Proceedings of the 1st workshop on Constraints in Software Testing, Verification and Analysis

(CSTVA'06)

a CP'06 workshop
September 25-29, 2006

Cité des Congrès - Nantes, France

Benjamin Blanc, Arnaud Gotlieb, Claude Michel (eds)

25 Sept. 2006

Preface

Recent years have seen an increasing interest in the application of constraint solving techniques to the field of testing and analysis of software systems. A significant part of constraint-based techniques have been proposed and investigated in program analysis (static and dynamic), structural testing, model-based testing, property-oriented testing, etc. These applications also introduced specific issues which was at the root of dedicated constraint solving techniques. Among these issues, we may cite the requirement for heterogeneous type handling, the ability to deal with control structure, or the need to solve constraints over specific domains like the floating-point numbers. Thus, we initiated this workshop with the hope that people coming from these communities could meet together to exchange experiences and fruitful ideas which can enhance their current and future works.

The need for such a workshop has raised from our discussions within the V3F project. This project started three years ago with the aim to address the specific features of floating-point computations in critical software verification and validation. The underlying technology of the project was based on constraint solving techniques over floating-point numbers. This first workshop on Constraints in Software Testing, Verification and Analysis (CSTVA'06), a workshop of the twelfth International Conference on Principles and Practice of Constraint Programming, emerged from the will of the participants of the V3F project to promote the ideas developed during the last three years on this topic. In addition, the organizers of the CPsec workshop, Giampaolo Bella, Stefano Bistarelli, Simon N. Foley and Fred Spiessens contacted us to propose to transfer some of their papers to CSTVA. CPsec is a workshop dedicated the applications of constraint satisfaction and programming to computer security. Therefore, it shares significant interests with the CSTVA workshop. After a careful reviewing process, the program committee has selected 6 papers. This set of papers address issues of program analysis and constraint programming as well as computer security and constraint programming:

- Michel Leconte and Bruno Berstel propose to extend a CP solver with congruence domains to avoid the slow convergence phenomenon due to the large domains that program analysis has often to handle.
- Erwan Jahier and Pascal Raymond describes algorithms to solve Boolean and numerical constraints, and to draw values among the set of solutions.
- Qiang Fu, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor detail a method based on geometric modeling using polyhedra to verify that automated and manual transformations of program preserve the functionality of the software.
- Fred Spiessens, Luis Quesada and Peter Van Roy propose the use of a constraint propagator for reachability in directed graphs to help to solve the problem of confinement.
- Yannick Chevalier and Mounira Kourjieh present a method relying on the reduction of constraint solving to take into account, at the symbolic level, that an in-

truder actively attacking a protocol execution may use some collision algorithms in reasonable time during an attack.

- Najah Chridi and Laurent Vigneron investigate a strategy, based on both a services driven model for group protocols and constraint solving, for flaws detection for group protocols properties.

In addition, a paper describing the V3F project was included in the proceedings. It covers many aspects of the role of constraint solving in software verification and security analysis.

Acknowledgement: We would like to thank Barry O'Sullivan for its support of this workshop, the program committee and also the external reviewers for the high quality of their reviews. We also address our gratitude to Lydie Mabil for the design and maintenance of the website dedicated to the workshop.

Benjamin Blanc, Arnaud Gotlieb, Claude Michel
(co-organizers of CSTVA'06)

Program Committee

- Fabrice Bouquet, INRIA Nancy
- Andy King, University of Kent
- Bruno Legeard, Leirios Technologies
- Bruno Marre, CEA Paris
- Christophe Meudec, Institute of Technology Carlow
- Fred Mesnard, University of La Réunion
- Andreas Podelski, Max-Planck-Institut fr Informatik
- Jean-Charles Régin, ILOG S.A.
- Michel Rueher, University of Nice
- Pascal Van Hentenryck, Brown University

Additional reviewers

Frédéric Besson, Martine Ceberio, Yannick Chevalier, Tristan Denmat, Katy Dobson, Patricia Hill, Bruno Martin, Matthieu Petit, Michael Rusinowitch, Fred Spiessens.

List of Papers

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")[5]. 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

[5] 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[6], the computation of $(10.00000000000004 - 10.0)/(10.000000000000004 - 10.0)$ yields[7] 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.

[6] This example comes from http://www.cs.princeton.edu/introcs/91float/.
[7] 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 $[1.4012984643248170e-45, 32767.9990234]$ 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[8]. 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.

[8] 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

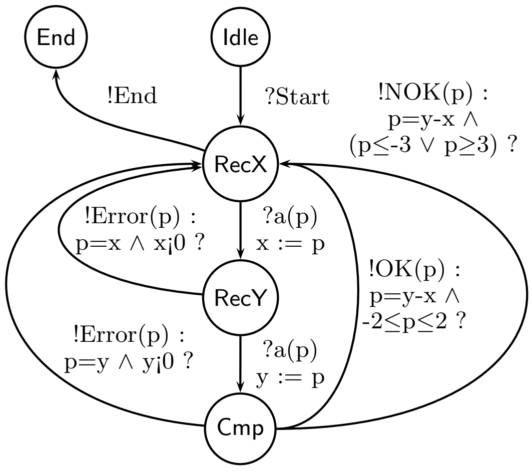
The aim is here to check whether or not a program is conform (with a given meaning) to a formal specification. To take floating-point numbers into account, a tolerance should be introduced in the conformance relation: floating-point values are allowed to differ to some degree from the real values of 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[9]. 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.

[9] 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[MA00] 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.[10]

[10] 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 [0.0199999999999999983, 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. 0.0199999999999999983) 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.

[MA00] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions: Gatel. In *In Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000)*, 2000.

[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.

Extending a CP Solver with Congruences as Domains for Program Verification

Michel Leconte[1] and Bruno Berstel[1,2]

[1] ILOG
9, rue de Verdun – 93250 Gentilly – France
[2] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85 – 66123 Saarbrücken – Germany
{leconte,berstel}@ilog.fr

Abstract. Constraints generated for Program Verification tasks very often involve integer variables ranging on all the machine-representable integer values. Thus, if the propagation takes a time that is linear in the size of the domains, it will not reach a fix point in practical time. Indeed, the propagation time needed to reduce the interval domains for as simple equations as $x = 2y + 1$ and $x = 2z$ is proportional to the size of the initial domains of the variables. To avoid this *slow convergence* phenomenon, we propose to enrich a Constraint Programming Solver (CP Solver) with *congruence domains*. This idea has been introduced by [1] in the abstract interpretation community and we show how a CP Solver can benefit from it, for example in discovering immediately that $12x + |y| = 3$ and $4z + 7y = 0$ have no integer solution.

1 Introduction

Programs made of production (or condition-action) rules [2–5], which are at the basis of Business Rules Management Systems (BRMS) [6, 7], are gaining more and more interest in the industry, as a way to externalize the rapidly changing behaviors of applications, due to frequent regulation updates or to competitive pressure. However, in order to deliver the expected agility and robustness, the business users expects from the BRMS that they assist them in mastering their rapidly growing rule bases. This concern includes the verification of properties on the rule programs, as well as code navigation tools that are aware of the rule program semantics.

Program verification problems, that is, the question of whether a program satisfies a given property, can often be formulated as satisfiability and non-satisfiability problems. Using a CP Solver to solve such problems has the advantage of being able to address a large class of formulas. This comes at the price of completeness, but practical experience shows that it is most of the time effective [8]. The solutions that are found correspond to answers (witnesses or counterexamples) to the program verification questions.

However, the constraint problems that derive from program verification questions carry specificities that challenge the efficiency of a “plain” CP Solver. In particular, and in constrast with combinatorial optimization problems, the domains of the input variables are very large, being typically only bounded by the machine representation of numbers. Also, because program verification often works by refutation, verification problems tend to produce unsatisfiable constraint problems. Since the time taken by CP Solvers to conclude to unsatisfiability may be proportional to the size of the domains of the variables, they result in being inefficient in some cases, and especially on bug-free programs, for which the constraint problems are unsatisfiable. This is illustrated by Example 1 in Section 2.

In this paper we propose to incorporate *congruences as domains* into CP Solvers, in order to prove the unsatisfiability of equations on integer variables more efficiently, that is, in a time independent from the size of the domains of the variables. Congruences are about the division of an integer by another, and the remainder in this division. Congruence analysis comes from the static analysis community. It has been introduced by Granger in [1]. As shown in that paper [1], congruence analysis is not restricted to linear equations, but can handle also general multiplication expressions. We extend its scope to other non-linear expressions such as absolute value, or minimum.

In Section 2 we describe the slow convergence issue, and how it relates to Program Verification. Then in Section 3 we study the existing related work. The technique of congruences as domains, as well as the scope we address, are introduced in Section 4. Section 5 provides the formulas for propagation, and Section 6 studies the cooperation between congruence and interval domains. Finally Section 7 illustrates the slow convergence phenomenon with experimental data, Section 8 presents the usage that was made of this technique in ILOG's commercial products, and Section 9 concludes.

2 Slow Convergence in Program Verification

In this paper we will use the following notations (all variables below are elements of $\mathbb{Z}$, the set of the integers).

- We will note $a\mathbb{Z} + b$ the set $\{az + b \mid z \in \mathbb{Z}\}$.
- For $x \in a\mathbb{Z} + b$, we will also note $x \equiv b\,[a]$.
- We will use $a \wedge b$ for the greatest common divisor of a and b.
- We will use $a \vee b$ for the least common multiplier of a and b.

As mentioned in the introduction, a particularity of constraint problems related to program verification, is that the input variables behave as if not bounded. For example, integer input variables are often supposed to take any value from machine integers according to their type. This is completely different from what happens when using a constraint solver for solving combinatorial problems, where we always try to reduce the initial domain of variables as much as it can be with respect to the problem.

Consider a simple constraint such as, say, $2x + 2y = 1$ where x and y are integer variables with value ranging from $-d$ to d, for some integer d. Obviously, there is no solution to this constraint, and the "usual" interval reduction will find it, by reducing the domains of x and y down to empty sets. But to achieve this, the interval reduction will have to step through all the domains $[-d, d]$, then $[-d+1, d-1]$, etc. up to empty ones.

We stress that this *slow convergence* phenomenon occurs during the propagation of constraints: the time taken to reach a fix point is asymptotically proportional to the width of the domains. Propagation occurs both initially and during labelling; as a result, slow convergence may happen when searching for a solution. For example, the equation $2x + 2y + z = 1$ leads to the (slow) propagation of the previous constraint $2x + 2y = 1$ if z has been assigned 0 during the search. It is to avoid both cases that we have implemented congruences as domains.

The slow convergence issue over real numbers has motivated the development of special interval narrowing techniques in [9]. Unfortunately, they do not apply to the integer issue.

From the point of view of program verification, such problems can occur in various situations. Consider for instance a program containing the following loop: `while (x is even) increment x`. This program always terminates. As mentioned in [10], a way to prove it is to prove that the conjunction of the loop test expressed on the program states before and after one loop step, is unsatisfiable. Here the program state is the value of x; after one loop step it equals $x + 1$. To conclude to the termination of the program, a prover may thus want to show that the constraint $\texttt{even}(x) \wedge \texttt{even}(x+1)$ is unsatisfiable. This boils down to the constraint problem exposed above, with the same slow convergence problem.

As other examples of program verification problems that lead to proving that a conjunction of integer equalities is unsatisfiable, consider the problem of overlapping conditions in a guarded integer program. The original motivation for the work presented in this paper comes from the verification of rule programs, but the overlapping conditions problem may arise in any context that involves guarded commands.

Example 1. Consider the following program, which implements in some fictious guarded command language a simple version of the function that returns the number of days in a Gregorian calendar year.

```
function nbOfDays (y : int) : int is
   y === 0 mod 4  ->  366  |
   y === 1 mod 4  ->  365  |
   y === 2 mod 4  ->  365  |
   y === 3 mod 4  ->  365
end
```

The question is whether the guards in this program overlap or not.

The answer is 'no', that is, the program is bug-free (with respect to the question). To prove it, we shall first translate the guards into constraints, which gives the four constraints $y = 4x_i + i$, for $i = 0, 1, 2, 3$. In these constraints y and x_i are integer variables lying in $[-d-1, d]$ for some integer d (potentially $2^{31} - 1$). Then we shall prove that for any two distinct i and j between 0 and 3, the conjunction $y = 4x_i + i \wedge y = 4x_j + j$ is unsatisfiable. As seen previously in this section, interval reduction can achieve this, but will need d steps. The rest of this paper shows that using congruences as domains allows a CP Solver to prove the unsatisfiability in a fixed number of steps.

3 Related Work

Congruence analysis has been introduced by Granger [1, 11] with applications to automatic vectorization. Today congruence analysis is an important technique, especially to verify pointer alignment properties [12, 13]. In this paper we extend its scope beyond linear equations and multiplication, to other non-linear expressions such as absolute value, or the minimum operator.

Congruence domains have also been extended to constraints of the form $x - y \equiv b\,[c]$ [14, 15]. Toman *et al.* proposed in [16] a $O(n^4)$ normalization procedure for conjunctions of such constraints, Miné improved it to $O(n^3)$ in [14]. Grids [17, 18] are another extension which addresses *relational* congruence domains, in the presence of equalities of the form $\sum a_i x_i \equiv b\,[c]$, while we address the more specific non-relational domains for $x \equiv b\,[c]$. In [19], Granger proposes an extension of the congruence analysis by considering sets of rationals of the form $a\mathbb{Z} + b$, where $a, b \in \mathbb{Q}$.

In this paper, we extend a solver based on the finite domain CP Solver ILOG JSolver [20] with congruence as domains for variables. Very few CP Solvers reason with congruence. The Alice system [21] and its successor Rabbit [22] implement some congruence reasoning capabilities as part of formal constraint handling. Let us look at an example, which was taken from [22].

Example 2. Find all integer *positive* solutions of $x^3 + 119 = 66x$.

From $x^3 < 66x$, Rabbit finds that $x^2 < 66$, and then $x < 8$. Rabbit then performs two factorizations, namely $119 = 7 \times 17$ and $66 = (3 \times 17) + 15$. It then uses a congruence reasoning to deduce from these factorizations that $x^3 \equiv 15x\,[17]$, which can also be written $x(x^2 - 15) \equiv 0\,[17]$. Rabbit then applies the deduction rule

$$\text{if } a \times b \equiv 0\,[k] \text{ and } k \text{ is prime, then } a \equiv 0\,[k] \text{ or } b \equiv 0\,[k]$$

to deduce that $x \equiv 0\,[17]$ or $x^2 \equiv 15\,[17]$. Since x is positive and $x < 8$, the constraint $x \equiv 0\,[17]$ is always false. Finally $x^2 \equiv 15\,[17]$ leads to $x = 7$ as this is the only admissible value for x among $[1, 7]$.

As we can see from Example 2, the congruence capabilities of Rabbit are pretty important, and they are based on redundant modular equations generation. In this example, it involves the factorization of 119 as 7×17.

To solve this example, congruence domains of the form $a\mathbb{Z} + b$ as we propose are not enough. However, here pure interval reduction is enough to find the solution. When the example is given to ILOG JSolver, the domain of x starts at $[1, 2^{31} - 1]$, and is reduced to $[2, 1290]$, then to $[3, 43]$, to $[5, 13]$, to $[6, 9]$, to end with $x = 7$.

As we will see, interval and congruence domains interact smoothly [1, 12, 13]. This is an example of the so-called reduced product operation of the theory of abstract interpretation [1, 23, 24]. Numeric domains such as intervals and congruences are available in Static Analysis Systems such as Astrée [25] or the Parma Polyhedra Library [26, 27].

Finally, another approach to avoiding the slow convergence problem on constraints such as $x = 2y + 1$ and $x = 2z$, is to use an integer linear solver inside the CP Solver. Note that this would handle only linear equations.

4 Congruences as Domains

4.1 Scope

The scope of constraints that we consider here extends to any equality constraint over integer variables and expressions. The integer expressions are built using the usual $+$, $-$, $\times$, $\div$ arithmetic operators, as well as the power, absolute value, minimum, and maximum ones.

We also consider *element expressions* in the form $\mathbf{t}[i]$, where $\mathbf{t}$ is an array and i is an integer variable. The element expression denotes the i-th element of the array. In simple element expression, this element is an integer; in generalized ones, the element is an integer variable. An *element constraint* is a constraint of the form $z \in \{\mathbf{t}[i]\}$, which amounts to the disjunction $\bigvee_i z = \mathbf{t}[i]$.

Finally, we also consider *if-then-else expressions* in the form $\mathsf{if}(c, e_1, e_2)$, where c is a constraint, and the e_i are integer expressions. The if-then-else expression denotes the e_1 expression if the constraint c is true, and the e_2 expression if the constraint c is false.

Although the whole range of integer constraints is covered, congruence analysis is of course not a decision procedure. That is, congruence analysis alone will not always detect the unsatisfiability of a set of integer constraints. And this does not harm, since it is simply meant to strengthen the constraint propagation.

4.2 Using Congruences on Interval Domains

Example 3. Find all integer solutions of $2x + 4y + 6z = 1$.

On the example above, the interval-based constraint propagation will perform no bound reduction at all. In particular, the unsatisfiability of the constraint will not be detected by constraint propagation.

A congruence reasoning shows that the expression $2x + 4y + 6z$ is even, and thus cannot be made equal to 1. At least, this illustrates a missing propagation.

Remember that a constraint $\sum_i a_i x_i = c$ has no solution if the greatest common divisor $\bigwedge_i a_i$ does not divide the constant c. We can use this property in the propagators of integer linear constraints: we compute the greatest common divisor of the coefficients of uninstantiated variables, and check if it divides the constant minus the sum of the $a_i x_i$ for instantiated variables.

This **passive** use of a congruence constraint, where congruences are used to update the bounds of interval domains, may already be useful. In addition it has little overhead on propagation time since this check has to be done up-front, and then only when a variable becomes instantiated.

4.3 Storing Congruence Information Separately

Example 4. Find all integer solutions of the problem made of the constraints $2x + 4y + 3z = 1$ and $z = 2t + 12$.

The passive use of congruence information just described will not detect that z cannot be even in $2x + 4y + 3z = 1$. Thus the unsatisfiability of the two constraints will be not detected. However, it would be a bad idea to use such a congruence constraint in an **active** way without caution.

Imagine for example that the congruence constraint not only checks for the constants dividing the greatest common divisor, but also *adjusts* the bounds of the domains of variables accordingly. Let us say that propagating $2x+4y+3z = 1$ would lead to adjust the bounds of z in such a way that these bounds are not even. Coming back to Example 4, an empty domain will be found for z, as the constraints will eventually lead to a domain with both odd and even bounds. Unfortunately, this would exhibit a slow convergence behavior since the bounds would change by one unit at a time.

The way to solve this last problem is to share the congruence information between constraints, that is to say to equip variables with congruence information, as opposed to hide it in the actual values of their bounds. Consequently, in the very same way we associate a point wise finite domain to each integer variable, we associate to each of them a congruence domain in the form of a pair (a, b) that represents the set $a\mathbb{Z} + b$. Then for each expression, the congruence domain of the expression can be computed from the congruence domains of the sub-expressions, using the formulas detailled in next section. Similarly the computed congruence domains are propagated by equalities constraints to reduce the congruence domains of the variables.

This way, the unsatisfiability of the two constraints $x = 2y$ and $x = 2z + 1$ is found by a congruence reasoning deducing that x should be both even and odd. This reasoning requires a number of steps which is independent from the size of the domains of the variables involved.

This active use of congruence information, where domains are reduced, subsumes the passive use described previously, which only performs divisibility checks.

5 Propagation of Congruences as Domains

5.1 Propagation Through Operations

Each integer variable has a congruence domain, noted $a\mathbb{Z} + b$, which represents all possible values for this variable. We use $0\mathbb{Z} + b$ to represent the constant b, and $1\mathbb{Z} + 0$ as the domain of a variable with all integers as possible values.

Now we have to define how to compute the congruence domain for expressions. We only give here the formulas for the addition and multiplication operations. These formulas, and those for substraction and division, can be found in [1]. Given $x \in a\mathbb{Z} + b$ and $y \in a'\mathbb{Z} + b'$:

$$x + y \in (a \wedge a')\mathbb{Z} + (b + b') \tag{1}$$

$$x \times y \in (aa' \wedge a'b \wedge ab')\mathbb{Z} + bb' \tag{2}$$

One can note that the square expression has a more precise characterization than the one derived from the general multiplication case. Given $x \in a\mathbb{Z} + b$:

$$x^2 \in (a^2 \wedge 2ab)\mathbb{Z} + b^2 \tag{3}$$

Let us look now at the union expressions, which result from constraints of the form $z \in \{x, y\}$. Given $x \in a\mathbb{Z} + b$ and $y \in a'\mathbb{Z} + b'$:

$$\text{if } z \in \{x, y\} \text{ then } z \in (a \wedge a' \wedge |b - b'|)\mathbb{Z} + b \tag{4}$$

The dissymmetry between b and b' in this formula is only apparent. Indeed, let α denote the greatest common divisor of a, a', and $|b - b'|$ appearing in (4): in particular it divides $|b - b'|$. That is, $\exists k \in \mathbb{Z}, b - b' = \alpha k$. In other words, we have $b \equiv b' \, [\alpha]$.

This formula for the union gives the formula for *if-then-else* expressions. Remember that $\mathsf{if}(c, e_1, e_2)$ is an expression taking the value e_1 when c is true and e_2 when c is false. If the constraint c is known to be true (resp. false), then the congruence domain for if-then-else is the congruence domain of e_1 (resp. e_2). However if the truth value of the constraint is unknown, then the expression has a congruence domain which is the union of the congruence domains of the two expressions. (This makes union an over-approximation of *if-then-else* expressions.) Given $x \in a\mathbb{Z} + b$, $y \in a'\mathbb{Z} + b'$, and a constraint c:

$$\mathsf{if}(c, x, y) \in \begin{cases} a\mathbb{Z} + b & \text{if } c \text{ is known to be true} \\ a'\mathbb{Z} + b' & \text{if } c \text{ is known to be false} \\ (a \wedge a' \wedge |b - b'|)\mathbb{Z} + b & \text{otherwise} \end{cases} \tag{5}$$

This also leads to the formula for a min expression, as $\min(x, y)$ is equivalent to $\mathsf{if}(x < y, x, y)$. Formulas for max and absolute value may be easily found if we remark that $\max(x, y) = \mathsf{if}(x < y, y, x)$ and $|x| = \mathsf{if}(x < 0, -x, x)$.

Finally, for an array $\mathbf{t}$ of integer variables and an integer variable i, the expression $z = \mathbf{t}[i]$ is equivalent to $z \in \{\mathbf{t}[j]\}$ for all values j which are non-negative, and less than the length of the array $\mathbf{t}$.

5.2 Propagation Through Equality

We now indicate how to deal with equality constraints. As usual when propagating through equality, we just have to compute the intersection of the domains. Given $x \in a\mathbb{Z} + b$ and $y \in a'\mathbb{Z} + b'$:

$$\text{if } x = y \text{ then } x \in \begin{cases} (a \vee a')\mathbb{Z} + b'' & \text{if } (a \wedge a') \text{ divides } (b - b') \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

The number b'' can be computed as follows. Let $x = au + b$ and $y = a'v + b'$, the equality $x = y$ gives $au+b = a'v+b'$, that is, $au-a'v = b'-b$. Since $a \wedge a'$ divides $b - b'$, this can be simplified by $a \wedge a'$ into $\alpha u - \alpha' v = \beta$. Since α and α' are relatively prime, Bezout's theorem ensures that there exist u_0 and v_0 such that $\alpha u_0 + \alpha' v_0 = 1$. These numbers can be computed using a generalized Euclid's algorithm [28]. Combining the last two equations gives $\alpha(u - \beta u_0) = \alpha'(v + \beta v_0)$. Since α and α' are relatively prime, $u - \beta u_0$ is a multiple of α'. From $x = au + b$, we have $x \in \alpha' a\mathbb{Z} + b''$, where $b'' = b + \beta u_0$.

If the equality constraint involves expressions instead of variables, then the congruence domains of the expressions are used to compute the intersection. This resulting domain is then downward propagated to the sub-expressions of the expressions until it falls back to the variables.

Example 5. Find all integer solutions to $4x = 3|y| + 2$.

Let us use Example 5 to illustrate how the propagation of congruence domains proceeds. In the absence of further information, we have $x, y \in 1\mathbb{Z} + 0$. The formulas for addition (1), multiplication (2), and absolute value (5) give that $4x \in 4\mathbb{Z} + 0$ and $3|y| + 2 \in 3\mathbb{Z} + 2$.

The formula (6) for the equality constraint gives that both expressions belong to $12\mathbb{Z}+8$. Since $4x \in 12\mathbb{Z}+8, x \in 3\mathbb{Z}+2$. Since $3|y|+2 \in 12\mathbb{Z}+8, |y| \in 4\mathbb{Z}+2$. The absolute value can be decomposed into the case where $y \in 4\mathbb{Z} + 2$, and the case where $-y \in 4\mathbb{Z} + 2$. This latter case gives $y \in 4\mathbb{Z} - 2$, which is the same as $4\mathbb{Z} + 2$. Eventually $y \in 4\mathbb{Z} + 2$. The domains cannot be further reduced: a fix point is reached.

6 Cooperation of Congruences and Intervals

The idea here is to merge the two notions and to consider domains of the form $a\mathbb{Z} + b \cap [min, max]$. In the Abstract Interpretation framework, this corresponds to the reduced cardinal product of congruence domains and interval domains. It is called Reduced Interval Congruence (RIC) in [12, 13]. By combining the two domains, information coming from interval domains will be used by the congruence domain and vice-versa.

Let us first examine how to communicate information from interval domains to congruence domains.

– When a variable is bound, as for instance in $x = b$, this can be formulated in congruences as $x \in 0\mathbb{Z} + b$.

– When it is found that $x \in \{b_i\}$ for some constants b_i, this implies that $x \in (\bigwedge_{i>0} |b_i - b_0|)\mathbb{Z} + b_0$.
– For an element constraint $z \in \{\mathbf{t}[i]\}$, the range of the variable i restricts the elements of $\mathbf{t}$ that are to be taken into account to compute the congruence domain of z.

To communicate information from congruence domains to interval domains, one will use the fact that the bounds of a variable must lie in the same congruence domain as the variable itself. That is, if $x \in [min, max]$ and $x \in a\mathbb{Z} + b$, then min and max must be adjusted in order to belong to $a\mathbb{Z} + b$. When $a \neq 0$, the adjusted min is $a\lceil (min - b)/a \rceil + b$ and the adjusted max is $a\lfloor (max - b)/a \rfloor + b$.

If the diameter $max - min$ is less than a, the variable will have a singleton or empty domain. For instance if the interval domain had been reduced to $[0, a-1]$, then the variable can be instantiated to b, which is the only element of $a\mathbb{Z} + b \cap [0, a-1]$. Similarly, if the interval domain had been reduced to $[0, |b| - 1]$, then the solver fails, as $a\mathbb{Z} + b \cap [0, |b| - 1] = \emptyset$ for any a.

Also, for an element constraint $z \in \{\mathbf{t}[i]\}$, the congruence domain of z is to be taken into account to remove from the index variable domain the values i_0 for which $z = \mathbf{t}[i_0]$ cannot be satisfied.

Let us close this section with a example showing non-trivial reductions.

Example 6. Consider the two constraints $4x = 3y + 2$ and $|x| - 12z = 2$.

We have already seen that the first constraint leads to $x \in 3\mathbb{Z} + 2$ and $y \in 4\mathbb{Z} + 2$. Now, looking at the second constraint, we deduce that $|x| \in 12\mathbb{Z} + 2$. Since $|x| = \mathsf{if}(x < 0, -x, x)$, we deduce that $x \in 12\mathbb{Z} + 10$ (if $x < 0$) or $x \in 12\mathbb{Z} + 2$ (if $x \geq 0$). Because $12\mathbb{Z} + 10 \cap 3\mathbb{Z} + 2 = \emptyset$, we are left with $x \in 12\mathbb{Z} + 2$ and $x \geq 0$.

7 Experimental Illustration

In this section we describe two examples that suffer from the slow convergence problem, and we provide experimental data that exhibits the phenomenon.

Example 7. Solve $2x + 3y + 6z = 2$, with $x, y, z \in [-10^d, 10^d]$.

In Table 1, the times mentioned are the time taken to generate the first solution. The variable x is instantiated first to -10^d, then to $-10^d + 1$ and finally to $-10^d + 2$ which leads to a solution. The numbers show that without congruence domains this time is proportional to the size of the domains, while it is not when using congruence domains. The result is then found in a time lower than what can be measured; this is interesting per se, but the table shows that this time does not depend on the size of the domains.

Example 8. Prove that $2x + 2y = 1$, with $x, y \in [-10^d, 10^d]$, has no solution.

In Table 2, the times mentioned are the time taken during the propagation, which concludes to the unsatisfiability of the constraint. Here also, the numbers show that without congruence domains this time is proportional to the size of the domains, while it is not when using congruence domains.

Value of d	Time without C.D.	Time with C.D.
4	0.04 s	0 s
5	0.19 s	0 s
6	1.88 s	0 s
7	18.85 s	0 s
8	241.82 s	0 s
9	946.57 s	0 s

Table 1. Times taken for solving Example 7.

Value of d	Time without C.D.	Time with C.D.
4	0.12 s	0 s
5	0.06 s	0 s
6	0.62 s	0 s
7	6.16 s	0 s
8	61.96 s	0 s
9	871.05 s	0 s

Table 2. Times taken in propagation for Example 8.

8 Industrial Usage

From a marketing perspective, Program Verification is an important feature for BRMS [29]. As mentioned in the introduction, it aims at helping the non-technical users to master the rule programs their author using the system. This takes both the form of verification of properties on the program, and of semantics-aware code navigation tools. For commercial products such as ILOG JRules, this is a key differentiator.

As an illustration, one of the verification tasks we perform is to detect when a rule is never applicable, that is, when the tests in its condition part are always unsatisfiable. When the user creates a rule within the Eclipse IDE [30], a rule that is never applicable will be immediately signaled, with the tests causing the unsatisfiability highlighted.

We have embedded in ILOG JRules a new verification solver, as a Java library built on a Constraint-based Programming Solver derived from ILOG JSolver [20]. This CP Solver is part of ILOG JRules since release 4.5 which was delivered in 2003, and has kept evolving since. The passive use of congruence as presented in section 4 is present in ILOG JRules since release 6.0. We are now implementing congruence as domains for the next release of ILOG JRules.

9 Conclusion

Integer constraint propagation exhibits a *slow convergence* phenomenon when the time to reach a fix point or to fail is proportional to the size of the domains of the variables.

To avoid this phenomenon for some integer equality constraints, we added to a CP Solver some congruence reasoning capabilities. We have taken the idea of equiping the variables with congruence domains from the abstract interpretation community [1], as it leads to efficient and scalable implementations. We have shown how a CP Solver can benefit from these congruence domains with several examples, concluding with illustrations on the interaction of interval and congruence domains.

This work is part of the already distributed ILOG JRules product, and will be completely integrated in the next ILOG JRules release.

References

1. Granger, P.: Static analysis of arithmetic congruences. International Journal of Computer Math (1989) 165–199
2. Allen Newell, H.A.S.: Human problem solving. Prentice Hall, Englewood Cliffs, NJ, USA (1972)
3. Davis, R., Buchanan, B.G., Shortliffe, E.H.: Production rules as a representation for a knowledge-based consultation program. Artif. Intell. **8**(1) (1977) 15–45
4. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. **19**(1) (1982) 17–37
5. Baralis, E., Widom, J.: An algebraic approach to static analysis of active database rules. ACM Trans. Database Syst. **25**(3) (2000) 269–332
6. ILOG: ILOG JRules. (2006) `http://www.ilog.com`.
7. JBoss: Drools. (2006) `http://www.drools.org`.
8. Collavizza, H., Rueher, M.: Exploration of the capabilities of constraint programming for software verification. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 182–196
9. Lhomme, O., Gotlieb, A., Rueher, M.: Dynamic optimization of interval narrowing algorithms. J. Log. Program. **37**(1-3) (1998) 165–183
10. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In Steffen, B., Levi, G., eds.: VMCAI. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 239–251
11. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In Abramsky, S., Maibaum, T.S.E., eds.: TAPSOFT, Vol.1. Volume 493 of Lecture Notes in Computer Science., Springer (1991) 169–192
12. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In Duesterwald, E., ed.: CC. Volume 2985 of Lecture Notes in Computer Science., Springer (2004) 5–23
13. Venable, M., Chouchane, M.R., Karim, M.E., Lakhotia, A.: Analyzing memory accesses in obfuscated x86 executables. In Julisch, K., Krügel, C., eds.: DIMVA. Volume 3548 of Lecture Notes in Computer Science., Springer (2005) 1–18
14. Miné, A.: A few graph-based relational numerical abstract domains. In Hermenegildo, M.V., Puebla, G., eds.: SAS. Volume 2477 of Lecture Notes in Computer Science., Springer (2002) 117–132
15. Bagnara, R.: Data-Flow Analysis for Constraint Logic-Based Languages. PhD thesis, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy (1997) Printed as Report TD-1/97.
16. Toman, D., Chomicki, J., Rogers, D.S.: Datalog with integer periodicity constraints. In: SLP. (1994) 189–203

17. Bagnara, R., Dobson, K., Hill, P.M., Mundell, M., Zafanella, E.: Grids: A domain for analyzing the distribution of numerical values. In: LOPSTR. (2006)
18. Müller-Olm, M., Seidl, H.: A generic framework for interprocedural analysis of numerical properties. In Hankin, C., Siveroni, I., eds.: SAS. Volume 3672 of Lecture Notes in Computer Science., Springer (2005) 235–250
19. Granger, P.: Static analyses of congruence properties on rational numbers (extended abstract). In Hentenryck, P.V., ed.: SAS. Volume 1302 of Lecture Notes in Computer Science., Springer (1997) 278–292
20. ILOG: ILOG JSolver. (2000) `http://www.ilog.com`.
21. Laurière, J.L.: A language and a program for stating and solving combinatorial problems. Artif. Intell. **10**(1) (1978) 29–127
22. Laurière, J.L.: Programmation de contraintes ou programmation automatique. Technical report, L.I.T.P. (1996) `http://www.lri.fr/~sebag/Slides/Lauriere/Rabbit.pdf`.
23. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
24. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Language Design for Reliable Software. (1977) 77–94
25. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée analyzer. In: ESOP'05. (2005)
26. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In Hermenegildo, M.V., Puebla, G., eds.: Static Analysis: Proceedings of the 9th International Symposium. Volume 2477 of Lecture Notes in Computer Science., Madrid, Spain, Springer-Verlag, Berlin (2002) 213–229
27. Parma Polyhedra Library: PPL. (2006) `http://www.cs.unipr.it/ppl`.
28. Knuth, D.E.: Seminumerical Algorithms. Second edn. Volume 2 of The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts (1981)
29. Hendrick, S.D.: Business Rule Management Systems: Addressing Referential Rule Integrity. IDC. (2006) `http://www.idc.com/getdoc.jsp?containerId=201262`.
30. The Eclipse Consortium: Eclipse 3.0. (2005) `http://www.eclipse.org`.

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 non-deterministic, 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

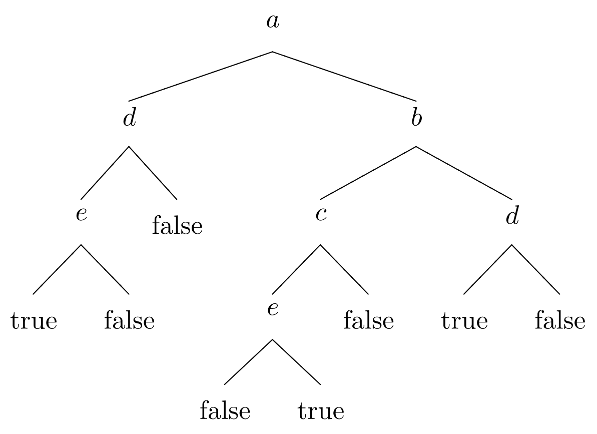


Figure 1: A BDD containing 10 solutions (ade, $\overline{a}bc\overline{e}$, and $\overline{a}\overline{b}d$).

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.

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

$\mathcal{N}$, 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.

Requirements for Constraint Solvers in Verification of Data-Intensive Embedded System Software[⋆]

Qiang Fu[1], Maurice Bruynooghe[1], Gerda Janssens[1], and Francky Catthoor[2]

[1] Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium
`qiang,maurice,gerda@cs.kuleuven.be`
[2] IMEC vzw, Kapeldreef 75, B-3001 Heverlee, Belgium
`catthoor@imec.be`

Abstract. In tuning data-intensive software such as multimedia and telecom applications for embedded processors in portable devices, designers use a combination of automated and manual transformations at the source level to optimize the resource consumption of the software. It is of crucial importance that the functionality of the software is preserved. For software with static control, a verification method exists that first transforms the code into dynamic single assignment form and next verifies the functional equivalence of the two versions. The verification is based on geometric modelling using polyhedra.
In this paper, we describe in detail the basic operations of the verification method, discuss the control issues that affect its overall performance, and analyze the functionalities that constraint solvers have to offer to handle this application.

1 Introduction

Embedded systems for the consumer electronics market run data-intensive multimedia and telecom applications on devices with severe resource constraints. Designing such systems is a complex task. Typically, designers start from an initial design assembled by a straightforward combination of trusted algorithms in a high level language. Then designers want to optimize performance, area on chip, power consumption and overall cost of the design. For that purpose, they use analysis tools and perform a mix of automated and manual transformations according to some design methodology (e.g. [4]) to parallelize the loops or to improve the array related memory management. Figure 1 illustrates a typical loop transformation which would benefit the memory size. The whole process is very error prone. Moreover, once the design in the high level language is frozen and implemented in the embedded system, the cost of bugs in the design becomes excessive. Hence there is a need for thorough testing and/or verification.

⋆ Work supported by FWO-Vlaanderen

```
void foo(int in, int b) {

  const int N=5;
  int i,j,p,k,l,a[N+1][N];

  for (i = 1; i <= N; ++i)
o1: a[0][i] = 5;
    for (j = 1; j <= N-i+1; ++j)
o2:   a[i][j] = in[i][j] + a[i-1][j];

  for(p = 1; p <= N; ++p)
o3: b[p][1] = f(a[N-p+1][p], a[N-p][p]);

  for(k = 1; k <= N; ++k)
    for (l = 1; l <= k; ++l)
o4:   b[k][l+1] = g(b[k][l]);
}

                              original program
```

```
void foo(int in, int b){

  const int N=5;
  int i,j,l,a[N+1][N];

  for(i = 1; i <= N; ++i)
t1: a[0][i] = 5;

  for(j = 1; j <= N; ++j) {
    for (i = 1; i <= N-j+1; ++i)
t2:   a[i][j] = in[i][j] + a[i-1][j];
t3: b[j][1] = f(a[N-j+1][j], a[N-j][j]);
    for (l = 1; l <= j; ++l)
t4:   b[j][l+1] = g(b[j][l]);
  }
}

                           transformed program
```

Fig. 1. Program before and after loop transformation.

For (parts of) systems with static control flow where conditions, index expressions, and bounds on iterators are (piecewise) affine functions of the bounds of the surrounding iterators and where no pointer references occur, the code can be transformed to so called Dynamic Single Assignment (DSA) code [5] where each array element is written only once by methods described in [5,15]. This meets the requirements of a very relevant subset of all possible application codes in our target domain. The resulting code is *functional*: Each element of an output array is a function of a set of elements of the input arrays. Verification of the equivalence of original and transformed code then reduces to checking for each output element that the function mapping inputs to outputs is equivalent. In order to ensure scalability to realistic data and loop sizes in the embedded system domain, the crux of the methods is to do this verification not element by element but to handle at once groups of elements for which the function is the same. Such methods are described in [1,13].

Those methods rely on the use of the *geometric model* (also called polyhedral/polytope model) for representing the meaning of programs. Geometric modelling of programs is well known in the parallel compiler and regular array synthesis research domains and is used extensively to analyze the execution of program statements [5,11,12]. The geometric model concisely represents all of the necessary information about the data and control flow in the program. The basic idea is to represent the iterations for which a statement is executed by the integer points in a polytope, i.e., a bounded polyhedron. A polyhedron is a subspace in n-dimensional space bounded by a finite number of hyperplanes. These hyperplanes can be represented as a system of linear inequalities. The latter can be extracted from the iterator bounds and conditions that control the execution of the statement.

This paper develops in more detail the method sketched in [13], discusses control issues that affect the performance, and analyzes the functional requirements for the constraint solving. In Section 2, we explain the geometric modelling of the program. The concept of proof obligations is introduced in Section 3. Also the basic operations to reduce proof obligations together with the functionality they require from the underlying constraint solvers are described in this section. In Section 4, we discuss how to avoid redundant computations during the reduction of proof obligations. Handling recurrences is presented in Section 5. In Section 6, we discuss in more detail which operations are required and how existing solvers provide support for them. Section 7 concludes.

2 Program representation under the geometric model

Geometric modelling is used by many authors. Here we recall the basics by means of some examples. Consider statement *o2* in Figure 1 (with `N` substituted by its value) which is a **writer** of array `a` and a **reader** of arrays `in` and `a`:

```
    for (i = 1; i <= 5; ++i)
       for (j = 1; j <= 5-i+1; ++j)
o2:         a[i][j] = in[i][j] + a[i-1][j];
```

The **iteration domain** is a relation over the iterators governing the statement. Each tuple (i, j) in the relation defines a value of the iterators for which the statement is executed. The relation is described by the integer points in a polytope: $D = \{(i, j) \mid i \geq 1 \wedge i \leq 5 \wedge j \geq 1 \wedge j \leq 5 - i + 1\}$.

The **definition domain** is a relation over the indices of the array in the left hand side of the statement. It defines which elements of the array are written by the statement. Also this relation can be described by the integer points in a polytope: $W_{\mathtt{a}} = \{(a_1, a_2) \mid \exists i, j : a_1 = i \wedge a_2 = j \wedge (i, j) \in D\} = \{(a_1, a_2) \mid \exists i, j : a_1 = i \wedge a_2 = j \wedge i \geq 1 \wedge i \leq 5 \wedge j \geq 1 \wedge j \leq 5 - i + 1\}$. Because the program is in DSA form, the constraints define a bijection between the indices a_1 and a_2 of the array and the iterators i and j. Using the two equalities, the existentially quantified variable i and j can be eliminated and one obtains $W_{\mathtt{a}} = \{(a_1, a_2) \mid a_1 \geq 1 \wedge a_1 \leq 5 \wedge a_2 \geq 1 \wedge a_2 \leq 5 - a_1 + 1\}$.

For each operand in the right hand side, one can define an **operand domain**. It is a relation over the indices of the operand array. It defines which elements of the array are read by the statement. Similar to the definition domain, it can be described by the integer points in a polytope. For the second operand, it is given by: $R_{\mathtt{a}} = \{(a_1, a_2) \mid \exists i, j : a_1 = i - 1 \wedge a_2 = j \wedge (i, j) \in D\}$. Note that there is a functional dependency from the iterators to the indices as one value is read in each iteration, but in general not from the indices to the iterators as the same value can be read in different iterations. Again, i and j can be eliminated and we obtain $R_{\mathtt{a}} = \{(a_1, a_2) \mid a_1 + 1 \geq 1 \wedge a_1 + 1 \leq 5 \wedge a_2 \geq 1 \wedge a_2 \leq 5 - a_1\}$.

Each executed instance of the statement reads values from elements in the operand arrays and writes a value in an element of the lhs array. For each operand, there is a **dependence mapping** that defines which operand element is read for each written element. As said above, the relation between the indices of the lhs array and the iterators is a bijection while there is functional dependency between the iterators and the indices of the operand array, hence the dependency mapping can be understood as a function from the indices of the lhs array to the indices of the operand array (and is in general not invertible). To stress that the dependence mapping encodes a functional dependency, we denote it as $M(\boldsymbol{i} \rightarrow \boldsymbol{j})$ with $\boldsymbol{i}$ the indices of the written array and $\boldsymbol{j}$ the indices of the operand array. Also this relation can be represented by the integer points of a polytope. For the second operand of our example, we have: $M((a'_1, a'_2) \rightarrow (a_1, a_2)) = \{a'_1, a'_2, a_1, a_2 \mid \exists i, j : a'_1 = i \wedge a'_2 = j \wedge a_1 = i - 1 \wedge a_2 = j \wedge (i, j) \in D\} = \{a'_1, a'_2, a_1, a_2 \mid a'_1 = a_1 + 1 \wedge a_2 = a'_2 \wedge a'_1 \geq 1 \wedge a'_1 \leq 5 \wedge a'_2 \geq 1 \wedge a'_2 \leq 5 - a'_1 + 1\}$. In general, with m the dimension of the array being written and n the dimension of the array being read, the dependence mapping is a polytope in a space of dimension $m + n$. The dimension of the polytope however can be lower, because the constraints can imply equalities.

Statements can be written in a **normal form** where all indices of the lhs array are distinct variables and the indices of the rhs arrays are functions of the indices of the lhs side. The latter correspond to the dependency mapping of the statement. This normal form, together with the constraints on the indices forms the *geometric model of the statement* and completely characterizes it. The statement *o2* is already in normal form as the indices are the iterators. The constraints are those of the iterator domain.

While in the above examples, the relation of interest is represented by *all* integer points inside a polytope, this is not always the case. Consider for example a for loop with a non-unit stride:

```
for i=1, i<=21; i= i+5
```

Its iteration domain is modelled by the formula $D = \{(i) \mid \exists k : i \geq 1 \wedge i \leq 21 \wedge i = 1 + 5k\}$ which contains an existentially quantified variable. The relation represents the points in the set $\{1, 6, 11, 16, 21\}$; these are not all the integer points inside $(i \geq 1 \wedge i \leq 21)$ which is the projection of the original two dimensional polyhedron $(i \geq 1 \wedge i \leq 21 \wedge i = 1 + 5k)$ upon i. The existentially quantified variable is also introduced when normalizing a statement such as `a[2i] = ...`. Indeed, normalization will replace it with `a[k] = ...` and compute a constraint $(\exists i : k = 2i \wedge \ldots \leq i \leq \ldots)$ in a normal form. Also modulo operations and integer division can give rise to such variables. These formulae with existentially quantified variables belong to the class of the Presburger formulae.

Now, a program can simply be represented by the geometric models of the statements. The dependencies between reads and writes are captured by the use-def chains which can be derived from the geometric models of the statements. Note that the exact execution order is not modelled since it is irrelevant to the verification purpose.

3 Verification method

Our verification task consists of proving that the original and the transformed programs compute the same outputs when their inputs are equal. To do so, one is given the names of the corresponding input and output arrays. Also, one can assume that the functions called by programs are side-effect free and have not been modified by the transformation, that the programs are in DSA form and that the data flow is correct, i.e., that values are read after being written (or are available as input). In what follows, we use some notational conventions. The arrays in the original and the transformed program are distinguished by the respective superscript $^{\mathsf{o}}$ and $^{\mathsf{t}}$. A vector $(i_1, \ldots, i_n)$ is denoted as $\boldsymbol{i}$; i_l refers its l^{th} element. In the verification task for the programs of Figure 1, the corresponding input arrays are $\mathtt{in}^{\mathsf{o}}$ and $\mathtt{in}^{\mathsf{t}}$; the corresponding output arrays are $\mathtt{b}^{\mathsf{o}}$ and $\mathtt{b}^{\mathsf{t}}$.

The verification task can be expressed by a (conjunction of) proof obligation(s). A **proof obligation** describes an equivalence relation that must hold between one expression from the original program and the other one from the transformed program. It is formalized as a tuple $(exp^{\mathsf{o}}(\boldsymbol{i}), exp^{\mathsf{t}}(\boldsymbol{j}), P(\boldsymbol{i}, \boldsymbol{j}))$, where $exp^{\mathsf{o}}(\boldsymbol{i})$ and $exp^{\mathsf{t}}(\boldsymbol{j})$ are expressions from the original and transformed program respectively; these expressions are parameterized by respective vectors of variables $\boldsymbol{i}$ and $\boldsymbol{j}$. $P(\boldsymbol{i}, \boldsymbol{j})$ is a set of constraints that specifies a relation between the vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ (the tuples in the relation are the integer points in the polytope). The meaning is: for each pair $(\boldsymbol{i}, \boldsymbol{j}) \in P(\boldsymbol{i}, \boldsymbol{j})$ (i.e., all integer solutions of P) the equality $exp^{\mathsf{o}}(\boldsymbol{i}) = exp^{\mathsf{t}}(\boldsymbol{j})$ has to be proven.

For example, the verification task of Figure 1 can be modelled by the proof obligation $(\mathtt{b}^{\mathsf{o}}[i_1, i_2], \mathtt{b}^{\mathsf{t}}[j_1, j_2], P((i_1, i_2), (j_1, j_2))\ )$, in which $P((i_1, i_2), (j_1, j_2)) = \{(i_1, i_2), (j_1, j_2) \mid i_1 = j_1 \wedge i_2 = j_2 \wedge i_1 \geq 1 \wedge i_1 \leq 5 \wedge i_2 \geq 1 \wedge i_2 \leq i_1 + 1 \wedge j_1 \geq 1 \wedge j_1 \leq 5 \wedge j_2 \geq 1 \wedge j_2 \leq j_1 + 1\}$. This proof obligation between output arrays $\mathtt{b}^{\mathsf{o}}$ and $\mathtt{b}^{\mathsf{t}}$ expresses that both programs are equivalent if one proves that $\mathtt{b}^{\mathsf{o}}[i_1, i_2] = \mathtt{b}^{\mathsf{t}}[j_1, j_2]$ for all pairs $(\boldsymbol{i}, \boldsymbol{j}) \in P$. While in the initial proof obligation, the relation between $\boldsymbol{i}$ and $\boldsymbol{j}$ is a bijection, this is in general not the case. Also, the given correspondence between input arrays can be expressed in this form (as assumptions). For our example it can be formulated as : $(\mathtt{in}^{\mathsf{o}}[i_1, i_2], \mathtt{in}^{\mathsf{t}}[j_1, j_2], \{(i_1, i_2), (j_1, j_2) | i_1 = j_1 \wedge i_2 = j_2 \wedge i_1 \geq 1 \wedge i_1 \leq 5 \wedge i_2 \geq 1 \wedge i_2 \leq 5)\}$.

The verification method then consists of using the geometric models of the program statements to reduce the initial proof obligations to proof obligations that trivially hold because they belong to the assumptions.

Reduction of proof obligations There are three basic reduction steps:

Reduction I Peeling expressions in a proof obligation of the form

$$(f(exp^{\mathsf{o}}_1(\boldsymbol{i}), \ldots, exp^{\mathsf{o}}_n(\boldsymbol{i})), f(exp^{\mathsf{t}}_1(\boldsymbol{j}), \ldots, exp^{\mathsf{t}}_n(\boldsymbol{j})), P(\boldsymbol{i}, \boldsymbol{j}))$$

The method leaves the functions uninterpreted and imposes (as sufficient condition) that the arguments of the functions must be pairwise equivalent[3]. Hence the proof obligation is replaced by the conjunction of n proof obligations of the form $(exp^{\mathsf{o}}_k(\boldsymbol{i}), exp^{\mathsf{t}}_k(\boldsymbol{j}), P(\boldsymbol{i}, \boldsymbol{j}))$. The proof fails if different top level function symbols are found. This indicates an error in the transformation.

[3] See [13] for a discussion how to extend the approach for handling commutative and associative functions.

Reduction II Propagation across an assignment. If one of the expressions is an array reference, the corresponding writers of the array can be used to reduce the proof obligation. Without loss of generality, let us assume a proof obligation of the form $(\mathtt{a}^{\mathrm{o}}[\boldsymbol{f(i)}], exp^{\mathrm{t}}(\boldsymbol{j}), P(\boldsymbol{i},\boldsymbol{j}))$, i.e., with an array reference from the original program. Let the normalized statement s: $\mathtt{a}^{\mathrm{o}}[\boldsymbol{k}] = exp^{\mathrm{o}}(\boldsymbol{k})$ be a writer of $\mathtt{a}^{\mathrm{o}}$ and $C(\boldsymbol{k})$ the constraints on $\boldsymbol{k}$. Using the set of equalities $\boldsymbol{k} = \boldsymbol{f(i)}$, the proof obligation can be rewritten as $(\,exp^{\mathrm{o}}(\boldsymbol{f(i)}), exp^{\mathrm{t}}(\boldsymbol{j}), P(\boldsymbol{i},\boldsymbol{j}) \wedge C(\boldsymbol{f(i)})\,)$ with $C(\boldsymbol{f(i)})$ the constraint with the elements of $\boldsymbol{k}$ substituted by the corresponding elements from $\boldsymbol{f(i)}$.
The new proof obligation can be discarded when the constraint is inconsistent, i.e., when none of the values referred to by $\mathtt{a}^{\mathrm{o}}[\boldsymbol{f(i)}]$ is actually written by s. Doing the propagation for each of the writers of $\mathtt{a}^{\mathrm{o}}$, the original proof obligation is replaced by one proof obligation for each of the writers that actually writes some of the values referred to by $\mathtt{a}^{\mathrm{o}}[\boldsymbol{i}]$.

Reduction III When both expressions are references to input arrays, i.e., of the form $(\mathtt{in}^{\mathrm{o}}[\boldsymbol{f(i)}], \mathtt{in}^{\mathrm{t}}[\boldsymbol{g(j)}], P(\boldsymbol{i},\boldsymbol{j}))$, the proof obligation can be dismissed as satisfied after checking that it can be proved from the given assumptions about the correspondence between input arrays. More precisely, given the input equivalence assumption $(\mathtt{in}^{\mathrm{o}}[\boldsymbol{k}], \mathtt{in}^{\mathrm{t}}[\boldsymbol{l}], C(\boldsymbol{k},\boldsymbol{l}))$, the integer points in $P(\boldsymbol{i},\boldsymbol{j})$ are a subset of those in $C(\boldsymbol{k},\boldsymbol{l})$ under the condition: $\boldsymbol{k} = \boldsymbol{f(i)} \wedge \boldsymbol{l} = \boldsymbol{g(j)}$, i.e., that $P(\boldsymbol{i},\boldsymbol{j}) \wedge \neg C(\boldsymbol{k},\boldsymbol{l})$ is unsatisfiable (or that $C(\boldsymbol{k},\boldsymbol{l})$ is entailed by $P(\boldsymbol{i},\boldsymbol{j})$).

The reduction of proof obligations requires the following primitive operations on relations (constraints): emptiness checking (consistency checking) and subset testing (entailment or negation).

However, a naive application of the above method results in a rather inefficient procedure because: ① Several statements can read (the same or different) values written by some statements, hence several proof obligations involving the same statement can be created. ② Recurrences (direct or indirect) are processed by complete loop unrolling, which is definitely not feasible in practice.

4 Control issues in the absence of recurrences

A sequence of different statements $s_0, s_1, \ldots, s_{n-1}$ represents a **recurrence** when each statement s_i is a writer of an array $\mathtt{a_i}[\boldsymbol{k_i}]$ and a reader of an array $\mathtt{a_{i+1}}[\boldsymbol{l_i}]$ with dependency mapping $M(\boldsymbol{k_i} \rightarrow \boldsymbol{l_i})$, $\mathtt{a_0}$ and $\mathtt{a_n}$ are the same array, and the equijoin of dependency mappings[4] i.e., $M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_n}) = M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_1}) \wedge M(\boldsymbol{k_1} \rightarrow \boldsymbol{k_2}) \wedge \ldots \wedge M(\boldsymbol{k_{n-1}} \rightarrow \boldsymbol{k_n})$ is a nonempty relation. $M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_n})$ is called a *self dependence mapping about array* $\mathtt{a_0}$. Note that a recurrence implies that each statement s_i is a writer of a value used to compute another value it writes; in particular, for statement s_0, we have that $\mathtt{a_0}[\boldsymbol{k_n}]$ is used to compute $\mathtt{a_0}[\boldsymbol{k_0}]$ for each tuple $(\boldsymbol{k_0}, \boldsymbol{k_n})$ in M.

[4] The dependency mapping can be viewed both as a relation and as a constraint, under the constraint view, the equijoin can be denoted as a conjunction of constraints.

In this section we discuss how to tackle the inefficiencies caused by multiple reads from elements written by the same statement in the absence of recurrences.

```
    for (i = 1; i <= 10; ++i)
s1:   b[i] = ...
    for (i = 1; i <= 10; ++i)
s2:      a[i] = ... b[i] ...;
    for (i = 1; i <= 10; ++i)
s3:      c[i] = ... 2*b[i] ...;

                          Case I
```

```
    for (i = 1; i <= 10; ++i)
s1:   b[i] = ...
    for (i = 1; i <= 10; ++i) {
      if (i > 5)
s2:      a[i] = b[i];
      else
s3:      a[i] = 2*b[i]; }

                          Case II
```

Fig. 2. Two programs illustrating the need for a good control.

Case I of Figure 2 shows a program where array `b` written in statement *s1* is read two times. This will give rise to two different proof obligations involving $\mathtt{b}[i]$ with the same boundaries on i. One should avoid proving it twice (there can be many steps before the input is reached).

Case II of the same figure shows a slightly different circumstance. Now, the two proof obligations refer to different pieces of the array `b`, so they will be different. However, substantial work could be saved if one could merge both proof obligations into a single one. Indeed, then only one proof obligation need be reduced to a condition between input arrays instead of two.

A simple way to tackle the inefficiency of Case I is by tabling all proof obligations. A new proof obligation can be dismissed if it is implied by an already tabled one (in the same way as a proof obligation between input arrays can be dismissed when implied by the assumptions, see Section 3). Permanently storing all proof obligations that occur during the verification may result in the huge table size.

In the absence of recurrences, it is possible to define a strategy that avoids such redundancies and keeps table size to a minimum by storing only the necessary proof obligations. Our strategy is to associate a counter with each statement s; this counter is initialized with the number of readers of the array written by s. Now consider a proof obligation of the form $(\mathtt{a}[\boldsymbol{f}(\boldsymbol{i})], exp^{\mathrm{t}}(\boldsymbol{j}), P(\boldsymbol{i}, \boldsymbol{j}))$ and let *s0* be one of the writers of `a`. The Reduction II step propagating the proof obligation across *s0* is delayed until the counter associated with statement *s0* is 0, i.e., until all proof obligations originating from readers of elements of `a` that are written by *s0* are available (and redundant ones have been removed). After propagation, all counters of writers of arrays $\mathtt{b}_{\mathtt{i}}$, for which *s0* is a reader, can be decreased by 1 as all proof obligations originating from the reader *s0* are now available. This works fine when *s0* is a copy statement as the new proof obligation is ready for propagation. However, when the rhs is an expression, then the proof obligation about $\mathtt{b}_{\mathtt{i}}$ is not ready for propagation until one or more peel steps have been applied. It blocks propagation across the writers of $\mathtt{b_i}$ until it has been further reduced by peel steps.

Our strategy implies that the propagation steps across statements are bundled in a different way. Instead of propagating one proof obligation ($\mathtt{a}[\boldsymbol{i}], \ldots$) across all writers of `a` at once, all proof obligations containing `a` are at once propagated across a single writer of `a` (when its counter is 0 and there are no pending peel operations). The correctness of the dataflow together with the absence of recurrences ensures that the verification will never be blocked as there will always be a zero counter.

A proof obligation with an array reference is tabled when created and remain *active* until it has been propagated to all writers of that array, at which point it can be removed. Hence the table consists of a set of active proof obligations which is only a fraction of the total number of proof obligations that are created during the verification.

5 Handling recurrences

Figure 1 contains two recurrences consisting of a single statement (*direct recurrences*), namely statements *o2* and *o4*. Also, there exists *indirect recurrences* in which several statements are involved. Figure 3 shows another example with a direct recurrence in statement *s3*. Note that the program slice that computes the value of $a[7]$ also contains the instances of *s3* for $i = 5$ and $i = 3$.

```
s1:  a[1] = 5;
s2:  a[2] = 6;
       ...
     for (i=3, i<=7, i++)
s3:    a[i] = a[i-2] + 5;
       ...
s4:  b[0] = a[7]
```

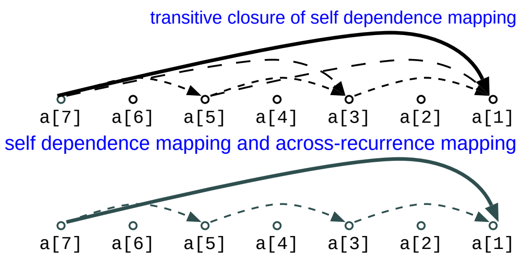

transitive closure of self dependence mapping
a[7] a[6] a[5] a[4] a[3] a[2] a[1]
self dependence mapping and across-recurrence mapping
a[7] a[6] a[5] a[4] a[3] a[2] a[1]

Fig. 3. A piece of code with a recurrence.

The control described in Section 4 will be blocked at recurrences. Consider the example; the counter of *s3* will never reach 0 because one of the readers of the values it writes is inside the recurrence. Simply not counting the reads inside the recurrence when initializing the counters will avoid the deadlock but will result in a naive algorithm that unrolls the recurrence, what results in an unacceptable performance degradation. In our example, the recurrence is entered when propagating a proof obligation about $\mathtt{a}[7]$ reaches *s3*. This is then reduced to a proof obligation about $\mathtt{a}[5]$ and next about $\mathtt{a}[3]$. Finally, the recurrence is exited by the proof obligation about $\mathtt{a}[1]$.

Note that the dependencies in a recurrence are always well-founded because the program is in DSA and the data flow is correct. Hence there are always statements that exit the recurrence when tracing the dependencies (statements *s2* and *s1* in our example). As we mentioned at the beginning of Section 4, a recurrence can be characterized by a self dependency mapping $M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_n})$ for some array a. If the distance between $\boldsymbol{k_0}$ and $\boldsymbol{k_n}$ is the same for all tuples $(\boldsymbol{k_0}, \boldsymbol{k_n})$ in M then we can use an approach that avoids completely unrolling the recurrence (for other recurrences, we simply use the naive approach mentioned above).

One way, sketched in [13] is to compute the positive transitive closure of the self dependence mapping (the arrows in the top right of Figure 3) and to use the difference between domain and range of the transitive closure relation to compute the across-recurrence mapping. Another approach is to combine the constant distance of the self-dependency mapping with domain information to extend the relation containing one tuple of the self dependency mapping into a relation covering all tuples of the mapping (the arrows in the bottom right of Figure 3) and to extend the proof obligation in the same way. Note that one must have a recurrence in both the original and the transformed program and that both have to be synchronous. This makes the whole technique rather involved and we omit further details.

6 Solvers

6.1 Requirements

As we have seen in Section 2, for simple statements, the relations of interest can be represented by the integer points in a polytope. However, for more complex statements, more complex constraints are required that involve existentially quantified variables. These existentially quantified variables cannot be eliminated by projection because, as we illustrated, the set of integer points in the projection of a polyhedron can be strictly larger than the set obtained by projecting the integer points in the original polyhedron. In other words, projection can introduce an overestimation; as our verification method requires exact modelling, it is not a safe operation.

As a summary, the basic operations required by the verification method are consistency checking (does the constraint has an integer solution) and entailment. Another useful operation is convex hull. It can be used to merge several active proof obligations into a single one, and hence to reduce the size of the table (Section 4) and the number of reduction steps (Section 3). In fact, there are two ways to simplify the set of active proof obligations between the same pair of expressions:

- When the constraint part of one proof obligation is entailed by the constraint part of another one, it can be discarded.
- When the convex hull of the constraints of two proof obligations entails their disjunction (in other words, the convex hull is equivalent to the disjunction),

they can be replaced by a single proof obligation that has the convex hull as constraint.

The example below (taken from case II in fig 2) shows an application of this simplification.

Example 1. In the program, statement *s2* gives rise to a proof obligation of the form $(\mathtt{a}[i], exp(\boldsymbol{k}), i \geq 1 \wedge i \leq 5 \wedge C(\boldsymbol{k}))$. A similar proof obligation $(\mathtt{a}[j], exp(\boldsymbol{l}), j \geq 6 \wedge j \leq 10 \wedge C(\boldsymbol{l}))$ is raised by statement *s3*. Imposing the equalities $i = j$ and $\boldsymbol{k} = \boldsymbol{l}$ one can derive that the convex hull is given by $(i \geq 1 \wedge i \leq 10 \wedge C(\boldsymbol{k}))$ and that it is equivalent to the disjunction $(i \geq 1 \wedge i \leq 5 \wedge C(\boldsymbol{k})) \vee (i \geq 6 \wedge i \leq 10 \wedge C(\boldsymbol{k}))$, hence both proof obligations can be replaced by $(a[i], exp(\boldsymbol{k}), i \geq 1 \wedge i \leq 10 \wedge C(\boldsymbol{k}))$.

6.2 Solvers

In simple verification tasks, all constraints correspond to polytopes (the constraints do not contain existential variables). However, solutions are the integer points in these polytopes, hence, more is required than basic capabilities for solving linear equalities and inequalities over rationals or reals.

CLP(Q) [3] is a library in SICSTUS Prolog [14] for solving linear programming problems with a limited support for mixed integer linear optimization problem. Besides the consistency check over the rational domain, it allows one to check that there is at least one integer solution (using the `bb_inf` operation). Using these primitive operations, one can build more complex operations needed by our verification. As described in [2], a convex hull operation can be constructed.

The PolyLib [8] library is a software package that is designed for manipulating polyhedra. PolyLib is designed to handle *polyhedral domains* which refer to the set of integer points in the union of a finite number of polyhedra. It provides functions for various operations including testing for the existence of an integer solution and calculating convex hull. So it provides all the functionality for handling simple verification tasks.

PPL [10] is another library, however it is oriented more towards the support of rational convex polyhedra, and includes the ability to handle the strict inequalities. But the lack of support for checking the existence of integer solutions makes it not suitable for our verification task.

When it comes to the verification of more complex problems involving existential variables, then all of the above systems are not suited. For certain kinds of constraints there may be work-arounds that eliminate the existential variables. In particular $\mathcal{Z}$-polyhedra [9] may be useful to represent certain types of constraints with existential variables. Also PIP [6], a tool which computes the lexicographic minimum of the integer points in a parametric polyhedron, can sometimes be helpful in eliminating certain existential variables [17]. However, not all existential variables can be eliminated. In such case we need a solver that supports general Presburger formulae that contains existentially quantified variables.

The Omega library [11] is designed specifically for handling full Presburger formulae. It is based on an extension of the Fourier-Motzkin method called Omega test. It also provides the other operations required by our application, such as entailment, convex hull, and even simplification that replaces a disjunction with its convex hull when they are equivalent. It also provides a transitive closure operation [7].

Presburger formulae have a super-exponential time complexity. Hence Omega necessarily employs various heuristics. Sometimes, these heuristics may fail. Then the calculation either continues running without returning a solution in a reasonable time or simply gives an UNKNOWN result, as reported in Section 5.2.1 of [16]. That is likely inherent to any solver for the general class of Presburger formulae. Moreover, as reported by [16], the implementation of Omega has other problems that may cause it to abort the calculation with an error message, or in very rare cases even produce incorrect results.

7 Conclusion

In this paper we analyzed a verification task for embedded software that was previously sketched in [13]. We described in more detail the basic operations of the verification process. We also analysed the functionality a solver has to offer to be useable in this application. For simple verification tasks, where existential variables do not appear, or can be eliminated by simple work-arounds, various solvers that can handle polyhedra can be applied. However, for more complex verification tasks, only Omega[11] offers all the needed functionality, though there are no guarantees that it will never fail.

Acknowledgement

We are grateful to Peter Vanbroekhoven and Sven Verdoolaege for the many discussions and useful comments.

References

1. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. Technical Report Report RR-4285, INRIA, Oct. 2001.
2. F. Benoy, A. King, and F. Mesnard. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming*, 5:259–271, 2005.
3. H. C. OFAI CLP(Q,R) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
4. F. Catthoor, S. Wuytack, E. de Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic Publishers, 1998.
5. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

6. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, Sep 1998.
7. W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming*, 24(6), 1996.
8. V. Loechner. PolyLib: A library for manipulating parameterized polyhedra. Technical Report PI-785, IRISA, 1999.
9. S. P. K. Nookala and T. Risset. A Library for Z-polyhedral Operations. Technical Report PI-1330, IRISA, Mai 2000.
10. PPL. http://www.cs.unipr.it/ppl/.
11. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, Aug 1992.
12. F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Prog. Lang. Syst.*, 22(5):773–815, Sep 2000.
13. K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Construction, 14th International Conference, CC 2005, Proceedings*, volume 3443 of *LNCS*, pages 221–236. Springer, 2005.
14. SICSTUS. http://www.sics.se/isl/sicstus.html.
15. P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems, Tsukuba, Japan*, 2005.
16. S. Verdoolaege. *Incremental loop transformations and enumeration of parametric sets*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 2005.
17. S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor. Experiences with enumeration of integer projections of parametric polytopes. In *Compiler Construction, 14th International Conference, CC 2005, Proceedings*, volume 3443 of *LNCS*. Springer, 2005.

Confinement Analysis with Graph Reachabilty Constraints

Fred Spiessens[1,2], Luis Quesada[2], and Peter Van Roy[2]

[1] Cork Constraint Computation Centre, Cork, Ireland
[2] Université catholique de Louvain, Louvain-la-Neuve, Belgium

Abstract. In security analysis, the problem of confinement is to find ways to prevent some entities in a system to get direct access to each other, or in the generalized form, to have a certain type of (direct or indirect) influence on each other. In this paper we propose the use of a constraint propagator for reachability in directed graphs, DomReachability [QVDC06], to help solve such confinement problems. We give examples of how the propagator can be applied, and we show that it can improve scalability, in comparison with a constraint based tool for generalized security analysis [SJV05].

1 Introduction

In software security, the execution of some actions is controlled (allowed or disallowed), in an attempt to restrict their (direct or indirect) effects. Allowed actions are called *permissions.* The parts of a program that can have permissions are called *subjects.* The ability of a subject to directly or indirectly induce an effect is the subject's *authority.*

Safety properties indicate authority that is illegal and should be prevented. A program breaks a safety property if the illegal effect cannot be prevented. When analyzing if a program can break a safety property, the following are important:

1. What permissions are initially available to the subjects.
2. How do the subjects use their permissions to generate effects.

It was shown in [HRU76] that these problems are not computable in general. Therefore, security analysis has to approximate the problem from the safe side, by looking for *proof* that the safety property remains unbroken. If no such proof can be found, the problem is *assumed* to be unsafe. We safely approximate a program using a monotonic approach that considers only the authority *enhancing* parts of the actions.

In the approximative models we will also consider requirements that express that certain authority should *not* be prevented, which we will call *liveness possibilities*, to distinguish them from the stronger *liveness properties* which express guarantees that cannot be made when using safe approximations.

In this paper, we show that graph reachability constraints have useful applications in safety analysis and enforcement. We do not claim that this approach is appropriate, useful, or feasible in *all* circumstances.

Section 2 explains how authority can be expressed using graphs. In section 3 we introduce the reachability constraint propagator. The rest of the paper describes the use of this constraint for safety analysis. Section 4 demonstrates how to calculate the strategic inter-positioning of controllable subjects in a network of interacting entities. Section 5 explains restricted behavior can be expressed using subgraphs and additional constraints.

Section 5 presents two extensions of the the reachability propagator, and discusses their additional expressive power. We extend the safety analysis to networks of interconnected systems in Section 7, and discuss the scalability of the graph based approach in comparison with the "Scollar" tool [SJV05]. We then present future work, that will combine the strength of both approaches.

2 Expressing Authority in a Directed Graph

The nodes of our digraphs represent subjects, and, instead of representing permissions, the arcs represent the *direct authority* attainable by using permissions. That way, the reachable authority corresponds to the graph's transitive closure.

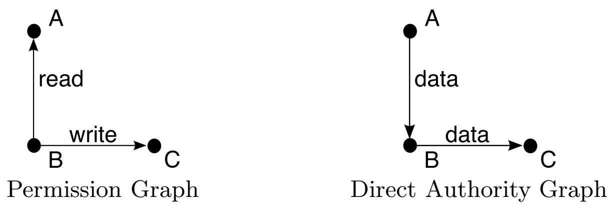


Fig. 1. From Permission Graphs to Direct Authority Graphs

For example, figure 1 shows how we can derive an authority digraph from a permission digraph that contains *read* and *write* permissions. Read and write permissions provide the same direct authority: *data-transmission*, be it in opposite directions. Therefore we can present both kinds of permissions in a single authority digraph.

3 The DomReachability Constraint

Before presenting the constraint, we introduce a set of concepts on which its definition relies. These concepts are explained in more detail in [QVDC06].

3.1 Extended dominator graph

Given a flow graph fg and its corresponding source s, a node i is a dominator of another node j if all paths from s to j in fg contain i (See [LT79,SGL97]).

Formally :

$$i \in \mathit{Dominators}(fg, j) \Leftrightarrow i \neq j \wedge \forall p \in \mathit{Paths}(fg, s, j) : i \in \mathit{Nodes}(p) \tag{1}$$

where

$$p \in \mathit{Paths}(fg, i, j) \Leftrightarrow \begin{cases} p \text{ is a subgraph of } fg \\ \mathit{Nodes}(p) = \{k_1, \ldots, k_n\} \wedge k_1 = i \wedge k_n = j \\ \mathit{Edges}(p) = \{\langle k_t, k_{t+1} \rangle \mid 1 \leq t < n\} \end{cases} \tag{2}$$

Note that the nodes unreachable from s are dominated by all the other nodes. However, the nodes reachable from s always have an *immediate* dominator, which is defined as :

$$i = \mathit{ImDominator}(fg, j) \Leftrightarrow \\ \begin{cases} i \in \mathit{Dominators}(fg, j) \\ \neg \exists k \in \mathit{Nodes}(fg) : i \in \mathit{Dominators}(fg, k) \wedge k \in \mathit{Dominators}(fg, j) \end{cases} \tag{3}$$

This property allows to represent the whole dominance relation as a tree, where the parent of a node is its immediate dominator.

Let us now consider the extended graph of fg, $\mathit{Ext}(fg)$, which is obtained by replacing the edges by new nodes, and connecting the new nodes accordingly. This graph can be formally defined as follows:

$$\langle N', E', s' \rangle = \mathit{Ext}(\langle N, E, s \rangle) \Leftrightarrow \begin{cases} s' = s \\ N' = N \cup E \\ e = \langle i, j \rangle \in E \Leftrightarrow \langle i, e \rangle \in E' \wedge \langle e, j \rangle \in E' \end{cases} \tag{4}$$

We call the dominator tree a flow graph's extended graph its *extended dominator tree*. Figures 2, 3 and 4 show an example of a flow graph, its extended graph, and its extended dominator tree, respectively. The extended dominator tree has two types of nodes: nodes corresponding to nodes in the original graph (*node dominators*), and nodes corresponding to edges in the original graph (*edge dominators*). The latter nodes are drawn in squares.

3.2 The *DomReachability* constraint

The *DomReachability* constraint is a constraint on three graphs:

$$\mathit{DomReachability}(fg, edg, tcg) \tag{5}$$

where

- fg is a flow graph, i.e., a directed graph with a source node, whose set of nodes is a subset of N;
- *edg* is the extended dominator graph of fg; and
- *tcg* is the transitive closure of fg, i.e,

$$\begin{gathered} \langle i, j \rangle \in \mathit{Edges}(tcg) \Leftrightarrow \langle i, j \rangle \in \mathit{Edges}(TC(fg)) \\ \langle i, j \rangle \in \mathit{Edges}(TC(g)) \Leftrightarrow \exists p : p \in \mathit{Paths}(g, i, j) \end{gathered} \tag{6}$$

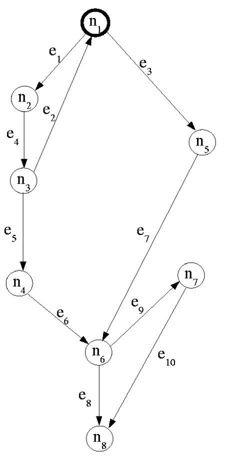


Fig. 2. Flow graph

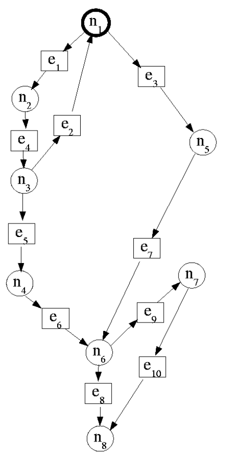


Fig. 3. Extended flow graph

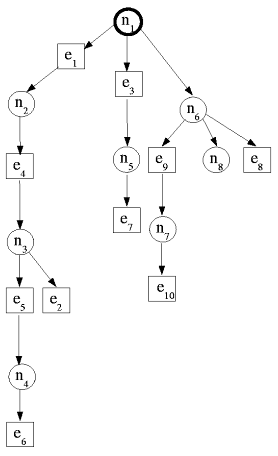


Fig. 4. Extended dominator tree

3.3 Expressing security constraints with DomReachability

Our security problems have two concerns:

1. some authority should not be reachable (safety properties)
2. other authority should be reachable (liveness possibilities)

Both concerns can be expressed in the *The Bounded Transitive Closure Problem (BTC)*: given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, find a directed graph g such that [3]:

$$\begin{gathered} g_{min} \subseteq g \subseteq g_{max} \\ \text{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{gathered} \tag{7}$$

The set of liveness possibilities will be tcg_{min}, tcg_{max} will be the complement of the set of safety properties, and g_{min} and g_{max} will just be suitable bounds for the safe configuration of permissions we are looking for.

The *BTC* instance can be directly modeled in terms of *DomReachability*: g_{min}, g_{max}, tcg_{min} and tcg_{max} are the bounds of fg and tcg.

Remark : *BTC* problems are NP-complete. That was recently shown in [Que06], by reducing the *The Disjoint Paths Problem* [GJ79] to *BTC*.

[3] There is no relation among g_{min}, g_{max}, tcg_{min} and tcg_{max}. In particular, it is true neither that tcg_{min} is the transitive closure of g_{min} nor that tcg_{max} is the transitive closure of g_{max}

4 Confinement by Interposition

Suppose we have a set of previously unconnected, uncontrollable components, and we want to find out how we can connect them, using controllable components, to allow them to perform their collaborative tasks, but also prevent them from breaking a given security policy. The tools we have to solve this problem are:

– a set of controllable components (subjects) to be strategically inter-positioned between the uncontrolled components.
– a set of permissions to be granted to the controllable components.

The assignment is: find a configuration (graph) with a *minimal number of controllable nodes* (not exceeding a fixed practical upper limit), that guarantees the requirements for possible liveness (the uncontrolled components get enough authority) as well as the requirements for safety (the uncontrolled components do not get too much authority).

4.1 Practical Example

We take a well known example, expressing a simple *Multi-Level Security Problem (MLS)* [BL74]. Two external subjects *Bond* and *Q*, with respective clearances *Top Secret* and *Confidential*, have to be given access to two external storage devices, one for *Top Secret* content, and one for *Confidential* content.

We have to construct the content of a black box in (e.g. Figure 5), with a minimal number of components. Since the uncontrollable components cannot be restricted, their connection to the box is bi-directional. Even the devices are not trusted to be passive containers, they are unknown components and could be of any type.

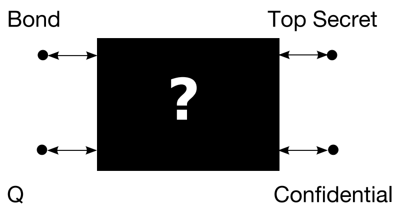


Fig. 5. The $*$-property black box

The security policy we want to enforce between these four entities is simply to make sure that no Top Secret information leaks (down) to the Confidential level. Therefore we will enforce the $*$-*property* (star-property) that states: agents should be able write to all levels above (and including) their own level of confidentiality, and read from all levels below (and including) their own level of confidentiality, but no agent should be able to write strictly below his confidentiality level, or read strictly above his confidentiality level. This is a policy that specifies both liveness requirements and safety requirements, so we will express it as suggested in section 3.

Expressing the problem in terms of DomReachability The *BTC* for the instance of the problem presented above is:

$$\begin{aligned} g_{min} &= \emptyset \\ g_{max} &= \{\langle x, y\rangle | x, y \in \{b, q, t, c\} \cup \{o_1, o_2, ..., o_{max}\}\} \\ tcg_{min} &= \{\langle b, t\rangle, \langle t, b\rangle, \langle q, c\rangle, \langle c, q\rangle, \langle c, b\rangle, \langle q, t\rangle\} \\ tcg_{max} &= g_{max} - \{\langle b, q\rangle\langle b, c\rangle\langle t, q\rangle\langle t, c\rangle\} \end{aligned} \tag{8}$$

In the problem, b stands for Bond, q for Q, t for the top-secret device, and c for the confidential device. The controllable nodes are o_1, o_2,...,o_{max}.

Apart from the *BTC* constraints, we have to express the fact that b, q, t, and c are uncontrolled, by making sure that all their connections are bi-directional. We therefore added the necessary implication constraints to the problem:

$$\forall 0 \leq x \leq max, i \in \{b, q, t, c\} : \langle i, o_x\rangle \in g \Leftrightarrow \langle o_x, i\rangle \in g \tag{9}$$

To minimize the number of controlled components, we can start with zero controlled nodes, and iteratively add one more, until we find a solution.

We also experimented with a labeling strategy that tends to find the solution with the least nodes first. By first trying to remove all possible edges from the controlled node that reaches the most nodes, the strategy tends to minimize both the number of edges and the number of controlled nodes, although there are cases where the number of controlled nodes used is not the smallest one possible.

The pruning performed by *DomReachability*, and the aforementioned labeling strategy provided the solution in Figure 6, in 40ms.

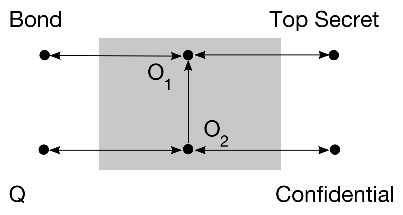


Fig. 6. A solution with the minimal number of controllable components

5 Confinement by Restricted Behavior

In the previous section we relied on the ability of the system to enforce the permissions. There could for instance have been a reference monitor that checked the permissions before they were exerted. Alternatively, the internal subjects could be trusted parts of the system: trusted to behave exactly as allowed by their permissions. Capability systems [DH65] rely on such subjects (called capabilities).

We could as well rely on our home-brewn trusted subjects to behave in "smarter" ways than simply using or not using certain permissions. We can program them to use their permissions in a way that would allow the desired *effects* and prevent the other ones. This allows for much more accurate analysis of the reachable authority in a system. An account of the different ways in which the boundaries of authority can be calculated is given in chapter 8 of [Mil06].

Suppose we want to express the behavior of a subject that only passes information if:

– other subjects wrote that information to it (it did not read the information itself from other subjects), and
– it writes that information itself to other subjects (it does not reveal that information to its own readers)

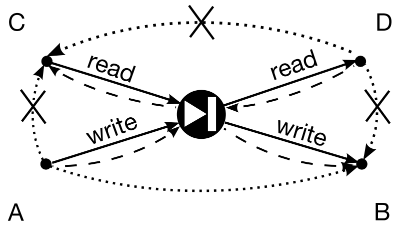


Fig. 7. Data Forwarder (dataflow diode)

Such a subject acts as a forward diode for data flow, depicted in figure 7. The full arcs denote the access rights and the dashed arcs represent the corresponding flow of data. The data-flow is only transitive in one direction: from A to B, as indicated by the dotted arcs. The behavior of the diode in the middle prevents data to pass in the three other directions.

We can express similar restricted behavior in a subgraph with four nodes: two $in - ports$ and two $out - ports$, one of each kind for reading, and the other one for writing. All external arcs will be connected to one of the four ports: the incoming flow to the in-ports, the outgoing flow to the out-ports, the flow via read permissions to the read-ports, and the flow via write permissions to the write-ports. These restrictions can directly be expressed in BTC, by removing the illegal external connections from g_{max}.

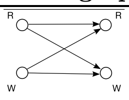
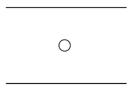
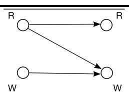
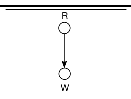
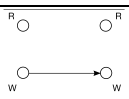
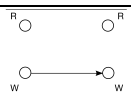
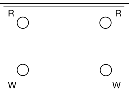
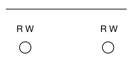
behavior graph	**simplified graph**	**behavior**
		unrestricted behavior
		hides its writers data from its readers
		data forwarder of figure 7
		non-tranparent subject

Fig. 8. Subgraphs for behavior-based internal dataflow

Figure 8 shows some behavior subgraphs with four *internal* ports. The internal arcs connect an in-port (left) to an out-port (right), as they relate incoming data to outgoing data. They correspond to the *dotted* arcs in figure 7.

These subgraphs replace the monolithic subject nodes in the *data-flow graph.* Depending on which of the four possible arcs are present, the behavior-graph can be simplified by collapsing two ports into one when the effect of connecting to/from either of them is the same (second column of figure 8).

6 Extending BTC for additional expressive power

In the previous sections we had to use additional constraints to express the security problems. We now propose two extensions to the BTC problem that incorporate the implication constraints that allow us to express interesting security problems.

6.1 The Conditional *BTC* Problem (CondBTC)

In section 4.1 we had to use extra constraints for all four uncontrolled components, to express that they should take only bidirectional connections. This means, if an edge $\langle A, B\rangle$ is in the graph, then so should $\langle B, A\rangle$. We can express this condition as an edge from $\langle A, B\rangle$ to $\langle B, A\rangle$ in a graph whose nodes are edges in other graphs and whose edges represent implications. This allows us to express inter-graph conditions on edges. If we consider also the complements of the graphs, (the complement of graph g is denoted as $(g)\prime$), we can express negations as well as implications.

Therefore we add a directed graph *condg* such that:

$$\begin{gathered} G_1, G_2 \in \{g, (g)', TC(g), (TC(g))'\} \\ \langle e_1^{G_1}, e_2^{G_2}\rangle \in condg \Leftrightarrow (e_1 \in G_1 \Rightarrow e_2 \in G_2) \end{gathered} \tag{10}$$

The security problems in sections 4 and 5 are direct applications of *CondBTC*. The implications involving edges of the solution graph and its transitive closure can be directly represented in terms of *condg*.

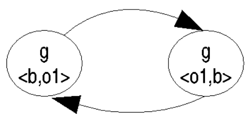


Fig. 9. A fraction of the *condg* for the problem in section 4.1

Figure 9 shows a bi-directional connection constraint as 2 edges in *condg*.

6.2 The Cardinal *BTC* Problem (CardBTC)

Instead of representing edges in another graph, let the nodes in *condg* now represent *mixed sets of edges from any of the DomReachability graphs.* An edge $\langle A, B\rangle$ in *condg* now represents a composite condition: if *all* edges in the set A are present, then so should *at least one* edge in the set B.

The extended definition of *condg* allows us to simplify the definition of the problem. The *BTC* graphs g_{min}, g_{max}, tcg_{min} and tcg_{max} can be defined in with *condg*, *transitive closure*, and *graph complement* as follows:

$$\begin{aligned} \forall e \in g_{min} &: \langle \emptyset, \{e^{g}\}\rangle \in condg \\ \forall e \notin g_{max} &: \langle \emptyset, \{e^{(g)'}\}\rangle \in condg \\ \forall e \in tcg_{min} &: \langle \emptyset, \{e^{TC(g)}\}\rangle \in condg \\ \forall e \notin tcg_{max} &: \langle \emptyset, \{e^{(TC(g))'}\}\rangle \in condg \end{aligned} \tag{11}$$

Notice that the number of nodes of *condg* is $O(|N|^2 + |conds|)$, where N is the set of nodes on which g_{min}, g_{max}, tcg_{min} and tcg_{max} are defined, and *conds* is the set of conditional constraints. This is because each graph has at most $|N|^2$ edges, and there are two nodes per conditional constraint. The number of edges is $O(|N|^2 + |conds|)$ too since there is one edge in *condg* per edge in g_{min}, g_{max}, tcg_{min} and tcg_{max}, and one edge per conditional constraint.

The expressivity of *CardBTC* can be further extended by labeling edges with constraints on the cardinality of the target set. For instance, figure 10 graphically shows the following constraint in extended Higraph notation [Har95] :

$$\begin{aligned} & b_1, b_2, b_3 \in \{0, 1\} \\ & b_1 = 1 \Leftrightarrow \langle b, c\rangle \in TC(g) \\ & b_2 = 1 \Leftrightarrow \langle b, f\rangle \in TC(g) \\ & b_3 = 1 \Leftrightarrow \langle b, e\rangle \in TC(g) \\ & \langle a, b\rangle \in TC(g) \Rightarrow b_1 + b_2 + b_3 > 1 \end{aligned} \tag{12}$$

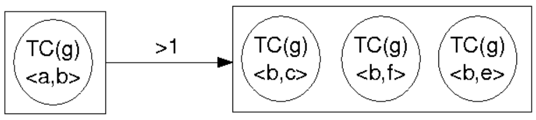


Fig. 10. A graphical presentation of a CardBTC constraint

We can say that, when the label of the edge is omitted, the implicit constraint is "> 0", i.e., at least one of the constraints in the set must be true.

The cardinality constraints involved in *CardBTC* can be managed by using standard approaches based on cardinality propagators [VD91]. However, we can reason at a higher level of abstraction by looking at the Boolean Satisfiability instance that results from associating each basic graph constraint with literals. This level of abstraction would let us take advantage of BDD propagators to narrow down the literals composing a given disjunction [HLS05]. We could also consider hybrid approaches, like the one suggested in [HS06], in order to inherit the advantages offered by SAT solvers.

6.3 Applying CardBTC for practical Security Problems

CardBTC allows us to express complex conditions on the propagation of authority in several ways we did not yet explore:

– It can be used to express more complex ways of authority propagation than transitive closure.
– It can be used to represent fine-grained conditional behavior of trusted subjects, without the need to represent every subject as a complex subgraph.

7 Secure Interoperation

In this section we present a security problem for which the scalability of the approach using DomReachability considerably exceeds that of *Scollar* [SJV05], a more general constraint-based tool for security analysis.

A system of interacting subjects can be secure, but when two or more secure systems become interconnected, the result may again introduce safety breaches. A secure reconfiguration removes a set of permissions, to make sure that no authority that was unreachable in any single system (and may have been forbidden by that systems policy), becomes reachable by the interconnection.

The use of constraint programming to find secure reconfigurations of interoperating systems, is proposed in [BFO05], for systems that are interconnected via shared subjects. The safety properties to be imposed on the interconnected systems, are the ones that hold in the individual systems. Optionally, additional safety and liveness requirements can then be added in the interconnected system.

Following this approach, we show how to use DomReachability, to find a minimal secure reconfiguration for a set of interconnected systems. A secure reconfiguration is a set of permissions such that, when each of these permissions are revoked in all systems that granted the permission before the interoperation, the interconnection will make no additional effects reachable between two subjects of the same system. This approach can then easily be extended to include additional constraints on the reachability of effects in the interconnected system.

7.1 Calculating Secure Reconfigurations with Scollar

The constraint based tool "Scollar", written in Mozart-Oz [VH04,Sch02], analyzes safety in configurations of permission-restricted and behavior-restricted subjects, and calculates the minimal (additional) restrictions that are necessary to guarantee the safety requirements.

Scollar was recently extended to compute safe reconfigurations for interoperating systems as well. To analyze secure interoperation, the tool first computes the transitive closures of every individual system, derives from these transitive closures the safety requirements for the global, interconnected system, and then calculates a minimal reconfiguration, that removes some initial permissions (and/or behavior of the trusted subjects, when specified)

Since Scollar's primarily aim is to analyze small patterns of interacting subjects with relatively complex behavior, its scalability in terms of number of subjects was not an initial concern. For secure interoperation, the problems tend to be larger in number of subjects, and marginally lower in complexity of the subject behavior. Recent experiments with the current implementation revealed that the practical limit allows no more than approximately 100 subjects in the interconnected system, even when the behavior complexity is reduced to simple on-off (active/passive), no propagation of permissions is modeled, and the mechanisms for effect propagation are reduced to simple transitive closure.

7.2 Comparing Scalability

We conducted a very preliminary set of experiments to get a rough idea of the relative scalability of DomReachability in comparison to Scollar.

We can expect that the DomReachability approach will perform best when the rules that model the propagation of authority are similar to reachability by transitive closure. At the same time we wanted to test the practical feasibility of modeling simple restrictive behavior as subgraphs.

We decided to run a very preliminary and small set of experiments with the following setup:

– N systems are inter-connected in a network that has a small-world topology, generated following the Watts-Strogaz approach [WS98] from a structured undirected graph in which every system (node) has 4 neighbors. A small world graph is a graph with a high clustering coefficient (of every system, most neighboring systems are connected) and a low characteristic path length (mean distance in the network between any two systems).

- Systems S_1 and S_2 are connected $\Leftrightarrow$ they share exactly two subjects.
- Unconnected systems have no common subjects
- Subjects are shared by at most two systems.
- Every system has exactly as many subjects as are required for its connections to its neighbors in the network.
- Half of the subjects in every system have unrestricted behavior, the other half are non-transparent (See Figure 8).

The same instances of the generated problems were fed to the Scollar based solver and to the DomReachability based solver. All safety properties were precalculated in Scollar, during the generation of the examples, and were not recalculated in the experiments.

The rules that govern data-flow in all systems were kept simple, and are illustrated by the following Horn Clauses, used in Scollar:

$readPermission(Y, X) \Rightarrow flow(X, Y)$
$flow(X, Y) \wedge flow(Y, Z) \wedge transparent(Y) \Rightarrow flow(X, Z)$

We made the experiments as simple as was reasonably possible, by considering a single permission (read), and two kinds of subject behavior: either transferring the data that was read or not. We did not introduce extra requirements for safety or liveness. We arranged for 50% of all subjects to be unrestricted (allowing data to flow through them in all directions), and the other 50% to be non-transparent (See Figure 8). The transparency of a subject was considered to be a fixed and was not optimized. Only the read permissions were optimized in the experiments.

All these restrictions where set up to restrict the influence of random choices on our measurements, and improve the accuracy with which our results reflect the influence of the size of the problem (number of systems).

All experiments where conducted on a dedicated Linux machine with 2GB of memory and 4 processors at 3.06 GHz. Figure 11 shows the time it took to find a first secure reconfiguration (in seconds), for networks with 8 to 52 systems (32 to 208 subjects). We performed only one calculation for every size of the problem. No results could be calculated in Scollar for problems of more than 24 systems (96 subjects), due to virtual memory exhaustion.

Even if only one problem instance was solved for every size, the results leave no doubt about the winner in this scalability contest. DomReachability is much more suited to solve problems of big size. We expect that Scollar is still more suitable for solving complex problems of small size, to model complex rules and subject behavior that express a refined approximation to how authority propagates. It will be interesting future work to find out exactly where the borderline for choosing between the two approaches lies, and even more interesting to see how the approaches can be combined to get the best of both.

8 Conclusion and Future Work

We have shown how the monotonic propagation of effects can be modelled with reachability constraints in a directed graph by associating nodes in the graph

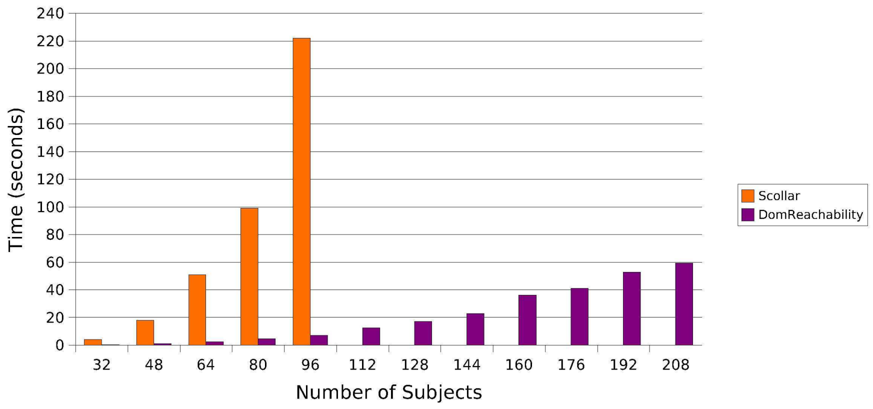


Fig. 11. *Scollar* and *DomReachability* calculating Secure Reconfigurations

with subjects, and edges with permissions between the corresponding subjects. We elaborated on the relation of the resulting constrained graph problem with the Bounded Transitive Closure problem (BTC) and suggested extensions of BTC that let us express more security requirements.

Some of the problems that we have presented can be solved in polynomial time. For instance, if there is no constraint on neither the lower bound of the interconnected graph of the interoperability problem nor on its transitive closure, the *BTC* instance can be solved in polynomial time. Indeed, the empty graph would be a valid solution to the problem. Even finding a maximal graph, i.e., a graph which is not included in another one respecting the safety properties, is still polynomial since it is always possible to find a graph not containing a particular edge that respects the safety properties.

The adoption of *DomReachability*, which is normally used in combinatorial problems, is justified because: (a) it offers an incremental approach for computing transitive closure, and (b) it discards invalid edges early on, since the addition of an edge may imply that some other edges are not part of the graph.

In section 4.1 we constrained the size of the graph by using a size constraint that takes the edges in the lower bound into account, but not the structure of the graph. A smarter global constraint would take into account the current boundaries of the graph and its transitive closure, to anticipate violations of the limit. For instance, suppose that i reaches j and the shortest path p from i to j contains x edges. Suppose also that the size of the graph is less than max. Then if the graph contains at least the edges in g_{min}, $max - |g_{min}| < |p - g_{min}|$ implies that there is no solution since reaching j from i would imply that the number of edges in the graph is greater than the limit(max). To detect this kind of information, we can use an approach like the one suggested in [Sel02] for incrementally keeping the shortest paths between each pair of nodes.

Towards a synergy of both approaches We expect DomReachability to be most useful in collaboration with our existing Scollar tool. Scollar is most suitable to express a system's rules that govern the propagation of permissions and authority, and a subject's behavior. System rules can express realistic models for propagation, that can take the restrictions of the behavior of the trusted subjects into account. Subject behavior can be expressed in a way that depends on the information that a subject has from initial conditions, and has required during the collaboration with other subjects. Its expressive power makes Scollar a tool that can (also) be used to study the propagation of authority in capability systems and patterns of collaborating entities.

The restriction to monotonic approximations (that are safe but may possibly be too crude) prevents us to directly express the revocation of authority. This is relevant for capability systems too because, even if access permissions cannot be revoked, it is very well possible (and easy) for a subject to revoke the authority it used to provide to its clients, for instance by refusing to collaborate any further, and no longer pass on any data or capabilities to them.

This is where the dominator part of DomReachability can be of direct use: to add expressive power to the safety requirements. Instead of simply stating that some effect (authority) should be prevented, we could instead require that all authority of a certain kind should only ever be available via a trusted subject that is able to revoke the authority. In the "authority-flow" graph (to be derived from the access-graph) a trusted subject Alice can revoke all Bob's authority over a third subject Carol, if Alice dominates Bob in the authority-flow graph that originates with Carol.

Acknowledgments

Part of the work presented here was funded by Science Foundation Ireland. The experiments were performed on planet-lab computers of the Université catholique de Louvain, funded by the EVERGROW project of the European Union "Complex Systems" Proactive initiative - 6th Framework Programme.

Part of the results in section 7 were extracted from joint work with Barry O'Sullivan, Simon Foley, and Stefano Bistarelli. The authors are grateful to Grégoire Dooms for many insightful discussions on the *BTC* problem.

References

[BFO05] Stefano Bistarelli, Simon N. Foley, and Barry O'Sullivan. Reasoning about secure interoperation using soft constraints. In *Workshop on Formal Aspects in Security and Trust (FAST)*, volume 173 of *FIP International Federation for Information Processing*. Kluwer, August 2005.

[BL74] D.E. Bell and L. LaPadula. Secure computer systems. In *ESD-TR*, pages 83–278. Mitre Corporation, 1974.

[DH65] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the The Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[Har95] David Harel. On visual formalisms. In Janice Glasgow, N. Hari Narayanan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning*, pages 235–271. The MIT Press, Cambridge, Massachusetts, 1995.

[HLS05] P.J. Hawkins, V. Lagoon, and P.J. Stuckey. Solving set constraint satisfaction problems using robdds. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[HS06] Peter Hawkins and Peter Stuckey. A hybrid bdd and sat finite domain constraint solver. In *PADL 2006 Proceedings*, volume 3819 of *Lecture Notes in Computer Science.* Springer, 2006.

[LT79] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[Que06] Luis Quesada. The bounded transitive closure problem, 2006. Available at *http://www.info.ucl.ac.be/˜luque/papers/btc.pdf.*

[QVDC06] Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In *PADL 2006 Proceedings*, volume 3819 of *Lecture Notes in Computer Science.* Springer, 2006.

[Sch02] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 2002.

[Sel02] Meinolf Sellmann. *Reduction Techniques in Constraint Programming and Combinatorial Optimization.* Doctoral dissertation, University of Paderborn, Paderborn, Germany, 2002.

[SGL97] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19(2):239–252, March 1997.

[SJV05] Fred Spiessens, Yves Jaradin, and Peter Van Roy. Using constraints to analyze and generate safe capability patterns. Research Report INFO-2005-11, Département d'Ingénierie Informatique, Université catholique de Louvain, Louvain-la-Neuve Belgium, 2005. Presented at CPSec'05. Available at `http://www.info.ucl.ac.be/∼fsp/rr2005-11.pdf`.

[VD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, March 2004.

[WS98] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

A Symbolic Intruder Model for Hash-Collision Attacks [⋆]

Yannick Chevalier and Mounira Kourjieh

IRIT Université Paul Sabatier, France
email: {ychevali,kourjieh}@irit.fr

Abstract. In the recent years, several practical methods have been published to compute collisions on some much used hash functions. Starting from two messages m_1 and m_2 these methods permit to compute m'_1 and m'_2 *similar* to the former such that they have the same image for a given hash function. In this paper we present a method to take into account, at the symbolic level, that an intruder actively attacking a protocol execution may use these collision algorithms in reasonable time during the attack. This decision procedure relies on the reduction of constraint solving for an intruder exploiting the collision properties of hash functions to constraint solving for an intruder operating on words, that is with an associative symbol of concatenation. The decidability of the latter is interesting in its own right as it is the first decidability result that we are aware of for an intruder system for which unification is infinitary, and permits to consider in other contexts an associative concatenation of messages instead of their pairing.

1 Introduction

Hash functions. Cryptographic hash functions play a fundamental role in modern cryptography. While related to conventional hash functions commonly used in non-cryptographic computer applications - in both cases, larger domains are mapped to smaller ranges - they have some additional properties. Our focus is restricted to cryptographic hash functions (hereafter, simply hash functions), and in particular to their use as cryptographic primitive for data integrity, authentication, key agreement, e-cash and many other cryptographic schemes and protocols. Hash functions take a message as input and produce an output referred to either as a *hash-code*, *hash-result*, or *hash-value*, or simply *hash*.

Collisions. A hash function is many-to-one, implying that the existence of collisions (pairs of inputs with the identical output) is unavoidable. However, only a few years ago, it was intractable to compute collisions on hash functions, so they were considered to be collision-free by cryptographers, and protocols were built upon this assumption. From the nineties on, several authors have proved the tractability of finding pseudo-collision and collision attacks over several hash

[⋆] supported by ARA-SSIA Cops and ACI JC 9005 functions. Taking this into account, we consider that cryptographic hash functions have the following properties:

– the input can be of any length, the output has a fixed length, $\mathrm{h}(x)$ is relatively easy to compute for any given x;
– pre-image resistance: for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that outputs, i.e., to find any x such that $y = \mathrm{h}(x)$ when given y;
– 2nd-pre-image resistance: it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x , to find x' different from x such that $\mathrm{h}(x) = \mathrm{h}(x')$;
– hash collision: it is computationally feasible to compute two distinct inputs x and x' which hash to the same output, i.e, $\mathrm{h}(x) = \mathrm{h}(x')$ provided that x and x' are created at the same time and independently one of the other.

In other words, a collision-vulnerable hash function h is one for which an intruder can find two different messages x and x' with the same hash value. To mount a collision attack, an adversary would typically begin by constructing two messages with the same hash where one message appears legitimate or innocuous while the other serves the intruder's purposes. For example, consider the following simple protocol:

$$A \rightarrow B : M, \sigma_A(M)$$

where $\sigma_A(M)$ denotes A's digital signature on message M using DAS digital signature scheme in which only the hash-value of M by a function h is considered. The following attack:

$$A \rightarrow B : M', \sigma_A(M)$$

can be launched successfully if the intruder first computes two different messages M and M' having the same hash value and then can lead Alice into executing the protocol with message M.

Collisions in practise. MD5 Hash function is one of the most widely used cryptographic hash functions nowadays. It was designed in 1992 as an improvement on MD4, and its security was widely studied since then by several authors. The first result was a pseudo-collision for MD5 [6]. When permitting to change the initialisation vector, another attack (free-start collision) has been found [8]. Recently, a real collision involving two 1024-bits messages was found with the standard value [19]. This first weakness was extended into a differential-like attack [22] and tools were developed [10, 9] for finding the collisions which work for any initialisation value and which are quicker than methods presented in [19]. Finally, other methods have been developed for finding new MD5 collisions [23, 17]. The development of collision-finding algorithms is not restricted to MD5 hash function. Several methods for MD4 research attack have been developed [20, 7]. In [20] a method to search RIPE-MD collision attacks was also developed, and in [2], a collision on SHA-0 has been presented. Finally, Wang *et al.* have developed in [21] another method to search for collisions for the SHA-1 hash function.

Goal of this paper. This development of methods at the cryptographic level to built collisions in a reasonable time have until now not been taken into account in a symbolic model of cryptographic protocols. We also note that the inherent complexity of these attacks make them not representable in any computational model that we are aware of. In this paper we propose a decision procedure to decide insecurity of cryptographic protocols when a hash function for which collisions may be found is employed. Relying on the result [3] we do not consider here other cryptographic primitives such as public key encryption, signature or symmetric key encryption, and assume that a protocol execution has already been split into the views of the different equational theories. The decidability proof presented here heavily relies on a recent result [4] that permits to reduce constraint solving problems with respect to a given intruder to constraint solving problems for a simpler one. This result relies on a new notion of *mode*. This notion aims at exhibiting a modular structure in an equational theory but has no simple intuitive meaning. In the case of an exponential operator as treated in [4] the separation was between an exponential symbol and the abelian group operations on its exponents, whereas here the separation is introduced between the application of the hash function and the functions employed by the intruder to find collisions.

Outline. We first give in Section 2 the definitions relating to terms and equational theories. We then present in Section 3 our model of an attacker against a protocol, and how we reduce the search for flaws to reachability problems with respect to an intruder theory. In Section 4 we describe in detail how we model the fact that an intruder may construct colliding messages, and how this intruder theory can be decomposed into simpler intruder theories. We give proof sketch of these reductions in Section 5 and conclude in Section 6.

2 Formal setting

2.1 Basic notions

We consider an infinite set of free constants C and an infinite set of variables $\mathcal{X}$. For any signature $\mathcal{G}$ (*i.e.* sets of function symbols not in C with arities) we denote $\mathrm{T}(\mathcal{G})$ (resp. $\mathrm{T}(\mathcal{G},\mathcal{X})$) the set of terms over $\mathcal{G} \cup \mathrm{C}$ (resp. $\mathcal{G} \cup \mathrm{C} \cup \mathcal{X}$). The former is called the set of ground terms over $\mathcal{G}$, while the latter is simply called the set of terms over $\mathcal{G}$. The arity of a function symbol f is denoted by $\mathrm{ar}(f)$. Variables are denoted by x, y, terms are denoted by s, t, u, v, and finite sets of terms are written $E, F, ...$, and decorations thereof, respectively. We abbreviate $E \cup F$ by E, F, the union $E \cup \{t\}$ by E, t and $E \setminus \{t\}$ by $E \setminus t$.

Given a signature $\mathcal{G}$, a *constant* is either a free constant or a function symbol of arity 0 in $\mathcal{G}$. We define the set of atoms Atoms to be the union of $\mathcal{X}$ and the set of constants. Given a term t we denote by $\mathrm{Var}(t)$ the set of variables occurring in t and by $\mathrm{Cons}(t)$ the set of constants occurring in t. We denote by $\mathrm{Atoms}(t)$ the set $\mathrm{Var}(t) \cup \mathrm{Cons}(t)$. A substitution σ is an involutive mapping from $\mathcal{X}$ to $\mathrm{T}(\mathcal{G},\mathcal{X})$ such that $\mathrm{Supp}(\sigma) = \{x | \sigma(x) \neq x\}$, the *support* of σ, is a finite set. The application of a substitution σ to a term t (resp. a set of terms E) is denoted $t\sigma$ (resp. $E\sigma$) and is equal to the term t (resp. E) where all variables x have been replaced by the term $\sigma(x)$. A substitution σ is *ground* w.r.t. $\mathcal{G}$ if the image of $\mathrm{Supp}(\sigma)$ is included in $\mathrm{T}(\mathcal{G})$.

An *equational presentation* $\mathcal{H} = (\mathcal{G}, A)$ is defined by a set A of equations $u = v$ with $u, v \in \mathrm{T}(\mathcal{G}, \mathcal{X})$ and u, v without free constants. For any equational presentation $\mathcal{H}$ the relation $=_{\mathcal{H}}$ denotes the equational theory generated by $(\mathcal{G}, A)$ on $\mathrm{T}(\mathcal{G}, \mathcal{X})$, that is the smallest congruence containing all instances of axioms of A. Abusively we shall not distinguish between an equational presentation $\mathcal{H}$ over a signature $\mathcal{G}$ and a set A of equations presenting it and we denote both by $\mathcal{H}$. We will also often refer to $\mathcal{H}$ as an equational theory (meaning the equational theory presented by $\mathcal{H}$). An equational theory $\mathcal{H}$ is said to be *consistent* if two free constants are not equal modulo $\mathcal{H}$ or, equivalently, if it has a model with more than one element modulo $\mathcal{H}$.

For all signature $\mathcal{G}$ that we consider, we assume that $<_{\mathcal{G}}$ is a total simplification ordering on $\mathrm{T}(\mathcal{G})$ for which the minimal element is a free constant $\mathrm{c}_{\mathrm{min}}$. *Unfailing completion* permits, given an equational theory $\mathcal{H}$ defined by a set A of equations, to build from A a (possibly infinite) set $R(A)$ of equations $l = r$ such that the ordered rewriting relation between terms defined by $t \rightarrow_{R(A)} t'$ if:

- There exists $l = r \in R(A)$ and a ground substitution σ such that $l\sigma = s$ and $r\sigma = s'$, $t = t[s]$ and $t' = t[s \leftarrow s']$;
- We have $t' <_{\mathcal{G}} t$.

This ordered rewriting relation is convergent, that is for all terms t, all ordered rewriting sequences starting from t are finite, and they all have the same limit, called the *normal form* of t. We denote this term $(t)\!\downarrow_{R(A)}$, or $(t)\!\downarrow$ when the equational theory considered is clear from the context. In the sequel we denote $\mathrm{C}_{\mathrm{spe}}$ the set consisting of $\mathrm{c}_{\mathrm{min}}$ and of all symbols in $\mathcal{G}$ of arity 0.

The *syntactic subterms* of a term t are denoted $\mathrm{Sub}_{\mathrm{syn}}(t)$ and are defined recursively as follows. If t is an atom then $\mathrm{Sub}_{\mathrm{syn}}(t) = \{t\}$. If $t = f(t_1, \ldots, t_n)$ then $\mathrm{Sub}_{\mathrm{syn}}(t) = \{t\} \cup \bigcup_{i=1}^{n} \mathrm{Sub}_{\mathrm{syn}}(t_i)$. The *positions* in a term t are sequences of integers defined recursively as follows, ε being the empty sequence. The term t is at position ε in t. We also say that ε is the root position. We write $p \leq q$ to denote that the position p is a prefix of position q. If u is a syntactic subterm of t at position p and if $u = f(u_1, \ldots, u_n)$ then u_i is at position $p \cdot i$ in t for $i \in \{1, \ldots, n\}$. We write $t_{|p}$ the subterm of t at position p. We denote $t[s]$ a term t that admits s as syntactic subterm. We denote by $\mathrm{top}(\_)$ the function that associates to each term t its root symbol.

2.2 Mode in an equational theory

We recall here the notion of *mode* on a signature, which is defined in [4]. Assume $\mathcal{H}$ is an equational theory over a signature $\mathcal{G}$, and let $\mathcal{G}_0$ be a subset of $\mathcal{G}$. Assume also that the set of variables is partitioned into two sets $\mathcal{X}_0$ and $\mathcal{X}_1$. We first define a signature function Sign(_) on $\mathcal{G} \cup \text{Atoms}$ in the following way:

$$\text{Sign}(\_) : \mathcal{G} \cup \text{Atoms} \to \{0,1,2\}$$
$$\text{Sign}(f) = \begin{cases} 0 \text{ if } f \in \mathcal{G}_0 \cup \mathcal{X}_0 \\ 1 \text{ if } f \in (\mathcal{G} \setminus \mathcal{G}_0) \cup \mathcal{X}_1 \\ 2 \text{ otherwise, i.e. when } f \text{ is a free constant} \end{cases}$$

The function Sign(_) is extended to terms by taking Sign(t)= Sign(top(t)).

We also assume that there exists a *mode* function $\text{m}(\cdot,\cdot)$ such that $\text{m}(f,i)$ is defined for every symbol $f \in \mathcal{G}$ and every integer i such that $1 \leq i \leq \text{ar}(f)$. For all valid f, i we have $\text{m}(f,i) \in \{0,1\}$ and $\text{m}(f,i) \leq \text{Sign}(f)$. Thus for all $f \in \mathcal{G}_0$ and for all i we have $\text{m}(f,i) = 0$.

Well-moded equational theories. A position different from ε in a term t is *well-moded* if it can be written $p \cdot i$ (where p is a position and i a nonnegative integer) such that $\text{Sign}(t_{|p \cdot i}) = \text{m}(\text{top}(t_{|p}), i)$. In other words the position in a term is well-moded if the subterm at that position is of the expected type w.r.t. the function symbol immediately above it. A term is *well-moded* if all its *non root* positions are well-moded. Note in particular that a well-moded term does not contain free constants. If a position of t is not well-moded we say it is *ill-moded* in t. A term is *pure* if its only ill-moded subterms are atoms. An equational presentation $\mathcal{H} = (\mathcal{G}, A)$ is well-moded if for all equations $u = v$ in A the terms u and v are well-moded and Sign(u)=Sign(v). One can prove that if an equational theory is well-moded then its completion is also well-moded [4].

Note that if $\mathcal{H}$ is the union of two equational theories $\mathcal{H}_0$ and $\mathcal{H}_1$ over two disjoint signatures $\mathcal{G}_0$ and $\mathcal{G}_1$, the theory $\mathcal{H}$ is well-moded when assigning mode i to each argument of each operator $g \in \mathcal{G}_i$, for $i \in \{0,1\}$.

Subterm values. The notion of mode also permits to define a new subterm relation in $\text{T}(\mathcal{G}, \mathcal{X})$.

We call a *subterm value* of a term t a syntactic subterm of t that is either atomic or occurs at an ill-moded position of t[1]. We denote Sub(t) the set of subterm values of t. By extension, for a set of terms E, the set Sub(E) is defined as the union of the subterm values of the elements of E. The subset of the maximal and strict subterm values of a term t plays an important role in the sequel. We call these subterm values the *factors* of t, and denote this set Factors(t).

Example 1. Consider two binary symbols f and g with $\text{Sign}(f) = \text{Sign}(g) = \text{m}(f,1) = \text{m}(g,1) = 1$ and $\text{m}(f,2) = \text{m}(g,2) = 0$, and $t = f(f(g(a,b), f(c,c)), d)$. Its subterm values are a, b, $f(c,c)$, c, d, and its factors are a, b, $f(c,c)$ and d.

In the rest of this paper and unless otherwise indicated, *the notion of subterm will refer to subterm values.*

[1] Note that the root position of a term is *always* ill-moded.

Unification systems. We review here properties of well-moded theories with respect to unification that are addressed in [4].

Assume $\mathcal{H}$ is a well-moded equational theory over a signature $\mathcal{G}$, and let $\mathcal{H}_0$ be its projection over the signature $\mathcal{G}_0$ of symbols of signature 0. Let us first define unification systems with ordering constraints.

Definition 1. *(Unification systems) Let $\mathcal{H}$ be a set of equational axioms on* $\mathrm{T}(\mathcal{G},\mathcal{X})$. *An* $\mathcal{H}$-unification system $\mathcal{S}$ *is a finite set of couples of terms in* $\mathrm{T}(\mathcal{G},\mathcal{X})$ *denoted by* $\{u_i \overset{?}{=} v_i\}_{i\in\{1,\ldots,n\}}$. *It is satisfied by a ground substitution* σ, *and we note* $\sigma \models_{\mathcal{H}} \mathcal{S}$, *if for all* $i \in \{1,\ldots,n\}$ *we have* $u_i\sigma =_{\mathcal{H}} v_i\sigma$.

We will consider only satisfiability of unification systems with ordering constraints. That is, we consider the following decision problem:

Ordered Unifiability

Input:	A $\mathcal{H}$-unification system $\mathcal{S}$ and an ordering $\prec$ on the variables X and constants C of $\mathcal{S}$.
Output:	SAT iff there exists a substitution σ such that $\sigma \models_{\mathcal{H}} \mathcal{S}$ and for all $x \in X$ and $c \in C$, $x \prec c$ implies $c \notin \mathrm{Sub}_{\mathrm{syn}}(x\sigma)$

3 Analysis of reachability properties of cryptographic protocols

We recall in this section the definitions of [3] concerning our model of an intruder attacking actively a protocol, and of the simultaneous constraint satisfaction problems employed to model a finite execution of a protocol.

3.1 Intruder deduction systems

We first recall here the general definition of intruder systems, as is given in [3]. We then recall the definition of a *well-moded intruder* that we will use in this paper. In the context of a security protocol (see *e.g.* [12] for a brief overview), we model messages as ground terms and intruder deduction rules as rewrite rules on sets of messages representing the knowledge of an intruder. The intruder derives new messages from a given (finite) set of messages by applying intruder rules. Since we assume some equational axioms $\mathcal{H}$ are satisfied by the function symbols in the signature, all these derivations have to be considered *modulo* the equational congruence $=_{\mathcal{H}}$ generated by these axioms. In our setting an intruder deduction rule is specified by a term t in some signature $\mathcal{G}$. Given values for the variables of t the intruder is able to generate the corresponding instance of t.

Definition 2. *An* intruder system $\mathcal{I}$ *is given by a triple* $\langle \mathcal{G}, \mathcal{S}, \mathcal{H}\rangle$ *where* $\mathcal{G}$ *is a signature,* $S \subseteq \mathrm{T}(\mathcal{G},\mathcal{X})$ *and* $\mathcal{H}$ *is a set of equations between terms in* $\mathrm{T}(\mathcal{G},\mathcal{X})$. *To each* $t \in S$ *we associate a* deduction rule $\mathrm{L}^t : \mathrm{Var}(t) \to t$ *and* $\mathrm{L}^{t,\mathrm{g}}$ *denotes the set of ground instances of the rule* L^t modulo $\mathcal{H}$*:*

$$\mathrm{L}^{t,\mathrm{g}} = \{l \to r \mid \exists\sigma, \textit{ground substitution on } \mathcal{G},\ l = \mathrm{Var}(t)\sigma \text{ and } r =_{\mathcal{H}} t\sigma\}$$

The set of rules $\mathrm{L}_{\mathcal{I}}$ *is defined as the union of the sets* $\mathrm{L}^{t,\mathrm{g}}$ *for all* $t \in \mathcal{S}$.

Each rule $l \to r$ in $\mathrm{L}_{\mathcal{I}}$ defines an intruder deduction relation $\to_{l\to r}$ between finite sets of terms. Given two finite sets of terms E and F we define $E \to_{l\to r} F$ if and only if $l \subseteq E$ and $F = E \cup \{r\}$. We denote $\to_{\mathcal{I}}$ the union of the relations $\to_{l\to r}$ for all $l \to r$ in $L_{\mathcal{I}}$ and by $\to_{\mathcal{I}}^{*}$ the transitive closure of $\to_{\mathcal{I}}$. Note that by definition, given sets of terms E, E' ,F and F' such that $E =_{\mathcal{H}} E'$ and $F =_{\mathcal{H}} F'$ we have $E \to_{\mathcal{I}} F$ iff $E' \to_{\mathcal{I}} F'$. We simply denote by $\to$ the relation $\to_{\mathcal{I}}$ when there is no ambiguity about $\mathcal{I}$.

A *derivation* D of length n, $n \geq 0$, is a sequence of steps of the form $E_0 \to_{\mathcal{I}} E_0, t_1 \to_{\mathcal{I}} \cdots \to_{\mathcal{I}} E_n$ with finite sets of ground terms $E_0, \ldots E_n$, and ground terms $t_1, \ldots, t_n$, such that $E_i = E_{i-1} \cup \{t_i\}$ for every $i \in \{1, \ldots, n\}$. The term t_n is called the *goal* of the derivation. We define $\overline{E}^{\mathcal{I}}$ to be equal to the set $\{t \mid \exists F \text{ s.t. } E \to_{\mathcal{I}}^{*} F \text{ and } t \in F\}$ *i.e.* the set of terms that can be derived from E. If there is no ambiguity on the deduction system $\mathcal{I}$ we write $\overline{E}$ instead of $\overline{E}^{\mathcal{I}}$.

We now define well-moded intruder systems and their properties.

Definition 3. *Given a well-moded equational theory $\mathcal{H}$, an intruder system $\mathcal{I} = \langle \mathcal{G}, S, \mathcal{H} \rangle$ is* well-moded *if all terms in S are well-moded.*

3.2 Simultaneous constraint satisfaction problems

We introduce now the constraint systems to be solved for checking protocols. It is shown in [3] how these constraint systems permit to express the reachability of a state in a protocol execution.

Definition 4. *(Constraint systems) Let $\mathcal{I} = \langle \mathcal{G}, S, \mathcal{H} \rangle$ be an intruder system. An $\mathcal{I}$-*Constraint system *$\mathcal{C}$ is denoted: $((E_i \rhd v_i)_{i\in\{1,\ldots,n\}}, \mathcal{S})$ and it is defined by a sequence of couples $(E_i, v_i)_{i\in\{1,\ldots,n\}}$ with $v_i \in \mathcal{X}$ and $E_i \subseteq \mathrm{T}(\mathcal{G}, \mathcal{X})$ for $i \in \{1, \ldots, n\}$, and $E_{i-1} \subseteq E_i$ for $i \in \{2, \ldots, n\}$ and by an $\mathcal{H}$-unification system $\mathcal{S}$.*

*An $\mathcal{I}$-*Constraint system *$\mathcal{C}$ is satisfied by a ground substitution σ if for all $i \in \{1, \ldots, n\}$ we have $v_i\sigma \in \overline{E_i\sigma}$ and if $\sigma \models_{\mathcal{H}} \mathcal{S}$. If a ground substitution σ satisfies a constraint system $\mathcal{C}$ we denote it by $\sigma \models_{\mathcal{I}} \mathcal{C}$.*

Constraint systems are denoted by $\mathcal{C}$ and decorations thereof. Note that if a substitution σ is a solution of a constraint system $\mathcal{C}$, by definition of constraint and unification systems the substitution $(\sigma)\!\downarrow$ is also a solution of $\mathcal{C}$. In the context of cryptographic protocols the inclusion $E_{i-1} \subseteq E_i$ means that the knowledge of an intruder does not decrease as the protocol progresses: after receiving a message a honest agent will respond to it. This response can be added to the knowledge of an intruder who listens to all communications.

We are not interested in general constraint systems but only in those related to protocols. In particular we need to express that a message to be sent at some step i should be built from previously received messages recorded in the variables $v_j, j < i$, and from the initial knowledge. To this end we define:

Definition 5. *(Deterministic Constraint Systems) We say that an $\mathcal{I}$-constraint system $((E_i \rhd v_i)_{i\in\{1,\ldots,n\}}, \mathcal{S})$ is* deterministic *if for all i in $\{1, \ldots, n\}$ we have* $\mathrm{Var}(E_i) \subseteq \{v_1, \ldots, v_{i-1}\}$

In order to be able to combine solutions of constraints for the intruder theory $\mathcal{I}$ with solutions of constraint systems for intruders defined on a disjoint signature we have, as for unification, to introduce some ordering constraints to be satisfied by the solution (see [3] for details on this construction). Intuitively, these ordering constraints prevent from introducing cycle when building a global solution. This motivates us to define the *Ordered Satisfiability* problem:

Ordered Satisfiability

Input: an $\mathcal{I}$-constraint system $\mathcal{C}$, $X = \mathrm{Var}(\mathcal{C})$, $C = \mathrm{Const}(\mathcal{C})$ and a linear ordering $\prec$ on $X \cup C$.

Output: SAT iff there exists a substitution σ such that $\sigma \models_{\mathcal{I}} \mathcal{C}$ and for all $x \in X$ and $c \in C$, $x \prec c$ implies $c \notin \mathrm{Sub}_{\mathrm{syn}}(x\sigma)$

4 Model of a collision-aware intruder

We define in this section intruder systems to model the way an active intruder may deliberately create collisions for the application of hash functions. Note that our model doesn't take into account the time for finding collisions, which is significantly greater than the time necessary for other operations. The results that we can obtain can therefore be seen as worst-case results, and should be assessed with respect to the possible time deadline in the actual specification of a protocol under analysis. Further works will also be concerned with the fact that given a bound on intruder's deduction capabilities, a collision may be found only with a probability p, $0 \leq p \leq 1$.

We consider in this paper five different intruder models. We will reduce in two steps the most complex one to a simpler one, relying on the notion of well-moded theories and on the results in [4]. We then prove decidability of ordered reachability for this simpler intruder system.

4.1 Intruder on words

We first define our goal intruder, that is an intruder only able to concatenate messages and extract *prefixes* and *suffixes*. We denote $\mathcal{I}_{\mathrm{AU}} = \langle \mathcal{F}_{AU}, S_{AU}, \mathcal{E}_{AU} \rangle$ an intruder system that operates on words, such that, if $\_ \cdot \_$ denotes the concatenation and ϵ denotes the empty word, the intruder has at its disposal all ground instances of the following deduction rules:

$$\begin{cases} x, y \to x \cdot y \\ x \cdot y \to x \\ x \cdot y \to y \\ \to \epsilon \end{cases}$$

We moreover assume that the concatenation and empty word operations satisfy the following equations:

$$\begin{cases} x \cdot (y \cdot z) = (x \cdot y) \cdot z \\ \qquad x \cdot \epsilon = x \qquad\qquad \epsilon \cdot x = x \end{cases}$$

Given these definitions, we can see terms over $\mathrm{T}(\mathcal{F}_{\mathrm{AU}}, \mathcal{X})$ as *words* over the alphabet $\mathcal{X} \cup \mathrm{C}$, and we denote letters$(w)$ the set of atoms (either variable or free constants) occurring in w. As usual, we extend letters$(_)$ to set of terms in $\mathrm{T}(\mathcal{F}_{\mathrm{AU}}, \mathcal{X})$ by taking the union of letters occurring in each term.

Pitfall. Note that this intruder model does not fit into the intruder systems definition of [3, 4]. The rationale for this is that, in the notation given here, the application of the rules is non-deterministic, and thus cannot be modelled easily into our "deduction by normalisation" model. We however believe that a deterministic and still associative model of message concatenation by means of an "element" unary operator, associative operator "$\cdot$", and Head and Tail operations may be introduced. This means that we also assume that unification problems are only among words of this underlying theory, disregarding equations that may involve these extra operators. We leave the exact soundness of our model for further analysis and concentrate on the treatment of collisions discovery for hash functions.

4.2 Intruder on words with free function symbols

We extend the $\mathcal{I}_{\mathrm{AU}}$ intruder with two free function symbols f and g of arity 4, and we add to the possible deductions of the intruder the application of the following deduction rules:

$$\begin{cases} x_1, x_2, y_1, y_2 \to \mathrm{g}(x_1, x_2, y_1, y_2) \\ x_1, x_2, y_1, y_2 \to \mathrm{f}(x_1, x_2, y_1, y_2) \end{cases}$$

This leads to an intruder system that will be an intermediate in the proof of our decision procedure. We denote it $\mathcal{I}_{\mathrm{free}}$, and we have:

$$\mathcal{I}_{\mathrm{free}} = \langle \mathcal{F}_{\mathrm{AU}} \cup \{\mathrm{g}, \mathrm{f}\}, S_{\mathrm{AU}} \cup \{\mathrm{f}(x_1, x_2, y_1, y_2), \mathrm{g}(x_1, x_2, y_1, y_2)\}, \mathcal{E}_{\mathrm{AU}} \rangle \,.$$

4.3 Hash-colliding intruder

We consider a signature modelling the following different operations:

- The *concatenation* of two messages, the extraction of a suffix or a prefix of a concatenated message and the production of an empty message, as in the case of the $\mathcal{I}_{\mathrm{AU}}$ intruder system;
- The application of a hash function h for which it is possible to find collisions, the hash-value of a message m denoted $\mathrm{h}(m)$;
- Two function symbols f and g denoting the (complex) algorithm being used to find collisions starting from two different messages m and m'.

We assume that the algorithm employed by the intruder to find collisions starting from two messages m and m' proceeds as follows:

1. First the intruder splits both messages into two parts, thus choosing m_1, m_2, m'_1, m'_2 such that $m = m_1 \cdot m_2$ and $m' = m'_1 \cdot m'_2$;

2. Then, in order to find collisions, the intruder computes two messages $g(m_1, m_2, m'_1, m'_2)$ and $f(m_1, m_2, m'_1, m'_2)$ such that:

$$(\mathrm{HC}) \quad h(m_1 \cdot g(m_1, m_2, m'_1, m'_2) \cdot m_2) = h(m'_1 \cdot f(m_1, m_2, m'_1, m'_2) \cdot m'_2)$$

A consequence of our model is that in order to build collisions starting from two messages m and m' the intruder must know (*i.e.* have in its knowledge set) these two messages. A side effect is that it is not possible to build three (or more) different messages with the same hash value by iterating the research for collisions. Formally, the core of the proof of this assertion is the following lemma that permits to prove that in an equivalence class of $\mathcal{E}_{\mathrm{h}}$ containing pure terms there exists only two different members modulo $\mathcal{E}_{\mathrm{AU}}$. The proof is based on the fact that occur-check analysis, and thus unification, would fail in a tentative counter-example.

Lemma 1. *Let* $t = h(t_1 \cdot \phi(t_2, t_3, t_4, t_5) \cdot t_6)$ *and* $t' = h(x_1 \cdot g(x_1, x_2, x_3, x_4) \cdot x_2)$ *be two pure terms such that* $\phi \in \{f, g\}$ *and the* t_i *are pure* $\mathcal{F}_{\mathrm{AU}}$ *terms and there exists* $u_1, u_2 \in \{t_2, \ldots, t_5\}$ *such that* $t_1 =_{\mathcal{E}_{\mathrm{AU}}} u_1, t_6 =_{\mathcal{E}_{\mathrm{AU}}} u_2$*. Then for any substitution* σ*, we have* $\sigma \models t \overset{?}{=}_{\emptyset} t'$ *iff* $\sigma \models \left\{ t_{i+1} \overset{?}{=}_{\mathcal{E}_{\mathrm{AU}}} x_i \right\}_{i \in \{1,\ldots,4\}}$ *and:*

$$\begin{cases} \phi = g \\ x_1\sigma = t_1\sigma \quad x_2\sigma = t_6\sigma \end{cases}$$

In a more comprehensive model we might moreover want to model that collisions cannot always be found using attacks published in the literature, but instead that given a deadline, the probability p of success of an attack is strictly below 1. This would imply that the application of this rule by the intruder would, assuming independence of collision attacks, reduce the likelihood of the symbolic attack found. In this setting our model would account for attacks with a non-negligible probability of success as is shown in [1].

Leaving probabilities aside, we express intruder's deductions in our setting by adding the rule $x \rightarrow h(x)$ to the deduction rules of the $\mathcal{I}_{\mathrm{free}}$ intruder. As a consequence, the previous description of the $\mathcal{I}_{\mathrm{free}}$ intruder enables us to model a collision-capable intruder

$$\mathcal{I}_{\mathrm{h}} = \langle \mathcal{F}_{\mathrm{h}}, S_{\mathrm{h}}, \mathcal{E}_{\mathrm{h}} \rangle$$

$$\text{with:} \quad \begin{cases} \mathcal{F}_{\mathrm{h}} = \mathcal{F}_{\mathrm{AU}} \cup \{f, g, h\} \\ S_{\mathrm{h}} = S_{\mathrm{AU}} \cup \{f(x_1, x_2, y_1, y_2), g(x_1, x_2, y_1, y_2), h(x)\} \\ \mathcal{E}_{\mathrm{h}} = \mathcal{E}_{\mathrm{AU}} \cup \{(HC)\} \end{cases}$$

For the following mode and signature functions the theory $\mathcal{E}_{\mathrm{AU}} \cup \{(HC)\}$ is a well-moded theory.

$$\text{mode:} \quad \begin{cases} \mathrm{m}(.,1) = \mathrm{m}(.,2) = \mathrm{m}(\mathrm{g}\,,i) = \mathrm{m}(\mathrm{f}\,,i) = 0 \ \forall i \in \{1,\dots,4\} \\ \mathrm{m}(\mathrm{h}\,,1) = 0 \end{cases}$$

$$\text{Signature:} \quad \begin{cases} \mathrm{Sign}(\cdot) = \mathrm{Sign}(\epsilon) = \mathrm{Sign}(\mathrm{f}\,) = \mathrm{Sign}(\mathrm{g}\,) = 0 \\ \mathrm{Sign}(\mathrm{h}\,) = 1 \end{cases}$$

The main result of this paper is the following decidability result.

Theorem 1 *Ordered satisfiability for the $\mathcal{I}_{\mathrm{h}}$ intruder is decidable.*

The rest of this paper is dedicated to the proof of this theorem. The technique employed consists in successive reductions to simpler problems and in finally proving that all simpler problems are decidable. These reductions are summarised in Figure 4.3. A proof sketch for the decidability of the $\mathcal{I}_{\mathrm{g}}$, $\mathcal{I}_{\mathrm{f}}$ and $\mathcal{I}_{\mathrm{AU}}$ is given in Section 5.2. Algorithm 1, that permits the first reduction, is based on the facts that the $\mathcal{I}_{\mathrm{h}}$ intruder is well-moded (as seen above) and that we can apply a reduction according to the criterion of [4] for well-moded intruder systems.

$$\begin{tikzcd}
& \mathcal{I}_{\mathrm{h}} \arrow[d, "\text{Algorithm 1}"] & \\
& \mathcal{I}_{\mathrm{free}} \arrow[dl] \arrow[d, "\text{Generic combination algorithm [3]}"] \arrow[dr] & \\
\mathcal{I}_{\mathrm{g}} & \mathcal{I}_{\mathrm{f}} & \mathcal{I}_{\mathrm{AU}}
\end{tikzcd}$$

Fig. 1. Reduction strategy

HYPOTHESIS 1: If $E \rightarrow_{\mathcal{S}_1} E, r \rightarrow_{\mathcal{S}_1} E, r, t$ and $r \notin \mathrm{Sub}(E, t) \cup \mathrm{C}_{\mathrm{spe}}$ then there is a set of terms F such that $E \rightarrow^*_{\mathcal{S}_0} F \rightarrow_{\mathcal{S}_1} F, t$.

If a well-moded intruder system system satisfies this hypothesis, then the following proposition holds. It is a cornerstone for the proof of completeness of Algorithm 1.

Proposition 1. *Let $\mathcal{I}$ be a well-moded intruder that satisfies hypothesis 1, and let $\mathcal{C}$ be a deterministic $\mathcal{I}$-constraint system. If $\mathcal{C}$ is satisfiable, there exists a substitution σ such that $\sigma \models_{\mathcal{I}} \mathcal{C}$ and:*

$$|\{t \in \mathrm{Sub}((\mathrm{Sub}(\mathcal{C})\sigma)\!\downarrow) \mid \mathrm{Sign}(t) = 1\}| \leq |\{t \in \mathrm{Sub}(\mathcal{C}) \mid \mathrm{Sign}(t) = 1\}| + |X|$$

5 Decidability of reachability

We present here a decision procedure for *Ordered Satisfiability Problem* for $\mathcal{I}_{\mathrm{h}}$ intruder system. Our technique consists in simplifying the intruder system $\mathcal{I}_{\mathrm{h}}$ to $\mathcal{I}_{\mathrm{free}}$. We then reduce the decidability problems of ordered reachability for deterministic constraint problems for $\mathcal{I}_{\mathrm{free}}$ to the decidability problems of ordered reachability for deterministic constraint problems for $\mathcal{I}_{\mathrm{g}}$, $\mathcal{I}_{\mathrm{f}}$ and $\mathcal{I}_{\mathrm{AU}}$. We finally prove the decidability for these intruder systems.

5.1 Reduction to $\mathcal{I}_{\text{free}}$-intruder

Algorithm We present here a procedure for reducing $\mathcal{I}_{\text{h}}$ intruder system to $\mathcal{I}_{\text{free}}$ intruder system that takes as input a deterministic constraint system $\mathcal{C} = ((E_i \rhd v_i)_{i\in\{1,\dots,n\}}, \mathcal{S})$ and a linear ordering $\prec_i$ on atoms of $\mathcal{C}$. Let $m = |\text{Sub}(\mathcal{C})|$ be the number of subterms in $\mathcal{C}$.

Algorithm 1

Step 1. Choose a number $k \le m$ and add k equations $h_j \stackrel{?}{=} \text{h}(c_j)$ to $\mathcal{S}$ where the h_j, c_j are new variables.

Step 2. For each $t \in \text{Sub}(\mathcal{C}) \cup \{c_1, \dots, c_k\}$ choose a *type* 0 or 1. If t is of type 1, choose $j_t \in \{1, \dots, k\}$ and add an equation $t \stackrel{?}{=} h_{j_t}$ to $\mathcal{S}$.

Step 3. For all $t, t' \in \text{Sub}(\mathcal{C})$, if there exists $h \in \{h_1, \dots, h_k\}$ such that $t \stackrel{?}{=} h$ and $t' \stackrel{?}{=} h$ are in $\mathcal{S}$, add to $\mathcal{S}$ an equation $t \stackrel{?}{=} t'$ to $\mathcal{S}$.

Step 4. Choose a subset H of $\{c_1, \dots, c_k\}$ and guess a total order $<_d$ on $L = H \cup \{v_1, \dots, v_n\}$ such that $v_i <_d v_j$ iff $i < j$. Write the obtained list $w_1, \dots, w_{n+k}$. Let $\mathcal{S}$' be the unification system obtained so far, and form: $\mathcal{C}' = ((F_i \rhd w_j)_{1\le j\le n+k}, \mathcal{S}')$ with:

$$\begin{cases} F_1 = E_1 & \\ F_{i+1} = F_i \cup (E_{j+1} \setminus E_j) & \text{if } w_i = v_j \\ F_{i+1} = F_i, w_i & \text{Otherwise} \end{cases}$$

Step 5. For all $t \in \text{Sub}(\mathcal{C})$ chosen of type 1, replace all occurrences of t in the F_i and all occurrence occurrences of t *as a strict subterm* in $\mathcal{S}'$ by the representant of its class h_{j_t}. Let F_i' be the set F_i once this abstraction has been applied

Step 6. Non-deterministically reduce $\mathcal{S}$' to a unification system $\mathcal{S}$" free of h symbols, and form the satisfiable $\mathcal{I}_{\text{free}}$ constraint system:

$$\mathcal{C}'' = ((F_i' \rhd w_i)_{1\le i\le n+k}, \mathcal{S}'')$$

Sketch of the completeness proof. Assume that the initial deterministic constraint system is satisfiable. By Proposition 1, there exists a bound substitution σ satisfying $\mathcal{C}$.

- Let the number k chosen at Step 1 be the number of subterms whose top symbol is h in $\text{Sub}((\text{Sub}(\mathcal{C})\sigma)\!\downarrow)$. The h_j represent the different values of the terms of signature 1. In the sequel we assume that σ is extended to the h_j such that all $h_j\sigma$ have a different value and are of signature 1.
- In Step 2, if $\text{Sign}((t\sigma)\!\downarrow) = 1$ we choose the j such that $(t\sigma)\!\downarrow = h_j\sigma$ and add the corresponding equation to $\mathcal{S}$.
- In Step 3, we had equations between terms whose normal form by σ are equals in order to simplify the reduction to $\mathcal{I}_{\text{free}}$.
- Step 4 is slightly more intricate. It relies on the fact that a rule in $\mathcal{S}_1$ may only yield a term whose normal form by σ is of signature 1.

The subset H correspond to the subterms of signature 1 of $\mathrm{Sub}((\mathrm{Sub}(\mathcal{C}\sigma))\downarrow)$ that are deduced by the intruder using a rule in $\mathcal{S}_1$. We then anticipate the construction of $h_j\sigma$ with the application of a rule in $\mathcal{S}_1$ by requiring that the corresponding $c_j\sigma$ has to be build just before. Given the bound on k, this means that all remaining deductions performed by the intruder are now instances of rules in $\mathcal{S}_0$. Since $\mathcal{C}$ is satisfied by σ there exists a choice corresponding to *quasi well-formed* derivations such that all remaining reachability constraints are satisfiable by instances of rules in $\mathcal{S}_0$.

– At Step 5 we "purify" almost all the constraint system by removing all occurrences of a symbol h but the ones that are on the top of an equality. By the choice of the equivalence classes it is clear that this purification does not loose the satisfiability by the substitution σ.
– The non-deterministic reduction is performed by guessing whether the equality of two hashes is the consequence of a collision set up by the intruder or of the equality of the hashed messages, and will produce a constraint system $\mathcal{C}$" without h symbol and also satisfiable by σ.

5.2 Decidability of reachability for the $\mathcal{I}_{\text{free}}$-intruder

We first reduce the $\mathcal{I}_{\text{free}}$ intruder system to simpler intruder systems using the combination result of [3]. We will consider the decidability of these subsystems in the remainder of this section.

Theorem 2 *Ordered satisfiability for the $\mathcal{I}_{\text{free}}$ intruder system is decidable.*

Decidability of reachability for the $\mathcal{I}_{\text{g}}$-intruder. In this subsection, we consider an $\mathcal{I}_{\text{g}}$ intruder system with $\mathcal{I}_{\text{g}} = \langle \mathrm{g}, \mathrm{g}(x_1, x_2, x'_1, x'_2), \emptyset \rangle$. This intruder has at its disposal all ground instances of the following deduction rule:

$$x_1, x_2, y_1, y_2 \rightarrow \mathrm{g}(x_1, x_2, y_1, y_2)$$

The proof of the following theorem consists in first proving the existence of well-formed derivations for the standard subterm relation in the spirit of [16], with the additional simplification that all rules are composition rules. One then guesses a minimal attack in non-deterministic polynomial time. Since the $\mathcal{I}_{\text{g}}$ and $\mathcal{I}_{\text{f}}$ intruder are isomorphic the result also applies to $\mathcal{I}_{\text{f}}$ after renaming of the symbol g into f.

Theorem 3 *Ordered satisfiability for the $\mathcal{I}_{\text{g}}$ intruder system is decidable.*

Decidability of reachability for the AU-intruder. We now give a proof sketch for the decidability of ordered satisfiability for the $\mathcal{I}_{\text{AU}}$ intruder since the procedure is new.

Theorem 4 *Ordered satisfiability for the $\mathcal{I}_{\text{AU}}$ intruder system is decidable.*

Proof. The algorithm proceeds as follows:

– Transform the deduction constraints $E \rhd v$ into an ordering constraint $<_d$;
– Check that $< = <_d \cup <_i$ is still a partial order on atoms of $\mathcal{C}$;
– Solve the unification problem with linear constant restriction $<$.

Let $\mathcal{C} = ((E_i \rhd v_i)_{0 \leq i \leq n}, \mathcal{S})$ be a deterministic constraint system for the $\mathcal{I}_{\mathrm{AU}}$ intruder, $<_i$ be a (partial) order on $\mathrm{Cons}(\mathcal{C}) \cup \mathrm{Var}(\mathcal{C})$, and let σ be a solution of the $(\mathcal{C}, <_i)$ ordered satisfiability problem.

Given a set of terms $E \subseteq \mathrm{T}(\mathcal{F}_{\mathrm{AU}}, \mathcal{X})$, let us denote $\mathrm{K}_{\mathcal{C}} = (\mathrm{Cons}(\mathcal{C}) \setminus \mathrm{letters}(E)) \setminus \mathcal{X}$. In plain words, $\mathrm{K}_{\mathcal{C}}(E)$ is the set of constants in $\mathcal{C}$ **not** occurring in E. We are now ready to define $<_d$ as a partial order on $\mathrm{Cons}(\mathcal{C}) \cup \{v_0, \ldots, v_n\}$: We set $v_i <_d c$ for all constants c in $\mathrm{K}_{\mathcal{C}}(E_i)$.

Claim. For all σ, we have $\sigma \models (\mathcal{C}, <_i)$ if, and only if, $\sigma \models (\mathcal{S}, <_i \cup <_d)$

Proof of the claim. Let us first prove the direct implication. Let σ be a ground solution of the $(\mathcal{C}, <_i)$ ordered satisfiability problem. By definition we have that σ is a solution of $(\mathcal{S}, <_i)$ ordered unifiability problem. Since for all $0 \leq i \leq n$ we have $\sigma \models E_i \rhd v_i$, we easily see that $\mathrm{letters}((v_i\sigma)\downarrow) \subseteq \mathrm{Cons}(E_i)$, and therefore $\mathrm{letters}((v_i\sigma)\downarrow) \cap \mathrm{K}_{\mathcal{C}}(E_i) = \emptyset$. Thus σ is also a solution of $(\mathcal{S}, <_d \cup <_i)$. Conversely, assume now that σ is a ground solution of $(\mathcal{S}, <_d \cup <_i)$. By definition for all $0 \leq i \leq n$ we have $\mathrm{letters}((v_i\sigma)\downarrow) \cap \mathrm{K}_{\mathcal{C}}(E_i) = \emptyset$, and thus $\mathrm{letters}((v_i\sigma)\downarrow) \subseteq \mathrm{letters}(E_i) \setminus \mathcal{X}$. Thus we have $(v_i\sigma)\downarrow \in \overline{(E_i\sigma)\downarrow}$ for all $0 \leq i \leq n$, and thus $\sigma \models (\mathcal{C}, <_i)$ $\Diamond$

Since unifiability with linear constant restriction is decidable for the *AU* equational theory [18], this finishes the proof of the theorem. Note that the exact complexity is not known, but the problem is NP-hard and solvable in PSPACE [13, 14], and it is conjectured to be in NP [15, 11]. $\square$

6 Conclusion

We have presented here a novel decision procedure for the search for attacks on protocols employing hash functions subject to collision attacks. Since this procedure is of practical interest for the analysis of the already normalised protocols relying on these weak functions, we plan to implement it into an already existing tool, CL-Atse. We also plan to formalise according to the model of [3] the underlying *AU* intruder system. In order to model hash functions we have introduced new symbols to denote the ability to create messages with the same hash value. This introduction amounts to the skolemisation of the equational property describing the existence of collisions We believe that this construction can be extended to model the more complex and game-based properties that appear when relating a symbolic and a concrete model of cryptographic primitives.

References

1. M. Baudet. Random polynomial-time attacks and Dolev-Yao models. In Siva Anantharaman, editor, *Proceedings of the Workshop on Security of Systems: Formalism and Tools (SASYFT'04)*, Orléans, France, June 2004.

2. E. Biham and R. Chen. Near-collisions of sha-0. In M. K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 290–305. Springer, 2004.
3. Y. Chevalier and M. Rusinowitch. Combining intruder theories. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP*, volume 3580 of *LNCS*, pages 639–651. Springer, 2005.
4. Y. Chevalier and M. Rusinowitch. Hierarchical combination of intruder theories. In Frank Pfenning, editor, *RTA*, volume 4098 of *LNCS*, pages 108–122. Springer, 2006.
5. R. Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *LNCS*. Springer, 2005.
6. B. den Boer and A. Bosselaers. Collisions for the compressin function of md5. In *EUROCRYPT*, pages 293–304. Springer, 1993.
7. H. Dobbertin. Cryptanalysis of md4. In D. Gollmann, editor, *Fast Software Encryption*, volume 1039 of *LNCS*, pages 53–69. Springer, 1996.
8. H. Dobbertin. Cryptanalysis of md5 compress. Presented at the rumps session *of Eurocrypt'96*, 1996.
9. V. Klìma. Finding md5 collisions - a toy for a notebook, 2005. Cryptology ePrint Archive, Report 2005/075. http://eprint.iacr.org/.
10. V. Klìma. Finding md5 collisions on a notebook pc using multi-message modificatons, 2005. Cryptology ePrint Archive, Report 2005/102. http://eprint.iacr.org/.
11. Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Automata, languages and programming, 25th international colloquium, icalp'98, aalborg, denmark, july 13-17, 1998, proceedings. In *ICALP*, volume 1443 of *LNCS*. Springer, 1998.
12. C. Meadows. The NRL protocol analyzer: an overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
13. W. Plandowski. Satisfiability of word equations with constants is in pspace. In *FOCS*, pages 495–500, 1999.
14. W. Plandowski. Satisfiability of word equations with constants is in pspace. *J. ACM*, pages 483–496, 2004.
15. W. Plandowski and W. Rytter. Application of lempel-ziv encodings to the solution of words equations. In *ICALP*, pages 731–742, 1998.
16. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc.14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001. IEEE Press.
17. Y. Sasaki, Y. Naito, N. Kunihiro, and K. Ohta. Wang's sufficient conditions of md5 are not sufficient, 2005. http://eprint.iacr.org/.
18. K. U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In K. U. Schulz, editor, *IWWERT*, volume 572 of *LNCS*, pages 85–150. Springer, 1990.
19. X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5 , haval-128 and ripemd. http://eprint.iacr.org/, 2004.
20. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions md4 and ripemd. In Cramer [5], pages 1–18.
21. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full sha-1. In V. Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
22. X. Wang and H. Yu. How to break md5 and other hash functions. In Cramer [5], pages 19–35.
23. J. Yajima and T. Shimoyama. Wang's sufficient conditions of md5 are not sufficient, 2005. http://eprint.iacr.org/.

Strategy for Flaws Detection based on a Services-driven Model for Group Protocols

Najah Chridi and Laurent Vigneron [⋆]

LORIA – UHP-UN2 (UMR 7503)
BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
{chridi,vigneron}@loria.fr

Abstract. Group key agreement is important in many modern public and dedicated applications. Nevertheless, as they have to be secure, their design is not straightforward. As such, the modelling and the verification of such protocols are necessary in order to avoid eventual weaknesses. This paper investigates a strategy for flaws detection for group protocols properties. The strategy is based on both a services driven model for group protocols and constraint solving. Our strategy has been applied to several group protocols such as GDH.2 and the Asokan-Ginzboorg protocol. This permits to pinpoint new attacks on them. The result found for the case of GDH.2 with four participants can be generalized to n participants. Another general attack has also been found for the case of the A-GDH.2 protocol.

1 Introduction

In recent years, applications requiring an unbounded number of participants have received increasing attention either for public domains or dedicated ones. As such, the design of secure group protocols [14] continues to be one of the most challenging areas of security research. To secure their communications, group members need to use a shared key, known as group key, which has to be updated following the dynamics of the group (join or leave operations, . . .). Therefore, several protocols dedicated to key establishment and updates have been proposed [8]. Among them, we are particularly interested in group key agreement protocols [9]. These protocols enable a group of participants to share a common key over insecure public networks, even when adversaries completely control all the communications.

Research in formal verification of cryptographic protocols has so far mainly concentrated on reachability properties such as secrecy and authentication. It has given so successful interesting results in the last years that this field could be considered as saturated. As such, many fully automatic tools have been developed and successfully applied to find flaws in published protocols, where many of these tools employ so-called *constraint solving* (see, e.g., [3]). Nowadays, dealing with the verification of group protocols arises several problems. Indeed, such protocols highlight new requirements and consider some complicated intended security properties other than secrecy and authentication. In fact, most of the verification approaches can only tackle specific models of protocols, and most of the time require the size of the group to be set in advance. This leads to the restriction of the chances to discover attacks. Besides, group membership is very dynamic; participants can join or leave the group at any time. As such, security requirements are more complicated to satisfy.

The main contribution of the present work is a strategy for flaws detection for the group protocols security properties. Our approach is based on both the services driven model described

[⋆] This work is supported by the QSL operation COWS.

in [4] and constraint solving. As mentioned, constraint solving has been successfully employed for reachability properties in the past and proved to be a good basis for practical implementations. The services driven model permits to specify security properties for group protocols as sets of constraints. This model specifies both group protocols and a large class of their intended properties, varying from standard secrecy and authentication properties to much trickier ones, such as key integrity, and backward and forward secrecy. Hence, our strategy paves the way for extending existing tools for reachability properties to deal with security properties for group protocols.

In this paper, we focus on group key establishment protocols. But this is worth mentioning that our method is also dealing with contributing protocols. To present our results, the paper is structured as follows. We first introduce our running example: The Asokan-Ginzboorg Protocol [2] (Section 2). This protocol will be used throughout this paper to illustrate every new notion introduced. In Section 3, we present the input required by our method. Then, Section 4 provides the necessary background concerning the services driven model. In Section 5, we show how this model can be used to search for attacks. The management of constraints and intruder knowledge is explained in Sections 6 and 7. We illustrate the application of our method to two examples in Section 8. And after a comparision with related work (Section 9), we summarize the results obtained by our approach and then discuss the related work (Section 10).

2 Running Example: The Asokan-Ginzboorg Protocol

Throughout this paper we will illustrate our ideas using a running example: the Asokan-Ginzboorg protocol [2]. It describes the establishment of a session key between a leader (A_n) and a random number n of participants (A_i where $1 <= i <= n$). The protocol proceeds by assuming that a short group password P is chosen and displayed, and then known by all. We assume also that there are two informations known by all the group: a one way hash function H and a commonly known function F. When the leader starts the execution of the protocol by sending the key of encoding (E), each participant generates two informations (a symmetric key (R_i) and a contribution to the group key (S_i)) and sends them encrypted by the key E. Messages exchanged throughout the protocol are expressed as follows:

$$\begin{array}{lll}
1.A_n \longrightarrow \text{ALL} & : & A_n, \{E\}_P \\
2.A_i \longrightarrow A_n & : & A_i, \{R_i, S_i\}_E,\ \text{i=1,\ldots,n-1} \\
3.A_n \longrightarrow A_i & : & \{\{S_j, \text{j=1,\ldots,n}\}\}_{R_i},\ \text{i=1,\ldots,n-1} \\
4.A_i \longrightarrow A_n & : & A_i, \{S_i, H(S_1, \ldots, S_n)\}_k \text{ some } i, k = F(S_1, \ldots, S_n).
\end{array}$$

In this messages exchange, E is a public key generated by the leader and used to encrypt the contribution (S_i) of each participant A_i. R_i denotes a fresh symmetric key generated by the participant A_i, sent to the leader with the contribution S_i. The leader will use it to encrypt all contributions (including S_n) in order to send the whole message to the participant A_i.

3 The Method's Input

As any communication protocol, a group protocol can be seen as an exchange of messages between several participants. This exchange is usually described by the set of actions executed by each participant in a normal protocol execution, i.e. without intervention of an Intruder.

Formally speaking, we define an instance of the protocol as the union of instances of roles and Intruder knowledge. An instance of a protocol is then given by $(\{\mathcal{R}_p \to \mathcal{S}_p\}_{p\in\mathcal{P}}, <_{\mathcal{P}}, S_0)$ where, $\mathcal{P}$ is a finite set and:

– $\{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p \in \mathcal{P}}$ denotes the set of rules of receive-send messages exchanged between honest participants. Each rule defines one step of the protocol: the messages sent by a honest participant ($\mathcal{R}_p$) and the expected response ($\mathcal{S}_p$). Note that $\mathrm{Var}(\mathcal{R}_p) \subseteq \mathrm{Var}(\mathcal{S}_p)$.
– $<_{\mathcal{P}}$ is a partial order over $\mathcal{P}$.
– S_0 is a set of terms representing the initial Intruder knowledge.

Let us return to our running example: the Asokan-Ginzboorg protocol. Having tested our method over several scenarios of this protocol, we have found an interesting result in the case of two parallel sessions. Thus, the modelling considered in the following corresponds to that scenario. In the first session, we have two participants: A_1 is the leader and A_2 is a normal member of the group. In the second session, the roles are exchanged.
Throughout this paper, while expressing informations related to the Asokan-Ginzboorg protocol, we adopt the following notations:

– p_{ijk} denotes the j-th step of the protocol played by the i-th participant in the k-th session.
– Alg_{ij} the point of vue of the group key of the i-th participant during the j-th session.
– Terms written in capital letters are informations known by the participants whether from the beginning of the execution or generated through the execution.
– Terms written in small letters denote variables. They may be instantiated by any values of the same type.
– S_{ij} denotes the contribution to the group key of the i-th participant in the j-th session.

The two sessions are expressed by the following steps:

$\mathbf{p_{111}}$:	$Init$	$\longrightarrow$	$A_1, \{E_1\}_P$	
$\mathbf{p_{121}}$:	$x_1, \{x_2, x_3\}_{E_1}$	$\longrightarrow$	$\{x_3, S_{11}\}_{x_2}$	
$\mathbf{p_{131}}$:	$x_1, \{x_3, H(x_3, S_{11})\}_{F(x_3, S_{11})}$	$\longrightarrow$	$End,$	$\mathbf{Alg_{11} = F(x_3, S_{11})}$
$\mathbf{p_{211}}$:	$x_4, \{x_5\}_P$	$\longrightarrow$	$A_2, \{R_1, S_{21}\}_{x_5}$	
$\mathbf{p_{221}}$:	$\{x_6, x_7\}_{R_1}$	$\longrightarrow$	$A_2, \{S_{21}, H(x_6, x_7)\}_{F(x_6, x_7)},$	$\mathbf{Alg_{21} = F(x_6, x_7)}$
$\mathbf{p_{212}}$:	$Init$	$\longrightarrow$	$A_2, \{E_2\}_P$	
$\mathbf{p_{222}}$:	$x_8, \{x_9, x_{10}\}_{E_2}$	$\longrightarrow$	$\{x_{10}, S_{22}\}_{x_9}$	
$\mathbf{p_{232}}$:	$x_8, \{x_{10}, H(x_{10}, S_{22})\}_{F(x_{10}, S_{22})}$	$\longrightarrow$	$End,$	$\mathbf{Alg_{22} = F(x_{10}, S_{22})}$
$\mathbf{p_{112}}$:	$x_{11}, \{x_{12}\}_P$	$\longrightarrow$	$A_1, \{R_2, S_{12}\}_{x_{12}}$	
$\mathbf{p_{122}}$:	$\{x_{13}, x_{14}\}_{R_2}$	$\longrightarrow$	$A_1, \{S_{12}, H(x_{13}, x_{14})\}_{F(x_{13}, x_{14})},$	$\mathbf{Alg_{12} = F(x_{13}, x_{14})}$

The set of steps $\mathcal{P} = \{p_{111}, p_{121}, p_{131}, p_{211}, p_{221}, p_{212}, p_{222}, p_{232}, p_{112}, p_{122}\}$ is ordered by the partial order $<_{\mathcal{P}}$: $p_{111} <_{\mathcal{P}} p_{121} <_{\mathcal{P}} p_{131},\ p_{211} <_{\mathcal{P}} p_{221},\ p_{212} <_{\mathcal{P}} p_{222} <_{\mathcal{P}} p_{232}$and $p_{112} <_{\mathcal{P}} p_{122}$.

4 The Services driven Model for Group Protocols

The services driven model presented in [4] permits to model contributed protocols, to study their characteristics and security properties. This model has been applied on several protocols, such as A-GDH.2, SA-GDH.2, Asokan-Ginzboorg and Bresson-Chevassaut-Essiari-Pointcheval and has permitted to pinpoint several existing attack types on them. A group protocol is modelled by three components $<\mathcal{A}, \mathcal{K}, \mathcal{S}>$, where

– $\mathcal{A}$: set of agents, members of the group,
– $\mathcal{K}$: set of members knowledge,
– $\mathcal{S}$: set of services. A *service* denotes the *contribution* of a participant to the generation of the group key. The contribution of an agent a_i to an agent a_j is all information (message) generated by a_i and necessary for a_j to deduce the group key.

Let $i \in \mathbb{N}$, each $\mathbf{a_i} \in \mathcal{A}$ (*i-th* participant) is linked with other sets defined as follows:

– $\mathbf{S_i} \subseteq \mathcal{S}$ is the *minimal* set of services necessary to a_i in order to generate the group key;
– $\mathcal{K}_\mathbf{i} \subseteq \mathcal{K}$ is the *minimal* set of *private* knowledge of a_i, useful for generating services and the group key; it includes the initial private knowledge and the information generated during the protocol's execution.
– $\mathcal{K}_\mathbf{ij} \subseteq \mathcal{K}$ is the set of knowledge *shared* between agents a_i and a_j; it denotes the *minimal* set of shared knowledge that is useful for generating the group key; this information is given by the protocol's specification. Note that $\mathcal{K}_{ij} = \mathcal{K}_{ji}$.

In addition to the services used for the group key generation, other subsets of services representing the services provided by an agent are defined. Thus, $\mathbf{S_{a_i}}$ denotes the subset of services to which a_i has contributed (directly or not) by providing a private information.
$S_{a_i} = \{s \in \mathcal{S} \mid \exists t \text{ subterm of } s, \text{ such as } t \in \mathcal{K}_i\}$
This system permits to formally define security properties related to group protocols. Some of them are strongly linked to the time evolution of the group, such as the independence of group keys, the forward secrecy or the backward secrecy. Moreover, other properties are time independent, like the implicit authentication, the secrecy, the confirmation or the integrity. The modelling of these security properties is based on the interaction of subsets defined above. For instance, the security property to be verified for the Asokan-Ginzboorg protocol is the key agreement property. It says that for the same session, the group members have to deduce the same group key. This property is specified in our model by: $\forall a_i, a_j \in \mathcal{A}\ with\ i \neq j,\ Alg_i = Alg_j$.

For our example with two participants (A_1 and A_2), the group key agreement is violated when: $Alg_{12} \neq Alg_{22}$, that is $F(x_{10}, S_{22}) \neq F(x_{13}, x_{14})$, or $Alg_{11} \neq Alg_{21}$, that is $F(x_3, S_{11}) \neq F(x_6, x_7)$. Therefore, the violation of the group key agreement property can be specified by the following constraints: $x_{10} \neq x_{13}$ or $x_{14} \neq S_{22}$ or $x_6 \neq x_3$ or $x_7 \neq S_{11}$.
For more details about the modelling and the verification of other properties, the reader must refer to the model presented in [4]. In the present paper we show how this model can be used to search for attacks in group protocols.

5 Searching for Attacks in Group Protocols

We present in this section the algorithm of searching for attacks. It is described as follows:

```
**Algorithm** *AttackSearch*(ppty,instance)
  execCorrect = True
  Exec = $\{(\emptyset, \{\mathcal{R}_p \to \mathcal{S}_p\}_{p \in \mathcal{P}}, S_0, \emptyset)\}$
  $SC_P$ = *ConstraintsPpty*(ppty,instance)
  **While** execCorrect = True **and** Exec $\neq \emptyset$ **Do**
    choose (PT,PTT,S,SC)$\in$ Exec
    canCompose = True
    **While** canCompose = True **and** $PTT \neq \emptyset$ **do**
      choose $p$ minimal such as $\mathcal{R}_p \to \mathcal{S}_p \in PTT$
      **If** *Compose*($\mathcal{R}_p$,S) **then**
        *Treat*($\mathcal{R}_p \to \mathcal{S}_p$,S,SC)
        *Take* $\mathcal{R}_p \to \mathcal{S}_p$ from PTT
        *Add* $\mathcal{R}_p \to \mathcal{S}_p$ to PT
      **else**
        canCompose = False
      EndIf
    End
    If canCompose = True Then
      If $Attack$(SC,$SC_P$) Then
        execCorrect = False
      EndIf
    EndIf
  End
```

The algorithm takes as parameters the instance of the protocol and the property to verify.

The first step of the procedure of searching for attacks consists in the generation of the constraints set (SC_P) related to the violation of the security property given in parameter as 'ppty'. This is done in the algorithm by the function *ConstraintsPpty*. With this intention, we follow the steps described below:

- the services driven modelling of the protocol's instance (parameter 'instance');
- the services driven modelling of the violation of the property (parameter 'ppty');
- the deduction of the set of constraints related to the violation of the security property.

The constraints set generated constitutes the first level of the constraints tree (to be explained in Section 6).

The idea behind the algorithm is to consider all the possible executions of the protocol in order to find one execution corresponding to an attack. An execution is defined by the quadruple (PT,PTT,S,CS) where,

- PT: the set of steps of the execution belonging to $\{\mathcal{R}_p \to \mathcal{S}_p\}_{p\in\mathcal{P}}$ which are already treated. They are messages already exchanged between honest participants and the Intruder. At the beginning of the procedure, this set is empty.
- PTT: the set of steps of the execution belonging to $\{\mathcal{R}_p \to \mathcal{S}_p\}_{p\in\mathcal{P}}$ which have not been treated yet. This set is provided with a total order to say that a step must be treated before an other and thus a message must be exchanged before an other. Initially, this set contains all the steps given as parameter to the procedure *AttackSearch*.
- S: the set of the Intruder knowledge after the last treated step. Initially, this set is equal to the set S_0 of the instance given as parameter to the procedure *AttackSearch*.
- CS: the constraints set of the protocol. At the beginning of the procedure, this set is empty.

Consider an execution of the protocol among the (finite) set of possible executions. An execution corresponds to an attack if the constraints generated throughout all the execution's steps (denoted SC) are coherent with the constraints of the violation of the secutity property. This test of coherence can be done at the end of the execution. In fact, for each step of the execution, we consider the rule $\mathcal{R}_p \to \mathcal{S}_p$. The aim of the Intruder is to compose from his current knowledge a term corresponding to the pattern of the term $\mathcal{R}_p$. In the algorithm, this is tested by the function *Compose*. This function permits to test if the Intruder can compose the message $\mathcal{R}_p$ from his current knowledge. We note that this function uses only the Intruder composition rules since we assume that the set of the Intruder knowledge S contains only terms that cannot be decomposed yet. This hypothesis is maintained thanks to the Intruder knowledge management (to be explained in Section 7). The function *Compose* returns a boolean result. If the result is negative, then the Intruder fails in composing such a message, and so, he cannot go on in the execution. Thus, this execution cannot lead to an attack. In this case, we consider another execution.

If the result is positive, then we have to treat the current step $\mathcal{R}_p \to \mathcal{S}_p$ of the instance. This is the role of the function *Treat*. The matching between composed messages and the message

expected leads to the construction of constraints joining informations (constants or variables) given in the message expected to the ones in the composed messages. We note that, in the case where the Intruder can compose several messages matching with the expected message, we obtain several alternatives and thus several choices of constraints (presented as a disjunction). These constraints are added to the set of the protocol constraints SC.
Behind this addition, there is an important point that we must focus on: the management of constraints, especially when there is a huge constraints' set to be added only in one step of an instance. This will be discussed in Section 6.

Once the messages are composed, the Intruder gets new knowledge that he will use in the next instance's steps. Therefore, the Intruder's knowledge must be updated for each acquisition of new information. Thus, we have to manage the Intruder knowledge set S for each step of a protocol instance. Then, we add to this set S the new information acquired while taking into account the definition of S as the set where we cannot apply the Intruder decomposition rules to its terms yet. This notion will be more studied in Section 7.

At the end of the execution, the two sets of constraints SC and SC_p are solved in order to get a solution that relates variables of the execution's steps with constants or with each other. This is the aim of the function *Attack*. It tests if the two constraints' sets are coherent. The two sets are coherent if and only if there exists at least one path in the constraints tree that contains only coherent constraints when the execution finishes (see Section 6).
If the two sets are coherent then the tested security property is violated for the concerned execution. In this case, the function permits to solve the constraints based on the union of the two sets SC and SC_P. While instantiating variables in the execution in question with the solution found, we obtain the execution's trace of the attack.

This method is efficient since the search for flaws is static as it corresponds to a resolution of two constraints systems. Nevertheless, for an execution step, generating the constraints matching all possible composed messages to the expected message can lead to a huge set of constraints. In order to minimize this set, we propose two kinds of suggestions:

- In the first one, for an execution step, we consider only constraints relying on variables used in the property's violation constraints (we note V this set of variables). For the other variables, we just save the information that the message must be composed from a certain set of knowledge: the **current** Intruder knowledge.
- The second proposition is based on the combination between the construction of the constraints set of the protocol SC and the test of coherence of the two sets of constraints SC et SC_P. Indeed, while generating the constraints of the protocol related to one step, we test the coherence of this subset of constraints with all the constraints built before. This permits to eliminate unnecessary constraints. This can be done by the use of the constraints tree (see Section 6).

6 Constraints Management

Our method for searching for flaws is based on the constraints' solving. The first part of constraints comes from the modelling of the violation of the security property to be tested in the services driven model (see Section 4). The second part is generated and updated at each step treated among the different steps composing the protocol's instance. It is to be noticed that these steps are totally ordered in an execution.
Since the aim of our procedure is to search for an eventual attack, it's goal is to find an execution that corresponds to two coherent sets of constraints SC and SC_P. Knowing the set SC_P, in order

to minimize the number of constraints added in an execution's step, we just add the necessary constraints that are coherent with the ones of the previous steps. To do this, we propose to associate the execution tree to a constraints tree. The idea behind the constraints tree is to allow only the addition of the necessary constraints to the set SC and thus to consider only constraints that are coherent with the ones of the previous steps.

The constraints tree is initially constructed from the constraints related to the violation of the security property to be tested. These constraints represent the first level of the tree. We note V the set of variables given in the current constraints of the protocol. This set is initiated to the variables manipulated in the first level of the constraints tree. Besides, as constraints of the first level can explain different choices to violate the property to be verified, the level concerned (the first) is composed of different states representing these alternatives. For each of these states, we consider the possible executions with the intention to find one corresponding to an attack.

Moreover, the tree has to be updated for each step $\mathcal{R}_p \rightarrow \mathcal{S}_p$ to be treated. We assume that we are at the level i of the constraints tree. For each state of the level i, we focus on constraints corresponding to the current step and coherent with constraints related to the current state of the level i. Once the Intruder is able to compose message(s) looking like the message $\mathcal{R}_p$, we can generate different alternatives for the constraints matching this (these) message(s) to the message $\mathcal{R}_p$ expected by the honest participant. Since these constraints can be different alternatives to form the message expected, they can be eventually represented by different states at the level $i+1$. Let us note the set of the constraints representing an eventual state of the level $i+1$ as *scc*. While updating the constraints tree, we have to distinguish different cases:

- Constraints of *scc* do not contain variables that already exist in a previous level relating the root to the direct parent. In this case, we may just save the information that this variable (the message in general) has to be generated from a certain set of knowledge (the current set of the Intruder knowledge). This information would be useful whenever another lower level use the same variable.
- Constraints of *scc* contain variables that already exists in a previous level relating the root to the direct parent. In this case, there are two possibilities:
  - Constraints of *scc* are incoherent with those of parents (constraints that exist between the direct parent (current state of the level i) and the root (alternative of the violation of the security property)). In this case, we do not add this state to the $(i+1)$-th level.
  - Constraints of *scc* are coherent with these of upper parents. In this case, we maintain these constraints *scc* as a possible son of the state at the level $i+1$.
    Besides, if the constrains of this possible state contains one constraint that already exists in previous states or may be deduced by transition, we may omit it in order to get rid of redundancy.
    Moreover, if a constraint of *scc* manage variables that already exist in V (already have values) and relating them to variables that are not yet in V, this constraint is replaced by a new one connecting the new variable to its value (by transition).

7 Intruder Knowledge Management

In our method for flaws detection we assume that our Intruder follows the most referred Intruder's model: the Dolev-Yao's model [7]. In this model, the Intruder has the entire control of the communication network. That is to say that the Intruder can intercept, record, modify, compose, send, encrypt and decrypt (if he has the appropriate key) each message. He has also the possibility to send faked messages in the name of another participant. Since he has such capabilities, the Intruder has to manage information he acquires from each step. We note the set of his knowledge

S. This set is defined as the set of all present knowledge in its maximality decomposed form. From the beginning of a protocol's execution, the Intruder knows some information. Terms composing these information constitute the set of his initial knowledge S_0. Then, initially, $S = S_0$. Throughout an execution, the set of the Intruder's knowledge has to be updated for each step treated. We distinguish two kinds of updates: when the Intruder has to compose a message(s) having the same pattern as the message expected by an honest participant, and when he gets some new information (as response from an honest participant).

Operation	**Composition Rules**	**Decomposition Rules**
Fresh	$\frac{}{k}$, $k \notin \mathcal{K} \cup \mathcal{A} \cup \mathcal{S}$	
Concatenation	$\frac{m1 \quad m2}{< m1, m2 >}$	$\frac{< m1, m2 >}{m1}$, $\frac{< m1, m2 >}{m2}$
Asymmetric Encryption	$\frac{m \quad k}{\{m\}_k^p}$	$\frac{\{m\}_k^p \quad inv(k)}{m}$
Symmetric Encryption	$\frac{m \quad b}{\{m\}_b^s}$	$\frac{\{m\}_b^s \quad b}{m}$
Product	$\frac{x \quad y}{x.y}$	$\frac{x.y \quad y^{-1}}{x}$
Inverse	$\frac{y}{y^{-1}}$	$\frac{y^{-1}}{y = \{y^{-1}\}^{-1}}$
Exponentiation	$\frac{t \quad \alpha}{\alpha^t}$	$\frac{\alpha^{x.y} \quad y^{-1}}{\alpha^x}$

Table 1. Rules of the Intruder's terms composition and decomposition

In the first case, we are treating the step $\mathcal{R}_p \rightarrow \mathcal{S}_p$ where the Intruder has to build a message(s) suiting the pattern of the message expected by an honest participant ($\mathcal{R}_p$). To do this, the Intruder follows composition rules defined in the first part of the Table 1.
In the second case, since all the messages sent by the participants acting in the protocol are sent to the Intruder, the last one has the possibility to decompose the message received by using terms already existing in S at this moment. This set S can also contain terms that have not yet be decomposed because some information was missing. Therefore, from information deduced from the last message received, the Intruder can decompose terms in S. In order to decompose terms, the Intruder follows decomposition rules defined in the second part of the Table 1.
We return now to our running example. In this paragraph, we consider the following execution's order ($<_e$): $p_{111} <_e p_{212} <_e p_{211} <_e p_{112} <_e p_{121} <_e p_{222} <_e p_{221} <_e p_{122} <_e p_{131} <_e p_{232}$
The aim of the Intruder is to build the patterns of the messages expected by the honest participants. We notice that to fulfill this goal, we need to manage the Intruder's knowledge whenever he gets a new information (that comes from a message received). For each message expected (taking into account the execution's order), the Intruder build every message that looks like the pattern of the message in question. By application of the Flaws Detection method to our example of the Asokan-Ginzboorg protocol, we find the constraints expressed in Table 2.

While solving the constraints system listed in Table 2 with the one of Section 4 (the one of the security property to verify) we find this solution: $x_{10} = x_6 = S_{21}$, $x_2 = R_2$, $x_7 = S_{22}$, $x_{12} = E_1$, $x_{13} = x_3 = S_{12}$, $x_5 = E_2$, $x_9 = R_1$and $x_{14} = S_{11}$.
The instantiation of variables in the execution by the values found above gives us the execution's trace of Figure 1.

At the end of this execution's trace, we have: $Alg_{11} \neq Alg_{21}$ and $Alg_{12} \neq Alg_{22}$. Thus, the group key agrement is violated for each one of the two sessions.

1	$x_5 = E_1$ *or* $x_5 = E_2$
2	$x_{12} = E_1$ *or* $x_{12} = E_2$
3	$x_2 = R_1, x_3 = S_{21}, E_1 = x_5$ *or* $x_2 = R_2, x_3 = S_{12}, E_1 = x_{12}$
4	$x_9 = R_1, x_{10} = S_{21}, E_2 = x_5$ *or* $x_9 = R_2, x_{10} = S_{12}, E_2 = x_{12}$
5	$x_6 = x_3, x_7 = S_{11}, x_2 = R_1$ *or* $x_6 = x_{10}, x_7 = S_{22}, x_9 = R_1$
6	$x_{13} = x_3, x_{14} = S_{11}, x_2 = R_2$ *or* $x_{13} = x_{10}, x_{14} = S_{22}, x_9 = R_2$
7	$x_3 = S_{21}, x_6 = S_{21}, x_7 = S_{11}$ *or* $x_3 = x_{13}, x_3 = S_{12}, x_{14} = S_{11}$
8	$x_{10} = S_{21}, x_6 = S_{21}, x_7 = S_{22}$ *or* $x_{10} = S_{12}, x_{13} = S_{12}, x_{14} = S_{22}$

Table 2. Constraints for the Asokan-Ginzboorg protocol

$$
\begin{array}{lll}
\mathbf{p_{111}}: & i \longrightarrow A_1 : & Init \\
 & A_1 \longrightarrow i & A_1, \{E_1\}_P \\
\mathbf{p_{212}}: & i \longrightarrow A_2 : & Init \\
 & A_2 \longrightarrow i & A_2, \{E_2\}_P \\
\mathbf{p_{211}}: & i \longrightarrow A_2 : & x_4, \{E_2\}_P \\
 & A_2 \longrightarrow i & A_2, \{R_1, S_{21}\}_{E_2} \\
\mathbf{p_{112}}: & i \longrightarrow A_1 : & x_{11}, \{E_1\}_P \\
 & A_1 \longrightarrow i & A_1, \{R_2, S_{12}\}_{E_1} \\
\mathbf{p_{121}}: & i \longrightarrow A_1 : & x_1, \{R_2, S_{12}\}_{E_1} \\
 & A_1 \longrightarrow i & \{S_{12}, S_{11}\}_{R_2} \\
\mathbf{p_{222}}: & i \longrightarrow A_2 : & x_8, \{R_1, S_{21}\}_{E_2} \\
 & A_2 \longrightarrow i & \{S_{21}, S_{22}\}_{R_1} \\
\mathbf{p_{121}}: & i \longrightarrow A_2 : & \{S_{21}, S_{22}\}_{R_1} \\
 & A_2 \longrightarrow i & A_2, \{S_{21}, H(S_{21}, S_{22})\}_{F(S_{21}, S_{22})},\ \mathbf{Alg_{21} = F(S_{21}, S_{22})} \\
\mathbf{p_{122}}: & i \longrightarrow A_1 : & \{S_{12}, S_{11}\}_{R_2} \\
 & A_1 \longrightarrow i & A_1, \{S_{12}, H(S_{12}, S_{11})\}_{F(S_{12}, S_{11})},\ \mathbf{Alg_{22} = F(S_{12}, S_{11})} \\
\mathbf{p_{131}}: & i \longrightarrow A_1 : & x_1, \{S_{12}, H(S_{12}, S_{11})\}_{F(S_{12}, S_{11})},\ \mathbf{Alg_{11} = F(S_{12}, S_{11})} \\
\mathbf{p_{232}}: & i \longrightarrow A_2 : & x_8, \{S_{21}, H(S_{21}, S_{22})\}_{F(S_{21}, S_{22})},\ \mathbf{Alg_{12} = F(S_{21}, S_{22})}
\end{array}
$$

Fig. 1. Execution's trace

8 Verification Results

By applying the strategy described in previous sections, we have found two authentication attacks for the protocol GDH.2 with 4 participants (attacking resp. participant A_3 and A_4) (See Figures 2 and 3). These Figures show either the normal execution of the protocol and the message to be changed in order to lead to an attack. In this section, we generalize these results to the GDH.2 with n participants and to the protocol A-GDH.2.

8.1 The GDH.2 protocol

Consider the GDH.2 [9] protocol with n participants $(A_1, ..., A_n)$. The Intruder can have the point of vue of the group key of the last member. Indeed, he intercepts the last message intended for the last participant $(x_1, ..., x_{n-1}, x_n)$ and alters the last component (x_n) by replacing it by any component of the last message varying from x_1 to x_{n-1}. When receiving the message expected, the last participant A_n exponentiates the components $x_1 ... x_{n-1}$ by R_n and send them to the other participants. The Intruder can then get the information varying from $Exp(x_1, R_n)$ to $Exp(x_{n-1}, R_n)$.

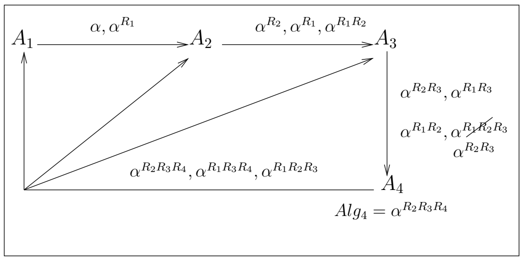

A_1
α, α^{R_1}
A_2
$\alpha^{R_2}, \alpha^{R_1}, \alpha^{R_1R_2}$
A_3
$\alpha^{R_2R_3}, \alpha^{R_1R_3}$
$\alpha^{R_1R_2}, \alpha^{\cancel{R_1}R_2R_3}$
$\alpha^{R_2R_3}$
$\alpha^{R_2R_3R_4}, \alpha^{R_1R_3R_4}, \alpha^{R_1R_2R_3}$
A_4
$Alg_4 = \alpha^{R_2R_3R_4}$

Fig. 2. First authentication attack for GDH.2

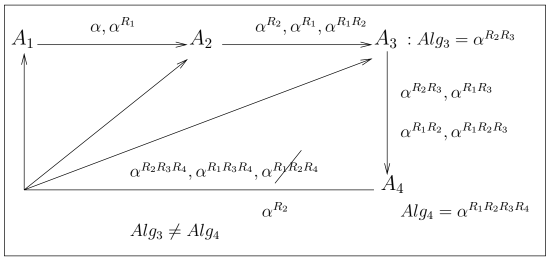

A_1
α, α^{R_1}
A_2
$\alpha^{R_2}, \alpha^{R_1}, \alpha^{R_1R_2}$
A_3
$: Alg_3 = \alpha^{R_2R_3}$
$\alpha^{R_2R_3}, \alpha^{R_1R_3}$
$\alpha^{R_1R_2}, \alpha^{R_1R_2R_3}$
$\alpha^{R_2R_3R_4}, \alpha^{R_1R_3R_4}, \alpha^{\cancel{R_1}R_2R_4}$
A_4
α^{R_2}
$Alg_4 = \alpha^{R_1R_2R_3R_4}$
$Alg_3 \neq Alg_4$

Fig. 3. Second authentication attack for GDH.2

Then, A_n uses the last component x_n in order to deduce the group key by exponentiating this by R_n. This key from the point of vue of the last member is then $Alg_n = Exp(x_n, R_n)$. Thus, if the Intruder replaces x_n by a message x_i from x_1 to x_{n-1}, the last participant deduces as group key $Exp(x_i, R_n)$. Nevertheless, this information is already available on the network and then known by the intruder.

The aim of the Intruder is now to have the point of vue of the group key of an intermediate participant A_i varying from A_1 to A_{n-1}. A_i deduces his key from the last message he received (X) from A_n where $X = x_1, .., x_{n-1}$. Since his group key will contain a private information: R_i, the Intruder has to make so that A_i generates for key of group a message which was transmitted before and which contains private information R_i. These messages are accessible to the intruder at the time of the first round of the protocol: when the members (in particular, intermediate members) are invited to give their contributions by exponentiating the messages received by the R_i. At the step p_i, A_i receives a message of form $x_{11}, .., x_{1i}$. In the message to be sent by A_i during this step, we find all the components x_{1j} (varying from x_{11} to x_{1i}) exponentiated by R_i. We assume now that, for an $i \in \{1, n-1\}$, for the message $X = x_1, .., x_i, ..., x_{n-1}$, x_i is one of the components $x_{11}, .., x_{1i}$. The participant A_i, while receiving X, takes his correspondent component x_i and generates his key by exponentiating it by R_i. Thus, $Alg_i = Exp(x_i, R_i)$. Nevertheless, the Intruder got already the information $Exp(x_i, R_i)$ from the step p_i.

8.2 The A-GDH.2 protocol

Consider the A-GDH.2 [9] protocol with n participants $(A_1, ..., A_n)$ where the Intruder is one of the participants and has as raw i $(I = A_i)$. The Intruder focus on the last message expected by the last participant. This message X is composed of n components $X = x_1...x_n$. The last participant exponentiates the (n-1) components of this message by R_n and the key of the corespondent participant. Thus, the component x_i is exponentiated by R_nK_{ni}. Instead of sending the message $X = x_1, ..., x_i, ..., x_n$ to A_n, the Intruder send the message $X = x_1, ..., x_i, ..., x_i$. As response to this message, A_n sends the message
X' = $Exp(Exp(x_1, R_n), K_{n1}), ..., Exp(Exp(x_i, R_n), K_{ni}), ..., Exp(Exp(x_{n-1}, R_n), K_{n(n-1)})$
and deduces as group key $Exp(x_i, R_n)$ as $x_n = x_i$. From the message X', the Intruder gets the component $Exp(Exp(x_i, R_n)K_{ni})$. He knows then $Exp(x_i, R_n) = Exp(x_n, R_n)$. Thus, he has the point of vue of the group key of the participant A_n. Moreover, for the other group members, as the Intruder does not alter the rest of the normal protocol's execution, he shares the same expected group key with the rest of the group (apart from A_n) $Exp(Exp(x_1, R_n), R_1) = Exp(Exp(x_i, R_n), R_i) = Exp(Exp(x_{n-1}, R_n), R_{n-1})$. The Intruder succeed then to divide the group on two parts and has the two points of view of the group key of the two parties.

9 Related Work

The verification of group protocols is a research topic on which there has been and there is still a lot of work. This is mainly due to the wide range of specific requirements imposed by this kind of protocols. Varying from an unbounded number of participants to very particular security properties, considering all those requirements is a real challenge, both theoretical and practical. Either modelling and verifying group protocols and their properties are very difficult.

It exists various research activities to formally specify group protocols and their specific requirements. For instance, Capsl has been extended to MuCapsl [6]. This language is to be translated to a multiset term rewriting rules (MuCIL) which is an extension of CIL in order to support multicast group management protocols. However, we only model the security property of secrecy.
In a recent work [5], Delicata and Schneider present a framework for reasoning about secrecy in a class of Diffie-Hellman protocols. The technique, which shares a conceptual origin with the idea of a rank function, uses the notion of a message-template to determine whether a given value is generable by an intruder in a protocol model. This work focus only on the security protocol of secrecy. Then, it is less general than Pereira's work as it deals with a sub class of Diffie-Hellman protocols (it claims the condition of I/O independence).

In the verification process, after the first step: the specification of either the protocol and the property, there is a more delicate step which is the verification step. It exists various research activities oriented on this task varying from manual methods to automatic ones. They lead to the discovery of several attacks that will be introduced in this section.

One of the most interesting techniques done by hand is suggested by Pereira and Quisquater in [9]. They have introduced a method converting the problem of ownership of some information by the intruder to a problem of resolution of a system of linear equations. With this method, several attacks have been found in the protocols suite CLIQUES [9]. This method has also permitted to get a generic result: it is impossible to design an authentication group key agreement protocol built on A-GDH for a number of participants greater than or equal to four [10]. Although this method is of great interest for analysing group protocols, its main drawback is that it has to be run by hand for discovering attacks.

Additionally, some tools have been extended in order to deal with the new requirements of group protocols. Significant attacks on such protocols have been found. In [12], Taghdiri and Jackson have modelled a multicast group key management protocol proposed by Tanaka and Sato [13]. They have been able to discover counterexamples to supposed properties. They have then proposed an improved protocol. However, in their model, no active attacker was included. Their improved protocol has been analyzed in [11] by CORAL and two serious attacks have been found. CORAL has also been used to discover other attacks concerning two protocols: Asokan-Ginzboorg and Iolus.

10 Summary and future work

Throughout this paper, we have presented a new strategy for dealing with group protocols and more generally contributed ones. The approach hinges around the use of the services driven model to deduce constraints related to the security property to verify. These constraints will be used with the protocol's execution constraints to obtain an attack execution's trace if it exists. This strategy permits to pinpoint new attacks in three different protocols. From these attacks, we have generalized the result to two protocols with n participants.

This work is nascent, but we are currently applying it to other protocols and to other security properties. We also plan to study the complexity of the suggested algorithm.

Since the analysis of a great number of protocols is generally done by automatic tools, we intend either to implement our strategy or to extend existing automatic tools that are based on constraints solving. Among these tools, we find Atse, one of four back-ends used in AVISPA [1], a tool that has already treated a large number of Internet security protocols. Its expressive protocol specification language permits, modulo some extensions, to model contributed protocols and their intended security properties. Since our basic constraints are based on equality and inequality constraints, they may be seen as booleen constraints and then the whole constraints can be considered as SAT constraints. Thus, we may integrate a SAT-solver in our solution.
The suggested approach can also be developed to consider another kind of group protocols such as hierarchical protocols that present additional verification constraints. Indeed, we have to extend the services driven model to deal with this kind of protocols.

References

1. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV'2005*, volume 3576 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, 2005. Springer.
2. N. Asokan and P. Ginzboorg. Key Agreement in ad hoc Networks. *Computer Communications*, 23(17):1627–1637, 2000.
3. Y. Chevalier and L. Vigneron. Strategy for Verifying Security Protocols with Unbounded Message Size. *Journal of Automated Software Engineering*, 11(2):141–166, 4 2004.
4. N. Chridi and L. Vigneron. Modélisation des propriétés de sécurité de protocoles de groupe. In *Actes du 1 er Colloque sur les Risques et la Sécurité d'Internet et des Systèmes*, pages 119–132, Bourges, France, October 2005. CRISIS.
5. R. Delicata and S. Schneider. A formal approach for reasoning about a class of diffie-hellman protocols. In *Formal Aspects in Security and Trust*, pages 34–46, 2005.
6. G. Denker and J. Millen. Modeling group communication protocols using multiset term rewriting, 2002.
7. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.
8. H. Hassan, A. Bouabdallah, H. Bettahar, and Y. Challal. Hi-kd: Hash-based hierarchical key distribution for group communication - ieee infocom poster, 2005.
9. O. Pereira. *Modelling and Security Analysis of Authenticated Group Key Agreement Protocols.* PhD thesis, Universit catholique de Louvain, May 2003.
10. O. Pereira and J.-J. Quisquater. Generic Insecurity of Cliques-Type Authenticated Group Key Agreement Protocols. In *17th IEEE Computer Security Foundation Workshop, CSFW*, pages 16–19, Pacific Grove, CA, 2004. IEEE Computer Society.
11. G. Steel and A. Bundy. Attacking group multicast key management protocols using coral. *Electr. Notes Theor. Comput. Sci.*, 125(1):125–144, 2005.
12. M. Taghdiri and D. Jackson. A Lightweight Formal Analysis of a Multicast Key Management Scheme. In *Formal Techniques for Networked and Distributed Systems, FORTE*, volume 2767, pages 240–256, Berlin, Germany, 2003. Springer.
13. S. Tanaka and F. Sato. A Key Distribution and Rekeying Framework with Totally Ordered Multicast Protocols. In *15thon Information Networking, ICOIN*, pages 831–838, Beppu City, Japan, 2001. IEEE Computer Society.
14. C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 68–79, 1998.