Contents

1 Introduction **2**

2 BDD and convex polyhedra **3**
2.1 Binary Decision Diagram (BDD) 3
2.2 Convex Polyhedra 4

3 The resolution algorithm **4**
3.1 The constraints domain 4
3.2 Get rid of if-then-else 5
3.3 A two-layered resolution scheme 6

4 Choosing solutions **6**
4.1 Random choice of Boolean values 6
4.2 Random choice of numeric values 7
4.2.1 Taking numerics into account during the BDD traversal . 7
4.2.2 Favoring limit cases 7
4.2.3 Drawing numerics uniformly 8
4.3 Fairness versus efficiency 8
4.3.1 Fairly choosing numerics is expensive 8
4.3.2 Combining Booleans and numerics 9

5 Available Tools **9**

6 Related work **10**

7 Conclusion **12**

Generating random values using Binary Decision Diagrams and Convex Polyhedra

Erwan Jahier Pascal Raymond

September 4, 2006

Abstract

This article describes algorithms to solve Boolean and numerical constraints, and to randomly select values among the set of solutions. Those algorithms were first designed to generate inputs for testing and simulating reactive real-time programs. As a consequence, the chose a solving technology that allow a fine control in the way solutions are elected. Indeed, a fair selection is sometimes required, while favoring limit cases is often interesting for testing.

Moreover, simulating a single reactive execution means generating several hundreds or even several thousands of atomic steps, and thus as many solving steps. Hence, the emphasis is put on efficiency, sometimes sacrificing precision or fairness.

Keywords: Constraint solving, Test sequences generation, Simulation, Reactive programs.

1 Introduction

Reactive embedded programs are often critical, and therefore need to be verified. The ideal is to verify programs exhaustively, using formal verification methods such as model-checking, deductive reasoning, or abstract interpretation. These methods face both theoretical problems like undecidability, and practical problems like state explosions. In practice, they are limited to relatively simple and small systems. Test and simulation, that do not explore the whole state space, remain the only tractable method for complex and huge systems.

Complex reactive systems are not supposed to behave correctly in a chaotic environment, and thus a completely random test generation is likely to produce irrelevant executions. As a matter of fact, the environment, while nondeterministic, is in general far from random: it satisfies known properties that must be taken into account to generate realistic test sequences.

A testing framework has been defined which includes languages for describing constrained random scenarios [16]. More precisely, an atomic step is described by a constraint on the current values of the variables. Those steps are then

combined with control structures describing the possible dynamic behavior (sequence, loop, non-deterministic choice). This dynamic aspect is not presented here (see [16, 8] for further detail). This article focuses on the basic problem of solving a constraint and generating a single step.

In order to tackle realistic problems, we want to handle both logical and numerical constraints. We also want the solver to be fully automatic, and thus we restrict ourself to a decidable domain: the domain of linear constraints.

The proposed solving method requires the construction of a normalized representation of constraints. This normal form is based on Binary Decision Diagrams for the logical part, and convex polyhedra for the numerical part.

The article is organized as follows. Section 2 first recalls the basic principles of BDDs and convex polyhedra; then Section 3 presents the solving process; Section 4 presents the solution selection; Section 5 presents associated tools; Section 6 presents related work.

2 BDD and convex polyhedra

The constraints we want to solve are a mixture of Boolean and linear numerical constraints. Basically, the formers are handled with BDD (Binary Decision Diagram), and the latter with convex polyhedra. We briefly review these representations before explaining how we use them.

2.1 Binary Decision Diagram (BDD)

A Binary Decision Diagram is a concise representation of the Shannon decomposition of a Boolean function [3]. More precisely, the BDD of a formula f is a Directed Acyclic graph (DAG) where each node is labelled by a variable of f. The *top-level* node is the only node that has no predecessor. The two only possible leaves are labeled by *true* and *false*. Each node has two successors: the *then* branch, and the *else* branch. During a traversal from the top-level node to a leaf, the variables always occur is the same order.

All solutions of a formula can be obtained by enumerating in its BDD all paths from the top-level node to the true leaf. For such a path, when a node is traversed using its *then* branch (resp. *else* branch), it means that the corresponding variable is true (resp. false).

Figure 1 shows a graphical representation of a BDD; *then* (resp *else*) branches are represented at the left-hand-side (resp right-hand-side) of the tree. This BDD contains 3 paths to the true leaf: ade, $\overline{a}bc\overline{e}$, and $\overline{a}\overline{b}d$. When we say that the monomial (conjunction of literals) $\overline{a}bc\overline{e}$ is a solution of the formula, it means that variables a and e should be false, variables b and c should be true, and variable d can be either true or false. The monomial $\overline{a}bc\overline{e}$ therefore represents two solutions, whereas ade and $\overline{a}\overline{b}d$ represents 4 solutions each, since 2 variables are left unconstrained.

In Figure 1 and in the following, for the sake of simplicity, we draw trees instead of DAGs. The key reason why BDDs work well in practice is that in

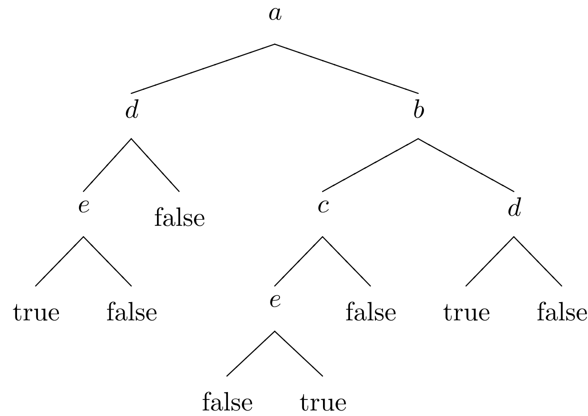

Figure 1: A BDD containing 10 solutions (ade, $\overline{a}bc\overline{e}$, and $\overline{a}\overline{b}d$).

their implementations, common sub-trees are shared. For example, only one node "true" would be necessary in that graph. Anyway, the algorithms in the sequel work on DAGS the same way as they work on trees.

2.2 Convex Polyhedra

The objective is to solve linear inequations, namely, to compute systems of the form $P = \{X|AX \leq B\}$, where A is a $n \times m$-matrix of constants, and B a m-vector of constants. Such a system define a convex polyhedron.

If all variables are bounded[1], solving such systems requires to compute a set of polyhedron *generators*, namely, to compute the vertices $v_1, \ldots, v_k$ such that $P = \{\sum_{i=1,k} \alpha_i.v_i | \sum_{i=1,k} \alpha_i = 1\}$. Reasonably efficient algorithms exist for that purpose, and several convex polyhedron libraries are freely available on the web [9, 1, 18]. They are all based on an algorithm due to Chernikova [4].

3 The resolution algorithm

3.1 The constraints domain

The input constraint language combines Boolean and numeric linear variables, constants, and operators. The syntax rules are given in Figure 2. The top-level constraint is a Boolean expression ($\langle e_b \rangle$).

[1]Existing libraries are not restricted to bounded polyhedron, but for software testing purposes, we are only interested in bounded ones.

$$\begin{array}{lcl}
\langle e_b \rangle & \rightarrow & V_b \mid \text{true} \mid \text{false} \mid \text{not } \langle e_b \rangle \mid \langle e_b \rangle \star_b \langle e_b \rangle \mid \langle e_n \rangle \star_n \langle e_n \rangle \mid (\langle e_b \rangle) \\
\langle e_n \rangle & \rightarrow & V_n \mid \mathcal{N} \mid \mathcal{N}.\langle e_n \rangle \mid \langle e_n \rangle \star_\pm \langle e_n \rangle \mid \text{if } \langle e_b \rangle \text{ then } \langle e_n \rangle \text{ else } \langle e_n \rangle \mid (\langle e_n \rangle) \\
\star_b & \rightarrow & \vee \mid \wedge \mid \text{xor} \mid \implies \mid ... \\
\star_n & \rightarrow & > \mid \geq \mid < \mid \leq \mid = \\
\star_\pm & \rightarrow & + \mid -
\end{array}$$

$\mathcal{N}$, V_b, and $V_n$respectively stand for numeric constants, Boolean variables, and numeric variables.

Figure 2: Constraint syntax rules

3.2 Get rid of if-then-else

The first step is to transform constraints to remove if-then-else constructs. Indeed, together with the comparison operators, the "if-then-else" construct lets one combine numeric and Boolean arbitrarily deeply. And this does not fit in the resolution scheme we propose later in Section 3.3. The key idea of the transformation is to put the formula into the normalized form:

$$if\ c_1\ then\ e_1\ else\ if\ c_2\ then\ e_2\ else\ \ldots\ else\ if\ c_n\ then\ e_n$$

where the Boolean expressions $c_1, \ldots, c_n$ do not contain "if-then-else". This transformation can be done recursively on the constraint syntax structure, as described in Figure 3. This transformation have the property to produce a set of conditions $\{c_1, \ldots, c_n\}$ that forms a partition ($i \neq j \implies c_i \wedge c_j = false$, and $\vee_{i=1,n} c_i = true$). Therefore, for the sake of conciseness, we note such expressions as a set of couples made of a condition and a numeric expression: $\{(c_i, num_expr_i)\}_{i=1,n}$.

If $t(e_1) = \{(c_1^i, e_1^i)\}_{i=1,n}$ and $t(e_2) = \{(c_2^j, e_2^j)\}_{j=1,m}$, then we have:

- $t(e_1 + e_2) = \{c_1^i \wedge c_2^j, e_1^i + e_2^j\}_{i=1,n}^{j=1,m}$ (ditto for "$-$", "$*$", etc.)
- $t(if\ c\ then\ e_1\ else\ e_2) = \{(t_{\mathbb{B}}(c) \wedge c_1^i, e_1^i)\}_{i=1,n} \cup \{(\overline{t_{\mathbb{B}}(c)} \wedge c_2^j, e_2^j)\}_{j=1,m}$
- $t_{\mathbb{B}}(e_1 \leq e_2) = t_{\mathbb{B}}(e_1 - e_2 \leq 0)$
- $t_{\mathbb{B}}(e_1 \leq 0) = \vee_{i=1,n}(e_1^i \leq 0 \wedge c_1^i)$ (ditto for "$\geq$", "$<$", "$>$", "$=$", "$\neq$")

Figure 3: Remove "if-then-else" from constraints. t transforms numeric expressions, and $t_{\mathbb{B}}$ transforms Boolean expressions.

During this transformation, one can simplify the resulting set by merging conditions corresponding to the same numeric expressions, and by removing couples where the condition is false. However, the transformation into BDD performed later will automatically do that.

3.3 A two-layered resolution scheme

Solving Booleans. We first replace numeric constraints by new intermediary Boolean variables: $\alpha_i = n_1 \star_n n_2$. The resulting expression contains only Boolean variables and operators, and can therefore be translated into a BDD. This BDD provides the set all the Boolean solutions of the constraint.

Solving Numerics. For each of the Boolean solution, namely, for each path in the BDD, we obtain a set of linear numeric constraints $\{\alpha_i\}_i$. Those constraints are sent to a numeric constraint *solver* that is based on a convex polyhedra library. On demand, the solver can return the set of generators corresponding to the convex polyhedron defined by the sent constraints. Of course, among the Boolean solutions, some of them are associated to an empty set of solutions.

In the end, each constraint is translated into a BDD that represents a union of (possibly empty) convex polyhedra.

4 Choosing solutions

In order to generate test sequences, once the set of solutions is computed, one of those has to be chosen. Using convex polyhedron, this set of solutions is represented by a set of generators, which makes it very easy to favor limit cases. A little bit more complex task is to perform a fair choice efficiently. However, as we discuss later, being fair sometimes costs too much. We present in the sequel some heuristics leading to reasonable trade-offs.

4.1 Random choice of Boolean values

The first step consists in selecting a Boolean solution. Once the constraint has been translated into a BDD, we have a (hopefully compact) representation of the set of solutions. We first need to randomly choose a path into the BDD that leads to a true leaf. But if we naively perform a fair toss at each branch of the BDD during this traversal, we would be very unfair. Indeed, consider the BDD of Figure 1; the monomial ade has 50% of chances to be tried, whereas $\overline{a}bc\overline{e}$, and $\overline{a}\overline{b}d$ have 25% each. One can easily imagine situation where the situation is even worse. This is the reason why counting the solutions before drawing them is necessary.

Note that in order to count the number of solutions, we cannot use integers, or even doubles. Indeed, we would be restricted to 32 or 1024 variables[2]. One possibility would be to use unbounded integers. However, for performance reasons, we have preferred to implement a kind of big float data structure, where the mantissa and the exponent are represented by unsigned integers. Indeed, we just need to add and to multiply positive integers, and such a representation

[2]Keep in mind that every atomic numeric constraint is encoded into a Boolean variable during the transformation, therefore 1024 is not that big.

makes it very cheap. The slight loss of precision is also insignificant for our purposes.

Once each branch of the BDD is decorated with its solution number, performing a fair choice among Boolean solutions is straightforward.

4.2 Random choice of numeric values

4.2.1 Taking numerics into account during the BDD traversal

From the BDD point of view, numeric constraints are just Boolean variables. This means that a solution from the logical variables point of view may lead to an empty set of solutions for numeric variables.

A naive method would be to select at random a path in the BDD, and then to check if that selection corresponds to a satisfiable problem for the numeric constraints. If it is not the case, then we should start again from the last choice point, namely, from the last node in the BDD path that corresponds to a numeric variable. Indeed, if we do not start from that last choice point but from the BDD top-level node, we change the probability because we give more chances to the BDD part that have less unsatisfiable paths for numeric reasons. The problem with this method is that it could lead to a big number of such backtracking steps before finding a valid numeric solution.

An alternative method that would avoid such useless backtracking consists in solving the numeric constraints during the traversal, in order to be able to cut zero-solution branches earlier. But then we are faced to the following efficiency issue: consider the following constraint : $a+b+c<1 \ \wedge \ a+b=2 \ \wedge \ b-c=3$, and suppose that the constraint $a+b+c<1$ appears first during the BDD traversal; this means that a polyhedron of dimension 3 will be created although the problem is of dimension 1. Maybe for dimension 3 it is not a major problem, but for higher dimensions it can be. Indeed, solving such linear constraints using convex polyhedron libraries is exponential in the dimension of the polyhedron.

Hence, we choose to implement an intermediary solution: take into account constraints of dimension 1 during the random selection[3], and delay constraints of higher dimensions until the a leaf is reached. If the set of solutions becomes empty during the draw, we backtrack to the previous choice point as in the first method. Whenever an equality is traversed during the draw, we apply the corresponding substitution to the set of delayed constraints, and check whether some of them become of dimension 1. If it is the case, such awaken constraints are sent to the solver. At the end of the BDD traversal, when a leaf is reached, delayed constraints are sent to the solver; and again, if the set of solutions is empty, we backtrack.

4.2.2 Favoring limit cases

In order to generate value sequences for feeding a program under test, it is often useful to try limit values at domain boundaries. Since convex polyhedron

[3] solving linear constraint on intervals does not require any convex polyhedron library.

libraries return the set of polyhedron generators, choosing randomly among vertices, or edges, or faces is easy.

One heuristic we use that is computationally cheap and that appears to be quite effective is the following. Consider a set of n generators $\{\gamma_i\}_{i=1,n}$ of a polyhedron of dimension k.

1. Draw one generator p.
2. Draw another generator γ_j in $\{\gamma_i\}_{i=1,n}$.
3. Draw a point p' between p and γ_j.
4. Go back to step 2 with $p = p'$, $k - 1$ times.

The advantage of this heuristic is that, since at step 2 the same γ_j can be chosen several times, vertices are favored, and then edges, and then faces, and so on, whatever the dimension of the polyhedron is.

4.2.3 Drawing numerics uniformly

At the end of the process, we have a valuation for each of the Boolean variables, plus a set of generators representing several possible valuations for the numeric variables. In order to complete the random selection process, one needs to randomly choose such a numeric valuation using the generators.

The only method we are aware of to perform this choice uniformly is to draw inside the smallest parallelepiped parallel to the origin axes containing the polyhedron until a point inside the polyhedron is found. That parallelepiped can be obtained by computing the minimum and the maximum values of generators for each of their components.

4.3 Fairness versus efficiency

4.3.1 Fairly choosing numerics is expensive

The algorithm proposed in 4.2.3 suffers from a major performance problem. Indeed, drawing into the smallest parallelepiped parallel to the axes is not that expensive: O(n.d), where d is the polyhedron dimension (d), and n the number of generators (the draw is O(d) by itself, but obtaining the parallelepiped is O(n.d)). But the number of necessary draws depend on the ratio between the volume of the polyhedron and the volume of the parallelepiped. And this ratio can be very small.

For example, when the dimension of the polyhedron is smaller than the one of the parallelepiped, the theoretic ratio is 0. It is not always true for the numeric values effectively representable on a machine, but still, the ratio is very small. By changing the base using a Gauss method, one can augment this ratio. But as the dimension increases (≥ 10), doing that is not sufficient.

A solution would be to compute the smallest surrounding parallelepiped (via rotations), but this ought to be very costly. We have also considered performing

a random walk in the polyhedron: but in order to know when to stop the walk, we need to know the volume of polyhedron, which is also very expensive [12].

A rather efficient algorithm to draw inside a convex polyhedron is to use a variant of the algorithm of Section 4.2.2, choosing a different generator each time at step 2. But this leads to a distribution that is not uniform: points tend to concentrate close to vertices. To our knowledge, there is no computationally simple way to perform such a uniform draw. However, for high dimensions, this seems to be a reasonable trade-off, especially for testing purposes.

Even if it means to lose completely the control over the distribution, another thing that could be done would be to use enumerative techniques based on Simplex.

4.3.2 Combining Booleans and numerics

In some cases, the algorithms we have presented so far may lead to counter-intuitive distribution. Consider for example the constraint over the integer variable x: "$0 < x < 100 \ \wedge \ x \neq 2$". In the corresponding BDD, one path will lead to a polyhedron made of the point $x = 1$, and the other one to the polyhedron made of points between 3 and 99. And each of those paths will have the same probability to be chosen (if we count the Boolean solution numbers).

In order to be fair, we need to compute the polyhedron volume for each path, and take it into account when counting the number of solutions. But this computation is very expensive for high dimensions. Moreover, since different polyhedra correspond to different paths in the BDD, we need to change a little bit our BDD representation as follows: a BDD node is not only associated to a Boolean variable or an atomic numeric constraint (noted α_i in 3.3); it is also associated the set of atomic numeric constraints that are between the node and the top-level node. Doing that, we loose some the shareness in the BDD: the one that concerned numeric constraints. Therefore, taking into account the volume of polyhedron definitely needs to be an option.

5 Available Tools

All the tools presented in the sequel are freely available on the web at the URL: `http://www-verimag.imag.fr/ synchron/index.php?page=tools`

LuckyDraw. The solving and drawing algorithms presented here are provided under the form of an Ocaml and a C API [4]. Both the underlying BDD and polyhedra library have been developed at Verimag and are available separately.

This library is used in Rennes by the STG tool (Symbolic Test Generation). STG aims at generating and executing test cases using symbolic techniques [10]. LuckyDraw is used at the final stage in order to generate a concrete trace sequence from a symbolic automaton describing several scenarii.

[4] Many thanks to B. Jeannet for the C-Ocaml interfacing work

Lucky, Lutin, Lurette. The LuckyDraw library is one of the main building-block of Lutin and Lucky[5], languages dedicated to the programming of stochastic reactive systems. Basically, the constraint language presented here is extended with (1) an explicit control structure, (2) a mechanism to instantiate input and memory variables, (3) and external function calls (to be applied on input and memory variables only). Those languages were originally designed to model reactive program environments in the Lurette testing tool [8].

Some issues with the current version of those tools. In our implementation, numeric values are represented by rationals, because the polyhedron library we use uses rationals. However, Using the same representation as the program under test (typically, floats or doubles) would certainly be better, in particular for testing.

Integers are also approximated by rationals: we draw a rational, and then we truncate it to obtain an integer. If the obtained solution is not valid, we draw another one. This process is problematic when the number of integer solutions is small, and pathologic for non-empty rational polyhedra that do not contain any integer solution. When no valid solution is found after a certain number of tries, our current implementation (maybe wrongly) pretends that there is no solution at all. It would be better to use a finite domain solver in that case, which should do quite well in such cases where the domain is small.

We could use such finite domains solvers from the beginning, but constraint solving for linear systems is very hard, in particular when the domain is big.

6 Related work

A lot of authors describe how to generate random-based test sequences using Constraint Logic Programming (CLP) or other external constraint solvers. Constraint-based techniques tackle quite general constraints, whereas we focus on linear constraints. Moreover, most authors uses enumerative techniques such as SAT for booleans and simplex for numerics, whereas we use more constructive techniques (BDD and convex polyhedron). The main advantage of constructive techniques is to provide a finer-grained control over the distribution of the values to be generated. Besides, very few author describe precisely the drawing heuristics they use, in particular with respect to numeric values.

Glass-box testing. Some authors [7, 2, 13] aim at generating input values in order to reach the maximal level of coverage with respect to a given criterion. The program under test is encoded into a CLP program, in such a way that generating inputs to cover a given path in the control flow graph consists in writing suitable CLP requests. Constraint filtering (constraint propagation) phases are combined with labelling phases, using several heuristics, such as: selecting the variable with the smallest domain; selecting the more constrainted

[5] www-verimag.imag.fr/∼synchron/tools.html

variable; or splitting domains. Somehow, using heuristics that way during the labelling leads to different ways of generating test data as in our work, but it is not clear which heuristics lead to what distribution for output variables in their framework (it was not their objective).

Drawing in a graph. Several works describe constraints-based methods [5] and heuristics [15] to generated random test values using graphs. But as already mentioned above and in Section 5, we also have an explicit control structure in order to control finely the distribution [16] (although we hardly describe this in this article).

Other work uses constraint solvers to generate test sequences for B and Z specifications [11]. Their test objective is to generate values that exercise their boundaries. A Finite State Automation (FSA) that represents a set of abstract executions is obtained via a reachability analysis. Then, they try to find a concrete path in the abstract FSA to reach desired states. The way they concretize a trace from a FSA is comparable to what we do with Lucky [16], the difference being that their FSA are automatically generated, whereas we provide a language to program them. In other words, we focus on the stochastic concretization whereas they focus on the generation of the FSA. In [6], we described how Lucky FSA can be generated using the Nbac abstract interpretation based tool in order to reach desired set of states. Those FSA were actually Lucky programs, that are simulated (concretized) using the algorithms presented here.

Generating floating-point numbers values. Another difference with most works using constraint solvers to generate test is that they use finite domain solvers, whereas we more specifically deal with floating-point numbers or rationals. The domain of floating-point numbers is also finite, but it is much bigger and finite domain solvers are quite inefficient with floats.

Solvers dedicated to floating-point numbers exist, although they are not always well-suited for program analysis in general, and test sequence generation in particular. [14] proposes specific constraint solving algorithms that pay particular attention to mismatch between reals and floats, as well as to rounding errors performed by usual solvers. The test generation performed using those algorithms is similar to previously mentioned article: they try to reach specific program points using verification techniques.

A language-oriented approach. Another difference with other works on constraint based program testing is that we adopt a language-oriented approach. Basically, when one wants to test a program using formal techniques, it is because the state space is too big to perform an exhaustive exploration. Instead of providing methods and algorithms to prune out some of the branches of the exploration graph, we provide random based programming languages (Lucky, Lutin) to explore the state space [16].

7 Conclusion

We have presented algorithms to solve linear constraints combining Boolean and numeric variables, as well as several heuristics to choose data values among the constraint solutions. Albeit they sometimes handle non-linear constraints, other constraint based techniques for generating test sequences generally targets finite domain variables (integers). Moreover, they are based on enumerative techniques (SAT, Simplex) that make it difficult to provide a fine-grained control over the distribution of the generated values. The algorithms and the associated library presented in this article are used as one of the main component of automatic test generation tools [17, 8].

References

[1] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *SAS*, volume 2477 of *LNCS*, pages 213–229. Springer, 2002.

[2] B. Botella, A. Gotlieb, C. Michel, M. Rueher, and P. Taillibert. Utilisation des contraintes pour la génération automatique de cas de test structurels. *Technique et Science Informatiques*, 21(9):1163–1187, 2002.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[4] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6), 1968.

[5] Alain Denise, Marie-Claude Gaudel, and Sandrine-Dominique Gouraud. A generic method for statistical testing. In *ISSRE*, pages 25–34. IEEE Computer Society, 2004.

[6] F. Gaucher, E. Jahier, F. Maraninchi, and B. Jeannet. Automatic state reaching for debugging reactive programs. In *AADEBUG, Fifth Int. Workshop on Automated and Algorithmic Debugging*. HAL - CCSd - CNRS, November 14 2003.

[7] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.

[8] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer (STTT)*, Special Section on Leveraging Applications of Formal Methods, 2006.

[9] B. Jeannet. *The Polka Convex Polyhedra library Edition 2.0*, May 2002. www.irisa.fr/prive/bjeannet/newpolka.html.

[10] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 349–364. Springer, 2005.

[11] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In L.H. Eriksson and P. A. Lindsay, editors, *FME*, volume 2391 of *LNCS*, pages 21–40. Springer, 2002.

[12] L. Lovász and M. Simonovits. Random walks in a convex body and an improved volume algorithm. *Random Structures and Algorithms*, 4(4):359–412, 1993.

[13] B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel. In *ASE*, pages 229–, 2000.

[14] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In Toby Walsh, editor, *CP*, volume 2239 of *LNCS*, pages 524–538. Springer, 2001.

[15] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In BRICS, editor, *Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01)*, pages 47–60, Aalborg, Denmark, 2001.

[16] P. Raymond, E. Jahier, and Y. Roux. Describing and executing random reactive systems. In *4th IEEE International Conference on Software Engineering and Formal Methods*, Pune, India, September 11-15 2006.

[17] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[18] D. Wilde. A library for doing polyhedral operations, 1993.