using intervals, but some algebraic treatments and the unreducible intervals strongly limit their apparition.

6 Conclusion and further work

Computation with floating-point numbers is a critical issue for many software systems. Very few tools are available to check that a program satisfies calculation hypothesis that have been done during the specification process. In the V3F project did investigate the capabilities of constraints techniques to handle these tasks.

We did develop a constraint solver over the floating-point numbers for structural testing of a program with floating-point numbers. We addressed the peculiarities of the symbolic execution of program with floating-point numbers. Issues in the symbolic execution of this kind of programs were carefully examined and

practical details on how to build correct and efficient projection functions over floating-point intervals have been described.

We did also develop different techniques to evaluate the distance between the semantics of a program over the real numbers and its semantics over the floating-point numbers. The key idea here is the introduction of a tolerance. A symbolic test generator for asynchronous reactive systems with the floating-point numbers has been developed. GATEL, a functional testing tool for synchronous systems, has also been adapted to handle Lustre specifications with numerical values.

First experimentations on academic programs are quite promising. Further work concerns the evaluation of all these techniques on more significant programs as well as their integration in software verification frameworks based on model checking or theorem proving.

References

[ANS85] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[BG05] B. Botella and A. Gotlieb. *FPSE: Floating-Point Symbolic Execution*. INRIA-IRISA, Rennes, 2005. Documentation of a floating-point interval constraint solver.

[BGM06] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*. LNCS. Springer-Verlag New York, Inc., 2005.

[BMH94] Frédéric Benhamou, David McAllester, and Pascal Van Hentenryck. Clp(intervals) revisited. In *Proc. of the ISLP'94*, pages 124–138, 1994.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a symbolic test generation tool. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, 2002.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[Lho93] Olivier Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of IJCAI'93*, pages 232–238, Chambéry(France), 1993.

[Mic02] C. Michel. Exact projection functions for floating point number constraints. In *Proc. of 7th AIMA Symposium*, Fort Lauderdale (US), 2002.

[Moo66] R. Moore. *Interval Analysis*. Prentice Hall, 1966.

[MRL01] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In *Constraint Prog. (CP'01)*, pages 524–538, LNCS 2239, Nov 2001.

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Integrated Formal Methods (IFM'00)*, volume 1945 of *LNCS*, 2000.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3), 1996.