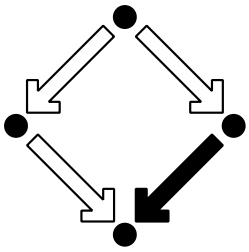
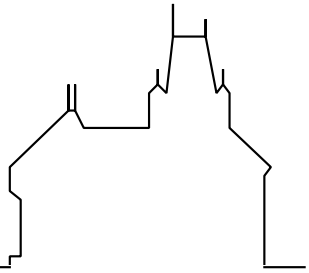


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



SCSS 2010
Symbolic Computation in Software Science

Tudor JEBELEAN, Mohamed MOSBAH
Nikolaj POPOV (eds.)

Hagenberg, Austria
July 29–30, 2010

RISC-Linz Report Series No. 10-10

Supported by:

SCSS 2010

Symbolic Computation in Software Science

3rd International Workshop

Hagenberg, July 29-30, 2010

Steering Committee

- Bruno Buchberger (Johannes Kepler University of Linz, Austria)
- Tetsuo Ida(University of Tsukuba, Japan)
- Mohamed Mosbah (University of Bordeaux, France)
- Tudor Jebelean (Johannes Kepler University of Linz, Austria)
- Nikolaj Popov (Johannes Kepler University of Linz, Austria)

Program Committee

- Adel Bouhoula (University of November 7th at Carthage, Tunisia)
- Bruno Buchberger (Johannes Kepler University of Linz, Austria)
- Adrian Craciun (IeAT, West University of Timisoara, Romania)
- Martin Giese (University of Oslo, Norway)
- Hoon Hong (North Carolina State University, USA)
- Tetsuo Ida (University of Tsukuba, Japan)
- Atsushi Igarashi (Kyoto University, Japan)
- *Tudor Jebelean (Johannes Kepler University of Linz, Austria) co-chair*
- Laura Kovacs (Technical University of Vienna, Austria)
- Yukiyoshi Kameyama (University of Tsukuba, Japan)
- Temur Kutsia (Johannes Kepler University of Linz, Austria)
- Alexander Letychevsky (Glushkov Institute of Cybernetics, Kyiv, Ukraine)
- Aart Middeldorp (University of Innsbruck, Austria)
- Yasuhiko Minamide (University of Tsukuba, Japan)
- *Mohamed Mosbah (University of Bordeaux, France) co-chair*
- Florina Piroi (Johannes Kepler University of Linz, Austria)
- Nikolaj Popov (Johannes Kepler University of Linz, Austria)
- Michael Rusinowitch (INRIA Lorraine, France)
- Masahiko Sato (Kyoto University, Japan)
- Wolfgang Schreiner (Johannes Kepler University of Linz, Austria)
- Yahya Slimani (University El Manar Tunis, Tunisia)

Local Organization Chair

- Nikolaj Popov (Johannes Kepler University of Linz, Austria)

Thursday, July 29

09:30-10:30 **Invited Talk.** Vampire: Past, Present and Future.
Andrei Voronkov

10:30-11:00 Coffee break.

12:30-14:00 Lunch at Hofwirt restaurant.

15:30-16:00 Coffee break.

18:00-21:00 Conference Dinner at Ars Electronica, Linz

Friday, July 30

08:30-09:30 **Invited Talk**. Challenges in Formalizing Geometric Reasoning:
A Case Study of Pick's Theorem.
John Harrison.

09:30-10:00 *Coffee break.*

10:00-10:30 Simple Non-Deterministic Strategy in Rewriting and Its Application for Verification.
Alexander Letichevsky, Alexander Letichevsky Jr.,
Vladimir Peschanenko. …….. 96

10:30-11:00 Multi-Domain Logic as a Tool for Program Verification.
Gabor Kusper and Tudor Jebelean. …….. 105

11:00-11:30 *Coffee break.*

11:30-12:30 **Invited Talk**. Symbolic Verification for Scientific Discovery.
Wei Li.

12:30-14:00 *Lunch at Hofwirt restaurant.*

14:00-14:30 Predicate transformers and system verification.
Oleksandr Letychevskyi, Alexander Letichevsky…… 118

14:30-15:00 The RISC ProgramExplorer: Reasoning about Programs as State Relations.
Wolfgang Schreiner…….. 131

15:00-15:30 Extended Web Services for Computational Origami.
Asem Kasem, Tetsuo Ida…….…….…. 144

15:30-15:40 Closing Remarks.

15:40-16:00 *Coffee break.*

16:00-17:00 Business Meeting.

A Formal Proof of Confluence for an Infinitely Generated Noncommutative Polynomial Ideal Parametrized over Integro-Differential Algebras

Loredana Tec*
Research Institute for Symbolic Computation, Johannes Kepler University,
Castle of Hagenberg, Austria 4032
ltec@risc.uni-linz.ac.at

1 Introduction

In this paper we outline a proof of confluence for an infinitely generated noncommutative polynomial ideal corresponding to rewrite rules for integro-differential operators. The notion of *integro-differential operator*—introduced in [21] as a generalization of the "Green's polynomials" of [20]—is an algebraic analogue of differential, integral and boundary operators in the context of linear ordinary differential equations (LODEs). As presented in [22], it is useful for modeling Green's operators as solutions for LODEs. Finding Green's functions—as canonical forms for the Green's operators—is a fundamental task in mathematics, science and engineering. Currently it is mostly done by pen-paper. It is very non-trivial, time-consuming and error-prone process. Hence, (semi)-automating the process is of utmost importance.

A rewrite system based method for automating the process was introduced in [20]. In developing a rewrite-system based program, a fundamental software science question is whether the program is *confluent*. If this is true, it gives the same output no matter what heuristics (choice of rewriting step) is used. In [20] a pen-paper proof of the (rewrite system) confluence was given. Proving confluence is usually a *non-trivial* task. Thus, (semi)-automating the process is also of fundamental importance. In this work, we show how to prove the confluence automatically via a Gröbner basis computation in the algebra of *integro-differential polynomials*. For this purpose, we use an *implementation* realized using the generic functor language of the THEOREMA system.

2 Integro-Differential Algebras

An *integro-differential algebra* $(\mathscr{F},\partial,\int)$ is defined as a differential algebra $(\mathscr{F},\partial)$ with a linear operation $\int\colon \mathscr{F}\to\mathscr{F}$ such that $\int$ is a section of ∂, i.e. $(\int f)' = f$, and the differential Baxter axiom $(\int f')(\int g') + \int (fg)' = (\int f')g) + f(\int g')$ holds.

2.1 Integro-Differential Operators

As mentioned in the Introduction, the integro-differential operators are useful for treating boundary problems for LODEs as they express both the problem statement (differential equation and boundary conditions) and its solution operator (an integral operator usually called "Green's operator"). Methods for solving and factoring boundary problems are described in [19, 22], both in a differential algebra and in an abstract setting.

The integro-differential operators are realized by a suitable *quotient* of noncommutative polynomials

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 1-4

*Recipient of a DOC-fForte-fellowship of the Austrian Academy of Sciences

$fg \to f \cdot g$	$\partial f \to \partial \cdot f + f\partial$	$\int f \int \to (\int \cdot f) \int - \int (\int \cdot f)$
$\phi\psi \to \psi$	$\partial\phi \to 0$	$\int f \partial \to f - \int (\partial \cdot f) - (\mathtt{E} \cdot f)\,\mathtt{E}$
$\phi f \to (\phi \cdot f)\,\phi$	$\partial \int \to 1$	$\int f \phi \to (\int \cdot f)\,\phi$

Table 1: Rewrite System for $\mathscr{F}[\partial, \int]$

over a given integro-differential algebra. They are built as an instance of the monoid algebra [3] for the word monoid over the infinite alphabet consisting of the letters ∂ and $\int$ along with all basis elements $x^n e^{\lambda x}$ ($n \in \mathbb{N}, \lambda \in \mathbb{C}$) of the exponential polynomials and all evaluations. Then the nine parametrized rewrite rules 1 introduced in [22] are factored out. These rules—modeling the operations of differentiation, integration and evaluation—form a noncommutative *Gröbner basis* in the underlying polynomial ring (see Section 4 for an automated proof).

Gröbner bases were introduced in [5, 6] and have became a crucial tool in computer algebra, specifically for solving several algebraic problems concerning ideals [9], see for instance [15, 16] for the commutative setting, and [4, 14, 18] for the noncommutative setting. There is a close relationship between Gröbner bases theory and theory of rewriting, where the analogue of *Buchberger's algorithm* is the *Knuth-Bendix procedure*, detailed in [2, 24]. The goal of the latter is to transform a set of equations (over terms) into a *noetherian* and *confluent* term rewriting system.

2.2 Integro-Differential Polynomials

The *integro-differential polynomials* over a given integro-differential algebra introduced in [22] form a commutative algebra. They model nonlinear differential and integral operators with an indeterminate u. Another interpretation is that they are polynomial functions (of x) involving an "unknown" function (namely u). A typical integro-differential polynomial is given by $2u\int u'^2 + \int (x^4 u u'' \int (x e^{3x} u^2 u'^3 \int u))$.

The algebra of integro-differential polynomials is realized as a special case of the general construction of polynomials in universal algebra, by adjoining one indeterminate function to an integro-differential algebra. For a comprehensive treatment of this notion of polynomials for arbitrary algebras we refer to [17]; useful surveys can be found in [1, 12]. For *computational purposes*, we have implemented a canonical simplifier for the induced congruence—identifying different expressions denoting the same integro-differential polynomial—thus solving the word problem in this variety.

3 The Theorema System

Theorema is a system designed as an integrated environment for doing mathematics, in particular proving, solving, and computing in various domains of mathematics [11]. Implemented on top of the computer algebra system Mathematica, its core language is higher-order predicate logic and contains a natural *programming language* such that algorithms can be coded and verified in a unified formal frame.

In this logic-internal programming language, *functors* are a powerful tool for building up hierarchical domains in mathematics in a modular and generic way that unites elegance and formal clarity. They were introduced and first implemented in Theorema by Bruno Buchberger. A short explanation of functors is given in [13]; for a general discussion of functor programming, see also [8, 23]. In order to achieve simultaneously genericity and efficiency, one can use the Theorema-Java compiler introduced in [25].

4 A Proof of Confluence

Our final goal is to provide a *complete automated proof* of noetherianity and confluence of the system of rules represented in Table 1 (see also Proposition 13 of [22]). We refer to [20] for a confluence proof for the so-called *analytic polynomials*. Equivalently, we show that the polynomials given by the difference between the left-hand and right-hand side of these rules form a *noncommutative Gröbner basis* in the underlying integro-differential algebra. Since every such rule is infinitely parametrized, the set of polynomials generates an *infinite noncommutative polynomial ideal*. Consequently, we need an algorithmic way to handle *noncommutative parametrized* polynomial reduction and S-polynomials. For this purpose we use a noncommutative adaption of reduction rings (rings with so-called *reduction multipliers* in the sense of [7, 10]). In order to prove that the set of these noncommutative polynomials represents a Gröbner basis for the infinite polynomial ideal generated by them, we must show that the corresponding S-polynomials reduce to 0. We first have to find a suitable domain where these S-polynomials can be computed.

Let us consider the rules $fg \to f \cdot g$ and $\partial f \to \partial \cdot f + f\partial$. Their corresponding S-polynomial is computed as $spol(fg - f\cdot g, \partial f - \partial \cdot f - f\partial) = \partial(fg) - (\partial f)g = (fg)\partial + (fg)' - f(\partial g) - f'g = (fg)' - f'g - fg'$. As we can notice in this example, we need to be able to compute in the algebra of integro-differential polynomials in two "unknown" functions. Using the generic functor approach, this is realized as an instance of the free vector space over a field K generated by the set of terms in an ordered monoid T. Then we extend this domain by introducing a particular type of multiplication, along with differentiation and integration. These operations require special attention, since the multiplication involves the shuffle product and the integral must be computed by a careful case distinction on the differential exponents. For the S-polynomial computations we use the functor of integro-differential operators over the algebra of integro-differential polynomials in two indeterminate functions producing a domain denoted here $\mathbb{G}$. The proof reduces to computations in a complicated domain where all the 72 parametrized S-polynomials are reduced to 0 and the corresponding Theorema command is given below.

With this computation, we conclude that the rules of Table 1 represent a Gröbner basis. Thus, equivalently, we have proved that the corresponding system is confluent.

References

[1] E. Aichinger and G. F. Pilz. A survey on polynomials and polynomial and compatible functions. In *Proceedings of the Third International Algebra Conference (Tainan, 2002)*, pages 1–16, Dordrecht, 2003. Kluwer Acad. Publ.

[2] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, Cambridge, 1998.

[3] T. Becker and V. Weispfenning. *Gröbner bases*, volume 141 of *Graduate Texts in Mathematics*. Springer, New York, 1993. A computational approach to commutative algebra, In cooperation with Heinz Kredel.

[4] G. M. Bergman. The diamond lemma for ring theory. *Advances in Mathematics*, 29(2):179–218, August 1978.

[5] B. Buchberger. *An algorithm for finding the bases elements of the residue class ring modulo a zero dimensional polynomial ideal (German)*. PhD thesis, Univ. of Innsbruck, 1965. English translation J. Symbolic Comput. **41**(3-4) (2006) 475–511.

[6] B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes Math.*, 4:374–383, 1970.

[7] B. Buchberger. A critical-pair/completion algorithm for finitely generated ideals in rings. In *Logic and machines: decision problems and complexity (Münster, 1983)*, volume 171 of *Lecture Notes in Comput. Sci.*, pages 137–161. Springer, Berlin, 1984.

[8] B. Buchberger. Mathematica as a rewrite language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Functional and Logic Programming (Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, November 1-4, 1996, Shonan Village Center)*, pages 1–13. Copyright: World Scientific, Singapore - New Jersey - London - Hong Kong, 1996.

[9] B. Buchberger. Introduction to Gröbner bases. In B. Buchberger and F. Winkler, editors, *Gröbner bases and applications*. Cambridge Univ. Press, 1998.

[10] B. Buchberger. Groebner rings and modules. In S. Maruster, B. Buchberger, V. Negru, and T. Jebelean, editors, *Proceedings of SYNASC 2001*, pages 22–25, 2001.

[11] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *J. Appl. Log.*, 4(4):359–652, 2006.

[12] B. Buchberger and R. Loos. Algebraic simplification. In B. Buchberger, G. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 11–43. Copyrigth: Springer Verlag, Vienna - New York, 1982.

[13] B. Buchberger, G. Regensburger, M. Rosenkranz, and L. Tec. General polynomial reduction with Theorema functors: Applications to integro-differential operators and polynomials. *ACM Commun. Comput. Algebra*, 42(3):135–137, 2008.

[14] G.-T. J. Bueso, José and A. Verschoren. *Algorithmic methods in non-commutative algebra*. Kluwer Acad. Publ., 2003.

[15] L.-J. Cox, David and D. O'Shea. *Ideals, varieties, and algorithms*. Springer, 3 edition, 2007.

[16] G.-M. Greuel and G. Pfister. *A* **Singular** *introduction to commutative algebra*. Springer, Berlin, extended edition, 2008.

[17] H. Lausch and W. Nöbauer. *Algebra of polynomials*. North-Holland Publishing Co., Amsterdam, 1973. North-Holland Mathematical Library, Vol. 5.

[18] H. Li. *Noncommutative Gröbner bases and filtered-graded transfer*. Springer-Verlag, 2002.

[19] G. Regensburger and M. Rosenkranz. An algebraic foundation for factoring linear boundary problems. *Ann. Mat. Pura Appl. (4)*, 188:123–151, 2008. DOI:10.1007/s10231-008-0068-3.

[20] M. Rosenkranz. A new symbolic method for solving linear two-point boundary value problems on the level of operators. *J. Symbolic Comput.*, 39(2):171–199, 2005.

[21] M. Rosenkranz and G. Regensburger. Integro-differential polynomials and operators. In D. Jeffrey, editor, *ISSAC'08: Proceedings of the 2008 International Symposium on Symbolic and Algebraic Computation*. ACM Press, 2008.

[22] M. Rosenkranz and G. Regensburger. Solving and factoring boundary problems for linear ordinary differential equations in differential algebras. *Journal of Symbolic Computation*, 43(8):515–544, 2008.

[23] W. Windsteiger. Building up hierarchical mathematical domains using functors in THEOREMA. In A. Armando and T. Jebelean, editors, *Electronic Notes in Theoretical Computer Science*, volume 23 of *ENTCS*, pages 401–419. Elsevier, 1999.

[24] F. Winkler. *The Curch-Rosser Property in Computer Algebra and Special Theorem Proving: An Investigation of Critical-Pair/Completion Algorithms*. PhD thesis, RISC-Linz, 1984. Verband der Wissenschaftlichen Gesellschaften Österreichs.

[25] A. Zapletal. *Compilation of Theorema Programs*. PhD thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, June 2008.

Guessing a Conjecture in Enumerative Combinatorics and Proving It with a Computer Algebra System

Alain Giorgetti
INRIA Nancy - Grand Est / CASSIS project
LIFC, University of Franche-Comté,
16 route de Gray, 25030 Besançon, France
alain.giorgetti@univ-fcomte.fr

Abstract

We present a theorem-proving experiment performed with a computer algebra system. It proves a conjecture about the general pattern of the generating functions counting rooted maps of given genus. These functions are characterized by a complex non-linear differential system between generating functions of multi-rooted maps. Establishing a pattern for these functions requires a sophisticated inductive proof. Up to now these proofs were made by hand. This work is the first computer proof of this kind of theorem. Symbolic computations are performed at the same abstraction level as the hand-made proofs, but with a computer algebra system. Generalizing this first success may significantly help solving algebraic problems in enumerative combinatorics.

1 Introduction

This work shows how a computer algebra system can help establishing new results in enumerative combinatorics. It is illustrated by the example of a recent conjecture about the general pattern of generating functions counting rooted maps by genus.

The general context is Problem 6 identified by Bender [2] in his list of ten unsolved problems in map enumeration. This problem is to find a "simple formula" defining the generating function $M_g(z)$ counting rooted maps of genus g by number of edges (exponent of z) for each positive genus. Rooted maps are combinatorial objects that were first enumerated by Tutte [7, 8] in the 1960's. A common pattern for all the $M_g(z)$, where g ranges over the positive integers, was first proposed in [3]. Each $M_g(z)$ was proved to be expressible as a rational function of $\rho = \sqrt{1-12z}$. However there is an unknown polynomial of ρ in the numerator of this function. An upper bound for its degree was conjectured but not proved. In [1] we provide the first proof of a more precise pattern, with a maximal degree for each unknown polynomial, when counting by number of vertices and faces. Focusing back on counting by number of edges, we [9] prove from it that a general pattern for the generating function $M_g(z)$ is

$$M_g(z) = m^{2g}(1-2m)^{4-5g}(1-3m)^{2g-2}(1-6m)^{3-5g}P_g(m), \tag{1}$$

where $P_g(m)$ is a polynomial of degree $6g-6$. After computing the explicit formulas for $g=1$ to $g=6$, we [9] conjecture that this polynomial is divisible by $(1-2m)^{2g-2}$ for all $g \geq 1$. The proof is obvious for $g=1$. The present work proves this conjecture for $g \geq 2$.

All the formerly mentioned proofs of general patterns were made by hand or with minimal computer assistance. These proofs are long, tedious and subject to errors. We expect to avoid errors and to reduce the proof construction effort by assisting it with a computer. The proof difficulty does not come from the underlying logical theory but from the size of the recursive definition of the polynomials $P_g(m)$. They indeed depend on two other families of polynomials. A challenge is to generalize the initial conjecture by guessing a more general conjecture for all the involved polynomials. Symbolic computations help to discover this generalized conjecture. Then three proofs by induction are constructed by substitution

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 5-18

and algebraic computations. Difficulties are reduced by replacing the exact system of equations by an abstraction of it that preserves the property to be proved.

The price to pay for computer assistance is to encode the problem and strategies for its resolution in a computer language. In the present case the problem is essentially algebraic. The definition of the polynomials $P_g(m)$ is composed of sums and products of polynomials (and anecdotally of some rational functions). This is a good reason for choosing a computer algebra system rather than a theorem prover. Certain unknown powers of $(1-2m)$ are conjectured to depend linearly on certain parameters. This idea comes from previous experience but is also motivated by the hope of obtaining a linear system to solve the generalized conjecture. Here again a theorem prover is not needed. The linear systems to be solved are of small size. Linear systems of equalities are solved by Gaussian elimination [6]. Linear systems of inequalities are solved by Fourier-Motzkin elimination [6] or with the simplex method [6] when it is possible to add an optimization goal. All these procedures are available in computer algebra systems.

Our contribution is to provide the first computer proof of an algebraic conjecture from enumerative combinatorics. The Maple code produces a trace of the conjectures discovered and of the three proofs by induction performed.

A definition of the polynomials $P_g(m)$ is given in Section 2. Section 3 presents the keys of an abstraction that simplifies this definition whilst preserving the divisibility property to be proved. Section 4 explains how symbolic computations help to guess a maximal power of $(1-2m)$ dividing each kind of polynomials appearing in the definition of the polynomial $P_g(m)$. Section 5 presents other symbolic computations performing proofs by induction of a divisibility property for each kind of polynomials.

2 Algebraic Problem

This section presents the large system of equations defining the polynomial $P_g(m)$. Since all the polynomials and rational functions defined hereafter are in the single indeterminate m, this indeterminate is omitted. For instance we write P_g instead of $P_g(m)$.

The polynomial P_g is defined in terms of another polynomial U_g by

$$P_g = U_g(1-m)^{4-4g}. \tag{2}$$

That polynomial is itself defined from a family of polynomials $S_g(n_1,\ldots,n_r)$ in the indeterminate m by

$$U_g = S_{g-1}(0,0) + m(1-2m)\,(1-m)^2 \sum_{j=1}^{g-1} S_j(0) S_{g-j}(0). \tag{3}$$

The goal is to construct a proof that $P_g(m)$ is divisible by $(1-2m)^{2g-2}$ for all $g \geq 1$. The proof difficulty comes from the size of the recursive definition of the polynomials $S_g(n_1,\ldots,n_r)$, called the *S-polynomials*. For any sequence $n_1,\ldots,n_r$ of non-negative integers, any non-negative integer g and any positive integer r such that $(g,r) \neq (0,1)$, the polynomial $S_g(n_1,\ldots,n_r)$ is indeed recursively defined in terms of some polynomials $S_j(p_1,\ldots,p_h)$ with j less than or equal to g. When j equals g the number h of parameters is less than or equal to r. When h equals r, the sum $p_1+\ldots+p_h$ is strictly less than $n_1 + \ldots + n_r$. It is also known [9] that the polynomials $S_g(n_1,\ldots,n_r)$ are symmetric in $n_1,\ldots,n_r$.

Before giving the recursive definition of the *S*-polynomials in Section 2.2 some convenient notations are introduced in Section 2.1.

2.1 Notations

For any positive integer r, $[r]$ denotes the sequence $(2,\ldots,r)$ if $r \geq 2$ and the empty sequence if $r = 1$. For any subsequence X of $[r]$, $[r]-X$ denotes the subsequence of the elements of $[r]$ that are not in X.

For any sequence $(n_2,\ldots,n_r)$ of integers, N_X denotes the sequence of those n_i such that i is in X and N_j denotes the sequence $(n_2,\ldots,n_{j-1},n_{j+1},\ldots,n_r)$. The polynomials K_i are defined for $i\geq 0$ by $K_0=-m$, $K_1=-1-m$, $K_2=-1$ and $K_i=0$ if $i\geq 3$. The polynomials L_k are defined for $k\geq 0$ by $L_0=-m$, $L_1=-1-2m$, $L_2=-2-m$, $L_3=-1$ and $L_k=0$ if $k\geq 4$. Finally we introduce an infinite family $(E_k)_{k\geq 1}$ of rational functions of m, all but the first two of which are polynomials, defined recursively by

$$E_1=\frac{1}{2m(1-2m)(1-m)^2},\qquad E_2=\frac{-5}{2(1-m)^2},\qquad E_3=-1, \tag{4}$$

and

$$E_k=-m(1-2m)(1-m)^2\sum_{i=2}^{i=k-1}E_iE_{k+1-i}\text{ for all }k\geq 4. \tag{5}$$

2.2 Recursive definition

The polynomials $S_0(n_1)$ are not defined. The recursive definition of the polynomials $S_g(n_1,\ldots,n_r)$ starts with $g=0$ and $r=2$. We have

$$\begin{aligned}S_0(n_1,n_2) &= (1-m)^2(-n_2(1-6m)(1-2m)E_{n_1+n_2+2}-(n_2+1)E_{n_1+n_2+3})\\ &\quad +2m(1-2m)(1-m)^2\sum_{\substack{i+j+k=n_1+1\\ i>0,k<n_1}}(-1)^{j+1}(1-6m)^j(1-2m)^jE_iS_0(k,n_2).\end{aligned} \tag{6}$$

If $(g,r)\neq(0,2)$, then $S_g(n_1,\ldots,n_r)=\text{term}_1+\text{term}_2+\text{term}_3+\text{term}_4$, where

$$\text{term}_1=2m(1-2m)(1-m)^2\sum_{\substack{i+j+k=n_1+1\\ i>0,k<n_1}}(-1)^{j+1}(1-6m)^j(1-2m)^jE_iS_g(k,n_2,\ldots,n_r), \tag{7}$$

$$\text{term}_2=m(1-2m)(1-m)^2\sum_{\substack{k+l+i=n_1+1\\ 0\leq j\leq g\\ X\subseteq[r]\\ (j,X)\neq(0,\emptyset)\\ (j,X)\neq(g,[r])}}K_i(1-6m)^i(1-2m)^iS_j(k,N_X)S_{g-j}(l,N_{[r]-X}), \tag{8}$$

$$\text{term}_3=\sum_{i+j+k=n_1+1}K_i(1-6m)^i(1-2m)^iS_{g-1}(k,j,N_{[r]}) \tag{9}$$

and

$$\text{term}_4=\sum_{j=2}^{r}\left(\begin{array}{l}n_j\sum_{k+l=n_1+n_j+2}L_k(1-6m)^{k+1}(1-2m)^{k+1}S_g(l,N_j)\\ +(n_j+1)\sum_{k+l=n_1+n_j+3}L_k(1-6m)^k(1-2m)^kS_g(l,N_j)\end{array}\right). \tag{10}$$

Formulas (2)-(10) are derived from formulas in [9] by replacing two parameters p and q by the parameter m. See [9] for details.

3 Abstraction

It is obvious from (2) that P_g is divisible by $(1-2m)^{2g-2}$ if and only if the same property holds for U_g. Each polynomial U_g (for $g\geq 1$) is defined by (3) as a sum of g terms. Proving that U_g is divisible by $(1-2m)^{2g-2}$ is easy if each of these terms is itself divisible by $(1-2m)^{2g-2}$. This sufficient but obviously not necessary condition is called the *term-by-term divisibility property* of (3). We conjecture that this property holds for (3) and try to prove it. Since each term in the right-hand side (RHS) of (3)

involves one or two S-polynomials, we conjecture that these S-polynomials are divisible by some power of $(1-2m)$ which is high enough for the initial conjecture to be provable.

The exact system of equations (6)-(10) defining the S-polynomials, called *the S system*, is large and complex. It is replaced by a simpler one by performing the following transformations. All of them preserve the term-by-term divisibility property.

3.1 Reasoning about indefinite sums

The sum of the last $g-1$ terms in (3) is an *indefinite* sum, i.e. a generalized summation, expressed with the mathematical sign $\sum$. The complexity of the S system mainly comes from the indefinite sums it contains.

An indefinite sum is a formal expression of the form

$$\sum_F E$$

also written $\sum_F E$. It denotes the summation of the expression E for all the models of the formula F, or 0 is F is not satisfiable. Technically, the $\sum$ sign is called a *binder*, the formula F is called the *sum constraint* and the expression E is called the *summed term*. The ($\sum$ sign of the) indefinite sum binds all the variables that appear in F and that are not bound or defined earlier in the formal expression where the indefinite sum appears.

In the S system, the variables defined earlier are g, r and the n_is for $1 \leq i \leq r$. Thus the set of variables bound by the indefinite sums in the expressions (6), (7), (8) and (9) is respectively $\{i,j,k\}$, $\{i,j,k\}$, $\{k,l,i,j,X\}$ and $\{i,j,k\}$. In (10) the external $\sum$ sign binds j and the two internal $\sum$ signs bind k and l. The bound variables i, j, k and l are non-negative integers, whereas X is a finite set of positive integers, or equivalently a strictly increasing sequence of such numbers.

To prove divisibility, we essentially need the factorization property that

$$\sum_{\ldots} AB = A \sum_{\ldots} B \tag{11}$$

when no variable bound by the indefinite sum appears in expression A, formally meaning that A is independent of these variables.

We plan to apply this property to all the indefinite sums of the S system, when A is $(1-2m)^d$ and d is an expression whose variables are not bound by the indefinite sum. If an algebraic expression E is divisible by $(1-2m)^d$, i.e. can be factorized as $(1-2m)^d B$, then by Property (11) any indefinite sum $\sum_F E$, where the variables in d are not bound in F, can also be factorized as $(1-2m)^d \sum_F B$, i.e. is divisible by $(1-2m)^d$. This sufficient condition of *term-by-term divisibility* allows the proof to be established on an abstract version of the S system where each indefinite sum is replaced by its summed term. At the same time, its sum constraint is transformed as described in the sext section.

3.2 Sum constraints

The sum constraint of an indefinite sum in the S system is composed of inequalities (in the large sense, including set inclusions), disequalities and exactly one equality. In (8) the inequalities are $0 \leq j$, $j \leq g$ and $X \subseteq [r]$, the disequalities are $(j,X) \neq (0,\emptyset)$ and $(j,X) \neq (g,[r])$, and the equality is $k+l+i = n_1+1$.

The equality is used to eliminate one of the bound variables in the summed term and in the sum constraint. For instance the summed term in (6) is

$$(-1)^{j+1}(1-6m)^j(1-2m)^j E_i S_0(n_1+1-(i+j),n_2) \tag{12}$$

and the sum constraint is $i > 0 \wedge n_1 + 1 - (i + j) < n_1$ whose simplification is $i > 0 \wedge 1 < i + j$. This transformation also eliminates n_1 from the other inequality (7) where it appeared.

Then all the inequalities and disequalities are turned into global hypotheses where their variables have been renamed for unicity in the whole system. The implicit constraints that all the bound integer variables are non-negative is made explicit. For instance, the inequalities in (6) are turned into the global hypotheses $i_0 > 0$, $j_0 \geq 0$ and $1 < i_0 + j_0$. The constraints $i_0 \leq n_1 + 1$ and $j_0 \leq n_1 + 1$ are thrown away because n_1 can be arbitrarly large. The other global hypotheses are $i_1 > 0$, $j_1 \geq 0$ and $1 < i_1 + j_1$ from (7), $0 \leq j_2$, $j_2 \leq g$, $X_2 \subseteq [r]$, $(j_2, X_2) \neq (0, \emptyset)$ and $(j_2, X_2) \neq (g, [r])$ from (8), $i_3 \geq 0$ and $j_3 \geq 0$ for (9), $2 \leq j_4$, $j_4 \leq r$ and $0 \leq k$ from (10).

3.3 Length and sum of a sequence of parameters

The second simplification comes from the fact that all the polynomials $S_g(n_1, \ldots, n_r)$ are symmetric in $n_1, \ldots, n_r$. The sequence $n_1, \ldots, n_r$ is abstracted by its length r and its sum $n = n_1 + \ldots + n_r$. In the S system the expression $S_g(n_1, \ldots, n_r)$ is replaced by the expression $\tilde{S}(g, r, n)$ such that $n = n_1 + \ldots + n_r$. For instance the term (12) is replaced by

$$(-1)^{j+1} (1 - 6m)^j (1 - 2m)^j E_i \tilde{S}(0, 2, n + 1 - (i + j)). \tag{13}$$

Simultaneously, any set Y of integers is replaced by its cardinality c_Y and the sum n_Y of its elements. For instance the global hypothesis $X_2 \subseteq [r]$ coming from (8) is replaced by the global hypothesis $c_{X_2} \leq r - 1$. The constraint $n_{X_2} \leq n_{[r]}$ is thrown away because $n_{[r]} = n - n_1$ can be arbitrarly large.

After these simplifications, there remain two occurrences of n_2 in (6) and two occurrences of n_j in (10) as multiplicative factors. They are considered as new formal symbols.

It is planned to automate all these transformations from a symbolic representation of the S system. At the current stage of this research the simplified system is directly written by hand in a Maple file. It is claimed that this Maple code is the correct definition of the simplified system. We do not reproduce this definition here on purpose, because writing it by hand presents a risk of typographical error and translating it from the Maple code into LaTeX syntax is not yet fully automatized.

3.4 Property preservation

Property preservation does not claim for an equivalence, but only for the following entailment: if the divisibility property of $\tilde{S}(g, r, n)$ is proved on the simplified system, then the same divisibility property holds for all the polynomials $S_g(n_1, \ldots, n_r)$ such that $n = \sum_{1 \leq i \leq r} n_i$ and the simplified proof can be lifted up on the S system.

In a near future we expect to derive from a formal specification of all the simplifications a formal proof that they preserve the term-by-term divisibility property. For the moment we can only justify this fact informally. Preservation by replacement of indefinite sums by their summed term has already been justified in Section 3.1. Once the sequences of integers have been replaced by their length and sum it cannot be conjectured anymore that the polynomials $S_g(n_1, \ldots, n_r)$ are divisible by a power of $(1 - 2m)$ which directly depends on the n_is but only on their sum and cardinality. Finally, the integers n_2 and n_j replaced by formal symbols are all multiplicative factors of products where divisibility by a power of $(1 - 2m)$ has to be observed; so divisibility after replacement entails divisibility before it, with the same power.

Altogether these simplifications lead to a proof of a stronger conjecture than the initial one. In case of failure, some of them will have to be relaxed in order to find a more subtle proof argument.

4 Conjecture Synthesis

The goal in this proof step is to guess a function of g, r, and n that computes a non-negative integer d such that (i) all the polynomials $\tilde{S}(g,r,n)$ are divisible by $(1-2m)^d$, (ii) this divisibility property is provable (term by term) by induction from the equations defining these polynomials, and (iii) the degree d is high enough to prove the divisibility conjecture for U_g. We try to synthetize d as a linear function of g, r and n by translating (i) into the property that there exists four integers c_g, c_r, c_n, c_1 and a polynomial $T(g,r,n)$ such that the equality

$$\tilde{S}(g,r,n) = (1-2m)^{c_g g + c_r r + c_n n + c_1} T(g,r,n) \tag{14}$$

always holds.

But the polynomials $\tilde{S}(g,r,n)$ are also computed from expressions E_k which probably contribute to the total degree of $(1-2m)$. We also have to conjecture this contribution and prove it.

The first two expressions E_1 and E_2 are not polynomials but rational functions. The factor $(1-2m)$ appears in the denominator of E_1. For sake of simplicity we prefer to consider only polynomials and non-negative powers of $(1-2m)$. Thus we introduce the polynomial D_k related to the polynomial E_k by

$$D_k = -2\,m\,(1-2\,m)\,(1-m)^2\,E_k \tag{15}$$

for all $k \geq 1$. From (4), (5) and (15) the computer easily yields the following recursive definition

$$D_1 = -1, \qquad D_2 = 5\,m\,(1-2\,m)\,, \qquad D_3 = 2\,m\,(1-2\,m)\,(1-m)^2 \tag{16}$$

and

$$D_k = \frac{1}{2}\sum_{i=2}^{i=k-1} D_i D_{k+1-i} \quad \text{for all } k \geq 4 \tag{17}$$

of the infinite family $(D_k)_{k\geq 1}$.

It is again expected that (iv) there exists two integers e_k, e_1 and a polynomial $F(k)$ such that

$$D_k = (1-2m)^{e_k k + e_1} F(k) \tag{18}$$

always holds, (v) there is a term-by-term divisibility proof by induction of this conjecture, and (vi) the degree $e_k k + e_1$ is high enough to contribute to the proof of conditions (i)-(iii) about $\tilde{S}(g,r,n)$. Note that conditions (iv), (v) and (vi) are respectively similar to conditions (i), (ii) and (iii), showing that the approach can be generalized.

The next two sections translate conditions (v) and (iii) into two equivalent systems of inequalities. Then they show how conjectures are elaborated by interpretation of values extracted from these systems.

4.1 Guessing a pattern for the polynomials D_k

The Maple code translates condition (v) into the equivalent system of four inequalities

$$\{e_k + e_1 \leq 0, \quad 2\,e_k + e_1 \leq 1, \quad 3\,e_k + e_1 \leq 1, \quad 0 \leq e_k + e_1\}\,. \tag{19}$$

The first three inequalities come from the three base cases $k = 1,2,3$ and the last one comes from the induction step of the proof by induction of term-by-term divisibility of D_k by $(1-2m)^{e_k k + e_1}$. The Maple function `solve()` decomposes this system into two systems, depending on the sign of e_k. Because we want to maximize the degree $e_k k + e_1$ of $(1-2m)$ we choose the reduced system

$$\left\{e_k \leq \frac{1}{2}, \quad 0 \leq e_k, \quad e_1 = -e_k\right\}, \tag{20}$$

where e_k is not negative. Unfortunately its unique solution with integral coefficients $e_k = e_1 = 0$ provides no contribution of the expressions E_k to the factorization of the polynomials $\tilde{S}(g,r,n)$.

We relax the condition that the coefficients should be integers and obtain a solution $e_k = \frac{1}{2}, e_1 = -\frac{1}{2}$ which maximizes the degree $e_k k + e_1$ of $(1-2m)$ at the value $\frac{k-1}{2}$. When k is even this degree is not an integer but a half-integer and condition (iv) is not established. We shall see in the next section that half-integers also arise from the same method applied to the polynomials U_g. We jointly discuss their interpretation and possible treatments in Section 4.3.

4.2 Guessing a pattern for the polynomials U_g and $\tilde{S}(g,r,n)$

We now apply the same method to condition (iii). We try to factorize the polynomials $\tilde{S}(g,r,n)$ with a power of $(1-2m)$ that is sufficiently high to make U_g divisible by $(1-2m)^{2g-2}$ for any $g \geq 2$.

The method proceeds as follows. Three instances of (14) are generated by setting the triple of variables (g,r,n) to the values $(g-1,2,0)$, $(j,1,0)$ and $(g-j,1,0)$. The three instances are then introduced in the RHS of the abstraction of (3) to eliminate the polynomials $\tilde{S}(g-1,2,0)$, $\tilde{S}(j,1,0)$ and $\tilde{S}(g-j,1,0)$. The result is simplified and then divided term by term by $(1-2m)^{2g-2}$. Condition (iii) is equivalent to the condition that the remaining degree of $(1-2m)$ in each term should not be negative. Due to the indefinite sum in (3) only two terms are considered: one coming from $S_{g-1}(0,0)$ and one coming from the term under the $\sum$ summation sign. The corresponding conditions respectively are

$$\forall g \,.\, g \geq 2 \Rightarrow c_g(g-1) + 2c_r + c_1 - 2g + 2 \geq 0 \tag{21}$$

and

$$\forall g \,.\, g \geq 2 \Rightarrow c_g g + 2c_r + 2c_1 - 2g + 3 \geq 0. \tag{22}$$

For $g = 2$ and $g = 3$ we know from explicit values that the divisibility property of the polynomials $\tilde{S}(g,r,n)$ is just sufficient for the corresponding property of the polynomials U_g to be true. In other words the corresponding inequalities in these conditions are equalities. The system of these four equalities for the cases $g = 2$ and $g = 3$ is over-constrained because there are only three unknowns but it admits the solution $c_g = 2$, $c_r = \frac{3}{2}$ and $c_1 = -3$, provided by the Maple function `solve()`. With this solution the two general conditions (21) and (22) are satisfied.

Condition (iii) does not determine c_n because all the n_is are 0 in (3). A value for c_n can be derived from a known S-polynomial with a non-null sum of n_i parameters. The polynomials

$$S_0(0,1) = (1-2m)(4m+1) \tag{23}$$

and

$$S_1(1,1) = (1-m)(1-2m)(16m^3 - 38m^2 + 21m - 4) \tag{24}$$

respectively provide the constraint $c_n \leq 1$ and $c_n \leq \frac{1}{2}$. We retain $c_n = \frac{1}{2}$ to complete the conjecture about the polynomials $\tilde{S}(g,r,n)$.

4.3 How to deal with half degrees?

Our best effort to guess a maximal power of $(1-2m)$ dividing the polynomials D_k and $\tilde{S}(g,r,n)$ has lead to the strange conjecture that this power is sometimes not an integer but a half-integer. How can this result be interpretated?

An accurate observation of the first values suggests that it should be possible to prove a divisibility by an integral power of $(1-2m)$. For instance a stronger conjecture for the polynomials D_k would be that D_{2k} and D_{2k-1} are divisible by $(1-2m)^k$ for $k \geq 1$. But this obversation and the previously guessed

conjecture also indicate that this proof should distinguish odd and even values of the parameters k, r and n. Taking the parity of the parameters k, r and n into account would multiplying by two the size of the proof by induction for the polynomials D_k and by four the one for the polynomials $\tilde{S}(g,r,n)$. Even if this multiplication of cases can be delegated to the computer, we prefer the more compact but more abstract approach exposed in the next section.

5 Proof synthesis

Let $H(E)$ by the property that E belongs to the ring $\mathbb{Q}[m]+(1-2m)^{\frac{1}{2}}\mathbb{Q}[m]$, where $\mathbb{Q}[m]$ is the ring of polynomials in the indeterminate m. This means that the algebraic expression E is either a polynomial in the indeterminate m or the product by $(1-2m)^{\frac{1}{2}}$ of a polynomial in the indeterminate m. We denote by *the theory of* H the axioms that the property H is satisfied by all the polynomials in the indeterminate m and by the expression $(1-2m)^{\frac{1}{2}}$ and is preserved by addition and multiplication.

With the help of the property H the divisibility properties to be proved are expressed in terms of powers $\frac{k-1}{2}$ and $2g+\frac{3}{2}r+\frac{1}{2}n-3$ that can be half-integers. The proofs proceed by application of the theory of H.

There is no divisibility proof to construct for the polynomials U_g because the coefficients c_g, c_r and c_1 have been guessed so that U_g is divisible by $(1-2m)^{2g-2}$ for all $g\geq 1$, as explained in Section 4.2. The three proofs to construct concern the expressions E_k (through the polynomials D_k for simplicity), the polynomials $\tilde{S}(0,2,n)$ and the polynomials $\tilde{S}(g,r,n)$ for $(g,r)\neq(0,2)$.

5.1 Polynomials D_k

We now prove by induction on k that there exists an expression $F(k)$ such that $H(F(k))$ holds and

$$D_k=(1-2m)^{\frac{k-1}{2}}F(k) \tag{25}$$

for all $k\geq 1$. The base cases $k=1,2,3$ are checked from (16) by instantiation of (25) and extraction of $F(1)$, $F(2)$ and $F(3)$.

The induction step consists of fixing k and assuming that there exists an expression $F(h)$ such that $H(F(h))$ holds and $D_h=(1-2m)^{\frac{h-1}{2}}F(h)$ for all $1\leq h<k$. This induction step is computer-assisted as follows. With the Maple function `subs` three substitutions replace the three expressions D_k, D_i and D_{k+1-i} in (17) by the corresponding RHS of (25). The result after simplification is the equality

$$F(k)=\frac{1}{2}F(i)F(k+1-i) \tag{26}$$

for $k\geq 4$. The theory of H is applied to this equality to state that $H(F(i))$ and $H(F(k+1-i))$ imply $H(F(k))$. This step ends the proof for the polynomials D_k.

The polynomials D_k are intermediate expressions to obtain a general pattern for the expressions E_k. The following lemma required for the next proofs is established by elimination of D_k according to (15).

Lemma 1. *For all $k\geq 1$ there exists an expression $F(k)$ such that $H(F(k))$ holds and*

$$E_k=\frac{-(1-2m)^{\frac{k-3}{2}}}{2m(1-m)^2}F(k). \tag{27}$$

5.2 Polynomials $\tilde{S}(0,2,n)$ and $\tilde{S}(g,r,n)$

It remains to prove the following lemma by induction on g, r and n.

Lemma 2. *For any $g \geq 0$, $r > 0$ and $n \geq 0$ such that $(g,r) \neq (0,1)$ there exists an expression $T(g,r,n)$ such that $H(T(g,r,n))$ holds and*

$$\tilde{S}(g,r,n) = (1-2m)^{2g+\frac{3}{2}r+\frac{1}{2}n-3} T(g,r,n). \tag{28}$$

The goal of this section is not to sketch this proof but to explain how it is produced by Maple code. The proof construction is divided into two cases, because the polynomials $\tilde{S}(0,2,n)$ are defined an equation different from the polynomials $\tilde{S}(g,r,n)$ when $(g,r) \neq (0,2)$. The two proof cases are treated the same way as in Section 5.1.

Let A be the simplified system of equations defining $\tilde{S}(g,r,n)$ and obtained from the S system by the abstraction defined in Section 3 (where the sequences $n_1,\ldots,n_r$ are abstracted by their sum n and their length r, some sets of integers are replaced by their cardinality and the sum of their elements, some multiplicative factors are abstracted by uninterpreted symbols, and indefinite sums are replaced by their summed term).

The induction hypotheses and lemma 1 are used to replace all the occurrences of polynomials $\tilde{S}(\ldots,\ldots,\ldots)$ and expressions $E_{\ldots}$ in the RHS of the equations in A by their factorized forms. Then an iterative process considers each monomial in these RHS one by one. For each monomial the simplification functions of Maple are called to sum up the total power of $(1-2m)$. This power is divided by $(1-2m)^{2g+\frac{3}{2}r+\frac{1}{2}n-3}$ and it is observed whether the result satisfies property H.

5.3 Final remark

There remains a final deductive step from the relaxed proofs to proofs of divisibility by an integral power of $(1-2m)$. From Lemma 2 and the fact that $\tilde{S}(g,r,n)$ is a polynomial we deduce more about $T(g,r,n)$ than $H(T(g,r,n))$, namely that $T(g,r,n)$ is a polynomial when $2g+\frac{3}{2}r+\frac{1}{2}n-3$ is even and is the product of a polynomial by $(1-2m)^{\frac{1}{2}}$ when $2g+\frac{3}{2}r+\frac{1}{2}n-3$ is odd. A similar argument is required for the polynomials D_k to complete the mathematical proofs. These final arguments are not supported by the actual symbolic computations.

6 Discussion

After the coefficients have been guessed from a part of the problem in Section 4, these coefficients are checked to satisfy the remaining conditions of the problem in Section 5. We could think that it is possible to remove the checking step by converting the whole problem into a system of inequalities. This approach of *full guessing* is attractive but failed during the present experiment, because some of the generated inequalities are not linear. The resulting problem is to satisfy a first-order formula of elementary number theory (first-order arithmetic over integers) and we know from Gödel's theorem [5] that there is no decision procedure for this theory. We expect that this formula belongs to a decidable fragment of this theory but we have not identified it yet. This is why the present proof construction is not fully automated, but only computer-assisted.

Some human decisions are required. The most difficult one was to select the subproblem to submit to the guessing method. This choice was guided by the layered structure of the recursive definition of the polynomials P_g. The problem has been successfully divided into smaller sub-problems that have

been solved one after the other. Some remaining non linearities required doing extra assumptions before finding a solution. Finally nonintegral powers appeared that required relaxing the conjectures. All these difficulties indicate that the initial problem was difficult.

6.1 Maple code

The conjecture and proof syntheses respectively described in Section 4 and 5 are implemented in the Maple program `abstrac.mpl`. They both start from a Maple encoding of the abstract system described in Section 3. At the present time the abstraction is performed by hand from the equations in [9] but an automation is planned.

The Maple code produces a trace in a LaTeX file. The result of its compilation is presented in Appendix A. Its first part explains the conjecture guessing process. Its second part contains the synthetized mathematical proofs by induction. All the equalities and inequalities shown in this trace are Maple expressions handled by the program to produce subsequent results. Many computed expressions are too large to be reproduced here. The Maple code writes them in another text file. It is envisaged to extend the function outputting LaTeX with line-breaking in order to fit a document width requested by the user.

6.2 Related work

The problem complexity mainly comes from the indefinite sums in the equalities composing the S system.

We need a theory for reasoning about these indefinite sums. An idea could have been to define these $\sum$ signs as a special kind of "big operators" specified in [4] for the Coq proof assistant. They indeed generalize the sum operator $(+)$ of two polynomials of m. The properties required during the proof search can certainly be found within the rich theory defined in [4]. The advantage of this theory is its generality, but it is a drawback in the present proof construction, because of the numerous prerequisites before applying it. We first have to define the ring of polynomials in m. We then have to reduce the multivariate summations to the univariate ones that are the only ones supported by this package. After all these preparations, we would have observed that the proof most often uses the lemmas gathered in the factorization property (11) that has inspired the abstraction of indefinite sums by their summed term in Section 3.1. Our approach is clearly less general but much lighter.

7 Conclusion

We have experienced constructing the proof of a conjecture of enumerative combinatorics with the help of a computer algebra system. To our knowledge it is the first strong computer assistance for this kind of conjectures.

The divisibility of the polynomials P_g by $(1-2m)^{2g-2}$ has been easily conjectured by human observation from the first computed explicit values of these polynomials. But the definition of these polynomials made it much harder to find a proof of this conjecture by hand. The definition is not complex from a mathematical point of view. Its complexity mainly comes from the indefinite sums in the equalities composing it and from the numerous parameters of the intermediate expressions introduced to decompose it. Finding a uniform way to treat all these cases is not only a way to reduce the proof search effort, but is also a key to assist it with a computer. Our proposal is a strong uniformization because the same factorization property is applied for all the indefinite sums, independently of the number and nature of the variables they bind.

The symbolic encoding of computations allows many tries to be repeated and many errors to be avoided. The symbolic computations in the present experiment are performed at the same abstraction level as the hand-made proof, but with a computer algebra system that avoids many errors and makes

it possible to fix the remaining ones quickly. The code outputs a mathematical proof of the divisibility property that looks like a proof by induction written by hand, for instance in [9]. The complete code represents about 700 lines of Maple code.

Many similar conjectures remain open in this research domain, for which the same approach could be re-used. The abstraction and conjecture synthesis techniques identified and presented here are general enough to be applied to these other conjectures. It is a part of our future work in collaboration with combinatoricians.

In a commented version of [2] published on his web site in 2002, Bender concludes Problem 6 (to provide a general pattern for the generating functions of rooted maps counted by number of edges) by writing "Arquès and Giorgetti [1] may have done as much as possible". This claim leaves implicit the means that can be employed to do more than has already been done. We agree with this claim if these means are limited to our human brain faculties (in any case mine). But we disagree if the proof search can be assisted by a computer. The present computer-assisted proof synthesis justifies this disagrement. It is a first success whose generalization may significantly contribute to the solution of many algebraic problems in the mathematical field of map enumeration.

8 Acknowledgment

We would like to thank Prof. T. R. S. Walsh for his stimulating discussion by electronic mail and for his corrections to a preliminary version of the present text.

References

[1] D. Arquès and A. Giorgetti. Énumération des cartes pointées de genre quelconque en fonction des nombres de sommets et de faces. *J. Combin. Theory Ser. B*, 77(1):1–24, sep 1999.

[2] E. A. Bender. Some unsolved problems in map enumeration. *Bull. Inst. Combin. Appl*, 3:51–56, 1991.

[3] E. A. Bender and E. R. Canfield. The number of rooted maps on an orientable surface. *J. Comb. Theory, Ser. B*, 53(2):293–299, 1991.

[4] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In O. Aït Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2008.

[5] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–98, 1931.

[6] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[7] W. T. Tutte. A census of planar maps. *Canad. J. Math.*, 15:249–271, 1963.

[8] W. T. Tutte. On the enumeration of planar maps. *Bull. Amer. Math. Soc.*, 74:64–74, 1968.

[9] T. R. S. Walsh and A. Giorgetti. Efficient enumeration of rooted maps of a given orientable genus by number of faces and vertices. Submitted.

A Synthetized mathematical text

The software outputs the following trace, composed of two parts. The first part explains how the conjectures are guessed. The second part sketches mathematical proofs by induction.

A.1 Conjecture synthesis

The goal is to find six coefficients e_k, e_1, c_g, c_r, c_n and c_1 such that there exists three families $V(g)$, $T(g,r,n)$ and $F(k)$ of elements of $\mathbb{Q}[m]+(1-2m)^{\frac{1}{2}}\mathbb{Q}[m]$ such that

$$U_g = (1-2m)^{2g-2}V(g), \tag{29}$$

$$\tilde{S}(g,r,n) = (1-2m)^{c_g g+c_r r+c_n n+c_1}T(g,r,n), \tag{30}$$

$$D_k = (1-2m)^{e_k k+e_1}F(k), \tag{31}$$

$$D_k = -2m(1-2m)(1-m)^2E_k \tag{32}$$

and the S system hold.

A.1.1 Polynomials D_k

Base cases

Case k = 1

$$(1-2m)^{e_k+e_1}F(1) = -1 \tag{33}$$

gives the constraint

$$e_k + e_1 \leq 0. \tag{34}$$

Case k = 2

$$(1-2m)^{2e_k+e_1}F(2) = -5m(-1+2m) \tag{35}$$

gives the constraint

$$2e_k + e_1 \leq 1. \tag{36}$$

Case k = 3

$$(1-2m)^{3e_k+e_1}F(3) = 2m(1-2m)(1-m)^2 \tag{37}$$

gives the constraint

$$3e_k + e_1 \leq 1. \tag{38}$$

Induction step

$$(1-2m)^{e_k k+e_1}F(k) = \frac{1}{2}(1-2m)^{2e_1+e_k k+e_k}F(i)F(k+1-i) \tag{39}$$

gives the constraint

$$0 \leq e_k + e_1. \tag{40}$$

Optimization The solution maximizing e_k is $e_1 = -\frac{1}{2}$ and $e_k = \frac{1}{2}$. With this solution, the hypothesis for the polynomials D_k is

$$D_k = (1-2m)^{\frac{k-1}{2}} F(k) \tag{41}$$

Pattern of the rational functions The hypothesis

$$E_k = -\frac{1}{2}(1-2m)^{\frac{k-3}{2}} F(k) m^{-1} (-1+m)^{-2} \tag{42}$$

is obtained by elimination of D_k.

A.1.2 Polynomials U_g

The constraints

$$2g-2 \leq c_g g - c_g + 2c_r + c_1 \tag{43}$$

and

$$2g \leq 3 + 2c_r + 2c_1 + c_g g \tag{44}$$

have to be satisfied.

A solution is $c_1 = -3$, $c_g = 2$ and $c_r = \frac{3}{2}$. The value $c_n = \frac{1}{2}$ is guessed from the case

$$S_1(1,1) = (-1+m)(-1+2m)(16m^3 - 38m^2 + 21m - 4) \tag{45}$$

A.2 Proofs by induction

A.2.1 Induction step for the polynomials D_k

After simplification the equality is

$$F(k) = \frac{1}{2} F(i) F(k+1-i) \tag{46}$$

A.2.2 Case $(g,r) = (0,2)$

Base case $(g,r,n) = (0,2,0)$

$$(1-2m)^{2c_r+c_1} T(0,2,0) = 1 \tag{47}$$

gives the constraint

$$2c_r + c_1 \leq 0 \tag{48}$$

which evaluates to true.

Induction step for $(g,r) = (0,2)$ Let $n \geq 1$. The induction hypothesis is

$$\tilde{S}(0,2,p) = (1-2m)^{2c_r + c_n p + c_1} T(0,2,p) \tag{49}$$

for all $0 \leq p < n$. The three terms in the RHS are considered one by one. The three resulting inequalities

$$2c_r + c_n n + c_1 \leq \frac{n+1}{2} \tag{50}$$

$$2c_r + c_n n + c_1 \leq \frac{n}{2} \tag{51}$$

and

$$2c_r + c_n n + c_1 \leq j - \frac{1}{2} + \frac{1}{2}i + 2c_r + nc_n + c_n - ic_n - jc_n + c_1 \tag{52}$$

are satisfied.

A.2.3 Case $(g,r) \neq (0,2)$

The induction hypothesis is

$$\tilde{S}(j,h,p) = (1-2m)^{c_g j + c_r h + c_n p + c_1} T(j,h,p) \tag{53}$$

with conditions on j, h and p too long to be reproduced here, but suitable for induction.

For term_1 the constraint is

$$0 \leq j - \frac{1}{2} + \frac{1}{2}i + c_n - c_n i - c_n j \tag{54}$$

which simplifies to

$$0 \leq \frac{1}{2}j \tag{55}$$

For term_2 the constraint is

$$0 \leq 1 + i + c_r + c_1 + c_n - c_n i \tag{56}$$

which simplifies to

$$0 \leq \frac{1}{2}i \tag{57}$$

For term_3 the constraint is

$$0 \leq k - c_g + c_r + c_n - c_n k \tag{58}$$

which simplifies to

$$0 \leq \frac{1}{2}k \tag{59}$$

For the first part of term_4 the constraint is

$$0 \leq k + 1 - c_r + 2c_n - c_n k \tag{60}$$

which simplifies to

$$0 \leq \frac{1}{2}k + \frac{1}{2} \tag{61}$$

For the second part of term_4 the constraint is

$$0 \leq k - c_r + 3c_n - c_n k \tag{62}$$

which simplifies to

$$0 \leq \frac{1}{2}k \tag{63}$$

All these constraints are satisfied, and this ends the proof.

Satisfiability of Non-Linear Arithmetic over Algebraic Numbers*

Harald Zankl René Thiemann Aart Middeldorp
Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria

Abstract

We propose an (incomplete) approach for satisfiability of non-linear arithmetic over algebraic real numbers by reducing the problem to non-linear arithmetic over the rationals/integers.

1 Introduction

The first order theory of non-linear arithmetic over the reals has been shown to be decidable by Tarski [2]. This decision procedure is of non-elementary computational complexity. Improvements of the original procedure are still of double exponential time complexity [1]. Existing implementations of these procedures can only cope with small instances. It is well-known that many termination criteria for term rewriting can be encoded in SAT/SMT. Typically such instances contain hundreds of variables and (tens of) thousands of arithmetic operations. In this note we suggest an alternative method for checking satisfiability of real arithmetic, capable of handling large instances. Our approach cannot be used to deduce unsatisfiability of arithmetic constraints since only (a subset of) algebraic real numbers are covered.

The remainder of this paper is organized as follows. In Section 2 we introduce the syntax of non-linear arithmetic constraints. Then Section 3 lists the encodings for the arithmetic operations while Section 4 shows their soundness (i.e., if the encoding evaluates to true then the original constraint is satisfiable). Section 5 then assesses our contribution while Section 6 concludes.

2 Syntax

In this section we fix the syntax for non-linear arithmetic constraints.

Definition 1. An *arithmetic constraint* φ is described by the BNFs

$$\varphi ::= \bot \mid \top \mid p \mid (\neg\varphi) \mid (\varphi \circ \varphi) \mid (\alpha \star \alpha) \quad \text{and} \quad \alpha ::= \mathrm{a} \mid r \mid (\alpha \diamond \alpha) \mid (\varphi\,?\,\alpha : \alpha)$$

where $\circ \in \{\vee, \wedge, \to, \leftrightarrow\}$, $\star \in \{>, =\}$, and $\diamond \in \{+, -, \times\}$.

Here $\bot$ ($\top$) denotes contradiction (tautology), p (a) ranges over Boolean (arithmetical) variables, $\neg$ ($\vee$, $\wedge$, $\to$, $\leftrightarrow$) is logical not (or, and, implication, bi-implication), $>$ ($=$) greater (equal), r ranges over the real numbers, and $+$ ($-$, $\times$) denotes addition (subtraction, multiplication). If-then-else is written as $(\cdot\,?\,\cdot : \cdot)$. The following example shows some (non-)well-formed constraints.

Example 2. The expressions 5, p_{100}, $(p_{10}\,?\,(2.1 \times \mathrm{a}_{12}) : 0)$, and $((((\mathrm{a}_{12} + (\sqrt{2} \times \mathrm{a}_{30})) + 7.2) > (0 - \mathrm{a}_5)) \wedge p_2)$ are well-formed whereas $-\mathrm{a}_{10}$ (unary minus) and $3 + \mathrm{a}$ (parentheses missing) are not.

The binding precedence $\times \succ +, - \succ >, = \succ \neg \succ \vee, \wedge \succ \to, \leftrightarrow, (\cdot\,?\,\cdot : \cdot)$ allows to save parentheses. Furthermore the operators $+$, $-$, $\times$, $\vee$, $\wedge$, and $\leftrightarrow$ are left-associative while $\to$ associates to the right. Taking these conventions into account the most complex constraint from the previous example simplifies to $\mathrm{a}_{12} + \sqrt{2} \times \mathrm{a}_{30} + 7.2 > 0 - \mathrm{a}_5 \ \wedge \ p_2$. To obtain smaller constraints already at the time of encoding trivial simplifications like $\varphi \wedge \top \to \varphi$, $\varphi \wedge \bot \to \bot$, $\cdots$ are performed.

Example 3. The constraint $\varphi := \mathrm{a}_1 \times \mathrm{a}_1 \times \mathrm{a}_1 = 3 \ \wedge \ \mathrm{a}_2 \times \mathrm{a}_2 = 2 \ \wedge \ \mathrm{a}_3 + \mathrm{a}_3 = 1$ is satisfiable. The assignment α with $\alpha(\mathrm{a}_1) = \sqrt[3]{3}$, $\alpha(\mathrm{a}_2) = \sqrt{2}$, and $\alpha(\mathrm{a}_3) = \frac{1}{2}$ is a model for φ.

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 19-24

* This research is supported by FWF (Austrian Science Fund) project P18763.

3 Reducing Reals to Integers

Next we show how to find models for constraints like the one in Example 3 automatically. To this end we extend the approach of dealing with algebraic real numbers from [3]. The idea is to reduce satisfiability of *real* arithmetic to arithmetic over *rational* numbers. A fast approach for the latter is introduced in [3]. To represent a real-valued variable we use pairs $([c_1,\ldots,c_n],m)$ (or in short $(\mathbf{c}_n,m)$) where the first element is a list of integer (or rational) valued variables and m is a variable taking integral values larger than one. The intended semantics of $(\mathbf{c}_n,m)$ is $c_1\sqrt[n]{m^0}+_{\mathbb{R}}\cdots+_{\mathbb{R}}c_n\sqrt[n]{m^{n-1}}$. Hence this approach can only represent (a subset of) algebraic numbers. For instance, $\sqrt[3]{1+\sqrt{2}}$ is not represented.

In the sequel we show that arithmetic operations can easily be encoded. To avoid confusion we denote the set of real (rational) numbers by $\mathbb{R}$ ($\mathbb{Q}$) and encodings of such numbers by $\mathbf{R}$ ($\mathbf{Q}$). Similarly a subscript $\mathbf{R}$ ($\mathbf{Q}$) is used to denote the encodings of arithmetic operations.

Definition 4. For $(\mathbf{c}_n,m)$ and $(\mathbf{d}_n,m)$ from $\mathbf{R}$ we define:

$$\begin{aligned}(\mathbf{c}_n,m)+_{\mathbf{R}}(\mathbf{d}_n,m) &:= ([c_1+_{\mathbf{Q}}d_1,\ldots,c_n+_{\mathbf{Q}}d_n],m)\\ (\mathbf{c}_n,m)-_{\mathbf{R}}(\mathbf{d}_n,m) &:= ([c_1-_{\mathbf{Q}}d_1,\ldots,c_n-_{\mathbf{Q}}d_n],m)\end{aligned}$$

The following example demonstrates addition for encodings of reals. To distinguish numbers from encodings of numbers the latter are represented in boldface.

Example 5. The computation $([\mathbf{1},\mathbf{2}],\mathbf{3})+_{\mathbf{R}}([\mathbf{5},\mathbf{3}],\mathbf{3})=([\mathbf{1}+_{\mathbf{Q}}\mathbf{5},\mathbf{2}+_{\mathbf{Q}}\mathbf{3}],\mathbf{3})=([\mathbf{6},\mathbf{5}],\mathbf{3})$ is valid since the left-hand side encodes $1+2\sqrt{3}+5+3\sqrt{3}$ and reduces to $6+5\sqrt{3}$ corresponding to the right-hand side.

Next we focus on multiplication.

Definition 6. For $(\mathbf{c}_n,m)$ and $(\mathbf{d}_n,m)$ from $\mathbf{R}$ we define:

$$(\mathbf{c}_n,m)\times_{\mathbf{R}}(\mathbf{d}_n,m) := (((\mathbf{c}_n,m)\cdot d_1)\gg 0)+_{\mathbf{R}}\cdots+_{\mathbf{R}}(((\mathbf{c}_n,m)\cdot d_n)\gg(n-1))$$

where

$$\begin{aligned}(\mathbf{c}_n,m)\cdot d &:= ([c_1\times_{\mathbf{Q}}d,\ldots,c_n\times_{\mathbf{Q}}d],m)\\ (\mathbf{c}_n,m)\gg 0 &:= (\mathbf{c}_n,m)\\ (\mathbf{c}_n,m)\gg(i+1) &:= ([m\times_{\mathbf{Q}}c_n,c_1,\ldots,c_{n-1}],m)\gg i\end{aligned}$$

Example 7. The computation

$$\begin{aligned}([\mathbf{1},\mathbf{2}],\mathbf{2})\times_{\mathbf{R}}([\mathbf{5},\mathbf{3}],\mathbf{2}) &= (([\mathbf{5},\mathbf{10}],\mathbf{2})\gg 0)+_{\mathbf{R}}(([\mathbf{3},\mathbf{6}],\mathbf{2})\gg 1)\\ &= ([\mathbf{5},\mathbf{10}],\mathbf{2})+_{\mathbf{R}}([\mathbf{12},\mathbf{3}],\mathbf{2})\\ &= ([\mathbf{17},\mathbf{13}],\mathbf{2})\end{aligned}$$

is justified by $(1+2\sqrt{2})\times(5+3\sqrt{2})=5+10\sqrt{2}+3\sqrt{2}+6\sqrt{2}\sqrt{2}=17+13\sqrt{2}$.

Next we encode comparisons. Here $\mathbf{c}_k=[c_1,\ldots,c_k]$ whenever $\mathbf{c}_{k+1}=[c_1,\ldots,c_k,c_{k+1}]$.

Definition 8. For $(\mathbf{c}_n,m)$ and $(\mathbf{d}_n,m)$ from $\mathbf{R}$ we define:

$$\begin{aligned}(\mathbf{c}_n,m)=_{\mathbf{R}}(\mathbf{d}_n,m) &:= (c_1=_{\mathbf{Q}}d_1)\wedge\cdots\wedge(c_n=_{\mathbf{Q}}d_n)\\ (\mathbf{c}_n,m)>_{\mathbf{R}}(\mathbf{d}_n,m) &:= (\mathbf{c}_n,m)>^0_{\mathbf{R}}(\mathbf{d}_n,m)\\ (\mathbf{c}_0,m)>^a_{\mathbf{R}}(\mathbf{d}_0,m) &:= a>_{\mathbf{Q}}0\\ (\mathbf{c}_{k+1},m)>^a_{\mathbf{R}}(\mathbf{d}_{k+1},m) &:= c_{k+1}+_{\mathbf{Q}}a\geqslant_{\mathbf{Q}}d_{k+1}\ \wedge\ (\mathbf{c}_k,m)>^{a+(c_{k+1}-d_{k+1})}_{\mathbf{R}}(\mathbf{d}_k,m)\end{aligned}$$

The next example shows that $=_{\mathbf{R}}$ and $>_{\mathbf{R}}$ only approximate $=_{\mathbb{R}}$ and $>_{\mathbb{R}}$, respectively. The intuition behind $>_{\mathbf{R}}$ is also demonstrated in the example.

Example 9. First we show that $=_{\mathbf{R}}$ is only an approximation of $=_{\mathbb{R}}$. The problem is that the representation of numbers need not be canonical, e.g., $([\mathbf{2},\mathbf{0}],\mathbf{4}) \neq_{\mathbf{R}} ([\mathbf{0},\mathbf{1}],\mathbf{4})$ since the lists are not equal componentwise but $2+0\sqrt{4} =_{\mathbb{R}} 0+1\sqrt{4}$. This poses problems when tests for equality appear at negative positions in Boolean formulas (cf. Section 4).

Next we show that $>_{\mathbf{R}}$ is only an approximation of $>_{\mathbb{R}}$ which is not only due to representation issues. Obviously $5+1\sqrt{2} \approx 6.41 >_{\mathbb{R}} 6.24 \approx 2+3\sqrt{2}$ holds. But we have $([\mathbf{5},\mathbf{1}],\mathbf{2}) \not>_{\mathbf{R}} ([\mathbf{2},\mathbf{3}],\mathbf{2})$ since $\mathbf{1} \not>_{\mathbf{Q}} \mathbf{3}$. On the other hand $([\mathbf{2},\mathbf{4}],\mathbf{2}) >_{\mathbf{R}} ([\mathbf{3},\mathbf{2}],\mathbf{2})$ which represents $2+4\sqrt{2} >_{\mathbb{R}} 3+2\sqrt{2}$. Since $4 \geqslant_{\mathbb{R}} 2$ also $4\sqrt{2} \geqslant_{\mathbb{R}} 2\sqrt{2}$. The leftover $2\sqrt{2}$ on the left-hand side is clearly greater than 2 which is used to obtain $2+2 >_{\mathbb{R}} 3$.

We note that $(\mathbf{c}_n, m) >_{\mathbf{R}} (\mathbf{d}_n, m)$ can be solved exactly if $n = 2$, i.e., when only square roots occur. The idea is to replace comparisons like $c_1 + c_2\sqrt{m} >_{\mathbb{R}} d_1 + d_2\sqrt{m}$ by $c_1 - d_1 + (c_2 - d_2)\sqrt{m} >_{\mathbb{R}} 0$. The latter can be encoded exactly based on a case analysis on the operands' signs while squaring the inequality.

In the final definition we exactly characterize $([c_1, c_2], m) >_{\mathbf{R}} 0$.

Definition 10. For the pair $([c_1, c_2], m)$ from $\mathbf{R}$ we define

$$\begin{aligned}([c_1,c_2],m) >_{\mathbf{R}} 0 \;:=\; & (c_1 \geqslant_{\mathbf{Q}} 0 \wedge c_2 >_{\mathbf{Q}} 0) \vee (c_1 >_{\mathbf{Q}} 0 \wedge c_2 \geqslant_{\mathbf{Q}} 0) \vee \\ & (c_1 \geqslant_{\mathbf{Q}} 0 \wedge c_2 <_{\mathbf{Q}} 0 \wedge \varphi) \vee (c_1 \leqslant_{\mathbf{Q}} 0 \wedge c_2 >_{\mathbf{Q}} 0 \wedge \chi)\end{aligned}$$

with $\varphi = c_1 \times_{\mathbf{Q}} c_1 >_{\mathbf{Q}} c_2 \times_{\mathbf{Q}} c_2 \times_{\mathbf{Q}} m$ and $\chi = c_1 \times_{\mathbf{Q}} c_1 <_{\mathbf{Q}} c_2 \times_{\mathbf{Q}} c_2 \times_{\mathbf{Q}} m$.

4 Soundness

In this section we show that our encodings are sound, i.e., that from a satisfying assignment for the encoding, a model for the original constraint can be inferred. By $[\![\cdot]\!]$ we denote the semantic evaluation function of the encodings. Since the encodings for $=_{\mathbb{R}}$ and $>_{\mathbb{R}}$ in general are not exact but only approximations, such constraints may not appear at negative positions in Boolean formulas, e.g. $\mathrm{a} = \mathrm{b} \to \bot$ is satisfiable if $\mathrm{a} = ([\mathbf{2},\mathbf{0}],\mathbf{4})$ and $\mathrm{b} = ([\mathbf{0},\mathbf{1}],\mathbf{4})$ but a and b both represent 2. We remark that none of the benchmarks from Section 5 contains comparisons at negative positions.

Addition: We have

$$\begin{aligned}[\![(\mathbf{c}_n, m) +_{\mathbf{R}} (\mathbf{d}_n, m)]\!] &= \sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}} + \sum_{1 \leqslant i \leqslant n} d_i \sqrt[n]{m^{i-1}} \\ &= \sum_{1 \leqslant i \leqslant n} (c_i + d_i) \sqrt[n]{m^{i-1}} \\ &= [\![([c_1 +_{\mathbf{Q}} d_1, \ldots, c_n +_{\mathbf{Q}} d_n], m)]\!]\end{aligned}$$

Subtraction: Analogous to addition.

Multiplication: First we show that $[\![(\mathbf{c}_n, m) \gg i]\!] = \left(\sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}}\right) \sqrt[n]{m^i}$ by induction on i. In the base case

$$[\![(\mathbf{c}_n, m) \gg 0]\!] = [\![(\mathbf{c}_n, m)]\!] = \sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}} = \left(\sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}}\right) \sqrt[n]{m^0}$$

In the inductive step

$$\begin{aligned}
&[\![(\mathbf{c}_n, m) \gg (i+1)]\!] \\
&\quad = [\![([m \times_{\mathbf{Q}} c_n, c_1, \ldots, c_{n-1}], m) \gg i]\!] = \left(m c_n \sqrt[n]{m^0} + \sum_{2 \leqslant j \leqslant n} c_{j-1} \sqrt[n]{m^{j-1}}\right) \sqrt[n]{m^i} \\
&\quad = \left(c_n \sqrt[n]{m^{n-1}} + \sum_{1 \leqslant j \leqslant n-1} c_j \sqrt[n]{m^{j-1}}\right) \sqrt[n]{m^{i+1}} = \left(\sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}}\right) \sqrt[n]{m^{i+1}}
\end{aligned}$$

Using this result multiplication is then justified by

$$\begin{aligned}
&[\![(\mathbf{c}_n, m) \times_{\mathbf{R}} (\mathbf{d}_n, m)]\!] \\
&\quad = \left(\sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}}\right) \times \left(\sum_{1 \leqslant i \leqslant n} d_i \sqrt[n]{m^{i-1}}\right) = \sum_{1 \leqslant i \leqslant n} \left(\sum_{1 \leqslant j \leqslant n} \left(c_j \sqrt[n]{m^{j-1}}\right)\right) \times d_i \sqrt[n]{m^{i-1}} \\
&\quad = \sum_{1 \leqslant i \leqslant n} \left(\left(\sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}}\right) \times d_i\right) \times \sqrt[n]{m^{i-1}} = [\![\sum_{1 \leqslant i \leqslant n} ((\mathbf{c}_n, m) \cdot d_i) \gg (i-1)]\!]
\end{aligned}$$

Equality: We have

$$\begin{aligned}
[\![(\mathbf{c}_n, m) =_{\mathbf{R}} (\mathbf{d}_n, m)]\!] &\iff \sum_{1 \leqslant j \leqslant n} c_j \sqrt[n]{m^{j-1}} = \sum_{1 \leqslant i \leqslant n} d_i \sqrt[n]{m^{i-1}} \\
&\iff \sum_{1 \leqslant j \leqslant n} (c_j - d_j) \sqrt[n]{m^{j-1}} = 0 \\
&\impliedby c_1 = d_1 \text{ and } \cdots \text{ and } c_n = d_n \\
&\iff [\![c_1 =_{\mathbf{Q}} d_1 \wedge \cdots \wedge c_n =_{\mathbf{Q}} d_n]\!]
\end{aligned}$$

Greater: First we give semantics to the $>^a_{\mathbf{R}}$ operator and show its soundness.

$$\begin{aligned}
[\![(\mathbf{c}_{k+1}, m) >^a_{\mathbf{R}} (\mathbf{d}_{k+1}, m)]\!] &\iff \sum_{1 \leqslant j \leqslant k+1} c_j \sqrt[n]{m^{j-1}} + a \sqrt[n]{m^k} >_{\mathbb{R}} \sum_{1 \leqslant i \leqslant k+1} d_i \sqrt[n]{m^{i-1}} \\
&\iff \sum_{1 \leqslant j \leqslant k} c_j \sqrt[n]{m^{j-1}} + (a + c_{k+1} - d_{k+1}) \sqrt[n]{m^k} >_{\mathbb{R}} \sum_{1 \leqslant i \leqslant k} d_i \sqrt[n]{m^{i-1}} \\
&\impliedby \sum_{1 \leqslant j \leqslant k} c_j \sqrt[n]{m^{j-1}} + (a + c_{k+1} - d_{k+1}) \sqrt[n]{m^{k-1}} >_{\mathbb{R}} \sum_{1 \leqslant i \leqslant k} d_i \sqrt[n]{m^{i-1}} \\
&\quad \text{and } a + c_{k+1} \geqslant d_{k+1} \\
&\iff [\![(\mathbf{c}_k, m) >^{a+c_{k+1}-d_{k+1}}_{\mathbf{R}} (\mathbf{d}_k, m)]\!]
\end{aligned}$$

Then soundness of $>_{\mathbf{R}}$, i.e., satisfiability of the encoding implies satisfiability of the constraint, follows.

Next we show soundness of the exact encoding (Definition 10), i.e., when $n = 2$. Note that $m > 0$ by assumption.

$$\begin{aligned}
&[\![([c_1, c_2], m) >_{\mathbf{R}} 0]\!] \\
&\quad \Longleftrightarrow c_1 + c_2\sqrt{m} >_{\mathbb{R}} 0 \\
&\quad \Longleftrightarrow c_1 >_{\mathbb{R}} -c_2\sqrt{m} \\
&\quad \Longleftrightarrow c_1 \geqslant_{\mathbb{Q}} 0 \text{ and } -c_2 <_{\mathbb{Q}} 0 \text{ or } c_1 >_{\mathbb{Q}} 0 \text{ and } -c_2 \leqslant_{\mathbb{Q}} 0 \text{ or} \\
&\qquad c_1 \geqslant_{\mathbb{Q}} 0 \text{ and } -c_2 >_{\mathbb{Q}} 0 \text{ and } c_1^2 >_{\mathbb{Q}} mc_2^2 \text{ or } c_1 \leqslant_{\mathbb{Q}} 0 \text{ and } -c_2 <_{\mathbb{Q}} 0 \text{ and } c_1^2 <_{\mathbb{Q}} mc_2^2 \\
&\quad \Longleftrightarrow c_1 \geqslant_{\mathbb{Q}} 0 \text{ and } c_2 >_{\mathbb{Q}} 0 \text{ or } c_1 >_{\mathbb{Q}} 0 \text{ and } c_2 \geqslant_{\mathbb{Q}} 0 \text{ or} \\
&\qquad c_1 \geqslant_{\mathbb{Q}} 0 \text{ and } c_2 <_{\mathbb{Q}} 0 \text{ and } c_1^2 >_{\mathbb{Q}} mc_2^2 \text{ or } c_1 \leqslant_{\mathbb{Q}} 0 \text{ and } c_2 >_{\mathbb{Q}} 0 \text{ and } c_1^2 <_{\mathbb{Q}} mc_2^2 \\
&\quad \Longleftrightarrow [\![(c_1 \geqslant_{\mathbf{Q}} 0 \wedge c_2 >_{\mathbf{Q}} 0) \vee (c_1 >_{\mathbf{Q}} 0 \wedge c_2 \geqslant_{\mathbf{Q}} 0) \vee \\
&\qquad (c_1 \geqslant_{\mathbf{Q}} 0 \wedge c_2 <_{\mathbf{Q}} 0 \wedge \varphi) \vee (c_1 \leqslant_{\mathbf{Q}} 0 \wedge c_2 >_{\mathbf{Q}} 0 \wedge \chi)]\!]
\end{aligned}$$

5 Assessment

We remark that our approach requires arithmetical variables to be based on (roots of) the same base. However, this does not immediately exclude solutions that can use different bases, as shown in the next example.

Example 11. Consider the constraint from Example 3 again. Our approach cannot find a model where $\alpha(\mathrm{a}_1) = \sqrt[3]{3}$ and $\alpha(\mathrm{a}_2) = \sqrt{2}$. However, it can determine the equivalent assignment $\alpha(\mathrm{a}_1) = \frac{1}{6}\sqrt[3]{648}$ and $\alpha(\mathrm{a}_2) = \frac{1}{18}\sqrt{648}$ since $648 = 2^3 \times 3^4$.

We implemented our approach for $n = 2$ in the SMT solver MiniSmt.[1] All tests have been performed on a single core of a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and on 64 GB of main memory. We considered the non-linear benchmarks matrix 1 and matrix 2 stemming from termination analysis mentioned in [3], since we are not aware of any specific test-beds for non-linear real arithmetic. In Table 1 we compare the approximated encoding of $>_{\mathbb{R}}$ from [3] (lpar) against the exact encoding of Definition 10 (scss).[2] The numbers in parentheses indicate how many bits have been used to encode integers and intermediate results, respectively. In the table "yes" refers to the number of systems that could be shown satisfiable, "avg" gives the average time in seconds for finding a model and "to" indicates for how many systems the execution was aborted since no result was produced within 60 seconds.

It is not surprising that the lpar-approach is faster than the scss-approach since the constraints for the latter involve more multiplications, which are costly. On the contrary the scss-approach supports reason-

Table 1: Evaluation of two benchmark families from termination analysis (1391 constraints each)

	lpar(2,4)	scss(2,4)	lpar(3,4)	scss(3,4)	lpar(3,5)	scss(3,5)
	yes/ avg/ to	yes/ avg/ to	yes/ avg/ to	yes/ avg/ to	yes/ avg/ to	yes/ avg/ to
matrix 1	308/1.43/ 27	308/2.99/ 62	306/1.92/ 44	303/ 4.28/ 99	311/ 3.72/123	294/ 5.50/175
matrix 2	296/6.92/315	276/8.31/391	286/7.79/344	264/11.58/459	237/10.95/543	221/13.70/637

[1] `http://cl-informatik.uibk.ac.at/software/minismt`

[2] Full details available from `http://cl-informatik.uibk.ac.at/software/minismt/experiments/scss`.

ing about numbers that involve square roots different from $\sqrt{2}$. However, the test-beds we considered do not seem to require such numbers.

6 Conclusion

Recently in [3] an approach that reduces satisfiability of non-linear real arithmetic to non-linear rational/integer arithmetic has been presented. In [3] real variables may take values of the shape $c + \sqrt{2}d$ and $>_{\mathbb{R}}$ is only approximated. In this paper we generalized the approach to real algebraic numbers of the more general shape $\sum_{1 \leqslant i \leqslant n} c_i \sqrt[n]{m^{i-1}}$ and presented suitable encodings in non-linear integer/rational arithmetic. Furthermore for the setting of [3], where always $n = 2$, we formulated an exact encoding of $>_{\mathbb{R}}$.

Acknowledgments. We thank Friedrich Neurauter for helpful discussion.

References

[1] Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Proc. 2nd International Conference on Automata Theory and Formal Languages. LNCS, vol. 33, pp. 134–183 (1975)

[2] Tarski, A.: A Decision Method for Elementary Algebra and Geometry. 2nd edn. University of California Press, Berkeley (1957)

[3] Zankl, H., Middeldorp, A.: Satisfiability of non-linear (ir)rational arithmetic. In: LPAR-16. LNCS (LNAI). (2010). To appear. Available from `http://cl-informatik.uibk.ac.at/~hzankl/new/publications`.

Automatic Correction of Firewall Mis-configurations

Nihel Ben Youssef Ben Souayeh and Adel Bouhoula
Higher School of Communication of Tunis, SUP'COM
Tunis, Tunisia
nihel.benyoussef@gmail.com
adel.bouhoula@supcom.rnu.tn

Abstract

Firewalls are the most widely adopted technology for protecting private networks. However, it has been observed that most firewalls on Internet have been highly error prone. A firewall policy error stems usually from policy changes. It creates serious security holes blocking some legitimate packets or allowing malicious attackers to sneak into private areas. In this paper, we propose an automatic approach to correct firewall mis-configurations with respect to a security policy given in a high level declarative language. We show that our method is both correct and complete. Finally, it has been implemented in a prototype. Experiments conducted on relevant case studies demonstrated the efficiency and the scalability of our approach.

1 Introduction

A firewall is a widely deployed device for improving the security of enterprise networks by accepting or discarding packets according to its configuration. However, configuring a firewall is an error-prone task, even for an experienced administrator. According to the study undertaken by Wool [12], most firewalls on Internet are plagued with policy errors. The main firewall configuration constraint is that the filtering rules of a *firewall configuration* FC file are treated in the order in which they are read in the configuration file, in a *switch-case* fashion. For instance, if two filtering rules associate different actions to the same traffic, then only the rule with the lower order is really applied. This is in contrast with the *security policies* (SP), used to express global security requirements and which is a set of rules considered without order. In this case, the action taken, for the flow under consideration, can be the one of the non executed rule. Several methods have been proposed in [11, 2] and [13] for the detection of such inter-rule conflicts in FC. These studies are limited to the problem conflict avoidance, and do not consider the more general problem of verifying whether a firewall reacts correctly with respect to a given SP. Indeed, when some inconsistencies in firewall rules are discovered by such methods, one can not make concrete judgements on the impact of such conflicts on the firewall's behavior. In this case, the resolution measures are leaved to the administrator.

Solutions are studied in [9], [6], and [10] for the analysis of the firewall's behavior. These methods consist on developing verification tools through which a user may send queries to judge whether inconsistencies exist with the security policy he/she wants to establish. Such interactive solutions are helpful for the administrator. However, no information are given about where and how to correct the firewall configuration once discrepancies are detected.

In [14], the author addresses the problem of checking a firewall configuration according to a given property. The proposed verification process is interactive: Once it fails, the user is warned. However, he/she should make its own resolution measures which seems to be a tedious task since no specific key elements are given to help the correction of FC.

In a previous work [5], we proposed a formal approach that takes as inputs the firewall configuration FC and the security policy SP that should be established. The method consists on checking, first, the coherence of the security policy and then, the soundness and the completeness of the firewall configuration FC with respect to the security policy *SP*.

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 25-34

If the soundness property is not verified, the approach returns the first flawed filtering rule. And, if the completeness checking fails, the method outputs a model of a packet which is considered by a specific security directive and ommited by the firewall configuration FC. These key elements should help the administrator to determine where exactly the mis-configurations are situated. But, even in this case, the administrator should intervene to complete the correction of the firewall configuration. This task seems, usually, quite difficult when it is performed manually. This is due to the complex overlapping domains often existing within the firewall filtering rules and the difficulty of considering, simultaneously, the entire security requirements during the correction process.

In this paper, we present an automatic resolution method of a misconfigured firewall that gives necessary and sufficient answers on how to correct the filtering rules by considering the key elements given as results in [5]. Our method can be used either in order to validate an existing FC with respect to a given SP or in order to assist the updates of FC, since adding, altering or deleting a filtering rule may change the semantic of the firewall policy.

The remainder of this paper is organized as follows. Section 2 settles the definition of the problems addressed in the paper. In Section 3, we introduce our resolution method of firewall mis-configurations and we prove its soundness and its completeness. In Section 4,we present an automatic resolution tool that we have implemented based on our method and some experiments on an example. Finally, section 5 overviews the performance of our proposed method.

2 Types of Firewall Mis-configurations

The main goal of this work consists in resolving mis-configurations of a FC with respect to a given SP. We distinguish two cases: when FC is not sound according to SP and when FC is not complete according to SP. In this section, we define formally these notions.

We consider a finite *domain* $\mathscr{P}$ containing all the headers of packets possibly incoming to or outgoing from a network. A *firewall configuration* (FC) is a finite sequence of *filtering rules* of the form $FC = (r_i \Rightarrow A_i)_{0 \leq i < n}$. Each precondition r_i of a rule defines a filter for packets of $\mathscr{P}$. The structure of r_i is described later in Section 4. Until then, we just consider a function *dom* mapping each r_i into the subset of $\mathscr{P}$ of filtered packets. Each right member A_i of a rule of FC is an *action* defining the behaviour of the firewall on filtered packets: $A_i \in \{accept, deny\}$. This model describes a generic form of FC which are used by most firewall products such as CISCO, *Access Control List*, `IPTABLES`, `IPCHAINS` and *Check Point Firewall...*

A *security policy* (SP) is a set SP of formulae defining whether packets are accepted or denied. It is presented as a finite set of *directives*: $SP = \{c_i \Rightarrow A_i | e_i \mid 1 \leq i \leq m\}$. Each directive can be simple or complex. A simple directive describes whether some traffic destined to one or more services that are required by one or more sources and given by one or more destinations (as described by the condition c_i) must be accepted or refused (according to $A_i \in \{accept, deny\}$). A complex directive is basically a simple directive with some additional exceptions defined in e_i.

Consider the following security policy whose directives are respectively simple and complex.

- *The sub network LAN_A has the right to access to the web service provided by the machine B .*
- *The machine A' belonging to LAN_A, has the right to access to all the services provided by the machine B except to its web service.*

The domain of SP is partitioned into $dom(SP) = \bigcup_{A\in\{accept,deny\}} SP_A$: the set of accepted and denied packets according to SP. SP is called *consistent* if $SP_{accept} \cap SP_{deny} = \emptyset$. We can, therefore, note that the above security policy is not consistent since the two directives have contradictory actions for the same traffic from the machine A' to the web service of the machine B. In our previous work [5], we proposed solutions to check the coherence of SP. For the remainder of this paper, we consider that SP is consistent. A FC is *not sound* with respect to a SP if there exists a packet p such that the action undertaken by the firewall for p, (i.e. the action of the first filtering rule matching p) is not the same as the one defined by the SP.

Definition 1 (soundness)**.** *FC is not sound with respect to SP iff* $\exists\ p \in \mathscr{P}$*, such that there exists a rule* $r_i \Rightarrow A_i$ *in FC with* $p \in dom(r_i)$ *and for all* $j < i$, $p \notin dom(r_j)$ *and* $p \in SP_{A_i}$.

A FC is *not complete* with respect to a SP if there exists a paket p such that the action defined by the SP for p is not really undertaken by the firewall.

Definition 2 (completeness)**.** *FC is not complete with respect to SP iff there exists* $p \in \mathscr{P}$ *and* $A \in \{accept, deny\}$*, such that* $p \in SP_A$ *and there exists no rule* $r_i \Rightarrow A$ *in FC such that* $p \in dom(r_i) \setminus \bigcup_{j<i} dom(r_j)$.

3 Our Solution

In [5], we presented a tool that generates the first flawed filtering rule when FC is not sound according to SP and outputs the directives of SP that handle some packets not treated by the filtering rules of FC. The above results are helpful but not sufficient to complete the resolution procedure. We propose, in this case: First, to determine the entire set of packets, designed as distinct filtering rules, causing conflict between FC and SP. And second, to add them in the appropriate order in FC: Above the flawed filtering rule to be considered by priority when FC is not sound according to SP or at the end of FC to completely implement SP. We note that the set of packets to determine in both situations is a result of complement operations. In the next sections, we develop in details this notion.

3.1 Basic complement operation

We refer to a security rule as, either a filtering rule, or a security directive. In this section, we define the set-theoretic difference of the domain of a security rule r_i and that of r_j. We consider afterwards $[r_i]$ as the domain of r_i. Let r_i and r_j be two filtering rules. We denote by ξ_{ik} the field's domain k of the rule i. For example, in table 2 , $\xi_{23} ::= [1\ \ 4]$.

Definition 3. *We define the complement* ζ_{ij} *of* $[r_i]$ *in* $[r_j]$ *as the union of sets of packets* $\mathscr{S}_{n(1\leq n\leq 4)}$ *presented in table 3.1. Each set* $\mathscr{S}_n$ *should follow the following conditions.*

$$\begin{cases} \forall k, \xi_{ik} \setminus \xi_{jk} \neq \varnothing \\ \forall k, \xi_{ik} \cap \xi_{jk} \neq \varnothing \end{cases}$$

Theorem 1 (correctness)**.** ζ_{ij} *is a solution of the complement* $[r_i] \setminus [r_j]$.

Proof. In each set S_n, there exists one field k that corresponds to a complement set $\xi_{ik} \setminus \xi_{jk}$ which is a sufficient condition to consider each $\mathscr{S}_n$ as a set of packets belonging to r_i and not to r_j. So, $\forall n, \mathscr{S}_n \subseteq [r_i] \setminus [r_j]$ holds. Since, $\zeta_{ij} = \bigcup_{1\leq n\leq 4} \mathscr{S}_n$, it follows that $\zeta_{ij} \subseteq [r_i] \setminus [r_j]$. For example, in table 3, $\mathscr{S}_1$ is the set of packets included in r_1 having as source address $[1\ \ 1][6\ \ 9]$, results of $\xi_{12} \setminus \xi_{21}$. Therefore, we can easly deduce that these packets do not belong to r_2.

$\mathscr{S}_1$	$\xi_{i1} \setminus \xi_{j1}$	ξ_{i2}	ξ_{i3}	ξ_{i4}
$\mathscr{S}_2$	$\xi_{i1} \cap \xi_{j1}$	$\xi_{i2} \setminus \xi_{j2}$	ξ_{i3}	ξ_{i4}
$\mathscr{S}_3$	$\xi_{i1} \cap \xi_{j1}$	$\xi_{i2} \cap \xi_{j2}$	$\xi_{i3} \setminus \xi_{j3}$	ξ_{i4}
$\mathscr{S}_4$	$\xi_{i1} \cap \xi_{j1}$	$\xi_{i2} \cap \xi_{j2}$	$\xi_{i3} \cap \xi_{j3}$	$\xi_{i4} \setminus \xi_{j4}$

Table 1: Complement of $[r_i]$ in $[r_j]$

	src_adr	*dst_adr*	*dst_port*	*protocol*
$[r_1]$	[1 2][6 9]	[8 10]	[3 8]	[0 1]
$[r_2]$	[2 3]	[1 9]	[1 4]	[0 0]

Table 2: Example of two filtering rules

Theorem 2 (completeness)**.** *The complement* $[r_i] \setminus [r_j]$ *is defined by* ζ_{ij}.

Proof. To proof that ζ_{ij} represents the complete set of packets corresponding to $[r_i] \setminus [r_j]$, we prove that the formulae $[r_i] \setminus \zeta_{ij} ::= [r_i] \cap [r_j]$ holds.$[r_i] \setminus \zeta_{ij}$ is equivalent to $r_i \setminus \bigcup_{1 \leq n \leq 4} S_n$ or also to $\bigcap_{1 \leq n \leq 4}([r_i] \setminus \mathscr{S}_n)$. According to *Definition 3*, we deduce the following complement operations:

$$[r_i] \setminus S_1 :: \quad \xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \;\; \xi_{i3} \;\; \xi_{i4}$$

$$[r_i] \setminus S_2 :: \quad [\xi_{i1} \setminus \xi_{j1} \;\; \xi_{i2} \;\; \xi_{i3} \;\; \xi_{i4}] \cup [\xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \cap \xi_{j2} \;\; \xi_{i3} \;\; \xi_{i4}]$$

$$\begin{aligned}[r_i] \setminus S_3 :: \quad & [\xi_{i1} \setminus \xi_{j1} \;\; \xi_{i2} \;\; \xi_{i3} \;\; \xi_{i4}] \cup [\xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \setminus \xi_{j2} \;\; \xi_{i3} \;\; \xi_{i4}] \\ & \cup [\xi i1 \cap \xi_{j1} \;\; \xi_{i2} \cap \xi_{j2} \;\; \xi_{i3} \cap \xi_{j3} \;\; \xi_{i4}]\end{aligned}$$

$$\begin{aligned}[r_i] \setminus S_4 :: \quad & [\xi_{i1} \setminus \xi_{j1} \;\; \xi_{i2} \;\; \xi_{i3} \;\; \xi_{i4}] \cup [\xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \setminus \xi_{j2} \;\; \xi_{i3} \;\; \xi_{i4}] \\ & \cup [\xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \cap \xi_{j2} \;\; \xi_{i3} \setminus \xi_{j3} \;\; \xi_{i4}] \\ & \cup [\xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \cap \xi_{j2} \;\; \xi_{i3} \cap \xi_{j3} \;\; \xi_{i4} \cap \xi_{j4}]\end{aligned}$$

By performing successive operations of intersection $\bigcap_{1 \leq n \leq 4}([r_i] \setminus \mathscr{S}_n)$, we deduce that $[r_i] \setminus \zeta_{ij} ::= [\xi_{i1} \cap \xi_{j1} \;\; \xi_{i2} \cap \xi_{j2} \;\; \xi_{i3} \cap \xi_{j3} \;\; \xi_{i4} \cap \xi_{j4}]$. Hence, $[r_i] \setminus \zeta_{ij} ::= [r_i] \cap [r_j]$.

3.2 Proposed Algorithm

As discussed in 3. , we propose an automatic method that determines the entire set of filtering rules causing conflict between FC and SP. We recall that in both situations (*non soundness* and *non completeness*), our goal shall perform the following complement operation: $\gamma_N ::= [r_i] \setminus \bigcup_{1 \leq j \leq N} [r_j]$. Its semantics differs, depending on the intended verification procedure. When we aim to correct a FC which is not sound according to SP, $[r_i]$ is the domain of the first flawed filtering rule and $\bigcup_{1 \leq j \leq N} r_j$ is the union of sets $\bigcup_{k<i}[r_k]$ and $\bigcup_{1 \leq l \leq m}[r_l]$. The former represents the accumulated domain of the previous rules of r_i and the later coresponds to the m security directives imposing the same action according to *SP*. The number N in this case is $m+i-1$. When we intend to correct a failure of the completeness verification, $[r_i]$ is the domain of a flawed security directive and $\bigcup_{1 \leq j \leq N}[r_j]$ represents the accumulated domain of the entire set N of filtering rules in *FC*. Our proposed algorithm is presented in Figure 1. It generates the final set γ_N by performing the set operatiosn $\bigcup_i(\gamma_{N-1}[i] \setminus [r_N])$.

Theorem 3 (correcteness & completeness)**.** *Our algorithm is both sound and complete. It is necessary and sufficient to prove that* γ_N *is the solution of the complement set operation* $[r_i] \setminus \bigcup_{1 \leq j \leq N}[r_j]$.

	src_adr	*dst_adr*	*dst_port*	*protocol*
S_1	[1 1][6 9]	[8 10]	[3 8]	[0 1]
S_2	[2 2]	[10 10]	[3 8]	[0 1]
S_3	[2 2]	[8 9]	[5 8]	[0 1]
S_4	[2 2]	[8 9]	[3 4]	[1 1]

Table 3: Complement Set C_{12} of the rules in table 2

Proof. The iterative part of our algorithm consists on performing forall $j > 1$, the complement operation $\bigcup_i(\gamma_{j-1}[i] \setminus [r_j])$. This expression is equivalent to $\bigcup_i(\gamma_{j-1}[i] \cap [\overline{r_j}])$ which is also $(\bigcup_i \gamma_{j-1}[i]) \cap [\overline{r_j}]$ or $(\bigcup_i \gamma_{j-1}[i]) \setminus [r_j]$. This equivalence shows that our algorithm generates γ_N by performing the following set operations: $((([r_i] \setminus [r_1]) \setminus [r_2]) \setminus [r_3] \ldots) \setminus [r_N]$. Its basic recursive part has the form $([r_i] \setminus [r_a]) \setminus [r_b]$ which is equivalent to $([r_i] \setminus [r_a]) \cap ([r_i] \setminus [r_b])$, or also $[r_i] \setminus ([r_a] \cup [r_b])$. Thus, step by step, we perform the final target: $[r_i] \setminus \bigcup_{1 \leq j \leq N}[r_j]$. Hence, our algorithm is both sound and complete.

Algorithm
Inputs
$r_i ::$ *A flawed security rule*
$\{r_j | (1 \leqslant j \leqslant N)\} ::$ *set of N security rules.*
Output:
$\gamma_N ::$ *Set of rules causing conflict between the firewall configuration and the security policy*
Begin
set $\gamma_0[1] \leftarrow [r_j]$
for *each rule* $r_j(j := 1 -> N)$ **do**
  set $\gamma_j \leftarrow \varnothing$
  for *each* $c(c := 1 -> |\gamma_{j-1}|)$ **do**
    $\gamma_j \leftarrow \gamma_j \cup (\gamma_{j-1}[c] \setminus [r_j])$
  end do
end do
return γ_N
End.

Figure 1: Generating the set of packets causing conflict

4 Automatic Resolution tool

The first input of our resolution tool is a set of firewall filtering rules. Each filtering rule $r_i \Rightarrow A$ is defined by a priority order *i* and composed of the following main fields: the source ip , the destination ip, the protocol and the destination port. The source and destination fields correspond to one or more machines identified by an IPv4 address and a mask coded both on 4 bytes. the protocol field refers to a transport layer protocol such as `TCP` or `UDP` and coded on 1 byte. Finally, the destination port is a specific number coded in 2 bytes. Both protocol and destination port define a service. For example, the web service is respresented by the protocol `TCP` and the port `80` and the DNS service by the protocol `UDP` and the port `53`.

Consider for instance the following rule r_i . This rule accepts `TCP` flow, coming from the source network `192.168.2.0/24` and reaching the machine `10.1.2.3` for any destination port belonging to the subrange `[0 - 2`^{16}`]`.

In this paper, we represent a field as a subrange `[a b]` that denotes the set of possible values `x` between `a` and `b`. That is, `a` $\leq$ `x` $\leq$ `b` . Therefore, the domain of the above rule r_i can be represented as follows.

	src_adr	*dst_adr*	*dst_port*	*protocol*	*action*
r_i	192.168.2.0/24	10.1.2.3	*	tcp	accept

	src_adr	*dst_adr*	*dst_port*	*protocol*
r_i	[192.168.2.0 192.168.2.255]	[10.1.2.3 10.1.2.3]	[0 65535]	[tcp tcp]

For the sake of readability, we consider that the source and destination fields $\in$ `[1 100]`, the protocol field $\in$ `[0 1]`, `0` refers to `TCP` and `1` refers to `UDP` and the destination port $\in$ `[1 6]`.

In order to illustrate the resolution procedure proposed, we have chosen to apply our method to a case study of a corporate network represented in Figure 2. The network is divided into three zones delineated by branches of a firewall whose initial configuration *FC* corresponds to the rules in Table 4.

The security policy *SP* that should be respected contains the following directives.

sp_1 : `net_1`, except the machine `admin`, has not the right to access to `net_2` .

sp_2 : `net_3` has the right to access to `net_1` except to the machine `admin`.

sp_3 : `net_3` has the right to access to `net_2` except the machine `B_1` which can only access to the DNS service (port : 6 (53), protocol : 1 (udp)) provided by the `DNS server` .

In the following, we note that sp'_i is the exception of the security directive sp_i. Our goal is, first, to

	src_adr	*dst_adr*	*dst_port*	*protocol*	*action*
r_1	[80 90]	[10 10]	*	*	deny
r_2	[80 90]	[5 40]	*	*	accept
r_3	[1 9] [12 40]	[60 70]	*	*	deny
r_4	[80 90]	[1 5] [11 50]	*	*	accept
r_5	[80 90]	[60 70]	*	*	accept
r_6	[11 50]	[60 70]	*	*	deny
r_7	[1 20]	[60 70]	[1 5]	*	accept

Table 4: Firewall Configuration to be corrected

verify that the configuration *FC* is conform to the security policy *SP* by checking the soundness and the completeness properties and second, once discrepancies detected, to determine the set of packets causing conflict and place them to the appropriate order as discussed in section 3. We can note, first, that the above security policy is consistent. This means that no contradictions exist within the security directives. This is an essential condition to guarantee the soundness of our results.

4.0.1 Soundness Verification

The result of the soundness verification obtained is displayed in Figure 3. The outcome shows that the firewall configuration *FC* is not sound with respect to the security policy *SP*, i.e. that there exists some packets that will undergo an action different of that imposed by the security policy. It indicates also that r_5 is the first rule that causes this discrepancy precisely with the exception of directive sp_3.

Indeed, the model returned corresponds to a packet accepted by the firewall through the rule r_5 while it should be refused according to the third directive of the security policy.

We proceed to our resolution algorithm to determine the entire set of packets that will be accepted by r_5 and should be denied according to the security policy. The results are shown in figure 4. This conflict can be resolved by adding the obtained rules immediately preceding the rule r_5, which allows to implement the third directive exception precised by the given model. Table 5 presents this modification.

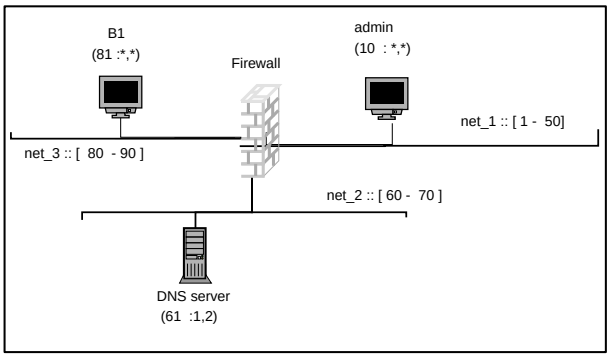


Figure 2: Network Topology Example

```
File  Edit  View  Terminal  Help
root@nanoupc:~/Desktop/yices-1.0.21/bin# ./yices -e soundness_check_2.ys

sat
(= ips1 81)
(= ipd1 62)
(= port 2)
(= protocol 1)
(= (sp 1) false)
(= (sp' 1) false)
(= (sp 2) false)
(= (sp' 2) false)
(= (sp 3) true)
(= (sp' 3) true)
(= (r 1) false)
(= (r 2) false)
(= (r 3) false)
(= (r 4) false)
(= (r 5) true)
(= (r 6) false)
(= (r 7) false)
```

Figure 3: Soundness Check

```
File  Edit  View  Terminal  Help
 ================================================================================
                              First wrong rule ::  r_5 ::
 ================================================================================

                          Set of packets Causing conflict with SP !
 ================================================================================

 [ 81  -  81 ]  | [ 60  -  60 ]  [ 62  -  70 ]  | [ 1  -  6 ]  | [ 0  -  1 ]

 [ 81  -  81 ]  | [ 61  -  61 ]  | [ 1  -  1 ]  [ 3  -  6 ]  | [ 0  -  1 ]

 [ 81  -  81 ]  | [ 61  -  61 ]  | [ 2  -  2 ]  | [ 0  -  0 ]
Time elapsed : 0.660000  s
 root@nanoupc:~/Desktop/yices-1.0.21#
```

Figure 4: Packets accepted by r_5 and should be denied according to *SP*

	src_adr	*dst_adr*	*dst_port*	*protocol*	*action*
r_1	[80 90]	[10 10]	*	*	deny
r_2	[80 90]	[5 40]	*	*	accept
r_3	[1 9] [12 40]	[60 70]	*	*	deny
r_4	[80 90]	[1 5] [11 50]	*	*	accept
r_5	[81 81]	[60 60] [62 70]	*	*	deny
r_6	[81 81]	[61 61]	[1 1] [3 6]	*	deny
r_7	[81 81]	[61 61]	[2 2]	[0 0]	deny
r_8	[80 90]	[60 70]	*	*	accept
r_9	[11 50]	[60 70]	*	*	deny
r_{10}	[1 20]	[60 70]	[1 5]	*	accept

Table 5: Firewall Configuration After correcting r_5

4.0.2 Completeness Verification

After that the soundness property has been established, we proceed to the verification of the completeness of the firewall configuration. We obtained the result displayed in Figure 5.

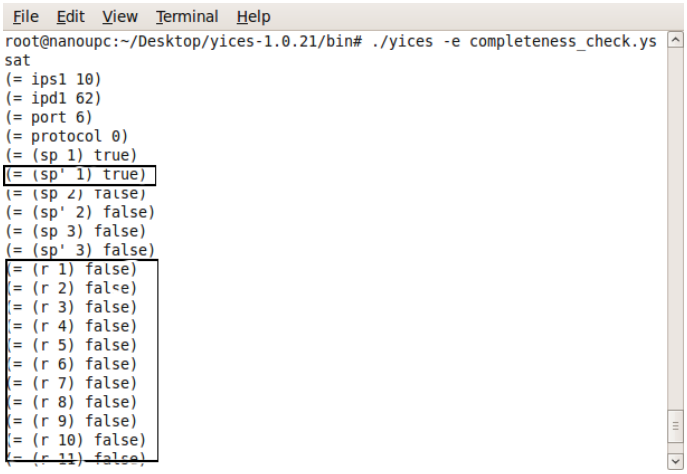


Figure 5: Completeness Check

According to this outcome, the configuration *FC* is not complete with respect to the security policy *SP*: some packets handled by the security policy are not treated by the filtering rules. Indeed, the model

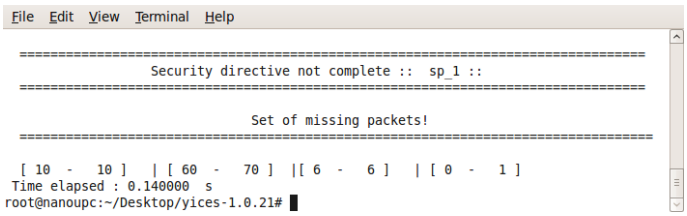


Figure 6: Set of packets considered by sp_1 and not treated by FC

returned corresponds to a packet considered by the exception of directive sp_1, and not treated by *FC*, precisely by the rules r_3, r_9 and r_{10}. To determine the entire set of the missing packets, we proceed to our resolution algorithm. The results are shown in figure 6. This conflict can be resolved by adding the obtained rules at the end of *FC*, which allows to implement the first directive exception precised by the given model. Table 6 presents this modification.

	src_adr	*dst_adr*	*dst_port*	*protocol*	*action*
r_1	[80 90]	[10 10]	*	*	deny
r_2	[80 90]	[5 40]	*	*	accept
r_3	[1 9] [12 40]	[60 70]	*	*	deny
r_4	[80 90]	[1 5] [11 50]	*	*	accept
r_5	[81 81]	[60 60] [62 70]	*	*	deny
r_6	[81 81]	[61 61]	[1 1] [3 6]	*	deny
r_7	[81 81]	[61 61]	[2 2]	[0 0]	deny
r_8	[80 90]	[60 70]	*	*	accept
r_9	[11 50]	[60 70]	*	*	deny
r_{10}	[1 20]	[60 70]	[1 5]	*	accept
r_{11}	[10 10]	[60 70]	[6 6]	*	accept

Table 6: A sound and complete Firewall Configuration

We note that our verification tool validates the properties after the modifications taken in Sections 4.0.1 and 4.0.2.

5 Performance and scalability

In order to better assess the efficiency of our tool, we generated synthetic firewalls based on the characteristics of real-life firewalls. The choice of generating such synthetic firewalls is due to the confidential access to some real-life firewalls that we have tried to check. Each filtering rule is composed by four fields: source IPV4 address, destination IPV4 address, destination port $\in [0-2^{16}]$ and transport port $\in [tcp, udp]$. The filtering rules originate from a specific security policy but could be not correct or not complete as shown in previous sections. First, we set the number of security directives to 4, then to 8 and finally to 15. The experimental tests were conducted on an Intel Dual core 1.6 GHz with 1 Gbyte of RAM. .We consider the average processing time of the resolution procedure. Figure 7 shows the performance evaluation of our method The worst case complexity of our algorithm is $O(n^r)$, with r, the number of rules to compute and n, the number of distinct sets of packets resulting of a basic complement operation. This number is bounded by 4 as seen in section 3.1. Thus, for a firewall where $r = 50$ (average number of rules in real-life firewalls [8])and $n = 4$ and with 15 security directives, one expects billions of solutions and a large processing time. However, in this case, only 11 distinct sets of packets are generated in 2.83 *seconds*. We note that these practical results are too much smaller than the expected ones. This can be explained as follows: Usually, the firewall configurations originate from a specific security policy. This implies that the filtering rules tend to be clustered in small groups instead of being randomly distributed. Consequently, numerous intermediate complement operations, in our algorithm, are very likely to be empty. Therefore, the processing time should be faster. In another side, we can intuitively note that when n gets larger, our complement s operation in the form $A \setminus \bigcup_n B_n$ would give us a solution smaller or equal in terms of number of packets. For instance, the set $A \setminus (B_1 \cup B_2)$ should be smaller or equal to $A \setminus B_1$.

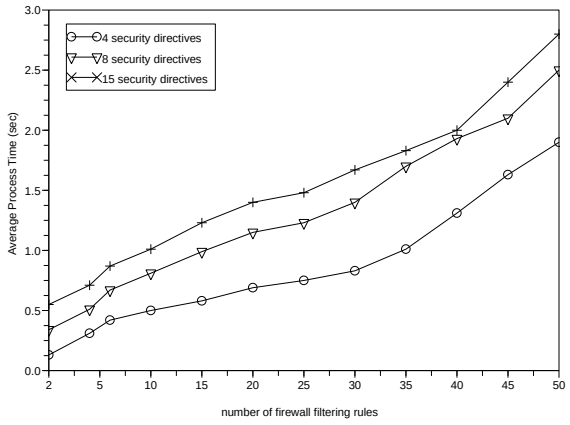


Figure 7: Processing Time Evaluation

6 Conclusion

In this paper, we proposed an automatic method to resolve firewall mis-configurations with respect to a given security policy. We distinguished two cases of firewall mis-configurations: First, when the firewall configuration is not sound according to the security policy and second, when the firewall configuration is not complete according to the security policy. We proved that our method is both sound and com-

plete. Finally, our method has been implemented offering full-coverage of possible IP packets used in production environments The experimental results obtained are very promising.

References

[1] T. Abbes, A. Bouhoula, and M. Rusinowitch. Inference system for detecting firewall filtering rules anomalies. In *Proc. of the 23rd annual ACM Symp. on Applied Computing*, 2008.

[2] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *IEEE/IFIP Integrated Management, IM'2003*, 2003.

[3] F. Cupens, N. Cuppens-Boulahia, T. Sans, and A. Miege. A formal approach to specify and deploy a network security policy. In *In Second Workshop on Formal Aspects in Security and Trust*, pages 203-218, 2004.

[4] B. Dutertre and L. Moura. The yices smt solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[5] N. Ben Youssef, A. Bouhoula and F. Jacquemard. Automatic Verification of Conformance of Firewall Configurations to Security Policies. In *Proc. of the 14th IEEE Symposium on Computers and Communications (ISCC'09).*, 2009.

[6] P. Eronen and J. Zitting. An expert system for analyzing firewall rules. In *Proc. of 6th Nordic Workshop on Secure IT Systems*, 2001.

[7] M. Gouda and A. X. Liu. Firewall design: consistency, completeness and compactness. *In Proc. of the 24th IEEE Int. Conf. on Distributed Computing Systems*, 2004.

[8] P. Gupta: Algorithms for Routing Lookups and Packet Classification. PhD thesis, Stanford University, 2000.

[9] S. Hazelhusrt. Algorithms for analyzing firewall and router access lists. TR, Univ. of the Witwatersrand, 1999.

[10] A. X. Lui, M. Gouda, H. Ma, and A. Ngu. Firewall policy queries. In *Proc. of the IEEE transactions on parallel and diatributed systems*, 2009.

[11] C. Pornavalai and T. Chomsiri. Firewall policy analyzing by relational algebra. In *The 2004 Int. Technical Conf. on Circuits/Systems, Computers and Communications*, 2004.

[12] A. Wool. A quantitative study of firewall configuration errors. In *IEEE Computer*, 37(6), 2004.

[13] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proc. of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[14] A. Liu . Formal Verification of Firewall policy. In *Proc of the IEEE Conference on Communications* ,2008.

Integrating Local Computation Models by Refinement ⋆

Dominique Méry[1], Mohamed Mosbah[2], and Mohamed Tounsi[2]

[1] Loria, Université Henri Poincaré Nancy 1 France
[2] LaBRI, Université Bordeaux 1 Talence France

Abstract. Our approach is directly related to the design of *correct-by-construction* programs. The main idea relies upon the development of *structured* programs following a top/-down approach, which is clearly well known in earlier works of Dijkstra [1, 2], and to use the refinement for controlling the correctness of the resulting program. It relies on a more fundamental question related to the notion of *problem to solve*. The methodology can be based on *incremental proof-based* developments. However, the link between the problem and the first model remains to be expressed and the refinement is a real help to justify in a very progressive way the choices of design. We propose in this work a framework combining local computations and refinement to prove the correctness of a large class of distributed algorithms.

Key words: Proof-based pattern, Event-B, Local computation, Distributed algorithm.

1 Introduction

1.1 Proof-based Development

Proof-based development methods [2–4] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete events. The relationship between two successive models in this sequence is that of *refinement* [3, 4]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination. A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [5]. At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations.

The goal of a event B development is to obtain a *proved model* and to implement the correctness-by-construction [6] paradigm. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process, while allowing for traceability of requirements.

B models rarely need to make assumptions about the *size* of a system being modeled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [7]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity. These solutions are, de facto, supporting the use of patterns which are stating general proof-based developments and are validating the expression *proof-based patterns*. Now, let us browse ideas on proof-based patterns and patterns-based development.

⋆ This work has been supported by the French research agency ANR under number **RIMEL** ANR-06-SETIN-015

1.2 Patterns for proof-based developments

Patterns and design patterns [8] provide a very convenient help in the design of object-oriented software. Originally, they were borrowed from architectures practices [9] and recently, J.-R. Abrial [10] suggested the introduction of a kind of patterns for the proof-based development. The action/reaction patterns have been applied to the press case study by J.-R. Abrial and they improve the proof process. Another pattern called re-usability pattern, has been suggested by Abrial,Cansell and Méry [11, 12] and applied to the development of voting systems [13] and greedy algorithms [12]. In the context of time-sensitive systems, J. Rehm et all [14] has identified patterns for introducing time in Event B models and for expressing time constraints in Event B models of the IEEE 1394 tree identification protocol. Dealing with access control-based problems, Benaissa et all [15] have proposed a general pattern for handling access control policies and for integrating access control policies into systems. Clearly, a growing activity on modeling patterns has started and is addressing many kinds of case studies and domains of problems. No classification is yet given and there is no real repository of patterns validated for a specific modeling language based on proof-based transformations.

In fact, the goal of patterns is to improve the modeling process based on proofs, like Event B, and the modeling process deals with different languages, when one considers the triptych of D. Bjoerner [16–18]- $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$. It means that in our case, we are using a programming language and a modeling language for analyzing the problem. The domain $\mathcal{D}$ is considering properties, axioms, sets, constants, functions, relations, ... theories. The system model $\mathcal{S}$ is expressing a model or a chain of models of the system; finally, $\mathcal{R}$ is expressing requirements for the system to design.

Local computations on graphs, and particularly graph relabeling systems, have been introduced in [19] as a suitable tool for encoding distributed algorithms, for proving their correctness and for understanding their power. In this model, a network is represented by a graph whose vertices denote processors, and edges denote communication links. The local state of a processor (resp. link) is encoded by the label attached to the corresponding vertex (resp. edge). A relabeling rule is a rewriting rule which has the same underlying fixed graph for its left-hand side and its right-hand side, but with an update of the labels. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may change according to some specific *computation rules*.

In the *Visidia* [20] research project, we are interested in formal specification and formal proof of local computation systems. This activity consists in giving a formal semantics to these systems, in developing tools for the certification of such algorithms : proof of invariants, proof of termination, and also in comparing the computational power of various subclasses of local computation systems or other kinds of distributed computation paradigms. In this paper, we combine this high level encoding of distributed algorithms with Event-B modeling to describe a pattern for proving distributed algorithms. We obtain a general schema for deriving distributed algorithms which are correct by construction. In fact, the proposed pattern proves distributed algorithms by combining the refinement and local computations. We think that, the most important idea behind the construction of this pattern is to simplify and to automate the design and the proof of distributed algorithms. However, we note that several existing works have tried to propose a general modeling process based on proofs for distributed algorithms, but unlike sequential algorithms, there is no universal methods for modeling the distributed ones [21].

1.3 Organization of the paper

We start in Section 2 by describing basic concepts of the Event-B modeling language. In Section 3, we present the local computations notion. In particularly we concentrate on the graph relabeling systems which can be defined as a framework to encode a distributed system. In Section 4, we introduce our pattern for combining local computations models and we formally develop the different contexts, as well as the different machines, of the pattern. We show in Section 5 how to apply our pattern to develop a distributed algorithm specification. Finally, we conclude this paper by summarizing our ideas and by describing the future directions of our research.

2 The modeling framework

We summarize basic concepts of the Event B modeling language developed by J.-R. Abrial [10, 22, 23] and we indicate links with the tool called RODIN [24]. We let the notion of *formal model* unspecified in this section and we will define it in the next section.

2.1 Modeling actions over states

The event-driven approach [10, 23, 25] is based on the B notation. It extends the methodological scope of basic concepts in order to take into account the idea of *formal models*. Roughly speaking, a formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*; an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. In the following, we briefly recall definitions and principles of formal models and explain how they can be managed by tools [5, 24, 26].

Generalized substitutions are borrowed from the B notation. They provide a means to express changes to state variable values. In its simple form, $x := E(x)$, a generalized substitution looks like an assignment statement. In this construct, x denotes a vector built on the set of state variables of the model, and $E(x)$ a vector of expressions. The interpretation we shall give here to this statement is not however that of an assignment statement. We interpret it as a *logical simultaneous substitution* of each variable of the vector x by the corresponding expression of the vector $E(x)$. There exists a more general normal form of this, denoted by the construct $x : |(P(x, x'))$. This should be read: "x is modified in such a way that the value of x afterwards, denoted x', satisfies the predicate $P(x, x')$ holds", where x' denotes the *new value* of the vector and x denotes its *old value*. This is clearly non-deterministic in general.

An event has two main parts: a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalized substitution. An event can take one of the three normal forms. The first form ($evnt \ \widehat{=}$ **begin** $x : |(P(x, x')$ **end**) shows an event that is not guarded: it is thus always enabled and is semantically defined by $P(x, x')$. The second ($evt \ \widehat{=}$ **when** $G(x)$ **then** $x : |(Q(x, x'))$ **end**) and third ($evt \ \widehat{=}$ **any** t **where** $G(t, x)$ **then** $x : |(R(x, x', t))$ **end**) forms are guarded by a guard which states the necessary condition for these events to occur. Such a guard is represented by WHEN $G(x)$ in the second form, and by
any t **where** $G(t, x)$ (for $\exists t \cdot G(t, x)$) in the third form. We note that the third form defines a possibly non-deterministic event where t represents a vector of distinct local variables. The, so-called, before-after predicate $BA(x, x')$ associated with each of the three event types, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the "execution" of event *evt*. The second and the third forms are semantically equivalent to $G(x) \ \wedge \ Q(x, x')$ resp. $\exists t \cdot (G(t, x) \ \wedge \ R(x, x', t)$. The following tables summarized the three possible forms for writing a B event.

Event e	Before-after Predicate BA(e)(x,x')
begin $x : \|(P(x, x'))$ **end**	$P(x, x')$
when $G(x)$ **then** $x : \|(Q(x, x'))$ **end**	$G(x) \ \wedge \ Q(x, x')$
any t **when** $G(t, x)$ **where** $x : \|(R(x, x', t))$ **end**	$\exists t \cdot (G(t, x) \ \wedge \ R(x, x', t))$

	Proof obligation
(INV1)	$Init(x) \ \Rightarrow \ I(x)$
(INV2)	$I(x) \ \wedge \ BA(e)(x, x') \ \Rightarrow \ I(x')$
(FIS)	$I(x) \ \wedge \ \mathsf{grd}(e)(x) \Rightarrow \exists y. BA(e)(x, y)$

Proof obligations (INV 1 and INV 2) are produced by the tool RODIN [24] from events in order to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from

the definition of the before-after predicate, $BA(e)(x, x')$, of each event e. Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false. When this is the case, the event is said to be *disabled*. The proof obligation FIS expresses the feasibility of the event e with respect to the invariant I.

2.2 Model Refinement

The refinement of a formal model allows us to enrich a model in a *step-by-step* approach, and is the foundation of our *correct-by-construction* [6] approach. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model in a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines *skip*, (3) no new event takes control for ever, and (4) relative deadlock-freeness is preserved. Details of the formulation of these proofs follows. We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. If $BA(e)(x, x')$ and $BA(f)(y, y')$ are respectively the abstract and concrete before-after predicates of the same event, respectively e and f, we have to prove the following statement, corresponding to proof obligation (1):

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ \exists x' \cdot (BA(e)(x,x') \ \wedge \ J(x',y'))}$$

Now, proof obligation (2) states that $BA(f)(y, y')$ must refine *skip* ($x' = x$), generating the following simple statement to prove (2):

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ J(x,y')}$$

For the third proof obligation, we must formalize the notion of the system advancing in its execution; a standard technique is to introduce a variant $V(y)$ that is decreased by each new event (to guarantee that an abstract step may occur). This leads to the following simple statement to prove (3):

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ V(y') < V(y)}$$

Finally, to prove that the concrete model does not introduce additional deadlocks, we give formalisms for reasoning about the event guards in the concrete and abstract models: $\mathsf{grds}(AM)$ represents the disjunction of the guards of the events of the abstract model, and $\mathsf{grds}(CM)$ represents the disjunction of the guards of the events of the concrete model. Relative deadlock freeness is now easily formalized as the following proof obligation (4):

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ \mathsf{grds}(AM) \ \Rightarrow \ \mathsf{grds}(CM)}$$

When one refines a model, one can either refine an existing event by strengthening the guard and/or the before-after predicate (effectively reducing the degree of non-determinism), or add a new event in order to refine the skip event. The feasibility condition is crucial for avoiding possible states which have no successor; for instance, the division by zero. Furthermore, such refinement guarantees that the set of traces of the refined model contains (up to stuttering) the traces of the resulting model. The Event B modeling language is supported by the RODIN platform [24] and is introduced in publications [10, 23] where there are a lot of case studies and elements on the language itself and its foundations of the Event B approach. The language of *generalized substitutions* is very rich and allows us to express any relation between states in a set-theoretical context. The expressive power of the language leads to require helps for writing relational specifications and this is why we should provide proof-based patterns for assisting the development of Event B models.

Name	Syntax	Definition
Binary relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Composition of relations	$r_1; r_2$	$\{x, y \mid x \in a \quad \wedge \quad y \in b \quad \wedge$ $\exists z.(z \in c \quad \wedge \quad x, z \in r_1 \quad \wedge \quad z, y \in r_2)\}$
Inverse relation	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \quad \wedge \quad y \in \mathcal{P}(b) \quad \wedge \quad y, x \in r\}$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \quad \wedge \quad \exists b.(b \in t \quad \wedge \quad a \mapsto b \in r)\}$
Range	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identity	$\mathsf{id}(s)$	$\{x, y \mid x \in s \quad \wedge \quad y \in s \quad \wedge \quad x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s); r$
Co-restriction	$r \rhd s$	$r;\ \mathsf{id}(s)$
Anti-restriction	$s \quad \lhd\!\!\!\!- r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \quad \rhd\!\!\!\!- s$	$r \rhd (\mathsf{ran}(r) - s)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Overriding	$q \quad \leftarrow\!\!\!\!\rhd r$	$(\mathsf{dom}(r) \quad \lhd\!\!\!\!- q) \cup r$
Partial Function	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \quad \wedge \quad (r^{-1}; r) \subseteq \mathsf{id}(t)\}$

2.3 Structures for models

Event B models are defined using two kinds of structures:

- contexts for stating mathematical properties of objects.
- machines for expressing actions modifying the state of the current systeM

The first structure is called a context and it provides the defintion of sets, constants, axioms for sets and constants and theorems that can be derived from the axioms of the context $\mathcal{D}$. The context $\mathcal{AD}$ is a previous context which is already defined and it extends the current context.A context is validated when sets, constants and axioms are well formed and when each theorem is proved.

A context is clearly stating static properties of the (system) model udner construction. The *extends* construct provides a way to resuse by extending former defined context. This point should be improved in the RODIN tool and it seems that further versions will include the processing of modules, for instance.

The proof process is based on the management of sequents and one can associate an environment for proof called $\Gamma(\mathcal{D})$; the proof environment includes axioms, properties and theorems already proved. It is given at the beginning but the rule of the game is to add new theorems. It means that we intends to prove the following properties in the sequent calculus style:

For any j in $\{1..q\}$, $\Gamma(\mathcal{D}) \vdash th_j : Q_j(S_1, \ldots S_n, C_1, \ldots, C_m)$

The dynamic part of a model is expressed using the notion of machine. A machine is either a basic machine, or a refinement of a more abstract machine. A machine is modelling a state by a list of variables x which are supposed to be modifiable by events listed in the machine. The view is supposed to be closed with respect to events. Each event is maintaining an assertion called invariant and which is a conjunction of logical statements calle inv_j. Eaxh reached state is satisfying properties of the theorems part called safety properties. Proofs obligations are given in the last section and they are generated and checkable by the RODIN framework. The validation of the machine M leads to the validation of the safety and invariance properties.

3 Local Computation Models

In this section, we illustrate, in an intuitive way, the notion of local computations, and particularly that of graph relabelling systems by showing how some algorithms on networks of processors may be encoded within this framework [27]. As usual, such a network is represented by a graph whose vertices stand for processors and edges for (bidirectional) links between processors. At every time, each vertex and each edge is in some particular state and this state will be encoded by a vertex or edge label. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may have changed according to some specific *computation rules*. Let us recall that graph relabelling systems satisfy the following requirements:

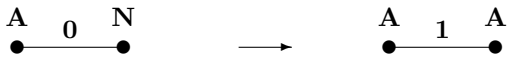
(C1) they do not change the underlying graph but only the labelling of its components (edges and/or vertices), the final labelling being the result,
(C2) they are local, that is, each relabelling changes only a connected subgraph of a fixed size in the underlying graph,
(C3) they are locally generated, that is, the applicability condition of the relabelling only depends on the local context of the relabelled subgraph.

For such systems, the distributed aspect comes from the fact that several relabelling steps can be performed simultaneously on "far enough" subgraphs, giving the same result as a sequential realization of them, in any order. A large family of classical distributed algorithms encoded by graph relabelling systems is given in [28]. In order to make the definitions easy to read, we give in the following an example of a graph relabelling system for computing a spanning tree. Then, the formal definitions of local computations will be presented.

3.1 Distributed computation of a spanning tree

Let us first illustrate graph relabelling systems by considering a simple distributed algorithm which computes a spanning tree of a network. Assume that a unique given processor is in an "active" state (encoded by the label **A**), all other processors being in some "neutral" state (label **N**) and that all links are in some "passive" state (label **0**). The tree initially contains the unique active vertex. At any step of the computation, an active vertex may activate one of its neutral neighbours and mark the corresponding link which gets the new label **1**. This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the links with label **1**.

An elementary step in this computation may be depicted as a *relabelling step* by means of the following relabelling rule R which describes the corresponding label modifications (remember that labels describe processor status):

R: 

An application of this relabelling rule on a given graph (or network) consists in (i) finding in the graph a subgraph isomorphic to the left-hand-side of the rule (this subgraph is called the *occurrence* of the rule) and (ii) modifying its labels according to the right-hand-side of the rule.

3.2 Formal definition of local computations

Local computations are characterized by applications of rules such that: an application of a rule to a ball depends exclusively on the labels appearing in the ball and changes only these labels. The previous examples can be described by the following general model. Let us introduce a few notations. We consider graphs which are finite, undirected and connected without multiple edges and self-loops. If G is a graph, $V(G)$ denotes the set of vertices and $E(G)$ denotes the set of edges. For a vertex v and a positive integer k; the *ball* of radius k with center v, denoted by $B_G(v,k)$, is the subgraph of G induced by the set of vertices $V' = \{v' \in V \mid d(v,v') \leq k\}$. Let L be an alphabet. A graph labelled over L will be denoted by (G,λ), where $\lambda : V(G) \cup E(G) \to L$ is the function labelling vertices and edges. The graph G is called the underlying graph, and the mapping λ is a labelling of G. Let $\mathcal{G}_L$ be the class of graphs labelled over some fixed alphabet L.

Definition 1. *A graph rewriting relation is a binary relation $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ closed under isomorphism. The transitive closure of $\mathcal{R}$ is denoted $\mathcal{R}^*$.*
An $\mathcal{R}-$rewriting chain is a sequence $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_n$ such that for every i, $1 \leq i < n$, $\mathbf{G}_i \ \mathcal{R} \ \mathbf{G}_{i+1}$. A sequence of length 1 is called an $\mathcal{R}-$rewriting step (a step for short).

By "closed under isomorphism" we mean that if $\mathbf{G}_1 \simeq \mathbf{G}$ and $\mathbf{G} \ \mathcal{R} \ \mathbf{G}'$, then there exists a labelled graph $\mathbf{G}'_1$ such that $\mathbf{G}_1 \ \mathcal{R} \ \mathbf{G}'_1$ and $\mathbf{G}'_1 \simeq \mathbf{G}'$.

Definition 2. *Let $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ be a graph rewriting relation.*

1. *$\mathcal{R}$ is a relabelling relation if whenever two labelled graphs are in relation then their underlying graphs are equal (not only isomorphic):*

$$\mathbf{G}\ \mathcal{R}\ \mathbf{H} \quad \Longrightarrow \quad G = H.$$

When $\mathcal{R}$ is a relabelling relation we shall speak about $\mathcal{R}-$relabelling chains (resp. step) instead of $\mathcal{R}-$rewriting chains (resp. step).
2. *A relabelling relation $\mathcal{R}$ is local if whenever $(G,\lambda)\ \mathcal{R}\ (G,\lambda')$, the labellings λ and λ' only differ on some ball of radius 1 :*

$$\exists\ v \in V(G) \text{ such that } \forall\ x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x).$$

We say that the step changes labels in $B_G(v,1)$.
3. *An $\mathcal{R}-$normal form of $\mathbf{G} \in \mathcal{G}_L$ is a labelled graph $\mathbf{G}'$ such that $\mathbf{G}\ \mathcal{R}^*\ \mathbf{G}'$, and $\mathbf{G}'\ \mathcal{R}\ \mathbf{G}''$ holds for no $\mathbf{G}''$ in $\mathcal{G}_L$. We say that $\mathcal{R}$ is noetherian if for every graph $\mathbf{G}$ in $\mathcal{G}_L$ there exists no infinite $\mathcal{R}-$relabelling chain starting from $\mathbf{G}$. Thus, if a relabelling relation $\mathcal{R}$ is noetherian, then every labelled graph has an $\mathcal{R}-$normal form.*

The next definition states that a local relabelling relation is *locally generated* if its restriction on centered balls of radius 1 determines its computation on any graph.

Definition 3. *Let $\mathcal{R}$ be a relabelling relation. Then $\mathcal{R}$ is locally generated if the following is satisfied: For any labelled graphs (G,λ), (G,λ'), (H,η), (H,η') and any vertices $v \in V(G)$, $w \in V(H)$ such that the balls $B_G(v,1)$ and $B_H(w,1)$ are isomorphic via $\varphi : V(B_G(v,1)) \longrightarrow V(B_H(w,1))$ and $\varphi(v) = w$, the following three conditions*

1. $\forall\ x \in V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \eta(\varphi(x))$ *and* $\lambda'(x) = \eta'(\varphi(x))$,
2. $\forall\ x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x)$,
3. $\forall\ x \notin V(B_H(w,1)) \cup E(B_H(w,1)),\ \eta(x) = \eta'(x)$,

imply that $(G,\lambda)\ \mathcal{R}\ (G,\lambda')$ if and only if $(H,\eta)\ \mathcal{R}\ (H,\eta')$.

Finally, local computations are the computations defined by a relation locally generated. The reader can find in [29] detailed definitions, formal properties and many examples of local computations.

Let us also note that labels can be sets or sets of sets. In particular, it is possible to handle graphs described as labels. For example, the Mazurkiewicz universal graph reconstruction is a distributed enumeration algorithm which allows the reconstruction of an anonymous graph. The manipulated labels for such an algorithm are sets standing for graphs (see [28]).

4 A pattern for combining local computation models

4.1 The pattern presentation

The Event-B modeling method provides the framework for supporting our method for developing distributed algorithms. In fact, this method can perfectly be used to construct a pattern for combining local computation models and Event-B language. After that it leads to a plugin for deriving a VISIDIA program.

In this section, we present an informal description of the pattern, and we explain how it allows to construct a correct specification of a distributed algorithm. We describe concepts for modeling state-based systems and we explain how models are defined in the Event-B. In Figure 1, the high-level structure of the distributed algorithm pattern is shown, followed by a very brief explanation of the different pattern levels:

In general, the Event-B design of a distributed algorithm starts with a very abstract model, then by successive refinement we obtain a concrete one that expresses the local behavior of the processor in the network. According to this development strategy and to our experience in the distributed algorithm specification, we conclude that three basic levels are necessary to build a correct specification of distributed algorithm. These levels are described as follows :

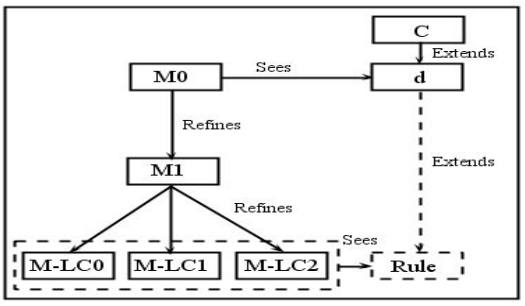


Fig. 1. The distributed algorithm pattern

1. **M0** is an abstract machine that computes the result in one step. It has only one event that states the result in one shot and does not describe the algorithmic process for computing the result.
2. The refinement of **M0** into **M1** will allow to express the inductive property which allows to explain the computation in the local computation model. We observed that, **M1** level may be a set of machines used to express the working of the algorithm and describe the necessary steps to move from an initial state to a final state which corresponds exactly to the calculated result in the first stage.
3. Finally, the **M**-LC (**L**ocal **C**omputation **M**achine) represents the last machine of the specification. In this step we introduce the rewriting rules of the algorithm.

With these machines, some contexts have an indispensable presence with a particular definition in the specification. The first one is the "c" context, it defines the application field of the distributed algorithm: i.e: graph, tree, ring.. . The second one is the "d" context, it is defined as an extension of "c". It includes static properties that describe the particularity of each algorithm. The last one is the "rule" context, it defines all rewrite rules of the algorithm. However, this context has a virtual presence. The designer is not supposed to define it, but it will serve to verify the consistency of the "M-LC" machine.

4.2 Formal development of the different contexts of the pattern

The "c" context The "c" context will describe the basic properties of the network on which distributed algorithms are designed to run. Formally, a network can be straightforwardly modeled as a connected, non-oriented and simple graph where nodes denote processors and edges denote direct communication links. A graph is simple if does not have more than one edge between any two nodes and no edge starts and ends at the same node (see axm4). An undirected graph, means that there is no distinction between two nodes associated with each edge (see axm3). A graph (oriented or not) is connected if only, for each pair of nodes, there exists a set of edges joining these nodes (see axm5). According to Jean-Raymond Abrial et al. [30, 31], a graph namely "**g**" is modeled by a set of nodes namely "**ND**" can be presented as follow:

CONTEXT C
SETS
ND
CONSTANTS
g
AXIOMS
axm1 : $g \subseteq ND \times ND$
axm2 : $dom(g) = ND$
axm3 : $g = g^{-1}$
axm4 : $id(ND) \cap g = \emptyset$
axm5 : $\forall s.s \subseteq ND \wedge s \neq \emptyset \wedge g[s] \subseteq s \Rightarrow ND \subseteq s$
END

The "d" context As presented in Section 3, local computations, particularly graph relabeling systems is a powerful model which provides general tools to encode distributed algorithms. In such

systems, each node and each edge is in some particular state and this state will be encoded by a specific label [32]. This label allow nodes to perform an elementary step of computation according to some relabeling rules. Formally, we specify labels as a finite set "LN" for **L**abel **N**odes and "LE" for **L**abel **E**dges. In order to specify correctly the rewriting system, it is necessary to mention that only nodes are forced to change their states. In other words "LN" must contain at least two different labels, but "LE" can include more than one label. In the beginning, for each node [resp. edge] a specific label is associated to it. The initial labeling and the local rewriting rules will define the behavior of a distributed algorithm on the network. Let "init_LE" (initial edge labels) and "init_LN" (initial node labels) be two sets to encode the graph initial labeling.
According to [27], if we only consider a noetherian graph relabeling systems (which means that from any initial graph, no infinite relabeling chain exists) any computation on a graph must give a result in a finite time. This means that after a finite computation steps and when no rule can be applied, the execution of the algorithm is finished and its solution is computed. Formally, we define "solution" as a non-empty set of possible and reachable solutions by the algorithm on the graph (when the graph becomes irreducible). It encodes a particular representation of the graph when nodes and edges are in a final state. We denote final state of nodes [resp. edges] by a set called "final_LN" [resp. "final_LE"] representing all node label [resp. edge] when the algorithm execution terminates. All these specific algorithmic properties are specified in the "d" context described in the following :

axm1 : $finite(LN) \land finite(LE)$
axm2 : $card(LN) > 1 \land card(LE) \geq 1$
axm3 : $final\_LN \subseteq LN \land final\_LE \subseteq LE$
axm4 : $init\_LN \subseteq LN \land init\_LE \subseteq LE$
axm5 : $final\_LN \neq \emptyset \land final\_LE \neq \emptyset$
axm6 : $init\_LN \neq \emptyset \land init\_LE \neq \emptyset$
axm7 : $init\_LN \cup final\_LN = LN$
axm8 : $init\_LE \cup final\_LE = LE$
axm9 : $solution \subseteq (g \rightarrow final\_LE) \times (ND \rightarrow final\_LN)$
axm10 : $solution \neq \emptyset$

The "rule" context A relabeling rule can be represented as a relabeling relation on the graph. In other words, it can be considered as a "before/after" relation that describes the change of states. Formally, we represent it as a binary relation called "rules" that could be shared by all synchronization algorithms. The "rules" is specified as follow:

$$\text{rules} \in (\text{LN} \times \mathbb{P}(\text{LN} \times \text{LE})\ ) \nrightarrow (\text{LN} \times \mathbb{P}(\text{LN} \times \text{LE}))$$

On left-hand side of "rules", we found the necessary condition to apply the rule. The first "LN" in the condition represents the state of the node center, where the powerset (LN × LE) represents the state of one (or more) neighbors as well as the state of edges connecting them to the center. The right-hand side of "rules" represents new labels of nodes and edges involved in the rule (result of the rule application).
In order to ensure the consistency of "rule" specification, we added some requirement properties: "rules" is a not empty set (axm3), a rule must perform a real label change (axm4) and the neighbors labels is a finite set (axm5). Formally, these properties are presented as follows :

axm3 : $rules \neq \emptyset$
axm4 : $\forall cond, act \cdot cond \in (LN \times \mathbb{P}(LN \times LE)) \land cond \mapsto act \in rules \Rightarrow cond \neq act$
axm5 : $\forall v1, v2, c1, c2 \cdot c1 \in LN \land c2 \in LN \land v1 \in \mathbb{P}(LN \times LE) \land v2 \in \mathbb{P}(LN \times LE) \land (c1 \mapsto v1) \mapsto (c2 \mapsto v2) \in rules \Rightarrow finite(v1) \land finite(v2) \land card(v1) = card(v2)$

4.3 Formal development of the different machines of the pattern

After describing the basic mathematical structure in the previous section, we can now proceed to the development of the distributed algorithm models. As we have previously explained, building a distributed algorithm specification consists of developing gradually a specification in three main levels:

The first level In this very abstract level, the machine will express only the goal of the distributed algorithm and not describe its process for computing the solution. As we presented in [23], the election algorithm (IEEE 1394 protocol) can be done in one step. The specification of the algorithm is made by a machine having only one event for selection of a node called leader in a finite time. We used the same approach to develop spanning tree algorithms in [33]. We have specified these algorithms with a machine that contains only one event which computes the solution in one shot and generates a spanning tree in the graph. In this context, other algorithms have been proposed like "mazurkiewicz" [34] and 3-coloration of a ring. In order to specify this event, we introduce the variable "result" that contain the result of the algorithm execution. Formally, "result" is defined by this very simple invariant :

$$\text{result} \in (\ g \leftrightarrow LE\ ) \leftrightarrow (\ ND \leftrightarrow LN\ )$$

The first machine is the generic solution for all distributed algorithms problems. Effectively, we consider it as a basis of refinement to generate a specific algorithm models. It includes an event called "oneshot" which avows the result of the distributed algorithm when its execution is completed. In other words, there is no protocol, only the formal definition of its intended result. The analogy of someone closing and opening their eyes. So the "oneshot" event will attribute an element from "solution" set to the "result" variable.
In order to avoid such difficulty in proofs we have chosen to declare the solution as a pair (x,y) instead of a single variable and therefore the generated PO will be proved using only the interactive prover of Rodin tool. Formally the basic definition of this event is represented as follows:

Event $oneshot \mathrel{\widehat{=}}$
  any
    x
    y
  where
    grd1 : $x \mapsto y \in solution$
  then
    act1 : $result := \{x \mapsto y\}$
  end
END

The second level In this level, we introduce machine(s) as well as event(s) to allow computation on the graph. As mentioned earlier, this computation does not change the underlying graph but only the labeling of its components (nodes/edges). This level remains in a high level abstraction, it encodes the algorithm and computes its result without considering relabeling rules (they will be introduced in the last level).
We add two functions "ln" and "le" that assign a label to each node and to each edge. The addition of these two variables involve the addition of new properties in the invariant component. The specification of these functions will take the following forms.

$$\text{ln} \in ND \rightarrow LN$$
$$\text{le} \in g \rightarrow LE$$

The following initialization establishes the invariants:

Initialisation
  begin
    act1 : $le :\in g \rightarrow init\_LE$
    act2 : $ln :\in ND \rightarrow init\_LN$
  end

The "oneshot" event is refined by modifying the guard and the action of the abstract event.

Event $oneshot \mathrel{\widehat{=}}$
refines $oneshot$
  when
    grd1 : $le \mapsto ln \in solution$
  with
    x : $\text{x} = \text{le}$
    y : $\text{y} = \text{ln}$
  then
    act1 : $result := \{le \mapsto ln\}$
  end

Remember that, a formal development with Event-B method of a distributed system is a sequence of models, linked by a refinement topology which is based on some methodological reasoning. Here, we observe that this reasoning is difficult to be systematic and until now it cannot be developed automatically. Because, each algorithm has its specificity, and with all formal methods (Event-B for instance), designer is always invited to reason on an abstract representation of the system. Therefore, finding all the system's invariants, and the suitable induction of the refinement is considered as the most complex task in the model building. Thus, refinement and the choice of the right abstraction make more tractable proof obligations, generated automatically from the text of B models and make us sure that the invariant is in fact an inductive property. The introduction of this level in our pattern, will help the designer to bridge this gap.
Considered as a small model, our pattern will guide designer to develop the necessary models of his own specification. As a generic event, we propose to define "Calcul+" that will specify the computation on an unknown numbers of nodes of the graph. It specifies the goal traced by all models appearing in this stage and that can be summarized in one word: the incursion to a final state.

The declared variables in this event are :

- "new_state_nodes" [resp. "new_state_edges"] is defined to model new states of nodes [resp. edges] belonging to "nodes" set [resp. "edges"] if a computation step takes place.
- "nodes" is the set of nodes that will change its states.
- "edges" is the set of edges connecting nodes of "nodes".
- "new_label_nodes" [resp. "new_label_edges"] is defined as a set including all future label of "nodes" [resp. "edges"].

The guards of "calcul+" event specification is given in the following :

$$
\begin{array}{ll}
\texttt{grd1}: & le \mapsto ln \notin solution \\
\texttt{grd2}: & nodes \subseteq ND \\
\texttt{grd3}: & edges \subseteq g \\
\texttt{grd4}: & \exists a \cdot a \in nodes \wedge a \in ln^{-1}[init\_LN] \\
\texttt{grd5}: & \exists at \cdot at \in edges \wedge at \in le^{-1}[init\_LE] \\
\texttt{grd6}: & (\forall t, v \cdot t \mapsto v \in edges \Rightarrow (t \in nodes \wedge v \in nodes)) \wedge \\
 & \qquad (\forall a, b \cdot a \in nodes \wedge b \in nodes \wedge a \mapsto b \in g \Rightarrow a \mapsto b \in edges) \\
\texttt{grd7}: & new\_label\_nodes \subseteq final\_LN \\
\texttt{grd8}: & new\_label\_edges \subseteq final\_LE \\
\texttt{grd9}: & new\_state\_nodes \in nodes \nrightarrow new\_label\_nodes \\
\texttt{grd10}: & new\_state\_edges \in edges \nrightarrow new\_label\_edges \\
\texttt{grd11}: & new\_state\_nodes \neq \emptyset
\end{array}
$$

In the invariant component, "grd1" states that the actual situation of the graph is diffrent from "solution". "grd4" [resp. "grd5"] is introduced to ensure that at least one active node [resp. edge] must appear in "nodes" [resp. "edges"] set.
Active nodes are defined as nodes that will change their states in a computation step. Whereas the neighbors of the active nodes which do not change their states but they are used to match the rewriting rule are called "passive" [35]. All the other nodes that are not participating in such elementary relabeling step are called "idle". From this definition, we can conclude that some nodes (or edges) don't change their state in a rewriting step. For that reason, we have defined "new_state_nodes" and "new_state_edges" (see grd9 and grd10) as two partial functions.
"grd11" ensures that the new state of "nodes" is different from their previous state. Finally, actions of the event will update labels of nodes [resp. edges] which belong to "nodes" [resp. "edges"].

$$
\begin{array}{ll}
\texttt{act1}: & ln := ln \lhd\!\!\!\!- new\_state\_nodes \\
\texttt{act2}: & le := le \lhd\!\!\!\!- new\_state\_edges
\end{array}
$$

However, after a bibliographic study we shall be able to confirm that sometimes a backward step in the computation may be considered by the distributed algorithm (Spanning tree: sequential computation nodes with ID for example). In other words, if we have nodes in a final state, they can decide to change its label to a one of the previous states. This kind of computation may be specified by Event-B. So, we add an optional event called "calcul-" that will be very similar to "calcul+". The difference between them can be summarized as follows :

- grd4 : The active node has to be an element from "final_LN" set,

- grd5 : The active edge has to be an element from “final_LE” set,
- grd7 : “new_label_nodes” becomes a sub set of “init_LN” (new_label_nodes $\subseteq$ init_LN)
- grd8 : “new_label_edges” becomes a sub set of “init_LE” (new_label_edges $\subseteq$ init_LE).

The third level Once machine(s) of the second level has (have) been specified and proven, the last machine can be refined for describing local label modification in a star (we denote by a star, a node with its neighbors in a graph). Now, we are able to observe the specification more precisely. Therefore, we can see more events, namely relabeling rules of the distributed algorithm. In this level, the “oneshot” event remains unchanged and the “Calcul+” (and “calcul-”) event still exists: it will be refined by another event(s) that simulates the elementary computation step of each node. Note that, the number of new events must always be equal to the number of algorithm rules.
As said in section 3, three kinds of local computations are considered for implementing distributed algorithms : LC0, LC1 and LC2. This section will show in more details a general model of event for each kind of synchronization.

(a) Local Computation of type 0 (LC0) In this level, we will refine the “calcul+” event to obtain an LC0 event. This event was called as “Local_computation”. It can perform only a computation on two adjacent nodes in the network. It can change labels of the two nodes as well as the label of the edge. To specify this event, we add some more details and we parameterize all abstract variables of the “calcul+”. The abstract variables were parameterized by replacing them with some concrete values by means of “witness”. In Event-B, witness is defined as a simple equality predicate involving the abstract parameters.
In the variable component of “Local_computation”, we declare “x” and “y” as two adjacent nodes involved in the computation step. Let “new_label_x” and “new_label_y” be the two labels that will encode the new states of “x” and “y” after applying the rule. And Let “new_label_edge” be the new label of the edge joining “x” and “y”. We note that “new_label_y” and “new_label_edge” are two optional variables in the specification.
In the witness component, we instantiate some abstract variables, for example, the witness “nodes: nodes = {x , y}” replaces the abstract parameter “nodes”, declared in the left hand side of the predicate, with the tow nodes “x” and “y”. Below the complete list of all witnesses.

$$
\begin{array}{l}
\texttt{edges} : \ \texttt{edges} = \{\texttt{x} \mapsto \texttt{y}, \texttt{y} \mapsto \texttt{x}\} \\
\texttt{nodes} : \ \texttt{nodes} = \{\texttt{x}, \texttt{y}\} \\
\texttt{new\_state\_nodes} : \ \texttt{new\_state\_nodes} = \{\texttt{x} \mapsto \texttt{new\_label\_x}, \texttt{y} \mapsto \texttt{new\_label\_y}\} \\
\texttt{new\_state\_edges} : \ \texttt{new\_state\_edges} = \{(\texttt{x} \mapsto \texttt{y}) \mapsto \texttt{new\_label\_edge}\} \\
\texttt{new\_label\_nodes} : \ \texttt{new\_label\_nodes} = \{\texttt{new\_label\_x}, \texttt{new\_label\_y}\} \\
\texttt{new\_label\_edges} : \ \texttt{new\_label\_edges} = \{\texttt{new\_label\_edge}\}
\end{array}
$$

In the guard component, “grd8” verify the existence of a rule that can be triggered with the initial state of nodes and/or edge. The “grd2” allow checking if the two declared nodes are neighbors or not. Formally, guards of the “Local_computation” for the LC0 relabeling rule can be encoded as follows :

$$
\begin{array}{l}
\texttt{grd1} : \ le \mapsto ln \notin solution \\
\texttt{grd2} : \ x \mapsto y \in g \\
\texttt{grd3} : \ new\_label\_x \in final\_LN \\
\texttt{grd4} : \ new\_label\_y \in final\_LN \\
\texttt{grd5} : \ new\_label\_edge \in final\_LE \\
\texttt{grd6} : \ ln(x) \in init\_LN \\
\texttt{grd7} : \ le(x \mapsto y) \in init\_LE \\
\texttt{grd8} : \ (ln(x) \mapsto \{ln(y) \mapsto le(x \mapsto y)\}) \mapsto (new\_label\_x \mapsto \{new\_label\_y \mapsto new\_label\_edge\}) \in rules
\end{array}
$$

(b) Local Computation of type 1 (LC1) LC1 is the star rule that updates a single node label and occasionally, labels of edges outgoing from the center. In a computation step, the label attached to the center of the star is modified according to some rules depending on the labels of the star. However, leaves labels are never modified (leaves are all the neighbors of the center). Like LC0, the parameterizing of abstract variables in LC1 event is also expressed by predicates in the witness component. They allow to introduce these four variables :

1. “c” : It represents by convention the center of the star. All the neighbors of “c” are represented by g[{c}].
2. “new_label_c” : It is the label that will encode the new state of “c” after the rule take place.
3. “new_label_edge” : It is a sub set of “final_LE”. It represent all the new states of the star edges.
4. “Boule_edge” : It is defined as a partial function mapped from each edge in the star to a label from “new_label_edge”. The goal of this variable is to hold the new leaves state.

“c” and “new_label_c” are considered as essential variables to specify LC1 rules when the other variables are considered as optional. In the guard component, we add an axiom to verify the existence of at least one edge in initial state. Also, we add an axiom to guaranty the existence of a relabeling rule that can coincide exactly with the initial state of the star and allows the triggering of the event.

(c) Local Computation of type 2 (LC2) LC2 is the star rule that updates labels attached to the center and/or the leaves, according to some relabeling rules. Compared to LC1 model, additional requirement must be enforced to specify an LC2 algorithm. This requirement is given below : A declaration of “Boule_node” is added to hold the new state of the star leaves. So, it is defined as a partial function mapped from each node of the star to a label from “new_label_node”. The “new_label_node” is defined as a subset of “final_LN”. The “Boule_node” specification is done as follow:

$$\begin{array}{|l} \text{Boule\_node} \in (g[\{\ c\}\ ] \cup \{\ c\}\ ) \nrightarrow \text{new\_label\_node} \\ \text{Boule\_node} \neq \emptyset \end{array}$$

The other difference with the LC1 algorithm resides in the action component; in fact, in the substitution component of a LC2 event, “ln” is overwritten by elements of “Boule_node”.

$$\boxed{\text{ln} := \text{ln} \lhd\!\!\!- \text{Boule\_node}}$$

5 Example

This section presents an application of our pattern and demonstrates how it can be used. To this end, we choose an example of the spanning tree algorithm implemented with the LC0 synchronization given in section 3.1. This algorithm can create a hierarchical tree that spans the entire network. It determines all redundant paths between processes and chooses only one which is active at a given time. Through this example, we have listed and discussed the initialization of the different pattern levels to generate quickly a correct specification.

The usage of design patterns in Object Oriented technology results in adapting and incorporating some pre-defined pieces of codes in a software project [36]:

- The adaptation of an Event-B design pattern essentially consists of instantiating its constants, variables and events in order to have them corresponding to some elements of the problem at hand.
- The incorporation of an Event-B design pattern within a larger model whose construction is in progress, consists of composing the design pattern events within some existing events of the model, so that the resulting effect is a refinement of the large model.

In this section, we apply these two techniques on our pattern to create an instance of the chosen distributed algorithm. We illustrate how we obtain an instance by applying refinement techniques.

The “c” context instantiation Having the definition of the current graph namely, “g” over the set of nodes namely, “ND” in the pattern, we will now extend the context “c” to define elements of the tree.
A tree can be defined as a connected acyclic subgraph that contains all the nodes of the graph and some edges. In order to specify a tree, we have to define a root “r”, a node: r ∈ ND (see axm1) and a parent function “t” (each node has an unique parent node, except the root). “r” is the root of “g”, if there exists a path joining “r” to each node of the graph “g”. (for more information about tree building, the reader should see [30]). Finally, we introduce the constant “trees” to be the set of all spanning trees (with root r) of the graph “g” (see axm2) :

axm1 : $r \in ND$
axm2 : $trees = \{t | t \in ND \setminus \{r\} \rightarrow ND \wedge (\forall q \cdot q \subseteq ND \wedge r \in q \wedge t^{-1}[q] \subseteq q \Rightarrow ND = q) \wedge t \subseteq g\}$

The "d" context instantiation Considering the network representation "g" and the set of possible trees in "g" namely, "trees", we now proceed to the definition of the visualization of the distributed algorithm. We implement labels describing all states of nodes and edges as defined previously in the relabeling system "R". Now, we can instantiate "LN" and "LE" as two sets; the first includes "A" and "N" and the second includes "Marked" and "not_Marked" labels. We notice that for this algorithm, edge labels are important to determine a spanning tree, they mark edges belonging to the spanning tree.
After that, we define "solution" as a particular representation of the graph when nodes and edges are in final state. In other words, "solution" constitutes the set of all combination of labeled graph that represent trees in "trees". The definition of "solution" is given by "axm9". The "d" context instancing extends at the same time the tow contexts: "d" and "c_instantiation". Formally, all added properties are listed by the following axioms:

axm1 : $LN = \{A, N\}$
axm2 : $init\_LN = \{A, N\}$
axm3 : $final\_LN = \{A\}$
axm4 : $LE = \{Marked, not\_Marked\}$
axm5 : $init\_LE = \{not\_Marked\}$
axm6 : $final\_LE = \{Marked, not\_Marked\}$
axm7 : $A \neq N$
axm8 : $Marked \neq not\_Marked$
axm9 : $solution \subseteq \{node, art, sol, a \cdot a \in trees \wedge node \in ND \wedge art \in a \wedge sol \in (\{(\{art\} \times \{Marked\}) \cup ((g \setminus \{art\}) \times \{not\_Marked\})\} \times \{\{node\} \times \{A\}\}) | sol\}$

The "rule" context instantiation The chosen algorithm is encoded by the graph relabeling system which is based on only one rule "R". The "rule_instantiation" context is specified as an extension of the two contexts "rule" and "d_instantiation". In this context we define "rules" as a formal specification of "R":

axm1 : $rules \subset rules$
axm2 : $rules = \{(A \mapsto \{N \mapsto not\_Marked\}) \mapsto (A \mapsto \{A \mapsto Marked\})\}$

The "m1" machine instantiation The first machine remains unchangeable. In the second level, a new machine which refines "M1" will be added. This machine, called "m1_instance", uses the "d_instance" context and it defines an instance of "calcul+" event, which gradually computes the spanning tree in a progressive way. It labels some edges and some nodes and this will allow to add a new edge to the tree under construction. Let "x" and "y" be the two variables from "ND" and let x $\mapsto$ y be the edge linking "x" to "y". Let "new_state_nodes" and "new_state_edges" be the new labels of the nodes (x,y) and the edge linking them. The guards specification of "calcul+" of "m1_instance" is done as follow:

grd1 : $x \mapsto y \in g$
grd2 : $le \mapsto ln \notin solution$
grd3 : $\{x, y\} \subseteq ND$
grd4 : $ln(y) \in init\_LN$
grd5 : $le(x \mapsto y) \in init\_LE$
grd6 : $(\forall t, v \cdot t \mapsto v \in \{x \mapsto y, y \mapsto x\} \Rightarrow (t \in \{x, y\} \wedge v \in \{x, y\})) \wedge (\forall a, b \cdot a \in \{x, y\} \wedge b \in \{x, y\} \wedge a \mapsto b \in g \Rightarrow a \mapsto b \in \{x \mapsto y, y \mapsto x\})$
grd7 : $\{A\} \subseteq final\_LN$
grd8 : $\{Marked\} \subseteq final\_LE$
grd9 : $new\_state\_nodes \in \{x, y\} \nrightarrow \{A\}$
grd10 : $new\_state\_edges \in \{x \mapsto y, y \mapsto x\} \nrightarrow \{Marked\}$
grd11 : $new\_state\_nodes \neq \emptyset$

The following witnesses replace the abstract variables with the concrete one.

nodes : nodes = {x, y}
edges : edges = {x ↦ y, y ↦ x}
new_label_nodes : new_label_nodes = {A}
new_label_edges : new_label_edges = {Marked}

The "m2-LC0" machine instantiation We will refer to the previous model to define a machine with respect to LC0 synchronization as presented in our pattern. This machine is an instantiation of the "m2-LC0" machine, it refines the instance of "m1" machine and it sees the "rule_instance" context. The last machine is given by the following specification:

Event $Local\_computation \widehat{=}$
refines $Local\_computation$
any
x
y
where
grd1 : $le \mapsto ln \notin solution$
grd2 : $x \mapsto y \in g$
grd3 : $ln[\{x\}] = \{A\}$
grd4 : $le[\{x \mapsto y\}] = \{not\_Marked\}$
grd5 : $(ln(x) \mapsto \{ln(y) \mapsto le(x \mapsto y)\}) \mapsto (A \mapsto \{A \mapsto Marked\}) \in rules$
grd6 : $ln[\{y\}] = \{N\}$
with
new_label_x : new_label_x = A
new_label_y : new_label_y = A
new_label_edge : new_label_edge = Marked
then
act2 : $ln := ln \Leftarrow (\{x \mapsto A, y \mapsto A\})$
act3 : $le := le \Leftarrow \{(x \mapsto y) \mapsto Marked\}$
end

6 Conclusion

This paper presents a formal pattern to design specification of distributed algorithms. Our contribution consits in the definition of a common and reusable pattern based on refinement techniques and on the encoding of distributed algorithms by graph rewriting systems. Therefore, this work is considered as extension to our work [33] in which we have used the refinement technique of Event-B to propose a unified modular development of distributed algorithms. We have illustrated this method by investigating examples of the distributed computation of spanning trees, and we have implemented them by using the Rodin platform. We plan to implement this pattern as a plug-in in Rodin platform to assist the user in the design of distributed algorithms.

References

1. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
2. Morgan, C.: Programming from Specifications. Prentice Hall International Series in Computer Science. Prentice Hall (1990)
3. Back, R.: On correct refinement of programs. Journal of Computer and System Sciences **23**(1) (1979) 49–68
4. Abrial, J.R.: The B book - Assigning Programs to Meanings. Cambridge University Press (1996)
5. Abrial, J.R., Cansell, D.: Click'n prove: Interactive proofs within set theory. In: TPHOL 2003. (2003) 1–24
6. Leavens, G.T., Abrial, J.R., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006), ACM (October 2006) 221–235
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, R., Gamma, P.: Design Patterns : Elements of Reusable Object-Oriented Software design Patterns. Addison-Wesley Professional Computing (1994)
9. Alexander, C., Ishikawa, S., Silverstein, M.: A pattern language: towns, buildings, construction. Oxford University Press (1977)
10. Abrial, J.R.: A Mechanical Press Controller. In: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2009)
11. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-b. Fundam. Inform. **77**(1-2) (2007) 1–28
12. Cansell, D., Méry, D.: Incremental parametric development of greedy algorithms. Electr. Notes Theor. Comput. Sci. **185** (2007) 47–62
13. Cansell, D., Gibson, J.P., Méry, D.: Refinement: A constructive approach to formal software design for a secure e-voting interface. Electr. Notes Theor. Comput. Sci. **183** (2007) 39–55

14. Cansell, D., Méry, D., Rehm, J.: Time Constraint Patterns for Event B Development. In Jacques Julliand, O.K., ed.: The 7th International B Conference - B 2007. Volume 4355 of Lecture Notes in Computer Science., Besançon, France, Springer (2007) 140–154
15. Benaïssa, N., Cansell, D., Méry, D.: Integration of security policy into system modeling. In Julliand, J., Kouchnarenko, O., eds.: B. Volume 4355 of Lecture Notes in Computer Science., Springer (2007) 232–247
16. Bjorner, D.: Software Engineering 1 Abstraction and Modelling. Texts in Theoretical Computer Science. An EATCS Series. Springer (2006) ISBN: 978-3-540-21149-5.
17. Bjorner, D.: Software Engineering 2 Specification of Systems and Languages. Texts in Theoretical Computer Science. An EATCS Series. Springer (2006) ISBN: 978-3-540-21150-1.
18. Bjorner, D.: Software Engineering 3 Domains, Requirements, and Software Design. Texts in Theoretical Computer Science. An EATCS Series. Springer (2006) ISBN: 978-3-540-21151-8.
19. Billaud, M., Lafon, P., Métivier, Y., Sopena, E.: Graph rewriting systems with priorities. Lecture notes in computer science **411** (1989) 94–106
20. ViSiDiA: http://visidia.labri.fr (2006)
21. Haddar, A., Hadj Kacem, A., Métivier, Y., Mosbah, M., Jmaiel, M.: Proving distributed algorithms for mobile agents. In Rao, S.; Chatterjee, M.J.P.M.C.S.S., ed.: International Conference on Distributed Computing and Networking (ICDCN). Volume 4904 of LNCS., Springer (2008) 286–291
22. Abrial, J.R., Mussat, L.: Introducing Dynamic Constraints in B. In: B98. (1998) 83–128
23. Cansell, D., Méry, D. In: The event-B Modelling Method: Concepts and Case Studies. Springer (2007) 33–140 See [37].
24. Project RODIN: Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/ (2004) 2004–2007.
25. Abrial, J.R.: $B^{\#}$: Toward a synthesis between z and b. In Bert, D., Walden, M., eds.: 3nd International Conference of B and Z Users - ZB 2003, Turku, Finland. Lectures Notes in Computer Science, Springer (June 2003)
26. ClearSy Aix-en-Provence (F): B4FREE. (2004) http://www.b4free.com.
27. Litovsky, I., Métivier, Y., Sopena, E.: Different local controls for graph relabelling systems. Math. Syst. Theory **28** (1995) 41–65
28. Bauderon, M., Métivier, Y., Mosbah, M., Sellami, A.: From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI (2002)
29. Litovsky, I., Métivier, Y., Sopena, E.: Graph relabelling systems and distributed algorithms. In Ehrig, H., Kreowski, H., Montanari, U., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation. Volume 3. World Scientific (1999) 1–56
30. Cansell, D., Méry, D.: Tutorial on the event-based b method. Technical report, INRIA a CCSD electronic archive server based on P.A.O.L [http://hal.inria.fr/oai/oai.php] (France) (2006)
31. Abrial, J.R., Cansell, D., Méry, D.: Formal derivation of spanning trees algorithms. In Bert, D., Bowen, J.P., King, S., Waldén, M.A., eds.: ZB. Volume 2651 of Lecture Notes in Computer Science., Springer (2003) 457–476
32. Mosbah, M., OSSAMY, R.: A programming language for local computations in graphs: Computational completeness. In IEEE, ed.: Proceedings of the 5th. Mexican International Conference in Computer Science Colima Mexico 20-24 September, Computer Society (2004) 12–19
33. Tounsi, M., Hadj Kacem, A., Mosbah, M., Méry, D.: A refinement approach for proving distributed algorithms : Examples of spanning tree problems. In: Integration of Model-based Formal Methods and Tools - IM_FMT'2009 - in IFM'2009, Düsseldorf Allemagne (02 2009)
34. ANR-RIMEL, P.: Développement d'algorithmes répartis. Technical report, MOSEL, LORIA, University Henri Poincaré - Nancy 1, ClearSy LABRI, University of Bordeaux & CNRS (February 2008)
35. Chalopin, J., Métivier, Y., Zielonka, W.: Election, Naming and Cellular Edge Local Computations. In: Graph transformation International Conference on Graph Transformation (ICGT 2004) (EATCS best paper award). Volume 3256 of Lecture notes in computer science., Italie, Springer (09 2004) 242–256
36. Abrial, J.R., Hoang, T.S.: Using design patterns in formal methods: An event-b approach. In: Proceedings of the 5th international colloquium on Theoretical Aspects of Computing, Berlin, Heidelberg, Springer-Verlag (2008) 1–2
37. Bjorner, D., Henson, M.C., eds.: Logics of Specification Languages. EATCS Textbook in Computer Science. Springer (2007)

Static Verification of Basic Protocols Systems With Unbounded Number of Agents

S. Potiyenko
Glushkov Institute of Cybernetics
Kiev, Ukraine
stepan@iss.org.ua

1 Basic protocols system

The structure of basic protocols system consists of the environment where agents interact. Agents work concurrently asynchronously and can analyze and change attributes of the environment and other agents. Any state of the system is defined by values of all attributes. During verification a need to define set of states occurs. Logic expressions are used for it and they are called formulas of base language. First order logic with multisort predicate calculus is used as base language.

Behavior of the agents is specified by basic protocols. Each basic protocol describes one atomic transition of particular agent and this agent is called key agent. Agents behavior differs by agent types and each agent instance is defined by name in formulas. So agent attributes occur as $a.r_1$ (a is agent name). The following attribute types are introduced: simple (numeric, enumerated and symbolic) and complex (functional types, arrays and lists). We will consider basic protocols where key agent name is a parameter. It means that given basic protocol specifies a transition of any agent of given agent type.

Basic protocol is defined as expression $\forall(a,x)(\alpha(a,x,r) \rightarrow < P(a,x,r) > \beta(a,x,r))$, where a - key agent name, x - list of (typed) parameters, $\alpha(a,x,r)$ - base language formula, $\beta(a,x,r)$ - conjunction of assignments, list update operators and base language formula, $P(a,x,r)$ - a process of the protocol (finite behavior of composition of the environment and agents), r is a list of attribute expressions used in the protocol. Attribute expression is an attribute name of simple type or a functional symbol $r_1(e_1,e_2,\ldots)$, where r_0 is a functional attribute or array name, $e_1,e_2,\ldots$ are expressions of corresponding argument types. Formula $\alpha(a,x,r)$ is called precondition, formula $\beta(a,x,r)$ - postcondition of the basic protocol. Basic protocol litself can be considered as temporal logic formula which expresses the following fact: if a state of the system satisfies condition α then the process P can be done and the next state of the system will satisfy condition β after it.

2 Predicate transformer

Only initial states of attributed transition system are known in the basic protocols model. To build other states partial transformation $\mu : S \rightarrow S$ exists on each state from the set of states S. A function of transformation of one set of states to another under the action of basic protocol is called predicate transformer:

$$E' = pt(E,\beta)$$

Here E,E' - states of the system before and after execution of basic protocol. States are represented as $E = D \wedge L, E' = D' \wedge L'$, where D and D' are base language formulas, L and L' - list equations. To build a formula which determines predicate transformer we need to consider basic protocol postcondition in details.

We split a set of attribute expressions r used in a basic protocol onto three subsets p,s and z. Set $p = (p_1,p_2,\ldots)$ consists of left parts of assignments in postcondition, $s = (s_1,s_2,\ldots)$ consists of attribute

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 51-54

expressions which have external occurrences in formula part of postcondition (denote it C), but does not coincide with attributes from p. And set $z = (z_1, z_2, \ldots)$ consists of attribute expressions which occurs in pre- and postcondition, but not included in two other sets. Now we can represent postcondition in the following form (bound variables are dropped here):

$$\beta(r) = (p_1(p,s,z) := t_1(p,s,z) \wedge p_2(p,s,a) := t_2(p,s,z) \wedge \ldots) \wedge U(p,s,z) \wedge C(p,s)$$

$U(p,s,z)$ - conjunction of list update operators. It's evident that attributes from z stays unchanged after basic protocol application. Now we can build transformed formula:

$$pt(D(p,s,z) \wedge L(p,s,z) \wedge \alpha(p,s,z), \beta((r,s,z)) = q_1 \vee q_2 \vee \ldots$$

where

$$q_i = \exists(x,y)(D(x,y,\xi_I) \wedge \alpha(x,y,\xi_i) \wedge L'(x,y,\xi_i) \wedge A_i(x,y,\xi_i) \wedge Q_i(x,y,\xi_i) \wedge C(r,s)),$$

$$A_i(x,y,\xi_i) = (p_1 = t_1(x,y,\xi_i) \wedge (p_2 = t_2(x,y,\xi_i)) \wedge \ldots)$$

Here ξ_i is the set z where some of functional expressions are replaced with variables from lists x and y.

Each of disjunctive members q_i corresponds to one of possible methods of identification of functional expressions, occurring in formulas D and β. To describe these methods, we consider the set M of all pairs of functional expressions of the form $(f(u), f(v)), u = (u_1, u_2, \ldots), v = (v_1, v_2, \ldots)$, where $f(u)$ is chosen from the set z, and $f(v)$ - from sets p and s. These functional expressions must be equal, if their arguments were equal before application of basic protocol.

For a construction of q_i we will choose arbitrary subset $N \subseteq M$ (including an empty set, the different indexes i corresponds to different successful selection of subset N, and in a sum they must cover all successful selections). For every pair $(f(u), f(v)) \in N$ we will consider conjunction of equalities $u = v, (u_1 = v_1 \wedge u_2 = v_2 \wedge \ldots)$. We will unite all such conjunctions in one and will add to it conjunctively negations of all equalities, which correspond to pairs not included into the set N. Denote obtained formula as $Q_i(p,s,z)$. If this formula is satisfiable, then the choice is successful. It's obviously that $f(u)$ is not independent and must change the value under condition that $Q_i(p,s,z)$ is true. Thus, $f(u)$ must change the value in the same way as $f(v)$. Therefore, in all formulas, related to the moment after application of protocol, occurrence of functional expression $f(u)$ must be replaced with a variable which corresponds to functional expression $f(v)$. Thus, if $f(u)$ coincides with several functional expressions, it is not important, what a variable is chosen (transitivity of equality). All such replacements for the chosen set of pairs specifies a substitution ξ_i.

Thus, a formula of a predicate transformer is disjunction of formulas q_i for all successful selections of set of pairs for unified functional expressions.

3 Static analysis

Verification can be done statically by analyzing basic protocols together with given properties. Static methods don't require state space search and avoid state space explosion. Consequently, systems with unbounded number of agents can be verified. It's reached by considering agent name as a parameter of basic protocol: we rewrite simple agent attributes $a.r_1$ to functional symbols $r_1(a)$ and functional agent attributes $a.r_1(x,\ldots)$ to $r_1(a,x,\ldots)$ interpreting agent name as the first argument of functional symbol. During following static analysis parameters of basic protocols are transformed to bound variables and

agent names is not an exception. So we can use quantifiers over agent names while constructing formulas and we consider relations between agents only (if there is the same agent in different protocols or several agents) avoiding their concrete number and names.

We can check invariance of given properties, prove determinacy of system behavior and absence of deadlocks.

4 Consistency

A state s of some agent a is called nondeterministic if there are more then one transition there, i.e. several basic protocols with key agent a can be applied in the state s. Let define consistency of basic protocols system: intersections of basic protocols preconditions should be absent for one key agent. If $\alpha(a,x,r)$ and $\alpha(a',y,r)$ are preconditions of two basic protocols with key agents a and a', x and y are lists of parameters and r is a list of used attributes. These basic protocols are consistent if formula

$$\neg\exists(a,a')(a = a' \wedge a'\exists x\alpha(a,x,r) \wedge \exists y\alpha'(a',y,r))$$

is identically true. If consistency of all pairs of basic protocols with the same key agent type is proved then the whole system is consistent and there is the only applicable basic protocol in each state for each agent. It means that all agents have deterministic behavior. Multiagent systems often have states where several transitions are possible under the action of different basic protocols. Such a transitions belong to different agents in consistent system. Here these states are not nondeterministic and a number of transitions is generated by interleaving of concurrently applied basic protocols of parallel agents.

To prove consistency of the system with n agent types, behavior of each of them is specified by $k_i (i = 1,\ldots,n)$ basic protocols, it's required $\sum_{i=1,\ldots,n} C_2^k$ pairs to be checked.

5 Completeness

The next goal is to prove absence of deadlocks in the system. State s is called deadlock if any transition is not possible in this state, so any basic protocol cannot be executed in the system. The property of completeness is defined as follows: disjunction of preconditions of basic protocols for the same key agent type shall be satisfiable for each agent instance:

$$\forall(a_1,a_2,a_3,\ldots)(a_1 = a_2 \wedge a_2 = a_3 \wedge \ldots \wedge$$

$$\wedge(\forall x_1\alpha_1(a_1,x_1,r_1) \vee \forall x_2\alpha_2(a_2,x_2,r_2) \vee \forall x_3\alpha_3(a_3,x_3,r_3) \vee \ldots))$$

where $\alpha_i(a_i,x_i,r_i)$ - precondition of basic protocol with key agent a_i parameterized by list x_i of variables and using a list r_i of attributes. We also can take into account user restrictions (known unreachable states, etc.) which should be specified by base language formula $R(a_1,r)$:

$$\forall(a_1,a_2,a_3,\ldots)(a_1 = a_2 \wedge a_2 = a_3 \wedge \ldots \wedge$$

$$\wedge(R(a_1,r) \vee \forall x_1\alpha_1(a_1,x_1,r_1) \vee \forall x_2\alpha_2(a_2,x_2,r_2) \vee \forall x_3\alpha_3(a_3,x_3,r_3) \vee \ldots))$$

If completeness is proved and there is at least one agent in the system in each moment of time then there is at least one applicable basic protocol in each state of the system. So deadlocks are absent in complete system. The set of basic protocols is split by agent types in typical multiagent systems. Here we can check completeness for each agent type separately and essentially reduce formula sizes for proving. If incompleteness is detected in all agent types then we can investigate incompleteness of whole system using formulas above. If there is at least one agent type in the system which completeness is proved for and there is at least one alive agent of this type at each moment of time then this system doesn't have deadlocks because of transitions of agents of this type.

6 Safety

Other properties are formulated individually for every system using base language formulas. Conception of safety conditions has been introduced for them. Let build algorithm of checking requirement of "truth everywhere" for each safety condition. Let $Q(r)$ - base language formula specifying safety condition, $s_0(r)$ - base language formula specifying initial state of the system (r - attribute list). At first, initial state should be checked: $s_0(r) \rightarrow Q(r)$. If initial state breaks safety it's useless to continue checking and algorithm should terminate. Then invariance of safety condition should be proved in the following way: if a basic protocol is applicable and safety condition is true then it shall be true after application of this protocol. It can be formally expressed as identical truth of the formula:

$$\forall(a,x)(pt(\alpha(a,x,r) \wedge Q(r), \beta(a,x,r)) \rightarrow Q(r))$$

.

Here predicate transformer *pt* is used to build the state of the system after protocol application before proving the formula.

7 Conclusions

Static analysis of basic protocols allows checking of properties of the system without exhaustive state space search. Agents are considered as parameters of basic protocols with further transformation to variables in formulas to be proved. So we can make assertions about proved properties for any number of agents working in the system.

Statically found violations of given properties are suspicions on errors. States of the system where errors occur can be obtained analyzing formulas which express consistency, completeness and safety conditions. Reachability of these states can be negated again, by static methods with formulation of new safety conditions. Otherwise, state space exploration methods can be used to reach target states.

References

[1] Lamport L. The temporal logic of actions, ACM Transactions on Programming Languages and Systems (TOPLAS). ACM New York, USA, 1994, Vol. 16, No. 3, p. 872-923. http://research.microsoft.com/pubs/64074/lamport-actions.pdf, last viewed June 2010, 1994-2010.

[2] A. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999. http://www.springerlink.com/content/2cratqgq5efr6vfh/, last viewed June 2010, 1999-2010.

[3] A. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 2005, 662-675. http://www.sdl-forum.org/issre04-witul/papers/litechevskiweigert.pdf, last viewed June 2010, 2005-2010.

[4] Potiyenko S. Static requirements checking and approaches to reachability problem Artificial Intelligence,2009, No. 1, p. 192-197. (on Russian) http://www.nbuv.gov.ua/portal/natural/ii/2009_1/4/00_Potienko.pdf, last viewed June 2010, 2009-2010.

[5] Letichevsky A.A., Godlevsky A.B., Letychevskyy A.A., Potiyenko S.V., Peschanenko V.S., Properties of a predicate transformer of VRS system. Kibernetika i sistemny analiz, 4, 2010, 3-16 (in russian). http://apsystem.org.ua/uploads/doc/ims/PPTVRS.pdf,last viewed June 2010, 2010-2010.

From Program Verification to Automated Debugging

Nikolaj Popov, Tudor Jebelean and Bruno Buchberger*
Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
popov@risc.uni-linz.ac.at

Abstract

We present an approach to verification of recursive functional programs and we put special emphasis on the possibility of falsifying incorrect programs. After finding automatically the buggy part of the program we use the "Lazy Thinking" approach for conjecturing a possible code correction.

1 Introduction

Verifying a program, or more precisely, proving that a program is correct with respect to a given specification, is normally done in the following distinguished steps: generate verification conditions; prove (discharge) the generated verification conditions.

We are developing a method for proving total correctness of certain kinds of functional recursive programs. As a distinctive feature of our approach, the verification conditions are not only sufficient, but also necessary for the program correctness [8].

In fact, even if a small part of the specification is missing – in the literature this is often the case – the correctness cannot be proven. Furthermore, a relevant counterexample may be constructed automatically [12].

There are various tools for proving program correctness automatically or semiautomatically, (see, e.g., [13],[1],[2]), and this is where our contribution falls into. As a distinctive feature of our prototype is the hint on "what is wrong" in case of a verification failure.

Furthermore, by the "Lazy Thinking" approach for program synthesis, we are able to conjecture a possible code correction.

This work is performed in the frame of the *Theorema* system [6], a mathematical computer assistant which aims at supporting all phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides both functional as well as imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked for validity. The system includes a collection of general as well as specific provers for various interesting domains (e. g. integers, sets, reals, tuples, etc.). More details about *Theorema* could be found at `www.theorema.org`.

2 Verification or Falsification

A Verification Condition Generator (VCG) is a device which takes the source code of a program together with the specification of its desired behaviour, and produces verification conditions (as logical formulae).

We approach the verification problem in an universal logical based setting. Namely, we consider that the objects manipulated by the program satisfy a certain collection of formulae T (the *object theory* of

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 55-65

*This research was partly supported by BMBWK (Austrian Ministry of Education, Science, and Culture) and Upper Austrian Government Project "Technologietransferaktivitäten".

the program). The formulae and the terms used in the program, as well as the specifications, are logical expressions using the signature of this theory. A *functional program* (without iteration) is in fact a logical formula in the object theory, because conditionals can be considered abbreviations of conjunctions of implications.

A VCG is *sound* if for any given program F, defined in the context of an object theory T, with an input condition $I_F[X]$ an output condition $O_F[X,Y]$: if the verification conditions are logical consequences of the object theory T then the program F satisfies its specification I_F and O_F.

Moreover, we are also interested in the following question: What if some of the verification conditions do not hold? May we conclude that the program is not correct? In fact, the program may still be correct. However, if the VCG is complete, then one can be sure that the program is not correct, and this is what we call *Falsification*. A VCG is *complete*, if whenever the program satisfies its specification, the generated verification conditions hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps practical debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong". Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

3 Generation of Verification Conditions

Before performing the "real" verification, we first make sure that our programs are coherent. It is not that programs which are not coherent are necessarily not correct, however, in order to construct a system of programs preserving modularity, we need to use only coherent programs.

3.1 Coherent Programs

In this subsection we state the principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principles are not our invention (similar ideas appear in [9]), we state them here because we want to emphasize on and later formalize them.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

– start from basic (trustful) functions e.g. addition, multiplication, etc.;

– define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we "forbid" using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives the possibility of easy replacement of existing functions. For example we have a powering function P, with the following program definition (implementation):

$$P[x,n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } P[x,n-1] * x$$

The specification of P is:

The domain $\mathbb{D} = \mathbb{R}^2$, precondition $I_P[x,n] \iff n \in \mathbb{N}$ and a postcondition $O_P[x,n,P[x,n]] \iff P[x,n] = x^n$.

Additionally, we have proven the correctness of P. Later, after using the powering function P for defining other functions, we decide to replace its definition (implementation) by another one, however, keeping the same specification. In this situation, the only thing we should do (besides preserving the name) is to prove that the new definition (implementation) of P meets the old specification.

Furthermore, we need to ensure that when defining a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs – we call this:

Appropriate values for the auxiliary functions. The following example will give an intuition on what we are doing. Let the program for computing F be:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x],$$

with the specification of F (I_F and O_F) and specifications of the auxiliary functions H (I_H and O_H), G (I_G and O_G). The two verification conditions, ensuring that the calls to the auxiliary functions have appropriate values are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x])$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]).$$

3.2 Recursive Programs and Generation of Verification Conditions

As is well-known, there is no universal VCG. Thus, in our research, we concentrate on constructing a VCG which is appropriate only for a certain kind of recursive programs – those which are defined by multiple choice *if-then-else* with zero, one, or more recursive calls on each branch (but without nested recursion). They are defined as those F:

$$\begin{aligned} F[x] = \ & \textbf{If } Q_0[x] \textbf{ then } S[x] \\ & \textbf{elseif } Q_1[x] \textbf{ then } C_1[x, F[R_1[x]]] \\ & \textbf{elseif } Q_2[x] \textbf{ then } C_2[x, F[R_2[x]]] \\ & \dots \\ & \textbf{else } Q_n[x] \textbf{ then } C_n[x, F[R_n[x]]]. \end{aligned} \tag{1}$$

where Q_i are predicates and S, C_i, R_i are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_i[x,y]$ are "combinator" functions, and $R_i[x]$ are "reduction" functions). We assume that the functions S, C_i, and R_i satisfy their specifications given by $I_S[x]$, $O_S[x,y]$, $I_{C_i}[x,y]$, $O_{C_i}[x,y,z]$, $I_{R_i}[x]$, $O_{R_i}[x,y]$. Additionally, assume that the Q_i predicates are non-contradictory, that is $Q_{i+1} \Rightarrow \neg Q_i$ and $Q_n = \neg Q_0 \wedge \dots \wedge \neg Q_{n-1}$, which we do only in order to simplify the presentation.

Note that functions with multiple arguments also fall into this scheme, because the arguments x, y, z could be vectors (tuples).

Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

Considering Coherent Recursive programs, we give here the appropriate definition:

Let S, C_i, and R_i be functions which satisfy their specifications. Then the program (1) is coherent if the following conditions hold:

$$(\forall x : I_F[x])\ (Q_0[x] \implies I_S[x]) \tag{2}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_F[R_1[x]]) \tag{3}$$
$$\dots$$
$$(\forall x : I_F[x])\ (Q_n[x] \implies I_F[R_n[x]]) \tag{4}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_{R_1}[x]) \tag{5}$$
$$\dots$$
$$(\forall x : I_F[x])\ (Q_n[x] \implies I_{R_n}[x]) \tag{6}$$

$$(\forall x : I_F[x])(Q_1[x] \wedge O_F[R_1[x], F[R_1[x]]] \implies I_{C_1}[x, F[R_1[x]]]) \tag{7}$$
$$\dots$$
$$(\forall x : I_F[x])(Q_n[x] \wedge O_F[R_n[x], F[R_n[x]]] \implies I_{C_n}[x, F[R_n[x]]]). \tag{8}$$

It is not that a program which is not coherent is necessarily not correct. However, non-coherent programs are somehow inconsistent, namely proving their correctness would involve knowledge about their auxiliary functions which is out of the official scope. Thus, if we allow them in our system of verified programs, the modularity would be lost.

After performing the coherence check, we go to the verification. The upcoming theorem gives the necessary and sufficient conditions for a program to be correct. These conditions are taken as the *Verification Conditions*.

Theorem. Let S, C_i, and R_i be functions which satisfy their specifications. Let also the program (1) be coherent. Then, (1) satisfies the specification given by I_F and O_F if and only if the following verification conditions hold:

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{9}$$

$$(\forall x : I_F[x])(Q_1[x] \wedge O_F[R_1[x], F[R_1[x]]] \implies O_F[x, C_1[x, F[R_1[x]]]]) \tag{10}$$
$$\dots$$
$$(\forall x : I_F[x])(Q_n[x] \wedge O_F[R_n[x], F[R_n[x]]] \implies O_F[x, C_n[x, F[R_n[x]]]]) \tag{11}$$

$$(\forall x : I_F[x])\ (F'[x] = 0) \tag{12}$$

where F' is defined as:

$$\begin{aligned} F'[x] = \ & \textbf{If } Q_0[x] \textbf{ then } 0 \\ & \textbf{elseif } Q_1[x] \textbf{ then } F'[R_1[x]] \\ & \textbf{elseif } Q_2[x] \textbf{ then } F'[R_2[x]] \\ & \dots \\ & \textbf{else } Q_n[x] \textbf{ then } F'[R_n[x]]. \end{aligned} \tag{13}$$

Based on this statement we construct a VCG, which takes as an input program (1) annotated with its specification I_F and O_F, and generates the verification conditions (9), (10), (11) and (12). Moreover, the theorem gives, in fact, two statements, namely:

– *Soundness*: If (9), (10), (11) and (12) hold, then the program (1) is correct, and

– *Completeness*: If (1) is correct, then (9), (10), (11) and (12) hold.

A precise proof of the theorem, based on the fixpoint theory of programs [11], is presented in [8], and completed in [10].

3.3 Proving the Verification Conditions

As we have already said, the coherence check is done at the beginning of the verification process—it is also realized by proving the validity of the respective conditions: (2), (3), (4), (5), (6), (7) and (8). Partial correctness is guarantied by (9), (10), (11), and termination—(12).

Proving any of the three kinds of verification conditions has its own difficulty, however, our experience shows that proving coherence is relatively easy, proving partial correctness is more difficult and proving the termination verification condition (it is only one condition) is in general the most difficult one. The latter one is expressed by using a *simplified version*(13) of the initial program (1), and the condition itself expresses a property of that *simplified version* (12). The proof typically needs an induction prover and the induction step may sometimes be difficult to find. Fortunately, due to the specific structure, the proof may be omitted, because different recursive programs may have the same *simplified version*.

Proofs of the verification conditions may be done by using a *Theorema* prover (see, e.g., [6],[7]) or by delivering the proof problem itself to another specialized tool. For serving the termination proofs, actually for omitting the proof redundancy, we are now creating libraries containing *simplified versions* together with their input conditions, whose termination is proven. The proof of the termination may now be skipped if the *simplified version* is already in the library and this membership check is much easier than an induction proof – it only involves matching against simplified versions.

4 Bug Localization

As commonly agreed, finding a bug in a program is the most difficult task in programming.

In our approach, due to its completeness, if any of the generated verification conditions does not hold, then the program is not correct with respect to the specification. Moreover, each verification condition corresponds to a concrete part of the program and having at hand such a condition we are able to isolate the buggy part of the program.

In order to make clear our experiments, we consider again a powering function P, however we provide this time a different implementation, namely *binary powering*:

$$\begin{aligned} P[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x*x, n/2] \\ & \textbf{else } x * P[x*x, (n-1)/2]. \end{aligned} \tag{14}$$

This program in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x,n] = x^n. \tag{15}$$

The (automatically generated) conditions for **coherence** are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow \mathbb{T}) \tag{16}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow \mathrm{Even}[n]) \tag{17}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow \mathrm{Odd}[n]) \tag{18}$$

$$(\forall x,n,m : n \in \mathbb{N})(n \neq 0 \wedge \text{Even}[n] \wedge m = (x*x)^{n/2} \Rightarrow \mathbb{T}) \tag{19}$$

$$(\forall x,n,m : n \in \mathbb{N})(n \neq 0 \wedge \neg\text{Even}[n] \wedge m = (x*x)^{(n-1)/2} \Rightarrow \mathbb{T}) \tag{20}$$

One sees that the formulae (16), (19) and (20) are trivially valid, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the 1 *constant-function-one* and the $*$ *multiplication*.

The formulae (17) and (18) are easy consequences of the elementary theory of reals and naturals. For the further check of **correctness** the generated conditions are:

$$(\forall x,n : n \in \mathbb{N})\ (n = 0 \Rightarrow 1 = x^n) \tag{21}$$

$$(\forall x,n : n \in \mathbb{N})\ (n \neq 0 \wedge \text{Even}[n] \Rightarrow n/2 \in \mathbb{N}) \tag{22}$$

$$(\forall x,n,m : n \in \mathbb{N})(n \neq 0 \wedge \text{Even}[n] \wedge m = (x*x)^{n/2} \Rightarrow m = x^n) \tag{23}$$

$$(\forall x,n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\text{Even}[n] \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{24}$$

$$(\forall x,n,m : n \in \mathbb{N})(n \neq 0 \wedge \neg\text{Even}[n] \wedge m = (x*x)^{(n-1)/2} \Rightarrow x*m = x^n) \tag{25}$$

$$(\forall x,n : n \in \mathbb{N})\ P'[x,n] = 0, \tag{26}$$

where

$$\begin{aligned} P'[x,n] = \quad & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \text{Even}[n] \textbf{ then } P'[x*x, n/2] \\ & \textbf{else } P'[x*x, (n-1)/2]. \end{aligned}$$

The proofs of these verification conditions are straightforward.

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program P is now almost the same as the previous one, but in the base case (when $n = 0$) the return value is 0.

$$\begin{aligned} P[x,n] = \quad & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \text{Even}[n] \textbf{ then } P[x*x, n/2] \\ & \textbf{else } x*P[x*x, (n-1)/2]. \end{aligned}$$

Now, for this buggy version of P we may see that all the respective verification conditions remain the same—and thus the program is correct—except one, namely, (21) is now:

$$(\forall x,n : n \in \mathbb{N})\ (n = 0 \Rightarrow 0 = x^n) \tag{27}$$

which itself reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debuging": we need to change the return value in the case $n = 0$ to 1.

Furthermore, in order to demonstrate how a bug might be located, we construct one more "buggy" example where in the "Even" branch of the program we have $P[x, n/2]$ instead of $P[x * x, n/2]$:

$$\begin{aligned} P[x,n] = \quad & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, we may see again that all the respective verification conditions remain the same as in the original one, except one, namely, (23) is now:

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x)^{n/2} \Rightarrow m = x^n) \tag{28}$$

which itself reduces to:

$$m = x^{n/2} \Rightarrow m = x^n$$

From here, we see that the "Even" branch of the program is problematic and one should satisfy the implication. The most natural candidate would be:

$$m = (x^2)^{n/2} \Rightarrow m = x^n$$

which finally leads to the correct version of P.

5 Correction: Finding Specification

Correcting a program in our case means finding a new program which is similar to the original one, however, obeying this time the given specification.

In order simplify the presentation, let us consider a bit simplified program schema, namely having only one "else" branch:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R[x]]]. \tag{29}$$

We assume the auxiliary functions S, C, and R are given with their specifications and the specification of $F : I_F$ and O_F is given as well. If the program schema is suitable for implementing this function F, there are in fact three possible places where a bug may occur: the function S is not suitable; the function C is not suitable; the function R is not suitable. Note that when we say *the function is not suitable* what we mean is that the specification of such a function is not suitable. This is because we want to preserve the modularity of our programs and therefore, as discussed earlier, the only knowledge concerning the auxiliary functions we are allowed to use is their specifications.

5.1 Bug due to the S function

Let us consider the case when the function S is not suitable. This means that the verification condition corresponding to the first branch of our conditionals will not be provable:

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]). \tag{30}$$

We need to find another function (program) S_{new} such that the formula

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S_{new}[x]]) \tag{31}$$

holds.

Due to the fact that the program S is specified by its specification I_S and O_S, we need to find a new output specification O_{Snew}, such that

$$(\forall x, y : I_F[x] \wedge I_S[x])\ (Q[x] \wedge O_{Snew}[x, y] \implies O_F[x, y]). \tag{32}$$

Going back to our buggy example, when in the base case (when $n = 0$) the return value is 0,

$$\begin{aligned} P[x,n] = \quad & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2], \end{aligned}$$

the unprovable verification condition is:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0 \Rightarrow 0 = x^n). \tag{33}$$

Looking for the new program S_{new} we obtain the following condition:

$$(\forall x, n, y : n \in \mathbb{N} \wedge \mathbb{T})\ (n = 0 \wedge O_{Snew}[x, n, y] \Rightarrow y = x^n), \tag{34}$$

which itself reduces to:

$$(\forall x, n, y : n \in \mathbb{N})\ (O_{Snew}[x, 0, y] \Rightarrow y = 1). \tag{35}$$

At this stage we are able to extract the specification of the S_{new} function: $I_{Snew}[x, n] \iff n \in \mathbb{N}$ and $O_{Snew}[x, 0, y] \iff y = 1$. One obvious solution is the constant function *One* on one argument x – here the second argument n is irrelevant. However, finding a program from a given specification may not be so obvious – it is called *program synthesis* and this is the subject of the next section.

5.2 Bug due to the R function

Let us consider the case when the function R is not suitable. This means that the verification condition corresponding to the second branch of our conditionals will not be provable:

$$(\forall x, y : I_F[x])\ (\neg Q[x] \wedge O_F[R[x], y] \implies O_F[x, C[x, y]]). \tag{36}$$

We need to find another function (program) R_{new} such that the formula

$$(\forall x, y : I_F[x])\ (\neg Q[x] \wedge O_F[R_{new}[x], y] \implies O_F[x, C[x, y]]) \tag{37}$$

holds.

Since the program R is specified by its specification I_R and O_R, we need to find a new output specification O_{Rnew}, such that

$$(\forall x, y, z : I_F[x] \wedge I_R[x])\ (\neg Q[x] \wedge O_{Rnew}[x, z] \wedge O_F[z, y] \Longrightarrow O_F[x, C[x, y]]). \tag{38}$$

One important remark: the function R_{new} may not be the identity function, because otherwise, intuitively, when invoking the recursive step, there will be no reduction on the argument and therefore, the program will not terminate. Moreover, one may prove that if the reduction function R (or R_{new} respectively) would be the identity function, then the termination condition (12) will never be satisfied.

In the example, when in the "Even" branch we have $P[x, n/2]$ instead of $P[x*x, n/2]$,

$$\begin{aligned} P[x,n] = \quad & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \text{Even}[n] \textbf{ then } P[x, n/2] \\ & \textbf{else } x * P[x*x, (n-1)/2], \end{aligned}$$

the unprovable verification condition is:

$$(\forall x, n, m : n \in \mathbb{N})\ (n \neq 0 \wedge \text{Even}[n] \wedge m = (x)^{n/2}\ \Rightarrow m = x^n) \tag{39}$$

which itself reduces to:

$$(\forall x, n, m : n \in \mathbb{N})\ (m = x^{n/2}\ \Rightarrow m = x^n).$$

We are now looking for a function (program) R_{new} which is similar to R, however obeying the respective verification condition. The type of R is $\mathbb{R} \rightarrow \mathbb{R}$ and therefore we suppose that R_{new} will have the same type.

The original R is defined on tuples in the following way: $R[x,n] = x^{n/2}$, i.e., its specification is $I_R[x,n] \iff T$, and $O_R[x,n,z] \iff z = x^n$. The specification of the new R_{new} should obey the condition:

$$(\forall x, n, m, z : n \in \mathbb{N} \wedge \mathbb{T})\ (n \neq 0 \wedge \text{Even}[n] \wedge O_{Rnew}[x,n,z] \wedge m = \mathit{Left}[z]^{\mathit{Right}[z]} \Longrightarrow m = x^n), \tag{40}$$

which itself reduces to:

$$(\forall x, n, m, z : n \in \mathbb{N})\ (n \neq 0 \wedge \text{Even}[n] \wedge O_{Rnew}[x,n,z] \Longrightarrow \mathit{Left}[z]^{\mathit{Right}[z]} = x^n). \tag{41}$$

At this stage we are able to extract the specification of the R_{new} function:

$$I_{Rnew}[x,n] \iff n \in \mathbb{N} \wedge n \neq 0 \wedge \text{Even}[n]$$

and

$$O_{Rnew}[x,n,z] \iff \mathit{Left}[z]^{\mathit{Right}[z]} = x^n.$$

As we discussed earlier, the identity function $Id[x,n] = \langle x, n \rangle$ may not be a solution, even though it obeys the specification for the unknown R_{new}. One possible solution is: $R_0[x,n] = \langle x*x, n/2 \rangle$, which in fact is in the original example (14).

5.3 Bug due to the C function

Let us consider the case when the function C is not suitable. This means that the verification condition corresponding to the second branch of our conditionals will not be provable:

$$(\forall x, y : I_F[x])\ (\neg Q[x] \wedge O_F[R[x], y] \Longrightarrow O_F[x, C[x,y]]). \tag{42}$$

Now, we need to find another function (program) C_{new} such that the formula

$$(\forall x, y : I_F[x])\ (\neg Q[x] \wedge O_F[R[x], y] \Longrightarrow O_F[x, C_{new}[x,y]]) \tag{43}$$

holds.

Since the program C is specified by its specification I_C and O_C, we need to find a new specification O_{Cnew}, such that

$$(\forall x, y, z : I_F[x] \wedge I_R[x])\ (\neg Q[x] \wedge O_F[R[x], y] \wedge O_{Cnew}[x,y,z] \Longrightarrow O_F[x,z]). \tag{44}$$

We do not go into more details, nether we display an example, because there is nothing essentially new.

Finding a relevant specification for the new (the unknown) function may require some human interaction. However, we are working on the automation of this discovery and we have obtained so far some promising results.

6 Correction: Generating a Program

Assume we have found a specification for an auxiliary function which would obey the verification conditions. Now, based on that specification, we need to generate a new program obeying that specification.

The method we use is the "Lazy Thinking" synthesis paradigm [3, 4] in the frame of the Theorema system. It is a deductive, scheme-based synthesis method, for algorithm invention and verification as a specific variant of systematic theory exploration. It is characterized

- by using a library of algorithm schemes
- and by using the information contained in failing attempts to prove the correctness theorem for an algorithm scheme in order to invent sufficient requirements on the auxiliary functions in the algorithm scheme.

The algorithm scheme is a definition of the desired algorithm in terms of unknown subalgorithms. The proof is likely to fail, as no information about the unknown subalgorithms is available. Following an analysis of the failing proof, conjectures are generated and added to the knowledge, such that the failure can be overcome. These conjectures turn out to be specifications for the unknown subalgorithms. Algorithms that satisfy the generated specifications can then either be retrieved from the knowledge base, or synthesized by lazy thinking in subsequent rounds of exploration.

The synthesis method is very powerful, and as it was recently shown [3], using the "Lazy Thinking" paradigm, one may (re)invent the Gröbner Bases algorithm [5].

7 Conclusions

The approach to program verification presented here is a result of an experimental work with the aim of combining programming, verification, proving, debugging and synthesis.

Although program synthesis may (or may not) replace completely the nowadays standard way of programming, we are convinced that interaction between the two complementary activities may lead to very fruitful results.

References

[1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[2] F. Blanqui, S. Hinderer, S. Coupet-Grimal, W Delobel, and A. Kroprowski. CoLoR, a Coq Library on Rewriting and Termination. In A. Geser and H. Søndergaard, editors, *Proceedings of 8th International Workshop on Termination*, Seattle, WA, USA, August 2006.

[3] Bruno Buchberger. Towards the Automated Synthesis of a Gröbner Bases Algorithm. In *RACSAM (Review of the Royal Spanish Academy of Science), Vol. 98/1*, pp. 65-75. 2005.

[4] Bruno Buchberger. Algorithm Invention and Verification by Lazy Thinking. In *Analele Universitatii din Timisoara, Seria Matematica - Informatica XLI*, pp. 41-70. 2003.

[5] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. (English translation Journal of Symbolic Computation 41 (2006) 475–511).

[6] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470-504, 2006.

[7] B. Buchberger and D. Vasaru. Theorema: The Induction Prover over Lists. Technical Report 97-20, RISC Report Series, University of Linz, Austria, June 9-10 1997.

[8] T. Jebelean, L. Kovacs, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*. To appear.

[9] M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.

[10] L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2006* pp. 67-74, Paphos, Cyprus, November 2006.

[11] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.

[12] N. Popov and T. Jebelean. A Prototype Environment for Verification of Recursive Programs. In Z. Istenes, editor, *Proceedings of FORMED'08*, pp. 121-130, March 2008. To appear as ENTCS volume, Elsevier.

[13] PVS group. PVS Specification and Verification System. http://pvs.csl.sri.com, 2009.

Case Studies for

Logical Based Synthesis in Theorema

Alois Altendorfer and Tudor Jebelean
RISC-Linz

Abstract

We study various case studies of program synthesis supported by theory exploration and proof generation. In this paper we present the case study on matching binary trees. In the context of a knowledge base containing the necessary properties of the theory, we start from the specification of the problem (input and output conditions) and we construct an inductive proof of the formula that for each input there exists an object satisfying the output condition. After constructing the proof the corresponding algorithm can be extracted immediately.

The approach in this work is practical and experimental. The purpose of the experiment is multifold: to construct the appropriate knowledge base necessary for this type of proofs, to find the natural deduction inference rules and the necessary strategies for their application, and finally to implement the corresponding prover in the frame of the *Theorema* system.

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue:1, pp. 66-81

1 Introduction

1.1 Approach

We study various cases of program synthesis supported by theory exploration and proof generation. The long term goal of the research is to design and implement methods for automatic synthesis of algorithms, based on the idea of extracting the algorithm from the proof of the existence of the result of the function. In more detail, we consider the specification of a function F ($I_F[X]$ - input, $O_F[X, Y]$ - output), and we construct the proof of the formula $\underset{X}{\forall} I_F[X] \Longrightarrow \underset{Y}{\exists} O_F[X, Y]$. The proof is usually constructed using an appropriate induction principle for the input domain, and the algorithm is usually easy to extract from the witnesses found during the proof.

In the current work we present various case studies in which we analyse the methods of the following mathematical process in the *Theorema* [2] system:

For a special domain we implement the basic constructs of the theory and explore further statements and properties (theory exploration), and then we focus on a certain function which we want to "implement" - that is we know certain properties (specification) of the function and we want the corresponding algorithm. First we construct an algorithm "by hand": a set of formulae which can be interpreted by the *Theorema* system. Secondly we construct the correctness proof for this algorithm. Last, we restart the proof with the algorithm as unknown and try to do synthesis according to the principle stated above.

As a concrete output, each case study provides a list of formulae, as well as the list of inference rules and proof methods, which are necessary for the proof to succeed.

1.2 Theory Exploration

As presented in [7], in mathematical logic, a theory is defined by its (first order) language, its knowledge base (collection of formulae), and its inference mechanism. The first order language consists of the set of predicate, function and constant symbols. The knowledge base K of a theory consists of first-order sentences over the language. We usually start with a collection of basic axioms and theorems known in the theory. The inference mechanism of the theory contains all reasoning rules useful for the development of the theory.

In our case studies we construct theories for natural numbers, for finite lists and for sets. For some case studies we use a combination of theories (e.g. theory of natural numbers and finite lists for computing the length of a list).

Following [4], in order to explore the theory of a certain special domain, we start with a small number of assumptions which define the basic constructs of the theory. After that we

introduce a new concept and we try to derive interesting properties of the new concept. The exploration can be done in two different ways namely in a bottom-up or top-down manner. On the one hand we try to derive new concepts using our knowledge and on the other hand we start with a desired property and the proof may fail. By analysis of the failing proof we find out which properties need to be added to the knowledge base.

In our case studies we use an inductive definition of the domain and therefore an appropriate induction principle is defined automatically. Different definitions lead to different principles and therefore we obtain different possibilities to proceed with proofs.

1.3 Proofs and Synthesis

On the one hand we do program verification in order to prove the correctness of algorithms and on the other hand we do program synthesis in order to obtain a correct algorithm. Starting with a specification and using different induction principles which are appropriate for the underlying domain we try to find a program satisfying our specifications. This is often done by using a formal definition as a specification in order to obtain a computable definition. Sometimes definitions in first order predicate logic using quantifiers are not "computable" (they cannot be automatically interpreted). A further field of program synthesis is of course finding a witness in a mathematical proposition because as soon as the witness is found in the proof an algorithm can be extracted from the proof - see e. g. [6].

The approach in this work is practical and experimental. Starting on basic definitions on a domain and appropriate induction principles we do proofs of different statements. This process helps us understanding and developing some insight into the problem. Generating these proofs may be done with the help of automated provers implemented in the *Theorema* system. Failing proofs may help us finding new lemmas or inference rules or just show that the desired goal is not proveable as expected. The main purpose of the work is to obtain methods on the different techniques which may be generalized. Additionally these methods may be implemented in the system. The experiments help us improving these methods and obtaining more insight into the synthesis problem.

2 Summary of Case Studies

The following examples are investigated: length of a finite list, reverse of a finite list, integer quotient and remainder, powerset of a finite set, composition of two substitutions, matching of pairs / tuples / general expressions, as well as unification of pairs / tuples / general expressions.

The approach has certain similarities with the one presented in [3, 4, 5], however it is essentially different because it does not use conjectures on the shape of the terms describing the algorithms.

We first start with examples using the domain of lists. The case studies on length and reverse of a list are quite easy but help us to gather some experience on proofs for program synthesis. The first version of our prover is able to do the correctness proofs of the algorithms for computing the length and the reverse as well as to perform program synthesis. Also some case studies for natural numbers (quotient and remainder computations) and for finite sets (computing the powerset) are analyzed. The inference rules and an appropriate proof strategy are detailed in the master thesis [1].

Our last case study is on matching and unification of expressions. Besides the analysis of the synthesis process of an algorithm for matching and unification we consider the problem of composing two substitutions. The detailed correctness proof of the composition algorithm turns out to be more complex as expected, but in the context of matching/unification the algorithm can be simplified. The current version of our prover is able to do program synthesis for length and reverse of a list and for matching binary trees. The latter is presented below and for details we refer to the master thesis.

3 Case Study on Matching Binary Trees

We want to solve the following problem:

- given a ground expression G and an expression E
- find a substitution σ
- such that IsMGU[G, E, σ] (i.e. ($G = E\,\sigma$))

The inductive definition of expressions automatically defines an induction principle, which can not be formulated in first order predicate logic. Thus, we lift it to the inference level in and obtain the following inference rule:

$$\frac{K \vdash P_{[\bullet c[a]]} \qquad K \vdash P_{[\bullet v[a]]} \qquad K, \mathrm{IsExpression}[\mathrm{E1}, \mathrm{E2}], P_{[\mathrm{E1}]}, P_{[\mathrm{E2}]} \vdash P_{[\langle \mathrm{E1}, \mathrm{E2} \rangle]}}{K \vdash \underset{\mathrm{IsExpression}[E]}{\forall} P_{[E]}}$$

i.e. in order to prove the universally quantified goal, prove first the base cases by using arbitrary but fixed constant and variable symbols instead of the universally quantified variable in the property P, then assume that the goal formula is true for arbitrary but fixed expressions E1 and E2 and try to prove the goal for the binary tree ⟨E1, E2⟩.

Excluding the case that the expression can be a variable $K \vdash P_{[\bullet v[a]]}$ leads to the inductive principle for ground expressions.

Note that the expression $P_{[E]}$ in the lower sequent does not denote an application, it only says that E occurs in the formula P and for instance $P_{[\bullet c[a]]}$ in the upper sequent denotes that the expression E is replaced by $\bullet c[a]$ in the formula P.

3.1 Synthesis Proof

Prove:

(Proposition (Matching)) $\underset{\text{IsGroundExpression}[G]}{\forall} \underset{\text{IsExpression}[V]}{\forall} \underset{\sigma}{\exists} (G = \text{Instance}[V, \sigma])$,

under the assumptions:

(Lemma (EmptySubstitution)) $\underset{E}{\forall} (E = \text{Instance}[E, \{\}])$,

(Algorithm (Instance1)) $\underset{a,\sigma}{\forall} (\text{Instance}[\bullet\text{c}[a], \sigma] := \bullet\text{c}[a])$,

(Algorithm (Instance2)) $\underset{Ex1,Ex2,\sigma}{\forall} (\text{Instance}[\langle Ex1, Ex2\rangle, \sigma] := \langle \text{Instance}[Ex1, \sigma], \text{Instance}[Ex2, \sigma]\rangle)$,

(Algorithm (Instance3)) $\underset{a,v,\sigma}{\forall} (\bullet\text{v}[v] \leftarrow \bullet\text{c}[a] \in \sigma \Rightarrow (\bullet\text{c}[a] = \text{Instance}[\bullet\text{v}[v], \sigma]))$,

(Lemma (ReplacementMembership)) $\underset{r,\sigma}{\forall} ((\sigma = \{r\}) \Rightarrow r \in \sigma)$,

(Lemma (ConstantUnequalTuple)) $\underset{a,Ex1,Ex2}{\forall} (\bullet\text{c}[a] \neq \langle Ex1, Ex2\rangle)$,

(Lemma (TupleUnequalConstant)) $\underset{a,Ex1,Ex2}{\forall} (\langle Ex1, Ex2\rangle \neq \bullet\text{c}[a])$,

(Algorithm (Instance4)) $\underset{Ex1,Ex2,v,\sigma}{\forall} (\bullet\text{v}[v] \leftarrow \langle Ex1, Ex2\rangle \in \sigma \Rightarrow (\langle Ex1, Ex2\rangle = \text{Instance}[\bullet\text{v}[v], \sigma]))$,

(Lemma (EqualityProperty)) $\underset{Ex1,Ex2,\sigma}{\forall} ((Ex1 = Ex2) \Rightarrow (\text{Instance}[Ex1, \sigma] = \text{Instance}[Ex2, \sigma]))$,

(Algorithm (Instance5)) $\underset{E,\sigma}{\forall} (\text{IsGroundExpression}[E] \Rightarrow (\text{Instance}[E, \sigma] := E))$,

(Lemma (TupleEquality)) $\underset{Ex1,Ex2,Ex3,Ex4}{\forall} ((Ex1 = Ex2) \bigwedge (Ex3 = Ex4) \Rightarrow (\langle Ex1, Ex3\rangle = \langle Ex2, Ex4\rangle))$,

(Lemma (InstanceComposition))

$\underset{Ex1,Ex2,\sigma,\theta}{\forall} ((Ex1 = \text{Instance}[\text{Instance}[Ex2, \sigma], \theta]) \Rightarrow (Ex1 = \text{Instance}[Ex2, \text{Comp}[\sigma, \theta]]))$,

We prove (Proposition (Matching)) by Induction on G and V .

1.: Take two constants which are equal:

Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:

Alternative proof 1: proved

In order to prove (Proposition (Matching)), we have to find $\sigma 2^*$ such that:

(1) $\bullet\text{c}[a1_0] = \text{Instance}\left[\bullet\text{c}[a1_0], \sigma 2^*\right]$.

Unifying (1) with (Lemma (EmptySubstitution)) we can find the witness $\left\{E \to \bullet\text{c}[a1_0], \sigma 2^* \to \{\}\right\}$ and we we obtain the Algorithm:

(AlgorithmBase) $\text{M}[\bullet\text{c}[a1_0], \bullet\text{c}[a1_0]] = \{\}$,

2.: Take two constants and assume:

(4) $\bullet\text{c}[a1_0] \neq \bullet\text{c}[b1_0]$.

Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:

Alternative proof 2: proved

We try to prove non-existence and try a proof by contradiction by assuming:

(5) $\bullet\text{c}[a1_0] = \text{Instance}[\bullet\text{c}[b1_0], \sigma 1_0]$.

Using the second part of (5) and by definition of (Algorithm (Instance1)) and we obtain:

(21) $\bullet\text{c}[a1_0] = \bullet\text{c}[b1_0]$,

Formula (21) contradicts to (4) and we obtain the Algorithm:

(AlgorithmBase) $M[\bullet c[a1_0], \bullet c[b1_0]] = \text{fail},$

3.: Take a constant and a variable symbol:
Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:
Alternative proof 1: proved
In order to prove (Proposition (Matching)), we have to find $\sigma 2^*$ such that:

(6) $\bullet c[a1_0] = \text{Instance}\left[\bullet v[v1_0], \sigma 2^*\right].$

Using (Algorithm (Instance3)) it remains to show:

(22) $\bullet v[v1_0] \leftarrow \bullet c[a1_0] \in \sigma 2^*$.

Using (Lemma (ReplacementMembership)) it remains to show:

(23) $\sigma 2^* = \{\bullet v[v1_0] \leftarrow \bullet c[a1_0]\}$.

By (23) we find a witness and we obtain the Algorithm:

(AlgorithmBase) $M[\bullet c[a1_0], \bullet v[v1_0]] = \{\bullet v[v1_0] \leftarrow \bullet c[a1_0]\}$.

4.: Take a constant symbol and a binary tree of expressions:
Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:
Alternative proof 2: proved
We try to prove non-existence and try a proof by contradiction by assuming:

(9) $\bullet c[a1_0] = \text{Instance}[\langle E1_0, E2_0\rangle, \sigma 1_0].$

Using the second part of (9) and by definition of (Algorithm (Instance2)) and we obtain:

(25) $\bullet c[a1_0] = \langle \text{Instance}[E1_0, \sigma 1_0], \text{Instance}[E2_0, \sigma 1_0]\rangle,$

Formula (25) contradicts to (Lemma (ConstantUnequalTuple)) and we obtain the Algorithm:

(AlgorithmBase) $M[\bullet c[a1_0], \langle E1_0, E2_0\rangle] = \text{fail},$

5.: Take a binary tree of ground expressions and a constant symbol:
Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:
Alternative proof 2: proved
We try to prove non-existence and try a proof by contradiction by assuming:

(11) $\langle G1_0, G2_0\rangle = \text{Instance}[\bullet c[b1_0], \sigma 1_0].$

Using the second part of (11) and by definition of (Algorithm (Instance1)) and we obtain:

(27) $\langle G1_0, G2_0\rangle = \bullet c[b1_0],$

Formula (27) contradicts to (Lemma (TupleUnequalConstant)) and we obtain the Algorithm:

(AlgorithmBase) $M[\langle G1_0, G2_0\rangle, \bullet c[b1_0]] = \text{fail},$

6.: Take a binary tree of ground expressions and a variable symbol:
Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:
Alternative proof 1: proved
In order to prove (Proposition (Matching)), we have to find $\sigma 2^*$ such that:

(12) $\langle G1_0, G2_0\rangle = \text{Instance}\left[\bullet v[v1_0], \sigma 2^*\right].$

Using (Algorithm (Instance4)) it remains to show:

(28) $\bullet v[v1_0] \leftarrow \langle G1_0, G2_0 \rangle \in \sigma 2^*$.

Using (Lemma (ReplacementMembership)) it remains to show:

(29) $\sigma 2^* = \{\bullet v[v1_0] \leftarrow \langle G1_0, G2_0 \rangle\}$.

By (29) we find a witness and we obtain the Algorithm:

(AlgorithmBase) $M[\langle G1_0, G2_0 \rangle, \bullet v[v1_0]] = \{\bullet v[v1_0] \leftarrow \langle G1_0, G2_0 \rangle\}$.

7.: Take a binary tree of ground expressions and a binary tree of expressions:

Since our goal (Proposition (Matching)) is existentially quantified we can try to find either a witness or try to prove that there does not exist one:

Alternative proof 1: proved

We assume :

(15) IsGroundExpression[$G1_0$],

(16) IsGroundExpression[$G2_0$],

(17) IsExpression[$\langle E1_0, E2_0 \rangle$],

(18) $G1_0$ = Instance[$E1_0$, M[$G1_0$, $E1_0$]],

and we have to find $\sigma 2^*$ such that:

(14) $\langle G1_0, G2_0 \rangle = \text{Instance}\left[\langle E1_0, E2_0 \rangle, \sigma 2^*\right]$.

Using the second part of (14) and by definition of (Algorithm (Instance2)) and we obtain:

(30) $\langle G1_0, G2_0 \rangle = \left\langle \text{Instance}\left[E1_0, \sigma 2^*\right], \text{Instance}\left[E2_0, \sigma 2^*\right] \right\rangle$

Using (18) as an instance in (Lemma (EqualityProperty)) and we obtain a new assumption:

(31) $\underset{\sigma}{\forall}$ (Instance[$G1_0$, σ] = Instance[Instance[$E1_0$, M[$G1_0$, $E1_0$]], σ]),

Using (15) as an instance in (Algorithm (Instance5)) and we obtain a new assumption:

(32) $\underset{\sigma}{\forall}$ (Instance[$G1_0$, σ] := $G1_0$),

By (31), we have to find metavariables such that:

(34) Instance[$G1_0$, σ^*] = Instance[Instance[$E1_0$, M[$G1_0$, $E1_0$]], σ^*].

Using the first part of (34) and by definition of (32) and we obtain:

(35) $G1_0$ = Instance[Instance[$E1_0$, M[$G1_0$, $E1_0$]], σ^*],

Using (35) as an instance in (Lemma (InstanceComposition)) and we obtain a new assumption:

(38) $G1_0$ = Instance[$E1_0$, Comp[M[$G1_0$, $E1_0$], σ^*]],

Using (Lemma (TupleEquality)) it remains to show:

(40) $\left(G1_0 = \text{Instance}\left[E1_0, \sigma 2^*\right]\right) \wedge \left(G2_0 = \text{Instance}\left[E2_0, \sigma 2^*\right]\right)$.

We can match (18) with a part of our goal (40) with the substitution $\left\{\sigma 2^* \rightarrowtail M[G1_0, E1_0]\right\}$ and we can try the following proof:

Alternative proof 1: failed

Show:

(41) $G2_0$ = Instance[$E2_0$, M[$G1_0$, $E1_0$]].

Using (Lemma (Lemma1)) and it remains to show:

(42) M[$G1_0$, $E1_0$] = M[$G2_0$, $E2_0$]

The proof of (42) fails. (No applicable inference rule was found.)

Alternative proof 2: proved

Go on and try to find an other witness:

We can match (38) with a part of our goal (40) with the substitution $\left\{\sigma 2^* \rightarrowtail \text{Comp}[M[G1_0, E1_0], \sigma^*]\right\}$ and we can try the following proof:

Alternative proof 1: proved

Show:

(43) $G2_0 = \text{Instance}[E2_0, \text{Comp}[\text{M}[G1_0, E1_0], \sigma^*]]$.

Using (Lemma (InstanceComposition)) it remains to show:

(45) $G2_0 = \text{Instance}[\text{Instance}[E2_0, \text{M}[G1_0, E1_0]], \sigma^*]$.

Consider this as a subproblem of the original one and it remains to show:

(46) $\sigma^* = \text{M}[G2_0, \text{Instance}[E2_0, \text{M}[G1_0, E1_0]]]$

By (46) we find a witness and we obtain the Algorithm:

(AlgorithmStep) $\text{M}[\langle G1_0, G2_0\rangle, \langle E1_0, E2_0\rangle] = \text{Comp}[\text{M}[G1_0, E1_0], \text{M}[G2_0, \text{Instance}[E2_0, \text{M}[G1_0, E1_0]]]]$.

□

3.2 Analysis

In the following we describe the inference rules and the proof strategy of the prover which generates the proof above.

3.2.1 Inference Rules

InductionPrinciple:

Now we consider that we have to show a property Q for two expressions, i.e. we have to show $\forall_{\text{IsGroundExpression}[G]} \forall_{\text{IsExpression}[E]} Q[G, E]$. Considering the following table we obtain an appropriate induction principle:

G \ E	•c[b]	•v[b]	⟨E₁, E₂⟩
•c[a]	•c[a] = •c[b] ⊢ Q[•c[a], •c[a]] •c[a] ≠ •c[b] ⊢ Q[•c[a], •c[b]]	⊢ Q[•c[a], •v[b]]	⊢ Q[•c[a], ⟨E₁, E₂⟩]
⟨G₁, G₂⟩	⊢ Q[⟨E₁, E₂⟩, •c[b]]	⊢ Q[⟨E₁, E₂⟩, •v[b]]	Q[G₁, E₁] ⊢ Q[⟨G₁, G₂⟩, ⟨E₁, E₂⟩]

In order to show a universally quantified goal with two expressions - E and a ground G - we have to show 6 base cases and one induction step. Note that in the step case we assume that our property Q just holds for the first part of the binary trees. This is for the reason that on the one hand the subproblem $Q[G_2, E_2]$ is not sufficient to find a witness for the combined problem $Q[\langle G_1, G_2\rangle, \langle E_1, E_2\rangle]$ and on the other hand a further subproblem using the solution of $Q[G_1, E_1]$ is generated during the proof.

IntroduceNewConstant:

$$\frac{K, P_{[t_0]} \vdash G}{K, \exists_x P_{[x]} \vdash G}\ t_0 \text{ new constant term}$$

The inference describes a common proof technique for an existentially quantified assumption. A new Skolem constant t_0 is introduced and the proof situation is reduced by assuming that P holds for t_0.

IntroduceNewConstant2:

$$\frac{K \vdash P_{[x_0]}}{K \vdash \underset{x}{\forall} P_{[x]}} \; x_0 \text{ new constant}$$

DeductionRule:

$$\frac{K, A \vdash B}{K \vdash A \Rightarrow B}$$

NCDeduction:

$$\frac{K, P_{[x_0]} \vdash Q_{[x_0]}}{K \vdash \underset{x}{\forall} P_{[x]} \Rightarrow Q_{[x]}} \; x_0 \text{ new constant}$$

The inference rule "IntroduceNewConstant2" describes a common proof technique for a universally quantified goal. In order to prove that a property P holds for all variables x we introduce a new constant symbol x_0 and the proof situation is reduced by proving that P holds for x_0.

The inference rule "DeductionRule" says, that if the goal formula is an implication $A \Rightarrow B$, then we can add A to the assumptions and we have to prove B from K and A.

Usually a universal quantifier has some properties for the variables and therefore we can combine the two inference rules introduced just above into one which is "NCDeduction".

IntroduceMetavariable:

$$\frac{K \vdash P_{[x^?]}}{K \vdash \underset{x}{\exists} P_{[x]}} \; x^? \text{ new metavariable}$$

This is an inference rule for proving an existential quantified goal. We introduce a new metavariable $x^?$ and try to show the goal formula P using $x^?$ instead of the variable. In the further proof we then hopefully find a witness term t satisfying P which then can be used instead of $x^?$.

IntroduceMetavariable2:

$$\frac{K, P_{[x^?]} \vdash G}{K, \underset{x}{\forall} P_{[x]} \vdash G} \; x^? \text{ new metavariable}$$

For a universally quantified assumption a metavariable can be introduced for which a witness has to be found.

AssumptionInstance:

$$\frac{K \vdash P_{[t]} \qquad K, Q_{[t]}, \underset{x}{\forall}(P_{[x]} \Rightarrow Q_{[x]}) \vdash G}{K, \underset{x}{\forall}(P_{[x]} \Rightarrow Q_{[x]}) \vdash G} \text{ where t is a suitable ground term}$$

satisfying $P_{[t]}$

This inference rule says that if we have a universal formula in our assumptions and t is a

suitable ground term satisfying $P_{[t]}$ (i.e. $P_{[t]}$ exists in the knowledge-base and can be matched with $P_{[x]}$) then we obtain $Q_{[t]}$ which can be added to the assumptions.

ReplaceEquality:

$$\frac{K_{\left[t_{[x_0]}\right]}, \underset{x}{\forall} f[x] := t_{[x]} \vdash G_{\left[t_{[x_0]}\right]}}{K_{[f[x_0]]}, \underset{x}{\forall} f[x] := t_{[x]} \vdash G_{[f[x_0]]}}$$

We want to prove a goal G from the knowledge base K and the definition of a function symbol f and we can match the variables of f with the constant symbols x_0. Then we can apply the definition of f and replace $f[x_0]$ with its definition $t_{[x_0]}$ in the goal G as well as in the knowledge K.

GoalTransformation:

$$\frac{K, \underset{x}{\forall}(P_{[x]} \Rightarrow Q_{[x]}) \vdash P_{[t]}}{K, \underset{x}{\forall}(P_{[x]} \Rightarrow Q_{[x]}) \vdash Q_{[t]}}$$

In order to prove a statement $Q_{[t]}$ by knowing the formula $P_{[x]} \Rightarrow Q_{[x]}$ for all variables x it is enough to prove the property P for the ground term t.

MatchAssumptionGoal:

$$\frac{\text{True}}{K, P_{[t]} \vdash P_{\left[x^?\right]}}$$ where $x^?$ is a metavariable and t a ground term

The conjecture should be returned by the system and after that this branch of the proof is True anyway because we have the same formula in the goal as in the assumptions, but this conjecture should also be used in other proof-branches afterwards. This means that if the metavariable $x^?$ exists in an other branch of the proof, then every occurrence of $x^?$ should be replaced by the witness term t. Note that the Skolem constants which occur in the term t must be introduced before the metavariable $x^?$, or if this is not the case it must be possible to reorder the proof in such a way that it is.

We can expand this inference rule to a more general one:

UnifyAssumptionGoal:

$$\frac{\text{True}}{K, P_{[y]} \vdash P_{[x]}}$$ where x and y are unifyable

So we are done if we are able to find witnesses for any metavariable and variable. This means that if we have a property $P_{[x]}$ in the goal and $P_{[y]}$ in the assumptions and a unifier of x and y can be found then we are done because suitable witnesses for the (meta-)variables exist.

ObtainWitness:

$$\frac{\text{True}}{K, \vdash G, x^? = t}$$ $x^?$ is a metavariable, t a ground term

We are done as soon as a witness can be found and $x^?$ is replaced by t in the other branches of the proof.

ConjunctiveGoal:

$$\frac{K \vdash A \qquad K \vdash B}{K \vdash A \bigwedge B}$$

In order to prove a conjunction in the goal we have to split the proof into cases and have to show all parts of the conjunction.

FinalProofSituation:

$$\frac{\text{True}}{K, A \vdash G, A}$$

FinalSituationContradiction:

$$\frac{\text{True}}{K, P, \neg P \vdash G}$$

A proof is finished if we obtain a contradiction in our assumptions.

FindSubproblem:

$$\frac{\text{True}}{K \vdash P_{[x_0]}}$$

We are done if we are able to detect that our goal is a subproblem of the initial problem $P_{[x]}$, e.g. $P_{[x_0]}$ can be solved by recursion.

3.2.2 Proof Strategy

Additionally to the rules we have to define a special order of the inference rules and some constraints on them. We obtain the strategy by our experience on other case studies and on the analysis of the proof tree.

Roughly speaking the proof proceeds in the following way: First we try to remove quantifiers in the way that for an universally quantifier in the goal we apply the induction principle, an existentially quantified variable in the assumption is replaced by a constant symbol as well and for an existentially quantified goal we introduce a metavariable.

In each branch we either try to find a substitution which unifies the two expressions or we try to prove that there does not exist such a substitution by assuming that there exists one and obtaining a contradiction.

In every attempt we transform our goal and try to obtain new knowledge until a final proof situation is reached and we are done.

In the induction step we introduce a new metavariable for a universally quantified assumption. This should be used with care and we just try this attempt because nothing else succeeds. A further difference to the other branches is that more than one witness can be found, but only one of them is suitable for all parts of the goal. Thus, for each witness an alternative proof branch is generated. At the end of the case step the system detects that the goal is a subproblem of the initial goal and returns the corresponding algorithm.

Therefore we obtain the following strategy:

- FinalSituationContradiction: A proof is finished as soon as we obtain a contradiction in the assumptions.
- FinalProofSituation
- MatchAssumptionGoal: For the special case that our goal is a conjunction we try to match an assumption with one part of the goal and prove that the obtained witness holds for the other parts of the conjunction. Additionally we construct an alternative proof in order to obtain a further witness if the one just obtained does not hold for the other parts of the conjunction.
- ObtainWitness: The proof branch is finished as soon as a witness is found.
- InductionPrinciple: Additionally to the different branches of the proof the basic construct of an algorithm is generated such that it can be returned as soon as the witnesses can be found.
- IntroduceNewConstant
- IntroduceNewConstant2
- IntroduceMetavariable: For an existentially quantified goal we either introduce a metavariable or try to disprove the goal by assuming that there exists a constant

fulfilling the respective property and try to obtain a contradiction. Thus, we construct two alternative proof-branches. If a witness can be found, the corresponding algorithm is returned and if we disprove the existence then an algorithm which returns "fail" is displayed in the proof.

- ReplaceEquality: Replace equalities in the assumptions and in the goal, i.e. apply definitions of functions in the proof situation.
- AssumptionInstance: We want to obtain new assumptions, not already existing ones. We differentiate our assumptions between formulae which are "global" and which are "local" by adding an additional information to each formulae. The knowledge-base which is used for the proof is set to be global and each formulae generated during the proof is set to be local. In order to do instances on the assumptions we now prefer to find a local instance on a global formula in order to obtain a new local formula.
- IntroduceMetavariable2: Find a witness for a universally quantified assumption. This rule is used not very often and is seen as a proof attempt if nothing else succeeds.
- GoalTransformation: We try to transform our goal by implications in the assumptions. Note that such a reduction of the goal does not always succeed and therefore we construct an alternative without reducing the goal by the corresponding implication.
- ConjunctiveGoal
- FindSubproblem: We try to detect a subproblem if nothing else can be applied.

3.2.3 Comments

A crucial difference to the other case studies mentioned above is that in the other examples the subproblems which can be solved by the induction hypothesis are stated at the beginning. Now the decomposition of the arguments does not work. Here the prover identifies a subproblem during the proof. Note that in the case of matching it is quite clear because the problem of computing the substitution of G_2 and $E_2\,\sigma_1$ where $\sigma_1 = M[G_1, E_1]$ is obviously a subproblem of the original one, because at least the first argument G_2 is smaller than the initial one which is $\langle G_1, G_2 \rangle$. Note that in unification this has not to be the case.

A further interesting point is the usage of the special version of the equality property. We take the assumption that two expressions are equal and obtain a new assumption that for any substitution the instance of the substitution on each expression is equal again. This property has to be used with care. Additionally the new equality holds for any substitution, but in the proof above the prover tries to find an appropriate one which then can be used for the original witness which has to be found.

Analysis of the proof above shows that the following subproblem which can be computed by the induction hypothesis (i.e. by recursion) has to be considered:

$$\underset{\sigma}{\exists}\left((G_1 = E_1\,\sigma) \bigwedge \underset{\theta}{\exists}\,(G_2 = (E_2\,\sigma)\,\theta)\right)$$

Note that following the process of program synthesis the second subproblem which uses the result of the first one is not stated at the beginning, it is generated during the proof. In this way we obtain a condition which has to be fulfilled such that we obtain a unifier.

A further crucial difference to the other case studies is that not in every case there exists a unifier of the two expressions, i.e. we have to disprove the property if we are not able to find a suitable witness.

As a concrete output our prover returns a routine of an algorithm of each induction-step which are the following:

(AlgorithmBase1) M[•c[$a1_0$], •c[$a1_0$]] = {},

(AlgorithmBase2) M[•c[$a1_0$], •c[$b1_0$]] = fail,

(AlgorithmBase3) M[•c[$a1_0$], •v[$v1_0$]] = {•v[$v1_0$] ← •c[$a1_0$]} .

(AlgorithmBase4) M[•c[$a1_0$], ⟨E1$_0$, E2$_0$⟩] = fail,

(AlgorithmBase5) M[⟨G1$_0$, G2$_0$⟩, •c[$b1_0$]] = fail,

(AlgorithmBase6) M[⟨G1$_0$, G2$_0$⟩, •v[$v1_0$]] = {•v[$v1_0$] ← ⟨G1$_0$, G2$_0$⟩} .

(AlgorithmStep) M[⟨G1$_0$, G2$_0$⟩, ⟨E1$_0$, E2$_0$⟩] = Comp[M[G1$_0$, E1$_0$], M[G2$_0$, Instance[E2$_0$, M[G1$_0$, E1$_0$]]]] .

These commands can be generalized to the following algorithm:

```
Algorithm["Matching- Binary- Trees", any[G1, G2, E1, E2, v1],
  M[⟨G1, G2⟩, ⟨E1, E2⟩] =                                    ""
    Comp[M[G1, E1], M[G2, Instance[E2, M[G1, E1]]]]
  M[G1, G1] = {}                                             ""]
  M[G1, •v[v1]] = {•v[v1] ← E1}                              ""
  M[G1, E1] = "fail"                                         ""
```

4 Conclusion and Further Work

The paper shows the main steps of program synthesis on the example of matching binary trees. Summing up the whole story we manage to do the following interesting points:

- Analysis of program synthesis: The construction of proofs helps us finding inference rules as well as methods of doing program synthesis. The analysis of different examples gives more insight and helps to develop more general mechanisms. The main point of the studies is the construction of a proof of an existencially quantified goal, which means that a witness for an unknown metavariable has to be found. It is known that an algorithm easily can be extracted from such a proof. Thus, we analyze the inference rules which are sufficient to run the proofs in natural style.
- Extraction of sufficient knowledge-base, inference rules and proof strategy: We provide a list of formulae which build up the knowledge-base, a list of inference rules as well as a suitable proof strategy which is necessary for the proof to succeed.
- Implementation of a prover: Using all the gathered knowledge and experience we implement our own prover.

The main goal of our further work is to improve the methods of program synthesis. Therefore we have to do more case studies in order to obtain more experience which helps us improving the prover and its strategy. Our current prover is able to do synthesis for the case studies on length and reverse of a list as well as matching and unification of binary trees. An extention to matching and unification of tuples and general expressions can be done easily because no new inference rules are necessary, we just have to modify our induction principle as well as our knowledge-base in an adequate manner. The long term goal is to design and implement further methods for automatic synthesis of programs, based on extracting the algorithm from the proof of the existence of the result of the function.

Since algorithm synthesis is very challenging task, there are several possible questions regarding the effectiveness and the efficiency of this approach in general. Since verification proofs are easier than synthesis proofs, why not just design the algorithm and verify it later? Moreover, in order to obtain very efficient algorithms it is often necessary to invent certain complex algorithm schemes, which is not harder then inventing the corresponding proof schemes (e. g. induction principles) for synthesis. Our answer to these questions is: We do not want to discuss here which approach from synthesis and verification is most suitable, but we

just want to investigate reasonable methods for algorithm synthesis. Our research shows that automatic reasoning can substantially improve the efficiency and reliability of algorithm invention, by completing the intelligent part of the algorithm design (in our case studies: the choice of the induction principle), with straightforward inference steps which can be easily performed by the machine. This may constitute the basis for further automation, e. g. by finding good principles for choosing the inductive schemes.

5 References

[1] A. Altendorfer. Case Studies in Proof Based Synthesis of Algorithms. Master Thesis. Technical Report 10-15, RISC, Johannes Kepler University Linz, Austria.

[2] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. Journal of Applied Logic 4(4), pp. 470-504. 2006.

[3] B. Buchberger. Mathematical Theory Exploration: Case Study Groebner Bases. Invited talk at SYNASC 2006, Timisoara.

[4] B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Strategy. AISC 2004, Springer LNAI 3249, pp. 236 - 250.

[5] B. Buchberger. A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In: MKM Workshop, Edinburgh, 2003. Electronic Notes in Theoretical Computer Science 93, pp. 24 - 59.

[6] L. Chiarabini, P. Audebaud. New Development in Extracting Tail Recursive Programs from Proofs. LOPSTR 2009.

[7] M. Hodorog. A. Craciun. A Case Study in Systematic Theory Exploration: Natural Numbers. Technical report no. 07-18 in RISC Report Series. 2007.

A Case Study in Systematic Exploration of Tuple Theory

Isabela Drămnesc, Adrian Crăciun*
West University of Timişoara,
Romania
{idramnesc,acraciun}@info.uvt.ro

Tudor Jebelean
RISC, Johannes Kepler University,
Linz, Austria
Tudor.Jebelean@jku.at

Abstract

We illustrate with concrete examples the systematic exploration of Tuple Theory in a bottom-up and top-down way.

In the bottom-up exploration we start from two axioms, add new notions and in this way we build the theory, check all the new notions introduced and prove some of them by the new prover which we created in the TH∃OREM∀ system.

In order to synthesize some algorithms on tuples (like e. g. insertion-sort) we use an approach based on proving. Namely, we start from the specification of the problem (input and output condition) and we construct an inductive proof of the fact that for each input there exists a solution which satisfies the output condition. The problem will be reduced to smaller problems, the method will be applied like in a "cascade" and finally the problem is so simple that the corresponding algorithm (function) already exists in the knowledge. The algorithm can be then extracted immediately from the proof. We present an experiment on synthesis of the insertion-sort algorithm on tuples, based on the proof existence of the solution. This experiment is paralleled with the construction (exploration) of the appropriate theory of tuples.

The main purpose of this research is to concretely construct examples of theories and to reveal the typical activities which occur in theory exploration, in the context of a specific application – in this case algorithm synthesis by proving.

1 Introduction

Mathematics is a technique of solving problems and the essence of mathematics is proving. By reasoning about the problems we obtain knowledge. This knowledge can be managed for theory exploration and reasoning about the knowledge can be implemented on computers.

In [3], the author talks about automated theorem proving versus mathematical theory exploration using computers and explains that automated theorem proving does not mean only to prove a single theorem, it means that by making the proof we are doing the process of exploring theory using computers. And he explains all the steps that one should follow in the process of exploration using the TH∃OREM∀ system.

In our paper we present a complete exploration of Tuple Theory, using the TH∃OREM∀ system. Our Tuple Theory is similar with the one described in [22], [27], [3], [11]. In contrast with the existent Tuple Theory from the TH∃OREM∀ system, based on higher order predicate logic, in our approach we use first order predicate logic, because this increases the feasibility of proving.

We use the TH∃OREM∀ system [8] – see `www.theorema.org` (implemented on top of Mathematica [26]), because this is a software system for automated theorem proving which uses mathematical logic and offers support for the user to formalize and to introduce new mathematical notions, to introduce conjectures and to prove the new notions introduced, to extract algorithms from the proved theorems, and to use these algorithms for computing, solving, for writing articles for publications, writing courses,

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 82-95

*Work carried out in the frame of the RISC Transnational Access Programme supported by the European Commission FP6 for Integrated Infrastructures Initiatives under the project SCIEnce - Symbolic Computation Infrastructure for Europe (contract No. 026133).

etc. The syntax is similar with mathematical logic and the proofs appear in a separate window and are generated in natural style, easy to read.

So, as a convention, in this paper all the definitions, axioms, propositions are written in the frame of the TH∃OREM∀ system.

1.1 Mathematical Theories

Mathematical theories (on computer or not) are built incrementally starting from a set of axioms, definitions and by adding propositions which we check and prove. A mathematical theory is expressed by the language, the knowledge base and the inference rules. The *language "$\mathscr{L}$"* contains a set of *predicate, function, and constant* symbols. The *knowledge base "$\mathscr{KB}$"* contains a set of predicate logic formulae over the language: *axioms, theorems, properties*. The *inference rules "$\mathscr{IR}$"* are rules that describe the reasoning system for the theory. The rules describe how to transform the current proof situation into a new one.

$$Theory = \langle \mathscr{L}, \mathscr{KB}, \mathscr{IR} \rangle$$

1.2 Systematic Exploration of Mathematical Theories

One can explore mathematical theories in a bottom-up or/and in a top-down way.

In the bottom-up exploration we start from a set of axioms and we add new notions that we check. We add properties describing the new notions introduced and we prove the propositions. In this way we build an entire theory. In section 2 we present the way we build the Tuple Theory in the bottom-up way.

In top-down exploration we start from the specification of the problem (input and output conditions) and we have to find an algorithm that satisfies the specification. This process is paralleled with exploring (building) the theory of tuples. In section 3 we describe, in general, the method that we use in order to synthesize algorithms and also an experiment on synthesis of the sorting algorithms for tuples, in particular the insertion-sort algorithm.

The process of systematic exploration of mathematical theories has a lot of advantages: all the information that the mathematician needs are visible and more accessible, he can use them without the need of thinking what is the notion that he has to know for proving something. Using computers it will be very useful for mathematicians because they do not have to do everything by hand and computers are much more faster than humans.

1.3 Related Work

Strategies for the systematic exploration of theories in a bottom-up and in a top-down way one can find in [3]. In fact many of the definitions and assumptions from Tuple Theory which we use in this paper are developed in that research work. Our previous work on bottom-up exploration of Tuple Theory is presented in [15].

Program synthesis in computational logic is a well studied field. For an overview of several approaches that have been investigated see [2]. A tutorial on synthesis of programs one can find in [20].

Deductive techniques for program synthesis are presented in [21] and techniques for constructing induction rules for deductive synthesis proofs are presented in [9].

Bruno Buchberger introduces a method for algorithm synthesis, called "lazy thinking", see [4] and also a model for theory exploration using schemes, see [5]. The "lazy thinking" method is implemented in the TH∃OREM∀ system [8], see [6]. In this approach, a “program schemata” (the structure of the solving term) is given to the synthesis procedure. In contrast, in our approach this schemata is also

discovered by the procedure, only the inductive principle of the domain is given. Another approach for automatize the synthesis of logic programs was introduced in [18].

A case study on exploration of natural numbers based on schemes (a bottom-up exploration) is done in [16] and also the decomposition of natural numbers (top-down exploration) in [12].

In [13] one can see how to use program transformation techniques to develop six versions of sorting algorithms. For classifications of the sorting algorithms see [19], [17], [23].

The synthesis of sorting algorithms on tuples in higher order predicate logic, using the "lazy thinking" method you can find in [11]. A case study on synthesis of the Merge-Sort algorithm one can find in [24], [25].

In this paper we explore the Tuple Theory in a bottom-up and in a top-down way. In the top-down way we use a method for proving in the context of constructive synthesis that is applied like in a "cascade" to all the reduced problems. Other case studies using this method on synthesis of some algorithms for tuples you can find in [14].

In contrast with the other case studies on sorting algorithms existent in the literature, in particular sorting of tuples, in our approach we use a different method for synthesize the algorithms and the use as much as possible of first order logic. This method was introduced in 2008, for details see [10]. A case study of transforming recursive procedures into tail recursive using this method of synthesis one can find in [1].

2 Context

2.1 Tuple Theory - Bottom-Up Exploration

The Tuple Theory is expressed by the language, the knowledge base and the inference rules.

The *language "Tuple-$\mathscr{L}$"* is similar with the one described in [22],[7] and is a set of *predicate, function, and constant* symbols. The predicates are: the unary predicate "IT" that describes the type of the objects from the theory ("IT" stands for "Is Tuple") and the binary equality "=" predicate. The functions are: the binary function "$\smile$" that adds an element at the beginning of a tuple and the unary identity "Id" function. As a constant we consider the empty tuple represented by $\langle\rangle$.

$Tuple-\mathscr{L} = \langle\langle IT, =\rangle, \langle\smile, Id\rangle, \langle\langle\rangle\rangle\rangle$

The *Knowledge base* consists from a set of axioms: the generation and the unicity. These axioms are describing a tuple. The "generation" axiom states that the empty tuple is a tuple and that by adding an element to a tuple we also obtain a tuple. The "unicity" axiom states that by adding an element to a tuple we cannot obtain the empty tuple, and that two tuples $u \smile X$ and $v \smile Y$ are equal if and only if $u = v$ and $X = Y$. We will express them using the syntax from the TH∃OREM∀ system:

$$Axiom[\,"generation",\ (IT[\langle\rangle]) \underset{a,IT[X]}{\forall} IT[a \smile X]\ ]$$

$$Axiom[\,"unicity",\ \underset{u,IT[X]}{\forall}(u \smile X \neq \langle\rangle) \underset{u,v,IT[X],IT[Y]}{\forall}((u \smile X = v \smile Y) \Rightarrow ((u = v) \wedge (X = Y)))\ ]$$

The induction axiom:

$$Axiom[\,"induction\ principle\ ",\\ (\mathfrak{F}[\langle\rangle] \bigwedge \underset{a,IT[X]}{\forall} (\mathfrak{F}[X] \Rightarrow \mathfrak{F}[a \backsim X])) \Rightarrow \underset{IT[X]}{\forall} \mathfrak{F}[X]\,]$$

The *inference mechanism* is lifted from the induction axiom, where $\mathfrak{F}$ stands for any predicate.

The notations: $\langle\rangle$ is the empty tuple, $a \backsim \langle\rangle$ is a tuple having one single element, $a \backsim X$ is an element added to the tuple X. This notations allows to use only first order predicate logic.

Note that in our formalism we use square brackets as in $f[x]$ for function application and for predicate application, instead of the usual round parantheses as in $f(x)$.

For the bottom-up exploration of this Tuple Theory we start from the two axioms "generation" and "unicity" and introduce in the knowledge base new notions like: concatenation of 2 tuples, the function that replaces an element from a tuple, the definition for the minimum element from a tuple, the definition for adding an element at the end of a tuple, the definition for the reverse of a tuple, the definition of a sorted tuple (the merge-sort algorithm).

All these notions are checked with concrete examples in the TH∃OREM∀ system and some propositions are automatically proved by the new prover created in the system.

E.g. We check if the "generation" axiom that we introduce is correct:

$$Compute[IT[2 \backsim (4 \backsim \langle\rangle)], using \rightarrow Axiom[\,"generation\ "]]$$

$$True$$

$$Compute[IT[today \backsim (is \backsim (Monday \backsim \langle\rangle))], using \rightarrow Axiom[\,"generation\ "]]$$

$$True$$

Introduce the definition for concatenation:

$$Definition[\,"concatenation\ ", any[u,X,Y],\\ \left.\begin{array}{c}((\langle\rangle \asymp Y) = Y)\\ (Y \asymp \langle\rangle = Y)\\ ((u \backsim X) \asymp Y = u \backsim (X \asymp Y))\end{array}\right]$$

and test it:

$$Compute[\langle\rangle \asymp (6 \backsim \langle\rangle), using \rightarrow Definition[\,"concatenation\ "]]$$

$$6 \backsim \langle\rangle$$

$$Compute[today \backsim (is \backsim \langle\rangle) \asymp a \backsim (beautiful \backsim (day \backsim \langle\rangle)), using \rightarrow Definition[\,"concatenation\ "]]$$

$$today \backsim (is \backsim (a \backsim (beautiful \backsim (day \backsim \langle\rangle))))$$

Introduce the definition for the function that replaces an element from a tuple

$$Definition[\,"replace\ ", any[u,v,n,X],\\ \left.\begin{array}{c}(replaceT[u \backsim X, 0, v] = v \backsim X)\\ (replaceT[u \backsim X, n, v]) = (u \backsim replaceT[X, n-1, v])\end{array}\right]$$

and test it using the Compute command:

$$Compute[replaceT[(3 \backsim (2 \backsim \langle\rangle)), 0, 5], using \rightarrow Definition[\,"replace\ "]]$$

$5 \smile (2 \smile \langle\rangle)$

$Compute[replaceT[(3 \smile (3 \smile (3 \smile \langle\rangle))), 1, 5], using \rightarrow Definition["replace"]]$

$3 \smile (5 \smile (3 \smile \langle\rangle))$

$Compute[replaceT[2 \smile (2 \smile (2 \smile \langle\rangle)), 2, 6], using \rightarrow Definition["replace"]]$

$2 \smile (2 \smile (6 \smile \langle\rangle))$

$Compute[replaceT[today \smile (is \smile (First \smile (March \smile (2010 \smile \langle\rangle)))), 3, June],$
$using \rightarrow Definition["replace"]]$

$today \smile (is \smile (First \smile (June \smile (2010 \smile \langle\rangle))))$

Introduce the definition for the minimum element from a tuple:

$$Definition["minimum", any[u, v, T],\ \begin{array}{c} minelem[u \smile \langle\rangle] = u \\ (minelem[u \smile (v \smile T)] = minelem[u \smile T]) \Longleftarrow (u \leq v) \\ minelem[u \smile (v \smile T)] = minelem[v \smile T] \end{array}\ ]$$

and test it with the Compute command:

$Compute[minelem[1 \smile (2 \smile \langle\rangle)], using \rightarrow \{Definition["minimum"]\}]$

1

$Compute[minelem[5 \smile (2 \smile \langle\rangle)], using \rightarrow \{Definition["minimum"]\}]$

2

$Compute[minelem[1 \smile (0 \smile (3 \smile \langle\rangle))], using \rightarrow Definition["minimum"]]$

0

$Compute[minelem[10 \smile (18 \smile (2 \smile (5 \smile \langle\rangle)))], using \rightarrow$
$Definition["minimum"]]$

2

Introduce the definition for adding an element at the end of a tuple:

$$Definition["adding at the end", any[a, b, X],\ \begin{array}{c} (\langle\rangle \frown b = b \smile \langle\rangle) \\ ((a \smile X) \frown b) = (a \smile (X \frown b)) \end{array}\ ]$$

and test it with the Compute command:

$Compute[\langle\rangle \frown 6, using \rightarrow Definition["adding at the end"]]$

$6 \smile \langle\rangle$

$Compute[(2 \smile (3 \smile \langle\rangle)) \frown 6, using \rightarrow Definition["adding at the end"]]$

$$2 \backsim (3 \backsim (6 \backsim \langle\rangle))$$

$$Compute[(a \backsim (b \backsim (b \backsim (c \backsim \langle\rangle)))) \frown End, using \rightarrow Definition[\,"adding\ at\ the\ end"]]$$

$$a \backsim (b \backsim (b \backsim (c \backsim (End \backsim \langle\rangle))))$$

Introduce the definition for the reverse of a tuple:

$$Definition[\,"reverse", any[u,X], \begin{array}{c} reverseT[\langle\rangle] = \langle\rangle \\ reverseT[u \backsim X] = reverseT[X] \frown u \end{array}]$$

and test it:

$$Compute[reverseT[1 \backsim (8 \backsim (3 \backsim (4 \backsim \langle\rangle))))], using \rightarrow \{Definition[\,"reverse"], Definition[\,"adding\ at\ the\ end"]\}]$$

$$4 \backsim (3 \backsim (8 \backsim (1 \backsim \langle\rangle))))$$

$$Compute[reverseT[Hagenberg \backsim (of \backsim (Castle \backsim (the \backsim \langle\rangle))))], using \rightarrow \{Definition[\,"reverse"], Definition[\,"adding\ at\ the\ end"]\}]$$

$$the \backsim (Castle \backsim (of \backsim (Hagenberg \backsim \langle\rangle))))$$

Introduce the merge-sort algorithm and all the notions that we need:

$$Definition[\,"simple\ tuple", any[u,X], \begin{array}{c} is[\langle\rangle] = True \\ is[u \backsim \langle\rangle] = True \\ is[u \backsim X] = False \end{array}]$$

$$Definition[\,"parts\ of\ the\ tuple", any[u,v,X], \begin{array}{c} ps[\langle\rangle] = \langle\rangle \\ ps[u \backsim \langle\rangle] = u \backsim \langle\rangle \\ ps[u \backsim (v \backsim X)] = u \backsim ps[X] \\ pd[\langle\rangle] = \langle\rangle \\ pd[u \backsim \langle\rangle] = \langle\rangle \\ pd[u \backsim (v \backsim X)] = v \backsim pd[X] \end{array}]$$

$$Algorithm[\,"combine\ the\ parts\ of\ the\ tuple", any[u,v,X,Y], \begin{array}{c} comp[\langle\rangle,\langle\rangle] = \langle\rangle \\ comp[\langle\rangle, v \backsim Y] = v \backsim Y \\ comp[u \backsim X,\langle\rangle] = u \backsim X \\ (comp[u \backsim X, v \backsim Y] = u \backsim comp[X, v \backsim Y]) \Leftarrow u \leq v \\ comp[u \backsim X, v \backsim Y] = v \backsim comp[u \backsim X, Y] \end{array}]$$

Add the algorithm:

$$Algorithm[\,"sorting\ merge\text{-}sort", any[X], \begin{array}{c} msort[\langle\rangle] = \langle\rangle \\ (msort[X] = X) \Leftarrow is[X] \\ msort[X] = comp[msort[ps[X]], msort[pd[X]]] \end{array}]$$

and check if it is correct:

$$Compute[msort[7 \frown (4 \frown (5 \frown (2 \frown (6 \frown (1 \frown (3 \frown \langle\rangle)))))))], using \rightarrow \{Definition["simple\ tuple(C)"], Definition[$$
$$Algorithm["combine\ the\ parts\ of\ the\ tuple"], Algorithm["sorting\ merge\text{-}sort"]\}]$$

$$1 \frown (2 \frown (3 \frown (4 \frown (5 \frown (6 \frown (7 \frown \langle\rangle))))))$$

2.2 Reasoning—the new prover "TuplesProverTM"

We created a prover, "TuplesProverTM", in the TH∃OREM∀ system that generates automatically some proofs of the propositions occurring in the process of theory exploration. This prover combines some of the existing provers with rewriting rules.

E.g. Introduce the Proposition "simple tuple" and prove it using the theory "tuples axioms" that contains the two axioms "generation" and "unicity" by the prover "TuplesProverTM":

$$Proposition["simple\ tuple", IT[a \frown (b \frown (c \frown (b \frown (b \frown \langle\rangle))))]]$$

$$Prove[Proposition["simple\ tuple"], using \rightarrow Theory["tuples\ axioms"],$$
$$by \rightarrow TuplesProverTM]//Last//Timing$$

$$\{0.452 Second, proved\}$$

Introduce the Proposition "equality":

$$Proposition["equality",$$
$$\underset{a,IT[X],IT[Y]}{\forall}((X = Y) \Rightarrow (a \frown X = a \frown Y))\Big]$$

and prove it by the prover "TuplesProverTM":

$$Prove[Proposition["equality"], using \rightarrow Theory["tuples\ axioms"],$$
$$by \rightarrow TuplesProverTM]//Last//Timing$$

$$\{0.452 Second, proved\}$$

Also, Introduce and prove the proposition:

$$Proposition["singleton\ f", \underset{u,IT[X]}{\forall}(\langle u\rangle \asymp X = u \frown X)\Big]$$

$$Prove[Proposition["singleton\ f"], using \rightarrow \{Theory["tuple\ 3"],$$
$$Definition["singleton"]\}, by \rightarrow TuplesProverTM, //Last$$

$$proved$$

Details about the implementation and the proofs you can see in [15].

3 Synthesis of algorithms on tuples – Top-Down Exploration

In this section we give a general description of our approach and a case study in synthesis of the insertion-sort algorithm using the method that we present.

In Program synthesis we deal with the question "Given a specification how one can find an executable program that satisfies the specification". There are different methods for synthesize programs in computational logic, see [2]. We use the constructive synthesis. This means that from the specification we generate a conjecture, prove the conjecture and from the proof extract the algorithm.

The novel specific feature of our approach is applying a method for proving the conjecture like in a "cascade".

3.1 Proof based synthesis

We start from the specification of the problem (input and output condition). Given the *Problem Specification:*

Input: $I_F[X]$
Output: $O_F[X,Y]$

find the definition of F such that $\underset{X:I_F}{\forall} O_F[X,F[X]]$.

Please note that we use square brackets for function application and for predicate application as in $f[x]$, instead of the usual round parantheses as in $f(x)$. Moreover the quantifier variable may be qualified with a property, as in the formula above which stands for: $(\forall X)(I_F[X] \Longrightarrow O_F[X,F[X]])$.

We synthesize F by proving $\underset{X:I_F}{\forall} \underset{Y}{\exists} O_F[X,Y]$ by some induction principle (which is not automatically chosen).

We represent a tuple like an element added to a tuple. By the notations we understand: $\langle\rangle$ the empty tuple, by $a \frown \langle\rangle$ a tuple having one single element, by $a \frown X$ we understand an element added to the tuple X. This notation allows to use only first order predicate logic.

Base case: We prove $\underset{Y}{\exists} O_F[\langle\rangle,Y]$. If the proof succeeds to find a ground term R such that $O_F[\langle\rangle,R]$, then we know that $F[\langle\rangle] = R$.

Induction step: We take arbitrary but fixed a and X_0 (satisfying I_F), we assume $\underset{Y}{\exists} O_F[X_0,Y]$ and we prove $\underset{Y}{\exists} O_F[a \frown X_0,Y]$. We Skolemise the assumption by introducing a new constant Y_0 for the existential Y. If the proof succeeds to find a witness $T[a,X_0,Y_0]$ (term depending on a,X_0, and Y_0) such that $O_F[a \frown X_0,T[a,X_0,Y_0]]$, then we know that $F[a \frown X] = T[a,X,F[X]]$.

Extracting the algorithm: Finally the algorithm that we extract from the proof is:

$$F[X] = \begin{cases} R, & \text{if} X = \langle\rangle \\ T[a,X,F[X]], & \text{if} X \neq \langle\rangle \end{cases}$$

In the induction step the problem will be reduced into smaller and smaller problems in case we cannot find directly the witness. Then we create a new problem (that is simpler) and we apply the same method. This process is repeated and the method is applied like in a "cascade". Finally our problem is that simple that the appropriate functions for constructing the witness term can be found in the knowledge base. From the proof we extract the algorithm as a case distinction equality in which the witnesses found during the proof occur on the right hand side.

During the proof we detect various propositions that we need for the proof to succeed. We prove these propositions and after that introduce them into the knowledge and use them in the proof. We will not go into details in this paper for the process of inventing propositions. We just want to mention that by adding propositions to the knowledge base we explore the theory. So the process of proving is paralleled to the process of building the theory.

3.2 Case study: Synthesis of Insertion-Sort

Problem Specification:

$I_{\text{InsertionSort}}[X] : True$

$O_{\text{InsertionSort}}[X,Y] : \begin{cases} X \approx Y \\ \text{IsSorted}[Y] \end{cases}$

The notation $X \approx Y$ means that X has the same elements as Y and $IsSorted[Y]$ means that Y is the sorted version of X.

Our *knowledge base* contains the definitions: " $\approx$ " (having the same elements), "IsSorted", " $\triangleleft$ " (an element is member into a tuple), "dfo" (delete first occurrence of an element from a tuple):

$$Definition\left[\ "\approx\ ", any[a,X,Y],\ \begin{array}{c}\langle\rangle\approx\langle\rangle\\ ((a\frown X)\approx Y)\Longleftrightarrow((a\triangleleft Y)\wedge X\approx dfo[a,Y])\end{array}\ \right]$$

$$Definition\left[\ "IsSorted", any[a,b,X],\ \begin{array}{c}IsSorted[\langle\rangle]\\ IsSorted[a\frown\langle\rangle]\\ IsSorted[a\frown(b\frown X)]\Longleftrightarrow((a\leq b)\wedge IsSorted[b\frown X])\end{array}\ \right]$$

$$Definition\left[\ "\triangleleft\ ", any[a,b,X],\ \begin{array}{c}a\not\triangleleft\langle\rangle\\ a\triangleleft a\frown X\\ (a\neq b)\Longrightarrow((a\triangleleft b\frown X)\Longleftrightarrow(a\triangleleft X))\end{array}\ \right]$$

$$Definition\left[\ "dfo", any[a,b,X],\ \begin{array}{c}dfo[a,\langle\rangle]=\langle\rangle\\ dfo[a,a\frown X]=X\\ (a\neq b)\Longrightarrow(dfo[a,b\frown X]=b\frown dfo[a,X])\end{array}\ \right]$$

In order to synthesize the sorting algorithm we generate the conjecture (by convention in this paper this is written in the TH∃OREM∀ system, see [8]):

$$Proposition\left[\ "problem\ InsertionSort", \underset{XY}{\forall\exists}\left\{\begin{array}{c}X\approx Y\\ IsSorted[Y]\end{array}\right.\ \right]$$

and try to prove it.

It is sufficient to prove the proposition "problem InsertionSort" because we do not take into consideration the type of the objects. We consider that all the objects are tuples and not checking the type does not influence the proof (that is why the input condition is $I_{\text{InsertionSort}}[X]:True$ and not $I_{\text{InsertionSort}}[X]:Xistuple$.

For proving the proposition "problem InsertionSort" we use the induction principle: $(P[\langle\rangle]\wedge\underset{a,X}{\forall}(P[X]\Longrightarrow P[a\frown X]))\Longrightarrow\underset{X}{\forall}P[X]$.

The induction principle is in second order logic, but by instantiation it becomes first order.

Base case: $X=\langle\rangle$

Prove $\underset{Y}{\exists}\left\{\begin{array}{c}\langle\rangle\approx Y\\ \text{IsSorted}[Y]\end{array}\right.$

Find witness $Y^?$ such that $\left\{\begin{array}{c}\langle\rangle\approx Y^?\ (\text{G1.1})\\ \text{IsSorted}\left[Y^?\right]\ (\text{G1.2})\end{array}\right.$

We take our first goal (G1.1) and try to do matching with the knowledge base. By matching $\langle\rangle\approx\langle\rangle$ (from the definition " ≈ ") with $\langle\rangle\approx Y^?$ we find *witness* $Y^?=\langle\rangle$.

We take this witness and ckeck it in the second goal (G1.2).

Check for (G1.2): IsSorted[$\langle\rangle$] is true by definition "IsSorted".

So, we obtain a branch of the algorithm $\mathscr{S}[\langle\rangle]=\langle\rangle$. (*)

Induction Step:

Assume: $\left\{\begin{array}{ll}X_0\approx Y_0 & (\text{H1.1})\\ \text{IsSorted}\,[Y_0] & (\text{H1.2})\end{array}\right.$ and

Prove: $\left\{ \begin{array}{l} a \frown X_0 \approx Y \\ \text{IsSorted}[Y] \end{array} \right.$ (G)

Find witness $Y^?$ such that: $\left\{ \begin{array}{l} a \frown X_0 \approx Y^? \\ \text{IsSorted}\left[Y^?\right] \end{array} \right.$

We rewrite this by the definition " $\approx$ " and we must prove:

$$\left\{ \begin{array}{ll} a \triangleleft Y^? & \text{(G2.1)} \\ X_0 \approx \text{dfo}\left[a, Y^?\right] & \text{(G2.2)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G2.3)} \end{array} \right.$$

Theory must contain the propositions:

$$Proposition\left[\textit{"IsElem in Insertion"}, \underset{a,X}{\forall}(a \triangleleft Insertion[a,X])\right]$$

$$Proposition\Big[\textit{"The same elements in Insertion"},$$
$$\underset{a,X,Y}{\forall}((X \approx Y) \Longrightarrow (X \approx dfo[a, Insertion[a,Y]]))\Big]$$

$$Proposition\Big[\textit{"Sorting using Insertion"},$$
$$\underset{a,X}{\forall}(IsSorted[X] \Longrightarrow IsSorted[Insertion[a,X]])\Big]$$

We instantiate proposition "IsElem in Insertion" and we obtain:

a $\triangleleft$ Insertion[a, X_0] (H2.1).

By matching (H2.1) with (G2.1) we find *witness* $Y^?$=Insertion[a,Y_0].

Check for (G2.2): $X_0 \approx \text{dfo}\left[a, \text{Insertion}\left[a, Y_0\right]\right]$.

We instantiate "The same elements in Insertion" and we know:

$(X_0 \approx Y_0) \Longrightarrow (X_0 \approx$ dfo[a, Insertion[a, Y_0]]) (H2.2)

By (H2.2), (H1.1), by Modus Ponens we know:

$X_0 \approx$ dfo[a, Insertion[a, Y_0]] (H2.3) that is the same with (G2.2).

Check for (G2.3): $\text{IsSorted}\left[\text{Insertion}\left[a, Y_0\right]\right]$

Instantiate proposition "Sorting using Insertion" and we know:

IsSorted[Y_0] $\Longrightarrow$ IsSorted[Insertion[a, Y_0]] (H2.4)

By (H2.4), (H1.2), by Modus Ponens we obtain:

IsSorted[Insertion[a, Y_0]] that is the same with (G2.3).

3.2.1 Case when we know the function "Insertion"

The theory contains also the definition "Insertion":

$$\begin{array}{l} Definition[\textit{"Insertion"}, any[a,b,X], \\ \qquad\qquad Insertion[a, \langle\rangle] = a \frown \langle\rangle \\ (a \leq b) \Longrightarrow (Insertion[a, b \frown X] = a \frown (b \frown X)) \\ (a > b) \Longrightarrow (Insertion[a, b \frown X] = b \frown Insertion[a,X]) \end{array}\Big]$$

We match this definition and we obtain our witness and if the knowledge have this definition this means that it contains also the properties for satisfying the conditions to be the witness. So, we obtain: $\mathscr{S}$[a$\frown X_0$] = Insertion$\left[a, Y_0\right]$ (**)

By (*), (**) and by the transformation rules $X_0 \longrightarrow X$, $Y_0 \longrightarrow \mathscr{S}[X]$ we extract the algorithm:

$$\underset{a,X}{\forall}\left\{ \begin{array}{l} \mathscr{S}[\langle\rangle] = \langle\rangle \\ \mathscr{S}[a \frown X] = \text{Insertion}[a, \mathscr{S}[X]] \end{array} \right.$$

3.2.2 Case when we do not know the function "Insertion"

In this case the problem is reduced into a simpler problem (synthesize the function insertion).

We write X as an element a added to the tuple T. The input condition is extended (by adding that T is sorted) and the output condition remains the same.

Reduced Problem:

Problem Specification: (X: a,T)

$I_{\text{Insertion}}[a,T]$: IsSorted[T]

$$O_{\text{Insertion}}[a,T,Y]: \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

Generate the conjecture:

$$\left[\textit{Proposition}[\textit{"reduced problem Insertion"}, \underset{a,T}{\forall}(\textit{IsSorted}[T] \Longrightarrow \underset{Y}{\exists}((a \frown T \approx Y) \wedge \textit{IsSorted}[Y]))\right]$$

Prove the proposition "reduced problem Insertion" by induction over T.

We use the Induction Principle Head/Tail (Decomposition)

$$\text{T: } \begin{cases} \langle\rangle & (P[\langle\rangle]) \\ b \frown T & (P[T] \Longrightarrow P[b \frown T]) \end{cases}$$

We take a arbitrary, but fixed and we start to prove.

Base case: T=$\langle\rangle$

Prove $\underset{Y}{\exists} \begin{cases} a \frown \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases}$

Find witness $Y^?$ such that $\begin{cases} a \frown \langle\rangle \approx Y^? & \text{(G3)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4)} \end{cases}$

We take our first goal (G3) and try to do matching with the knowledge base.

Theory must contain the proposition:

$$\left[\textit{Proposition}\left[\textit{"reflexivity in } \approx \textit{"}, \underset{X}{\forall}(X \approx X)\right]\right]$$

We use the proposition "reflexivity in ≈ " and by matching with (G3) we find witness $Y^? = a \frown \langle\rangle$.

Check for (G4): IsSorted[a$\frown\langle\rangle$]. This is true by the definition "IsSorted".

So, we obtain Insertion[a, $\langle\rangle$] = a $\frown\langle\rangle$. (*I*)

Induction Step: T: b$\frown$T

Assume: IsSorted[T]$\Longrightarrow\underset{Y}{\exists}\begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H*4)

Prove: IsSorted[b$\frown$T]$\Longrightarrow\underset{Y}{\exists}\begin{pmatrix} a \frown (b \frown T) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G*4)

For proving (G*4) we assume IsSorted[b$\frown$T] (H4) and

prove: $\underset{Y}{\exists}\begin{pmatrix} a \frown (b \frown T) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G4)

By (H4) and "property of sort", by Modus Ponens we obtain IsSorted[T] (H5).

The theory must contain the proposition:

$$\left[\textit{Proposition}\left[\textit{"property of sort"}, \underset{a,X}{\forall}(\textit{IsSorted}[a \frown X] \Longrightarrow \textit{IsSorted}[X])\right]\right]$$

From (H5), (H*4), by Modus Ponens we know: $\underset{Y}{\exists}\begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H6)

In (H6), by Skolem we obtain: $\begin{cases} a \frown T \approx Y_0 & \text{(H6.1)} \\ \text{IsSorted}\left[Y_0\right] & \text{(H6.2)} \end{cases}$

By the definition " $\approx$ " we rewrite (H6.1) into: $\begin{cases} a \lhd Y_0 & (\text{H7.1}) \\ T \approx \text{dfo}\,[a, Y_0] & (\text{H7.2}) \\ \text{IsSorted}\,[Y_0] & (\text{H6.2}) \end{cases}$

For proving (G4):

Find witness $Y^?$ such that $\begin{cases} a \smile (b \smile T) \approx Y^? (\text{G4.1}) \\ \text{IsSorted}\left[Y^?\right] (\text{G4.2}) \end{cases}$

Alternative I:

Using proposition "reflexivity in $\approx$" we do matching in (G4.1) and obtain witness: $Y^? = a \smile (b \smile T)$.

Check in (G4.2): IsSorted[a$\smile$(b$\smile$T)].

By definition "IsSorted" we have to prove: (a$\leq$b) $\wedge$ IsSorted[b$\smile$T].

IsSorted[b$\smile$T] is true by (H4).

So, (a$\leq$b)$\Longrightarrow Y^? = a \smile (b \smile T)$ and we obtain:

Insertion[a,b$\smile$T]=a$\smile$(b$\smile$T) iff a$\leq$b. (*II*)

Alternative II:

By IsSorted$\,[Y_0\,]$ (H6.2), a $\lhd Y_0$ (H7.1) and by $\neg$(a$\leq$b) (from the proposition "order"), and if we consider that a is the smallest element from Y_0, we can use the property "Sorting II" and we obtain that IsSorted$\,[b \smile Y_0]$ (H5).

The theory must contain the propositions:

$$\textit{Proposition}\left[\textit{"order"}, \underset{a,b}{\forall} \begin{array}{c} (\neg(a \leq b)) \Longrightarrow (a > b) \\ a > b \Longrightarrow a \neq b \end{array}\right]$$

$$\textit{Proposition}\left[\textit{"Sorting II"}, \underset{a,b,X}{\forall}((\textit{IsSorted}[b \smile X] \wedge (b \lhd (b \smile X)) \wedge (a \leq b)) \Longrightarrow \textit{IsSorted}[a \smile (b \smile X)])\right]$$

We do matching of (H5) with (G4.2) and we obtain witness $Y^? = b \smile Y_0$.

Check in (G4.1): a$\smile$(b$\smile$T)$\approx b \smile Y_0$.

By the definition " $\approx$ " we rewrite this into:

$\begin{cases} a \lhd b \smile Y_0 & (G4.12) \\ b \smile T \approx \text{dfo}\,[a, b \smile Y_0] & (G4.13) \end{cases}$

By the proposition "order" we know that a$\neq$b (H5.1).

By (H5.1) and by the definition " $\lhd$ " in (G4.12) we must prove that a $\lhd Y_0$ that is true by (H7.1).

Check for (G4.13):

By the definition "dfo" and by (H5.1) we rewrite (G4.13) into:

$b \smile T \approx b \smile \text{dfo}\,[a, Y_0]$ (G4.14)

By proposition "dfo $\approx$ " we have to prove:

$T \approx \text{dfo}\,[a, Y_0\,]$ that is true by (H7.2).

Theory must contain the proposition:

$$\textit{Proposition}\left[\textit{"dfo} \approx \textit{"}, \underset{a,T,X}{\forall}((a \smile T \approx a \smile X) \Longrightarrow (T \approx X))\right]$$

So, our witness is correct: $Y^? = b \smile Y_0$.

We obtain Insertion[a,b$\smile$T]=b$\smile Y_0$ iff a>b. (*III*)

From (*I*), (*II*), (*III*) and by the transformation rules $Y_0 \longrightarrow$Insertion[a, T], T$\longrightarrow$X we obtain the algorithm:

$$\underset{a,b,X}{\forall}\left(\begin{array}{c} Insertion[a,\langle\rangle] = a \smile \langle\rangle \\ Insertion[a, b \smile X] = \left\{\begin{array}{c} a \smile (b \smile X), a \leq b \\ b \smile Insertion[a, X] \ , a > b \end{array}\right. \end{array}\right)$$

4 Conclusion and Future Work

We presented in this paper the systematic exploration of Tuple Theory. In section 2 we describe the way how one can build the theory of tuples starting from two axioms, the "generation" and the "unicity" axioms, by adding new notions and properties of these notions in the frame of the TH∃OREM∀ system. We check all the new notions introduce and also we prove some of the propositions introduced by the prover created in the TH∃OREM∀ system. In section 3 we describe how one can build the theory of tuples by synthesizing algorithms on tuples, in particular the synthesis of insertion-sort algorithm. The problem was reduced into simpler problems and we apply the same method like in a "cascade". During the proof we introduce in the knowledge base the propositions that we need to continue the proof, in this way the process of proving is paralleled to the process of theory exploration.

The method for synthesize algorithms presented here and also the case study on synthesis of the insertion-sort algorithm are subject for implementation in the frame of the TH∃OREM∀ system. We plan to extend this work by investigating other tuple operations and other sorting algorithms (like e.g. merge-sort algorithm).

References

[1] P. Audebaud and L. Chiarabini. New Development in Extracting Tail Recursive Programs from Proofs. In *Pre-Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'09)*, Coimbra, Portugal, September 9-11, 2009, 2009.

[2] D. Basin, Y. Deville, P. Flener, A. Hamfelt, J. F. Nilsson, and Mathematical Modelling. Synthesis of Programs in Computational Logic. In *Program Development in Computational Logic*, pages 30–65. Springer, 2004.

[3] B. Buchberger. Theory Exploration with Theorema. *Analele Universitatii Din Timisoara, Ser. Matematica-Informatica*, XXXVIII:9–32, 2000.

[4] B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. *Analele Universitatii din Timisoara, Seria Matematica - Informatica*, XLI:41–70, 2003. special issue on Computer Science - Proceedings of SYNASC'03.

[5] B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Strategy. In J. Campbell B. Buchberger, editor, *Proceedings of AISC 2004*, volume 3249 of Springer LNAI, pages pages 236–250, 2004.

[6] B. Buchberger and A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. *Electr. Notes Theor. Comput. Sci.*, 93:24–59, 2004.

[7] B. Buchberger and A. Craciun. Algorithm synthesis by lazy thinking: Using problem schemes. In *SYNASC 2004*, pages 90–106. Mirton, Timisoara, Romania, 2004.

[8] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[9] A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing Induction Rules for Deductive Synthesis Proofs. *Electron. Notes Theor. Comput. Sci.*, 153(1):3–21, 2006.

[10] L. Chiarabini. Extraction of Efficient Programs from Proofs: The Case of Structural Induction over Natural Numbers. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Local Proceedings of the Fourth*

Conference on Computability in Europe: Logic and Theory of Algorithms (CiE'08), pages 64–76, Athens, Greece, June 15-20, 2008, 2008.

[11] A. Craciun and B. Buchberger. Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking. Technical Report 04-16, RISC–Linz, Austria, October 2004.

[12] A. Craciun and M. Hodorog. Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration. In D. Petcu, D. Zaharie, V. Negru, and T. Jebelean, editors, *SYNASC07*, 2007.

[13] J. Darlington. A Synthesis of Several Sorting Algorithms. *Acta Inf.*, 11:1–30, 1978.

[14] I. Dramnesc and T. Jebelean. Proof Based Synthesis of Sorting Algorithms. Technical Report 10-17, RISC Report Series, University of Linz, Austria, 2010.

[15] I. Dramnesc, T. Jebelean, and A. Craciun. Case Studies in Systematic Exploration of Tuple Theory. Technical Report 10-09, RISC Report Series, University of Linz, Austria, May 2010.

[16] M. Hodorog and A. Craciun. A Case Study in Systematic Theory Exploration: Natural Numbers. Technical Report 07-18, RISC–Linz, Austria, October 2007.

[17] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison-Wesley Longman Publishing Co.,Inc., Redwood City, CA, USA, 1998.

[18] I. Kraan, D. Basin, and A. Bundy. Middle-Out Reasoning for Logic Program Synthesis. In *Proc. 10th Intern. Conference on Logic Programing (ICLP '93)* (Budapest, Hungary), pages 441–455, Cambridge, MA, 1993. MIT Press. Also available as Technical Report MPI-I-93-214.

[19] K. K. Lau. A Note on Synthesis and Classification of Sorting Algorithms. *Acta Informatica*, 27:73–80, 1989.

[20] K. K. Lau and G. Wiggins. A Tutorial on Synthesis of Logic Programs from Specifications. In P. Van Hentenryck, editor, *Proc. 11th Int. Conf. on Logic Programming*, pages 11–14. MIT Press, 1994.

[21] Z. Manna and R. Waldinger. Synthesis: Dreams ? programs. *IEEE Transactions on Software Engineering*, 5:294–328, 1979.

[22] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*, volume Volume 1: Deductive Reasoning. Addison-Wesley, 1985.

[23] Susan M. Merritt. An Inverted Taxonomy of Sorting Algorithms. *Commun. ACM*, 28(1):96–99, 1985.

[24] D. R. Smith. A Problem Reduction Approach to Program Synthesis. In *IJCAI*, pages 32–36, 1983.

[25] D. R. Smith. Top-down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, 1985. (Reprinted in "Readings in Artificial Intelligence and Software Engineering", Eds. C. Rich and R. Waters, Morgan Kaufmann Pub. Co., Los Altos, CA, 1986, pp.35-61).

[26] S.Wolfram. *The Mathematica Book*. Wolfram Media Inc., 2003.

[27] R. Waldinger Z. Manna. *The Logical Basis for Computer Programming*, volume Volume 2: Deductive Systems. Addison-Wesley, 1990.

Simple Non-Deterministic Strategy in Rewriting and Its Application for Verification

Alexander Letichevsky and Alexander Letichevsky Jr.
Glushkov Institute of Cybernetics
Kiev, Ukraine
let@cyfra.net
lit@iss.org.ua

Vladimir Peschanenko
Kherson State University
Kherson, Ukraine
vladimirius@gmail.com

Abstract

The article is dedicated to Algebraic Programming System (APS) - the first term rewriting system (TRS), which uses the rewriting rules system (RRS) and strategies separately and to Insertion Modelling System (IMS). IMS is a basic system for the Verification of Formal Specification (VFS) system. The main differences of TRS ELAN, MAUDE, STRATEGO from APS are described.

1 Introduction

Algebraic Programming System APS [1] was developed by the departments 100,105 of Glushkov Institute of Cybernetics of the National Academy of Science of Ukraine [2] in 1987. It was the first system of term rewriting which used the RRS and strategies separately. The last version of APS system was created with colabaration of Research Institute of Information Technologies of Kherson State Univeristy [3] in 2009.

Unlike traditional approach oriented to the usage of canonical RRS with "transparent" strategy of their application, in APS it is possible to combine arbitrary RRS with different strategies of rewriting. Such an approach essentially extends the possibilities of rewriting technique enlarging the flexibility and expressiveness of it. The APS integrates four main programming paradigms in the following way. The main part of the program can be written in the form of rewriting systems. Imperative and functional programming is used for the definition of strategies. Logic paradigm is realized on a base of rewriting using built-in unification procedure. One of it's most important application is the Insertion Modelling System (IMS). The main defereneces between APS and ELAN[4], MAUDE[5], STRATEGO[6] are represented.

IMS system was created in 2010 as a realisation of insertion modelling theory[7]. It consists of APS, Model Drivers (MD) and Insertion Machines (IM), where are MD - is a tool for traces generation of transitional systems(such tool depends of traces algorithms generation only)[7], IM - realization of insertion functions[8] which depends on application arias of IMS system (for example IM's for verification: Semantics of MSC[9], IM for evidence algorithm[10], IM for Semantics of Basic Protocols[11] etc). We name IMS system and such insertional machines the Verification of Formal Specification (VFS) system.

So, the article is devoted to a special rewriting strategy of APS system and its application for VFS system.

The section 2 desribes a functional posibilities between APS system and ELAN, MAUDE, STRATEGO systems. The section 3 devoted to RRS representation in APS and some special algorithms of work with RRS. We describe our proposal for non-deterministic rewriting strategy (ND Strategy) in section 4. In section 5 VFS system we propouse IM for Semantics of Basic Protocols of VFS system. The section 6 presents such special strategy application in verification.

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 96-104

2 APS and Other

Let's demonstrate the comparison of functional possibilities between APS system[1] and ELAN[4], MAUDE[5], STRATEGO[6] systems.

Table 1: Comparison of Functional Possibilities Between TRS's

No	Name	Strategies Number	None Typing Strategies and rules	Procedural Language	Possibilities of Language Extension	User Manual Publication	Connection to the External Modules	Compilation	Dynamical Creation of the of the RRS	Support	Application Area	Commercial Products	Country
1	ELAN	arbitrary	-	-	+	1992	-	+,-	-	+	EA	*	France
2	STRATEGO	arbitrary	+	-	-	1994	-	+,-	+	-	SA	*	Netherlands
3	MAUDE	7	+	-	-	1995	-	+,-	-	+	MA	*	USA
4	APS	arbitrary	+	+	+	1987	**	***	+	+	AA	ACP	Ukraine

Where EA,SA is transformation systems including compilers, interpreters, static analyzers, domain-specific optimizers, code generators, source code refactorers, documentation generators, and document transformers; MA is general logics and logical frameworks, specification languages, declarative programming languages, semantics of programming languages and models of computation, distributed systems, formal tools and formal interoperability, reflection and metaprogramming, object-oriented modelling and programming, real-time systems, bio informatics, mobile languages, network protocols and active networks; AA - algebraic programming, insertional modelling, program transformation, general logics and logical frameworks, specification languages, declarative programming; ACP - VRS (Verification of Requirement Specification), TERM(School System of Computer Algebra); * - can't find any information about concrete projects; ** - to the binary files and system commands; *** - C - version of the arbitrary paths of program; +,- means that the system supports compilation of some small sub-set of system's language.

Without doubts due to quite developed typification in the system, MAUDE has more benefits than other systems (the process of evaluation with integers numbers).

It is clear from the table that APS doesn't concede to well-known systems of terms rewriting as per all criterions.

Let's compare the capacity of terms rewriting systems taking the example of finding of n - number of Fibonacci (in this case we are interested in total operation time of the program which is used for rewriting only). We are going to perform test on DELL VOSTRO 1500 (CPU Intel Core duo 2.0, Memory 2 Gb, HDD 160 Gb). The results of launching of this program in different systems of term rewriting are presented:

Table 2: The results of launching of algorithms finding of Fibonacci n-number

No	System names	Fibonacci number (in seconds)					
		15	20	21	22	23	24
1	Interpreter of ELAN	0	2	6	11.5	18.5	28
2	Interpreter of Stratego	0	3	7	12	21	34
3	Interpreter of MAUDE	0.004	0.04	0.068	0.072	0.104	0.236
4	Procedures of APS	0	1	1	3	4	7
5	Rewriting systems of APS	0	2	2	4	6	10

Without doubts due to quite developed typification in the system, MAUDE has more benefits than other systems (the process of evaluation with integers numbers uses some additional structures etc). In this connection, there is a quite limited number of rewriting strategies in the system that considerably complicates the algorithms realized in it.

From the other side, APS was considered to be one of the slowest systems of term rewriting. Talking into consideration the results of capacity of rewriting, it can be said that APS is a quite quick system of term rewriting (after the elimination of some deficiencies). Surely, it doesn't have compiler, but instead of it we propose a number of tools for convenience of programming in APLAN (Algebraic Programming LANguage, the language of APS system) as well as in C++. These tools take APS to the new more qualified level (the compilers of ELAN, MAUDE, Stratego don't support language possibilities completely).

Let's examine in details the mentioned above deficiencies of APS system. The first version of APS system had the memory leaks. The "garb" operator for collection of waste in the program was realized in APS system but as practice proved this operator didn't delete memory completely. In a course of analysis of the source code the designers discovered places of memory leaks. It was precipitated out 7 bytes of memory at the procedure calling, described in APLAN language. However, the actual C++ code which performs the calling of these procedures does not contain obvious calling functions of memory selection. It led to the fact that the used memory increased very fast at execution of the program and some small instances simply were terminated due to lack of memory in computer. As a result we have taken the following decision to implement the technology Smart Pointers in APS[12]. Thus, by means of this technology in the second version of APS system it was possible to be saved of this deficiency.

3 APS Rewriting

General definition of syntax of RRS is the following:

$$
\begin{array}{l}
< \textit{rewriting system} >::= \textit{rs}(< \textit{list of variables separated by ","} >) \\
\quad (< \textit{list of rules separated by ","} >) \\
< \textit{rule} >::= < \textit{simple rule} > \mid < \textit{conditional rule} > \\
< \textit{simple rule} >::= < \textit{algebraic expression} > = < \textit{algebraic expression} > \\
< \textit{conditional rule} >::= < \textit{condition} > - > < \textit{simple rule} > \\
< \textit{variable} >::= < \textit{identifier} >
\end{array}
$$

Each application of RRS in APS satisfies the following conditions now:

1. One of the rules of the system is applied or arithmetic operation is performed at each step of rewriting.
2. The choice of a rule is made according to the sequence in which the rules have been written.

Each RRS in APS applies to a term $t \in T_\Omega(Z)$, where $T_\Omega(Z)$ is algebra of terms of some algebraic program (see subsection, Ω is a signature of operation of this algebra and Z is a generic set of terms with zero arities, with some strategy: $applr$ applies RRS to a term ones, $appls$ applies while it's possible etc.

Rewriting machine of APS realized strategy $applr$ which is a base for all strategies in APS. APS used a special language for faster application of REM - REM (REwriting machine) language[13]. Algebric definition of REM language is represented by the next algebra.

Let $T_\Omega(Z)$ be the base algebra of terms for some algebraic program. Set Ω is an operation signature of this algebra, Z - is a generating set of terms of arity 0. Rewriting machine (REM) programs produce many-sorted algebra $R = (R_k)_{k \in Z}$ above $T_\Omega(Z \cup V)$, where V - is a set of variables of this program, R_k - set of programs of rank k. The corresponding signature contains the next operation:

1. $+ : R_k \times R_k \to R_k, k > 0$.
2. $test(\omega((),\ldots,())) : R_{n+m-1} \to R_n, n > 0, m > 0, \omega \in \Omega, m = ART(\omega)$. (function ART returns arity of term).
3. $match(t) : R_n \to R_{n+1}, n \geq 0, t \in T_\Omega(Z \cup V)$.
4. $rewrite(t) \in R_0, t \in T_\Omega(Z \cup V)$.
5. $If(u, rewrite(t)) \in R_0; u, t \in T_\Omega(Z \cup V)$.
6. $hash(t) : R_k \to R_k, k > 0, t \in S_{T_\Omega(Z \cup V)}$, where $S_{T_\Omega(Z \cup V)}$ - is set of marks and term of arity 0 from $T_\Omega(Z \cup V)$.

Operations 4 and 5 have zero arity and they produce elements for algebra of REM language. As follows from definitions.

SSR which was successfuly converted in representation of REM language is called $REM - program$.

3.1 Dynamical Adding and Removing of Rules from the RRS

The process of application of RRS to current term has a few stages: conversion to REM-program (one time for each system only), interpreting of REM-program by the kernel of rewriting machine. It means that if we have to update RRS (add,remove or update some rule), APS system has to rebuild it into REM-program each time. But this operation demands more time, especially for big RRS. So, in APS system we have realized two operators:

- *remove_rule_rs* - `the function for removing of rule from RRS without rebuilding of it.`

- *add_rule_rs* - `the function for adding of rule to RRS without rebuilding, dynamical adding of rule in REM language.`

More interesting function is *add_rule_rs*. This function adds new rule into RRS without its rebuilding. Its means that we should add new operation from signature of algebra which corresponds to insertion of rule into REM-program. We should create new RRS by the next ideas to this effect:

1. Using list of variables from current RRS and for insertion of rule we should create new RRS.

2. Conversion of new RRS into REM-program.

3. Insertion of new REM-program into current RRS (there are two possibilities to add new rule into RSS: to make it the first and to make it the last).

4. Using already known size of previous RSS making of number of insertion rewriting rule (this number can be used by appls strategy).

The insertion algorithm of two REM-programs uses the next ideas (*new_rs* - new REM-program, *old_rs* - old REM-program):

1. In any case *new_rs* will have the,next template $match(new\_rs\_m)rewrite(\_)$ or $match(new\_rs\_m)If(\_,\_)$.

2. If *old_rs* has the next template (in terms of APS) $hash(\_)\_$ then if hash contains the main mark of *new_rs_m* then we should call recursive to $hash(type(new\_rs\_m))$ and *ne_rs* if it's possible or if not then we should use $+$ operator from signature of algebra.

3. If *old_rs* has the next template $\_+\_$ then we should try to add in first argument and if it is not possible then to try to add in second argument.

4. If *old_rs* has the next template $match(old\_rs\_m)last\_part$ (it is possible only in one case: if body of match of *old_rs* and *new_rs* has equal main mark) then we should eliminate the *new_rs_m* from *new_rs_m* and call algorithm recursively with $last\_part$.

5. At last, we should use *hash* operator if it's possible or $+$ operator if it's not.

4 ND Strategy

A non-deterministic rewriting strategy is an enhancement of theory of Set Functions for Functional Logic Programmig[14] by the means of fuzzy sets and its application of rewriting. We use the next conditions for realization of a non-deterministic rewriting strategy :

1. After each successful application of some rule from RRS rewriting we continue with rules written below current.

2. Results of application of non-deterministic rewriting are separated by the special non-deterministic operation $+$.

Let show how it defences from the $applr$ strategy. The strategy $applr$ is represented as a function with two arguments: term which should be rewritten and REM-program. The high level of realization $applr$ strategy in APLAN language is $napplr$:

$$
\begin{array}{l}
NAME\ napplr; NAME\ appl;\\
napplr := proc(t,p)loc(pr,Yes,s)(\\
\quad let(p,Rs\ pr\ p);\\
\quad s := (p,t\ Nil,pr)+Nil;\\
\quad appls(s,appl);\\
\quad let(s,1:s);\\
\quad yes \to t := s\\
);
\end{array}
$$

The main part of it is represented by RRS $appl$, which is applied interactively to a state of rewriting machine s. The initial state contains program p, initial term t joines with constant Nil, and array pt which consists of skips. It is a precondition which defines requirement to $applr$. The postcondition of it is the following: if some system R which corresponds to a program p is applicable to a term t, then after stopping $s = (1 : R(t))$. If not then $s = 0$ after stopping. The high level of realization of $applr$ strategy in APLAN language is $appl$:

$$
\begin{array}{l}
appl := rs(p,q,r,t,pr)(\\
\quad (1:t)+r = (1:t),\\
\quad (p+q,t,pr)+r = (p,t,new(pr))+(q,t,pr)+r,\\
\quad (p,t,pr)+r = perform(p,t,pr)+r\\
);
\end{array}
$$

The information about functions $appls, applr, napplr, appl, perform, new$ is represented in[15].

For execution of these conditions we should rebuild the result s after application of $appl$ and to add non-deterministic application of RRS $appl$ to r. So, lets consider non-deterministic rewriting strategy nds_napplr in APLAN language:

$$
\begin{array}{l}
NAMES\ nds\_applr, elim\_colon, nds\_appl, nds\_appls;\\
nds\_napplr := proc(t,p)loc(pr,Yes,s)(\\
\quad let(p,Rs\ pr\ p);\\
\quad s := (p,t\ Nil,pr)+Nil;\\
\quad appls(s,nds\_appl);\\
\quad elim\_colon(s);\\
\quad yes \to t := s\\
);\\
elim\_colon := rs(x,y)(\\
\quad x+y = elim\_colon(x)+elim\_colon(y),\\
\quad x:y = y\\
);\\
nds\_appl := rs(p,q,r,t,pr)(\\
\quad (1:t)+r = nds\_appls(1:t),r),\\
\quad (p+q,t,pr)+r = (p,t,new(pr))+(q,t,pr)+r,\\
\quad (p,t,pr)+r = perform(p,t,pr)+r\\
);
\end{array}
$$

$$
\begin{array}{l}
nds\_appls := proc(nds\_res, other)(\\
\quad appls(other, nds\_appl); \\
\quad return(nds\_res + other) \\
);
\end{array}
$$

The optimization of ND Strategy is a very important thing, because if RRS is bigger, checking all of non-deterministic choices will be considerably slower. So, the main question is whether such ND strategy is optimal ot not?

Theorem of optimization of ND Strategy 4.1. *All non-deterministic simultaneously impossible cases will be eliminated from a consideration of REM-programs on a one step of interpretation with the help of hash operator of signature* Ω.

Let's consiter a simultaneously impossible cases on a one step interpretation: different mark of a term t of set of term with arity 0. But all of thouse cases will be in *hash* operator (it follows from its definition). It means that we choose only one case from all simultaneously impossible cases for current step of interpretation. So, the current realization of ND Strategy is optimal.

5 Semantics of Basic Protocols of VFS system

Each basic protocol is a Hoare triple $\alpha \rightarrow < P > \beta$, where P is a process, α and β are precondition and postcondition of process P, respectively. α and β are represented by logical expressions of the base language and define conditions on the set of states of a system. A process of a basic protocol is a finite convergent process over the set C of environment actions, which may contain the set A of agent actions. We shall use the following notation for arbitrary basic protocols: $\boldsymbol{pre}(b) = \alpha, \boldsymbol{post}(b) = \beta$, and the process of B is denoted as P_b.

Each basic protocol defines properties of a system and can be understood as a statement of temporal logic: if the precondition is true then the process of a protocol can start, and after it is successfully terminated, the postcondition must be true[11].

6 Non-deterministic Rewriting Strategy Application for Verification

Let $B_n = \{\alpha_i \rightarrow < P_i > \beta_i | i \in (1, \ldots, n)\}$ be a set of basic protocols of a project, a predicate transformer $\boldsymbol{Tr}(\alpha, \beta)$ is a function defined on formulae of the base language returning a new formula such that $\boldsymbol{Tr}(\alpha, \beta) \rightarrow \beta$, e - define formulae. A predicate transformer strengthens the postcondition of a basic protocol by adding residual properties from the precondition.

We can represent a process of one-step application of each protocol from e set B_n to a formulae e by the ND Strategy and the following RRS:

Then,

$$
\begin{array}{l}
S_\alpha = rs(e, u)(\\
\quad sat(e \wedge \alpha_1) \rightarrow e[P_1] = \boldsymbol{Tr}(e \wedge \alpha_1, \beta_1), \\
\quad \ldots \\
\quad sat(e \wedge \alpha_n) \rightarrow e[P_n] = \boldsymbol{Tr}(e \wedge \alpha_n, \beta_n), \\
);
\end{array}
$$

where n - a number of basic protocols in project, function *sat* checks satisfiability of conjunction of precondition and current environment.

Other good example for application of ND Strategy in verification is based on experiments for effective term hashing algorithm realization on APS.

Let $e_n = \{e_i | i \in (1,\ldots,n)\}$ be a set of formulae,$B_n = \{\alpha_i \rightarrow < P_i > \beta_i || i \in (1,\ldots,n)\}$ be a set of basic protocols of a project. How we could determine the set of basic protocols $B_i \in B_n$ from which we could get state e_i. To determine the set of basic protocols B_i we can build the next RRS which should dynamicaly updates after each step application of basic protocol:

$$
\begin{aligned}
&B_\alpha = rs(x)(\\
&\quad e_0(x) = B_0,\\
&\quad \ldots,\\
&\quad e_n(x) = B_n\\
&);
\end{aligned}
$$

7 Conclusion

The system of algebraic programming APS exceeds the majority of criterions of well-known TRS. Among these criterions we can outline the most two important ones: the presence of procedural language (allows using simultaneously the paradigms of declarative and imperative languages) and the commercial usage (using of system in a real big commercial products and not only in different researches).

The non-deterministic rewriting strategy is optimal and together with a REM-program updating functions can be apllied in different arias of APS and IMS systems applications.

References

[1] A. Letichevsky, A. Letichevsky Jr., V. Peschanenko Algebraic Programing System. http://apsystem.org.ua,last viewed June 2010, 1987-2010.

[2] Department of Theory of Digital Automatic Machines Institute of Cybernetics of the National Academy of Science of Ukraine. http://icyb.kiev.ua,last viewed June 2010, 2010.

[3] Laboratory for Development and Implementation of Pedagogical Software Research Institute of Information Technologies of Kherson State Univeristy. http://riit.ksu.ks.ua/index.php?q=en/node/88,last viewed June 2010, 2006-2010.

[4] The Elan Team Elan System Official Site. http://elan.loria.fr,last viewed June 2010, 1992-2010.

[5] The MAUDE Team Maude System Official Site. http://maude.cs.uiuc.edu,last viewed June 2010, 1995-2010.

[6] The Stratego Team Stratego System Official Site. http://www.program-transformation.org/Stratego/WebHome,last viewed June 2010, 1994-2010.

[7] V.A. Volkov A.A. Letichevsky, Y.V. Kapitonova and others Insertion Programming. http://apsystem.org.ua/uploads/doc/ims/IM1.rus.pdf,last viewed June 2010, 2003.

[8] A. Letichevsky, J. Kapitonova,V. Kotlyarov and others Insertion Modeling in Distributed System Design. http://apsystem.org.ua/uploads/doc/ims/IMDSD.eng.pdf,last viewed June 2010, 2008.

[9] A. Letichevsky, J. Kapitonovam,V. Kotlyarov and others Semantics of Timed Msc Language. http://apsystem.org.ua/uploads/doc/ims/STMSCL.eng.pdf,last viewed June 2010, 2002.

[10] A. Letichevsky A. Degtyarev, J. Kapitonova and others Evidence Algorithm and Problems of Representation and Processing of Computer Mathematical Knowledge. http://www.springerlink.com/content/q0540m70v17u7716,last viewed June 2010, 1999.

[11] A. Letichevsky, J. Kapitonova, V. Volkov and others Systems Specification by Basic Protocols. http://portal.acm.org/citation.cfm?id=1103717,last viewed June 2010, 2005.

[12] Y. Sharon Smart pointers - what, why, which? http://ootips.org/yonat/4dev/smart-pointers.html,last viewed June 2010, 1999.

[13] A. Letichevsky,V. Khomenko A Rewriting Machine and Optimization of Strategies of Term Rewriting. http://www.springerlink.com/content/r42356668304580w,last viewed June 2010, 2002.

[14] M. Hanus, S. Antoy Set Functions for Functional Logic Programming. http://portal.acm.org/citation.cfm?id=1599420&dl=ACM,last viewed June 2010, 2009.

[15] A.A.Letichevsky, J.V.Kapitonova, S.V.Konozenko Computations in APS. http://apsystem.org.ua/uploads/doc/aps/CAPS.eng.pdf,last viewed June 2010, 1993.

Multi-Domain Logic as a Tool for Program Verification

Gábor Kusper
Eszterházy Károly College
Eger, Hungary
gkusper@aries.ektf.hu

Tudor Jebelean
RISC, Johannes Kepler University
Linz, Austria
Tudor.Jebelean@jku.at

Abstract

We discuss the advantages of using Multi-Domain Logic (MDL – a version of signed logic) for solving Bounded Model Checking (BMC) problems. On one hand, MDL can encode BMC problems in a more compact and natural way. On the other hand, the resulting MDL problems can be solved faster then their boolean versions.

Many software and hardware verification problems are reduced to boolean satisfiability (SAT) of relatively large formulae. Moreover, various formal verification techniques, notable model checking and in particular bounded model checking generate satisfiability problems which are not genuinely boolean, but they have more natural encoding in signed propositional logic (in which a variable may have more than two values). Currently these problems are reduced to boolean SAT because the availability of SAT tools, however this involves a certain loss of information (wrt. the original problem) and also a certain loss of efficiency.

Multi-Domain Logic (MDL) is a generalization of signed logic, in which every variable has its own domain. This aspect increases the efficiency of direct solving of MDL satisfiability, because the solving process proceeds by reducing the size of the domains (contradiction appears as an empty domain). In contrast to the usual approach of translating signed logic satisfiability into boolean satisfiability, we implement the generalized DPLL directly for MDL, using a specific version of the techniques used for signed logic. Moreover, we use a novel technique – *variable merging*, which consists in replacing two or more variables by a new one, whose domain is the cartesian product of the old domains. This operation is used during the solving process in order to reduce the number of variables. Moreover, variable merging can be used at the beginning of the solving process in order to translate a boolean SAT problem into an MDL problem.

Our experiments with a prototype eager solver show the effects of variable merging, as well as the effects of different design decisions on the efficiency of the solver.

1 Introduction

Many software and hardware verification problems are reduced to boolean satisfiability (SAT) of relatively large formulae. Therefore, an important issue in practical verification is the development of *very efficient SAT algorithms*, because this will enlarge significantly the class of verification problems which can be solved. Moreover, in various software verification techniques, notable in model checking and in particular in bounded model checking, if some model variables may more than two values, then the generated satisfiability problems are not boolean, but they refer to a generalized model of propositional logic in which a variable may have more than two values. Currently these problems are reduced to boolean SAT because the availability of SAT tools, however this involves a certain loss of information (wrt. the original problem) and also a certain loss of efficiency. Therefore direct solving *algorithms for multi-valued logic* are also very important for software verification. The approach presented here addresses both of these problems:

- it increases the efficiency of boolean SAT by transforming it into multi-domain SAT which propagates units more efficiently, and

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 105- 117

- it solves directly SAT problems in multi-valued logic.

Signed logic [2, 7] is a special type of multi-valued logic in which the set of satisfying values for the variables may differ in different clauses. Namely, a *signed formula* is a conjunction of *signed clauses*, and a signed clause is a disjunction of *signed literals* of the form $S : p$, where S is a set (the *sign*) and p is a variable. (S is called the *support* of the variable in the respective clause.) The union of all signs constitute the *domain* N of the formula. An interpretation I is a mapping from the set of variables P to the set of truth values N, and it satisfies a literal $S : p$ if $I(p) \in S$. We may assume that each variable occurs at most once in each clause (otherwise we merge the corresponding literals by union of their supports). When $S = \emptyset$ the literal may be omitted from the clause, and when $S = N$ then the whole clause is redundant.

The classical methods from boolean logic generalize in a natural way to signed logic – see e. g. [2], which also describes the generalization of the DPLL algorithm [3], including a specific aspect of it for signed logic, namely the elimination from of branches corresponding to certain redundant truth values. (We will describe and use this strategy in the sequel under the name of *elimination of weak assignments*.) However, we found only few implementations of a direct method for solving signed logic problems – e. g. [7], which is targeted at a restricted class of formulae. Rather, most approaches are based on translating signed logic into boolean logic and using some version of a SAT algorithm. For instance, [1] uses the information from the original signed logic problem in order to guide the SAT search.

We present here a direct approach for solving signed logic problems, based on the generalization of DPLL method, which exhibits two novel aspects:

- separation of the domains of the variable,
- *dynamic merging* of the domains of the variables.

In contrast to the current approaches, we demonstrate how to solve boolean SAT problems by transforming them into signed logic problems and then applying our direct solver. This may constitute an efficient alternative to the current SAT solvers based on unit propagation, because, in the context of signed logic, more boolean constraints can be propagated simultaneously. For the representation of the signs we use strings of bits in the current implementation, but this is not essential for the main algorithm.

We call *Multi-Domain Logic (MDL)* the generalization of signed logic in which the domains of the variables may differ. Although from the theoretical point of view the expressivity of MDL does not differ from signed logic, in practice this distinction leads to more efficient solving methods. This is because *the domains can be reduced* during the solving process, and finding a contradiction is expressed as the reduction of the domain of a variable to the empty set. Initially the domain of a variable which does not occur in unit clauses is the union of all its supports. Otherwise, the domain is the intersection of all the supports from the unit clauses containing the respective variable. Unit resolution consists in intersecting the support of a non-unit clause with the respective domain - when this is empty then the literal disappears. If a support includes the corresponding domain, then the whole clause can be deleted (unit subsumption). Whenever a new unit is obtained, this will reduce the respective domain. When no new unit can be obtained, then one must branch on one of the variables, by splitting its domain. (We present here experiments with few splitting strategies.)

While the operations above are straightforward generalizations of boolean constraint propagation, MDL also benefits from specific strategies. Similarly to signed-logic, one may detect certain redundant elements in domains, which we call *weak assignments*: If an element *a* of a domain occurs in all supports together with another element *b*, then we say *a is weaker then b* and *a* can be eliminated from the domain. Novel in our approach is the use of a technique which we call *variable merging*. This consists in replacing two or more variables by a new one, whose domain is the cartesian product of the old domains. In each

clause, the disjunction of the literals containing the old variables is replaced by one literal whose support is constructed in a straightforward way. Variable merging is used at the beginning of the solving process in order to translate a boolean SAT problem into signed logic, but also during the solving process in order to reduce the number of variables, when some domains become relatively small.

Multi-domain logic (MDL) was introduced in [4]which also presents the first practical experiments with this method for solving signed logic problems directly, and in particular those which are constructed from boolean SAT problems. The idea of MDL solving improves on earlier boolean solvers based on simultaneous propagation of several boolean units [5, 6].

The current paper describes the status of our work in progress concerning the implementation of the MDL solver and its testing on various classes of propositional problems, in particular on problems originating in bounded model checking. We illustrate, on one hand, the relative improvement of translation from BMC problems into MDL as opposed to boolean logic. On the other hand, we illustrate the effectiveness of the MDL version of the DPLL algorithm and we investigate the efficiency of different combinations of specific strategies. For rapid prototyping we use an eager algorithm implemented in Java, thus both the size of the SAT instances as well as the absolute timings are not impressive. However we obtain interesting facts when comparing different strategies details,

- Increasing the number of the boolean variables during the original translation leads to a significant speed-up until a certain threshold, which depends on the implementation environment but also on the structure of the original problem.
- The domain splitting strategies during branching have a significant effect on certain classes of problems.
- Both the deletion of weak assignments, as well as the dynamic merging have a notable impact on efficiency.

These findings constitute a good motivation for a more complex implementation using lazy techniques, and for further experiments and possible theoretical developments.

2 Encoding BMC Problems in MDL

Bounded Model Checking [2, 2] is one of the most successful formal methods for practical verification of hardware and software. The process to be verified is represented as a transition system and the desired property is expressed in temporal logic. In order to use a SAT (propositional satisfiability) solver, one choses a certain bound k on the number of steps of the process, and then one translates the behaviour of the transition system as well as the temporal logic formula into a boolean formula.

We repeat here the illustration of this method as given in [2], in which this approach was introduced. Consider a binary counter as depicted in Fig. 1 (00 is the initial state). We want to check the property: "The counter will eventually reach the state 11." Each state is characterized by two variables a, b (the least and the most significant bit, respectively). The formula $I(s_0) : \neg a_0 \wedge \neg b_0$ expresses the distinctive property of the initial state. The transition relation $T(s, s')$ expresses the transition relation between states s, s':

$$(a' \leftrightarrow \neg a) \wedge (b' \leftrightarrow (a \oplus b))$$

(Note that $\leftrightarrow$ is the same as equality and $\oplus$ is the same as XOR or disequality.) For $k = 2$, the characterization of the behaviour of the counter is represented by the formula:

$$T(s_0, s_1) : (a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow (a_0 \oplus b_0))$$
$$T(s_1, s_2) : (a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow (a_1 \oplus b_1))$$

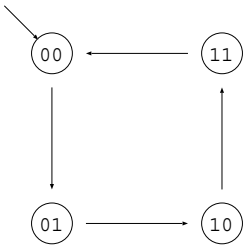


Figure 1: Two bit counter.

For $k = 2$ the statement "p is eventually satisfied" is represented by:

$$(a_0 \wedge b_0) \vee (a_1 \wedge b_1) \vee (a_2 \wedge b_2).$$

In order to check whether the property holds, one translates the whole propositional formula in CNF which can be treated by a SAT solver. During the CNF translation one introduces (recursively) new propositional variables for the subexpressions which are non-atomic (in order to prevent exponential grow of the formulae). If the resulting formula is unsatisfiable, then the property is not satisfied, otherwise it is.

These formulae have special characteristics which make them suitable for direct transformation into Multi-Domain Logic, if we group together the variables which describe a single state. In this case $I(s_0)$ generates a unary clause, and each transition $T(s,s')$ generates a number of binary clauses. If the number of variables is small (like in the example above) then we can even group the variables from two or more states together, and obtain many unary clauses. The MDL clause representation of the final formula $p(0) \vee p(1) \vee p(2)$ is one ternary clause, while in boolean representation it will contain 8 binary clauses. Of course for more complex examples the differences between Boolean and MDL representations becomes more significant.

This particular problem does not need branching in the DPLL algorithm because it represents a deterministic finite automaton with a single start state. Therefore, both in boolean logic as in MDL, the problem is solved by unit propagation, so we can compare directly the efficiency of the solvers in this case, by comparing the number of unit propagations. For 2 bits and 2 steps (as in the formalization above) the number of unit propagations for boolean DPLL is 16, while for MDL (cluster factor 2) it is 4. If we increase the problem to 9 bit counter and 511 steps, then the number of unit propagations in boolean DPLL is 12 786, while in MDL (clustering factor 5) it is 1024.

In the BMC experiments section below we present the concrete encoding for different problems and the differences in the number of basic proof steps (unit propagations), which suggests that the improvement in performance can be quite large.

3 Proof Techniques

Variable merging (VM). Merging two variables x, x' (with domains D, D') consists in replacing x, x' by a new variable y, which (intuitively) represents the pair $\langle x, x' \rangle$ and ranges over $D \times D'$. A disjunction $A : x \vee A' : x'$ becomes $((A \times D') \cup (D \times A')) : y$. By induction, merging extends to an arbitrary number of variables.

Variable clustering. Classical boolean variables can be seen as ranging over the domain $\{0,1\}$. As a first step of the MDL based SAT solving algorithm, the boolean problem is transformed by clustering the

variables. The *clustering factor* (number of boolean variables per MDL variable) may be fixed (as it is used in the present experiments) or variable. Before clustering we perform in fact a preprocessing step for detecting which variables occur more often together in clauses, and we try to group them together. The experiments presented in our previous work [4] show that the preprocessing results in a moderate increase in the efficiency of the MDL solving process.

Dynamic Variable merging. We also experiment with *dynamic merging*, that is binary merging of variables during the execution of the DPLL algorithm. When no new units can be obtained, then merging of two variables which occur together in a binary clause creates a new unit, thus split can be avoided. We apply merging if the size of the new domain is not bigger than the a 2^{k+t}, where k is the original clustering factor and t is a *merging threshold*.

Generalized DPLL. The Davis-Putnam-Logemann-Loveland algorithm [3] generalizes to MDL by extending unit subsumption and unit resolution. Let x be a variable symbol, A, B constant sets, and $\mathscr{C}$ a disjunction of MDL literals. The unit clause $A : x$ *subsumes* the clause $(B : x) \vee \mathscr{C}$ if $A \subset B$. The *resolvent* of the unit clause $A : x$ and the clause $(B : x) \vee \mathscr{C}$ is the clause $(A \cap B : x) \vee \mathscr{C}$. In contrast to propositional logic, the literal containing x is not always *canceled*, but only when $A \cap B = \emptyset$. There may be more units containing the same variable. These *collapse* into one unit by resolution, and the resulting set is the new *domain* of the respective variable. During DPLL, each time a new unit is obtained, it is intersected with the current domain of the respective variable in order to obtain the new domain. When the domain becomes empty, we have a contradiction on the respective search branch. As in the DPLL method, we use unit subsumption and unit resolution for performing *unit propagation (UP)*, and *constraint propagation (BCP)*. When all units have been propagated, then either all clauses have been subsumed (we have a solution), or we can create a new unit out of a binary clause by variable merging, or we must *branch* the search tree – which is can done by splitting of one of the domains using in various strategies (see below).

Branching strategies. In MDL we branch on a partition of the domain of a selected variable. Our present experiments use various choices for the variable and for the partition.
Choice of variable:

- MinDom: a minimal domain;
- MinDomClause: a minimal domain from a minimal clause;
- MinLit: a minimal literal from a minimal clause.

Minimality of a clause, domain, and literal refer to the number of literals, cardinality of the set, and cardinality of the sign, respectively. *Choice of partition:*

- SplitLit: the sign of the literal and its complement (only with MinLit);
- SplitHalf: half of the domain on each branch;
- SplitAll: one branch for each assignment.

When SplitHalf and SplitAll are used together with MinLit, then we generate additionally a branch corresponding to the complement (w.r.t. the domain) of the sign of the respective literal. In signed logic the SplitLit and SplitAll techniques are known and they are described in [2], however without implementation.

4 Implementation and Experimental Results

Data representation. The representation of a MDL formula is described in [4]. Here we only illustrate the idea of our representation by exhibiting the representation of clusters of two propositional variables a, b. The table below lists the bit strings, the corresponding sign sets, and the formula which is encoded.

0000	$\{\}$	$\mathbb{F}$	1000	$\{00\}$	$\neg a \wedge \neg b$
0001	$\{11\}$	$a \wedge b$	1001	$\{00,11\}$	$a \Leftrightarrow b$
0010	$\{10\}$	$a \wedge \neg b$	1010	$\{00,10\}$	$\neg b$
0011	$\{10,11\}$	a	1011	$\{00,10,11\}$	$a \vee \neg b$
0100	$\{01\}$	$\neg a \wedge b$	1100	$\{00,01\}$	$\neg a$
0101	$\{01,11\}$	b	1101	$\{00,01,11\}$	$\neg a \vee b$
0110	$\{01,10\}$	$a \times b$	1110	$\{00,01,10\}$	$\neg a \vee \neg b$
0111	$\{01,10,11\}$	$a \vee b$	1111	$\{00,01,10,11\}$	$\mathbb{T}$

These strings of bits can be represented as a list (or array) of computer words, and then union and intersection can be computed using hardware logical operations on words: union becomes *bitwise or* and intersection becomes *bitwise and*. For instance, if we merge 5 propositional variables into a multi-variable, then its domain has $2^5 = 32$ elements, so we can represent it on a 32-bit computer word. For higher clustering factors, however, the length of the representation in words grows exponentially, thus from a certain value the positive effect of clustering will be canceled by the higher cost of multi-word operations (as also confirmed by our experiments).

Experimental Results In order to demonstrate the effectiveness of multi-domain logic and to check the basic implementation principles, we implemented a variant of DPLL method described in Section3.

The implementation is realized in Java using eager principles, with the main purpose is not to compete with the modern SAT solvers based on lazy data structures, but to allow us to determine what is the effect of using various techniques in various combinations. We used a HP Compaq nx6110 notebook (32 bits) with Pentium M 1.86 MHz processor and 512 MB memory.

We tested our implementation on some unsatisfiable problems from the SATLIB – Benchmark Problems homepage (`www.satlib.org`). Namely, we used the `uuf-50-01.cnf` problem and the `hole6.cnf` problem, because they belong to a scalable class of problems, and are of a complementary nature. The former is randomly generated and has no structure or symmetry, while the latter is symmetric and well structured.

We experiment with unsatisfiable problems, because this gives more reliable information on the general behavior of the program. On satisfiable problems the running time may be influenced by the randomness of early found solutions.

We investigate the main aspects of the novel method: cluster preprocessing, branching strategies, deletion of weak assignments (DoWA), variable merging (VM), and the variable merging threshold.

All experiments are performed with increasing clustering sizes (number of original propositional variables per MDL variable).

Cluster preprocessing. The effect of cluster preprocessing confirms what we reported in the previous paper: a speed-up between $40-60\%$ for the randomly generated problems, and almost no speed-up for the pigeon–hole problems (because they are already well clustered). Therefore we will show in detail only the effect on the randomly generated problem `uuf-50-01.cnf`.

Branching strategies. In the previous section we presented 3 split strategies and 3 choice strategies, which we present in 4 combinations – which appear to be more efficient.

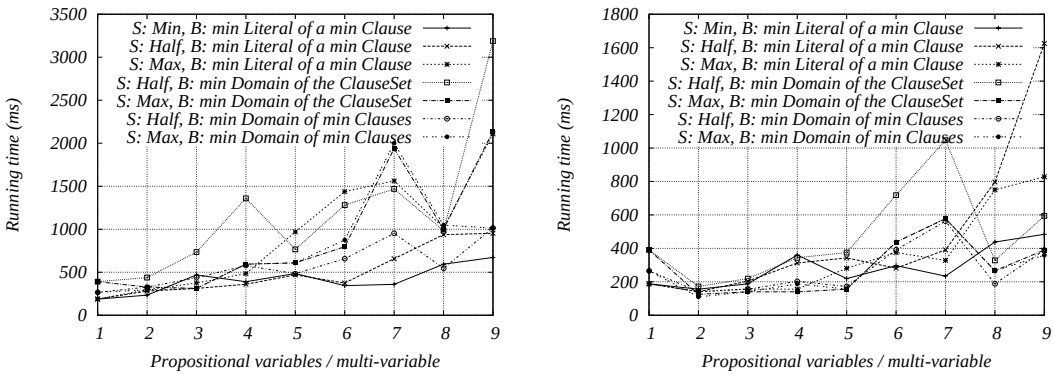


Figure 2: `uuf50-01.cnf`: Absolute timings without-, and relative timings with clustering.

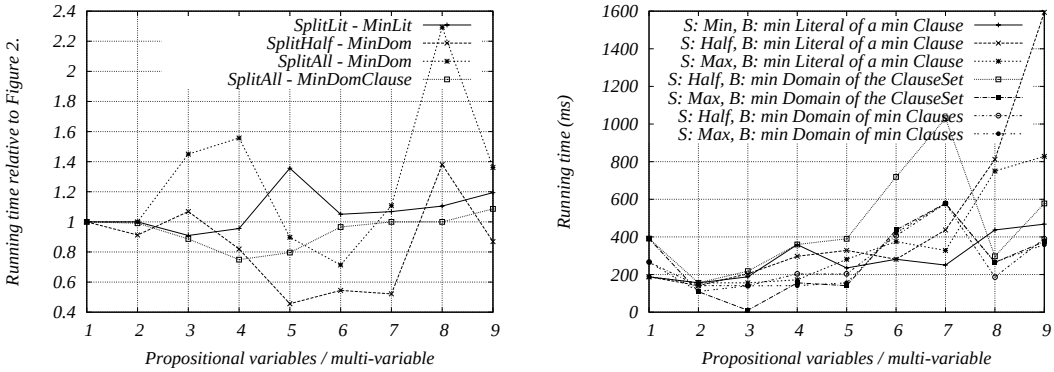


Figure 3: `uuf50-01.cnf`: DoWA and VM (timings relative to Fig. 1. with clustering).

First we show the running time of this methods on the `uuf50-01.cnf` problem from the SATLIB page. Figure 2 shows the case where we do not use any simplification techniques.

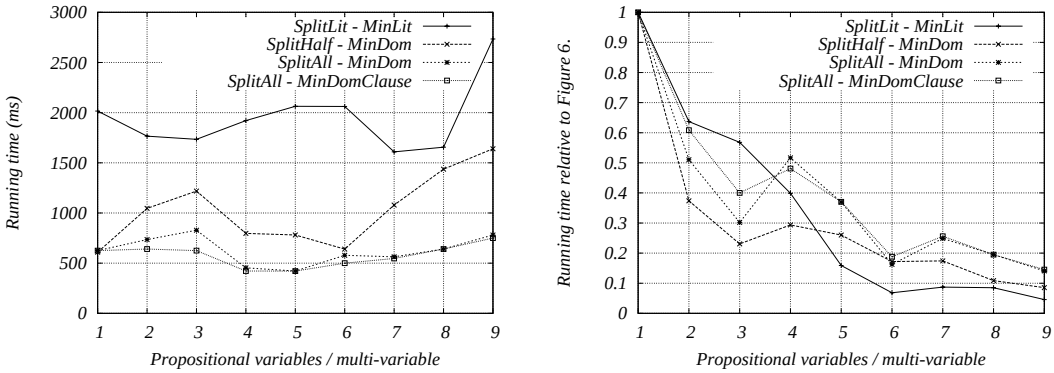


Figure 4: `hole6.cnf`: Absolute-, and relative timings when using DoWA and VM.

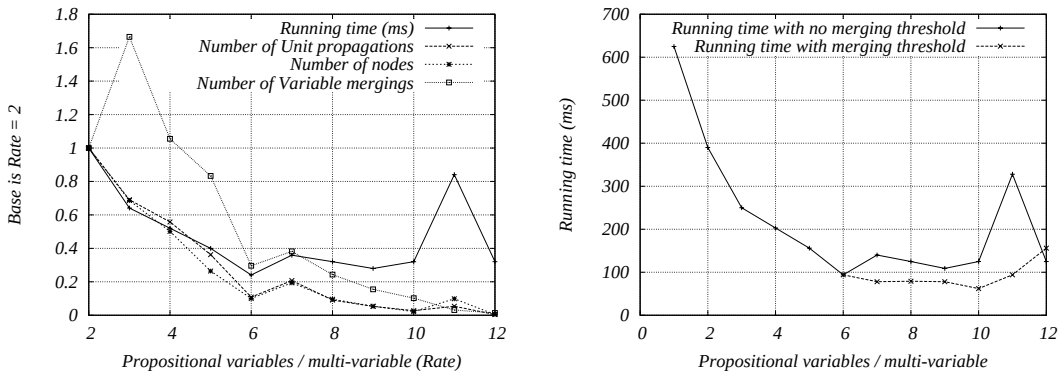


Figure 5: `hole6.cnf`: Statistics of SplitAll - MinDomClause ; Effect of merging threshold.

Dynamic variable merging. To create a new unit out of a binary clause is quite expensive, because we have to know the exact domains of the two variables. The current implementation is lazy in the sense that it recalculates a domain only if a new unit is found for the corresponding variable. Therefore, we use the following heuristic for deciding wether to compute the domains or not. Assume the binary clause is $A : x \vee B : y$. If $|A| * |B| \leq 2^{k+t-1}$, where k is the clustering factor and t is a merging threshold, then we calculate the domains of x and y. This heuristic performed very well. It covered the 88% of the cases where the the new domain really fits in 2^{k+t} bits and its guess was never wrong.

Deletion of weak assignments and Variable merging. The problems are already cluster preprocessed.

In Figure 3 we can see the effect of deletion of weak assignments, for short DoWA. Note that in some cases it even slows down the search, and in some cases it gives $5-15\%$ speed-up. The average of these values is 101%. For a clustering factor of 5 the average is 86%.

In the same Figure 3 we can see the effect of variable merging (VM). The effect ranges from slow down to $2-8\%$ speed-up. It appears that DoWA gives slightly better speed-up than VM, but in fact the average of these values are 99%, which is a bit better than in case of DoWA, but DoWA gives bigger speed-up.

In conclusion, on this uniform random 3-SAT problem, the effect of cluster preprocessing is large but the one of DoWA and VM is small.

We have also tested this simplification techniques on the "pigeonhole problem" `hole6.cnf` (7 pigeons and 6 holes). Figure 4 shows the absolute running time of 4 branching strategies, as well as the improvement of efficiency (as relative time) when using DoWA and VM together. It is difficult to judge which branching method is the best. The effect of DoWA and VM together on `hole6.cnf` is very significant, like $40-90\%$ speed-up. The average of these values is 39%. So in this case we obtain very good results with the same simplification techniques which gave virtually no speed-up in case `uuf50-01.cnf`. This is probably due to the regular structure of the pigeonhole problem.

Dynamic merging of variables. As mentioned in section 3, merging of variables can be further stimulated by allowing new MDL variables of greater size than the original ones (as controlled by the parameter *merging threshold*). In Figure 5 (left) we see that bigger merging threshold leads in general to a higher number of variable merging and, therefore, less nodes in the search tree. We would expect here an exponential curve since the bigger the merging threshold is the bigger the probability that two domains can be merged (note, that in this implementation the size of a domain is limited to 2^k, where k is rate

of propositional variables / multi-variable, k is fixed). If the merging threshold is $2k$, then virtually all binary clauses can be merged (only those not, which contains an already merged multi-variable). But we see that the number of variable merging starts to increase and then remains almost the same, then drops.

The explanation of this observation is that the number of unit propagations steps starts to decrease quite fast. It is so because each variable merging prevents a split and allow a unit propagation instead (note that we merge multi-variables only if they occur in a binary clause, which, of course, becomes a unit after variable merging). This cuts the search spaces (an early variable merge during the search can dramatically cut it) and, therefore, the number of unit propagation steps decreases. But if the search space is smaller then the probability of variable merges is also smaller. Furthermore, a merged multi-variable is less likely takes place again in a merge, because its domain has $A*B$ elements, where A,B are the size of the domains before merging. Hence, the number of variable merges cannot grow exponentially as the merging threshold grows.

In Figure 5 (right) we see that to solve the `hole6.cnf` problem we need only few milliseconds if we use variable merging, rate 6 (which is fixed in this figure) of propositional variables / multi-variable, and merging threshold 7 . This is even better if we have rate 13 but no dynamic merging. The explanation of this observation is that with rate 6 all clauses of the problem are already binary and they can be immediately merged because of the high merging threshold, $6+7 \geq 2*6$. Note that merging threshold 6 also was this property; indeed lot of the input clauses can be merged, but not those which contain already merged multi-variables. So with this high merging threshold after variable merging we have all units, so we spend only time on initialization and variable merging to solve hole6.cnf. One would except the same behavior in case rate 13 if we have no dynamic merging. But this does not guarantee that all clauses are units, we have lot of units but also some binaries, this is why it needs more time. To summarize, it is better to use rate k and merging threshold t than rate $k+t$.

This would suggest to choose very big numbers for the clustering factor k, and and for the threshold t, but then we need 2^{k+t} bits to store a literal (at least in the current implementation), which results in the unfeasibility of the operations on literals after certain limits.

5 Test Results on Bounded Model Checking Problems

The test result of this section is obtained on a NEC VERSA M370 notebook with Core2 Duo M 2 GHz processor (32 bits) and 1 GB memory.

MDL file format. This paper is motivated by the hope that multi-domain logic is useful to solve bounded model checking SAT problem instances. In particular, we expect that it will help to overcome the state space explosion problem, which bounds the size of the models which can be check by current tools. In order to check this assumption we performed several test with our MDL based SAT solver.

CNF files containing bounded model checking SAT problem instances can be downloaded from http://www.cs.cmu.edu/ modelcheck/bmc/bmc-benchmarks.html, and from the web pages of the hardware model checking competition (HWMCC).

The main idea is that instead of translating CNF files into multi-domain logic SAT problem instances, we create these instances directly from the model description. Therefore we added to our SAT solver the feature of recognizing multi-domain logic formulae in conjunctive normal form (similar to DIMACS file format – see below), having the file extension *.MDL. Previously, our solver recognized only propositional logic formulae in conjunctive normal form given in DIMACS file format (usual file extension *.CNF or *.DIMACS).

An example MDL file (counter.2bit.k2.mdl) is given to illustrate this file format:

```
{0}:x0
```

```
{1,2,3}:x0 {1}:x1
{0,2,3}:x0 {2}:x1
{0,1,3}:x0 {3}:x1
{0,1,2}:x0 {0}:x1
{1,2,3}:x1 {1}:x2
{0,2,3}:x1 {2}:x2
{0,1,3}:x1 {3}:x2
{0,1,2}:x1 {0}:x2
{3}:x0 {3}:x1 {3}:x2
```

This example has 10 clauses. The first clause is a unit, and it consists of one (multi-)literal: $\{0\} : x_0$, where $\{0\}$ is the sign set of the variable x_0. Next we have 8 binary clauses, which describe the implication: $\{n\} : x_i \Rightarrow \{(n+1) \ mod \ 4\} : x_{i+1}$, where $i = 0 \ldots 1$. The last clause is a ternary clause.

The file format is not bound to the use of natural numbers in sign sets. A sign set can contain any symbols delimited by commas. Variables can be also any symbols. Literals are delimited by spaces and clauses are delimited by end–of–line.

Our implementation automatically decides what shall be the *rate* of propositional variables per a multi-variable. Maybe this is the most crucial option of or SAT solver. The decision is done in the following way: we create the domain of each multi-variable as the union of its sign sets. The cardinality of the largest domain is chosen to be the rate. In the current implementation this rate must be the same for each multi-variable.

Bit counter example. The MDL file above describes a classical bounded model checking problem. Given a counter with n bits, then it has 2^n states and transitions. In this case it has 2 bits, 4 states, and 4 transitions: $00 \to 01$, $01 \to 10$, $10 \to 11$, and $11 \to 00$. It is the same example as in Figure 1, except the extra transition. The variable x_0 describes the initial state of the system, the variable x_1 describes its state in the next time step, and so on. We can move from a state to the next one by a transition, i.e. each transition takes one time step.

The number of time steps (or number of transition), usually called k, is an important parameter of bounded model checking problem. The file MDL file above (counter.2bit.k2.mdl) states that in the initial state is $x_0 = 0$, the goal state is that either $x_0 = 3$ or $x_1 = 3$ or $x_2 = 3$, i.e. the number of transition is $k = 2$. Of course from 0 we cannot reach 3 in two steps, so this clause set is unsatisfiable.

The corresponding CNF file can be generated from example by the BMC tool (see http://www.cs.cmu.edu/ modelcheck/bmc.html). The bit counter example can be stated in many different ways using the BMC tool. We give only that model description (counter.2bit.bmc) which results in the smallest CNF file.

```
VAR  x[2]
ASSIGN  init(x) := 0;
        next(x) := inc(a);
SPEC  AG x != 3
```

In the VAR clause we state that x is a 2 bit unsigned int variable. In the BMC tool each variable has unsigned int type, 1 bit variables may be used also as boolean ones. In the ASSIGN clause we state that the initial value of x is zero. Then we define a transaction which gives x the new value $x+1$. Note, that inc is the built–in function for increment. Finally we give a goal in the SPEC clause using temporal logic quantifiers. We state that for every path $x \neq 3$ holds at every time step in the future.

From this model we can generate CNF files by issuing the command:

```
./bmc -dimacs -k 2 counter.2bit.bmc > counter.2bit.k2.cnf
```

Here the $-k\ 2$ option indicates that we bound the model checking to use only 2 time steps.

In this way we have generated the following files: counter.2bit.k2.cnf, counter.2bit.k3.cnf, counter.-3bit.k6.cnf, counter.3bit.k7.cnf, ..., counter.9bit.k510.cnf, counter.9bit.k511.cnf. Note, that the instances with $k = 2^{\#bits} - 2$ are (still) unsatisfiable, but the instances with $k = 2^{\#bits} - 1$ are satisfiable, provided that for any number of bits the goal is $AGx! = 2^{\#bits} - 1$.

We do note have a model interpreter like the BMC tool for multi-domain logic, but we a subroutine, which can generate MDL files for any bit width and number of time steps. By this routine we have generated the following files: counter.2bit.k2.mdl, counter.2bit.k3.mdl, counter.3bit.k6.mdl, counter.3bit.k7.mdl, ..., counter.9bit.k510.mdl, counter.9bit.k511.mdl.

We used our MDL based SAT solver to solve these CNF and MDL instances. For the CNF files we used two settings. In the first one the rate of propositional variables per a multi variable is fixed to 1 (that is, we use the classical boolean DPLL). In the second one the rate was the same as in case of MDL files. This means that the rate was equal to the number of bits of the counter up to 5, then the half of the number of bits (rounded up), i.e., rate=1#bits up to 5, then #bits2. This change at 5 bit is motivated by the fact that we used a 32 bit processor for the test, and for a multi-literal with 5 propositional variables we need 32 bit, which is a machine word.

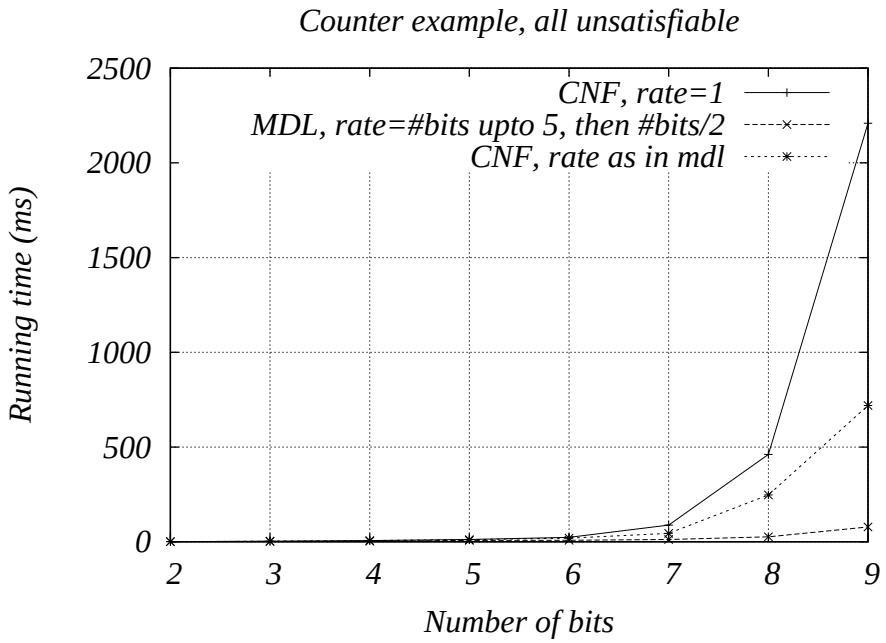


Figure 6: Counter example encoded in CNF and Multi-Domain Logic, $k = 2^{\#bits} - 2$

The test result are depicted on Figures 6. and 7. In the first one each instance is unsatisfiable, in the second one each one is satisfiable. We can see that our SAT solver could solve the bit counter problem encoded in MDL faster than the CNF encoding. The slowest one is CNF with rate fixed to 1. In this case our SAT solver just performs a normal DPLL algorithm. The second best running time is measured in case of CNF with bigger rate.

These figures shows that the MDL encoding helps us to postpone the space explosion problem. Why? When the BMC tool generates the CNF files out of the model, then it introduces new variables to keep the length of clauses to be small (around 3). These short cuts are resolved by unit propagations during the run of the DPLL method. Unit propagation is the fastest part of any DPLL based SAT solver. In

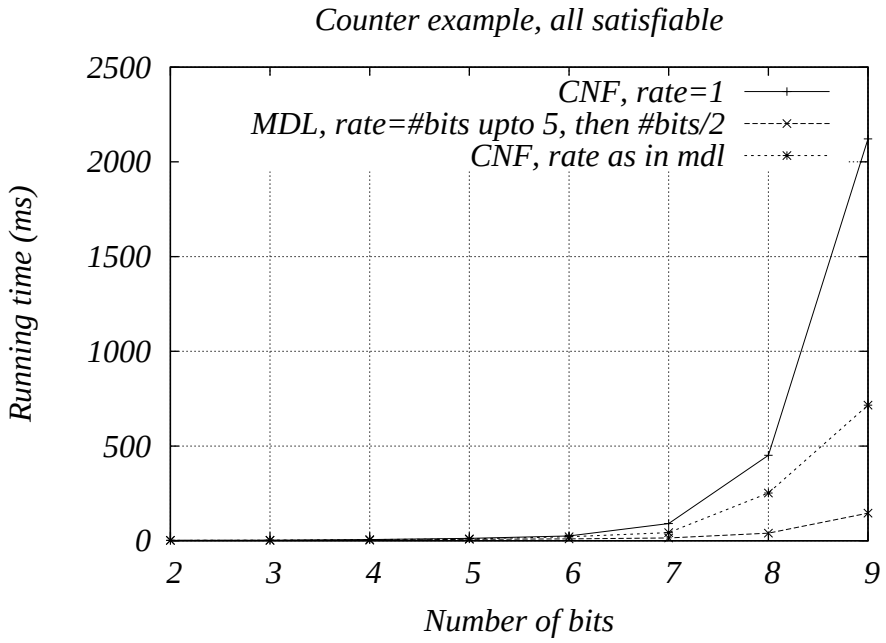


Figure 7: Counter example encoded in CNF and Multi-Domain Logic, $k = 2^{\#bits} - 1$

the case of MDL we have very short clauses, because one multi-literal can substitute many propositional literals. On Figure 8 we can see the number of performed unit propagation for the satisfiable instances.

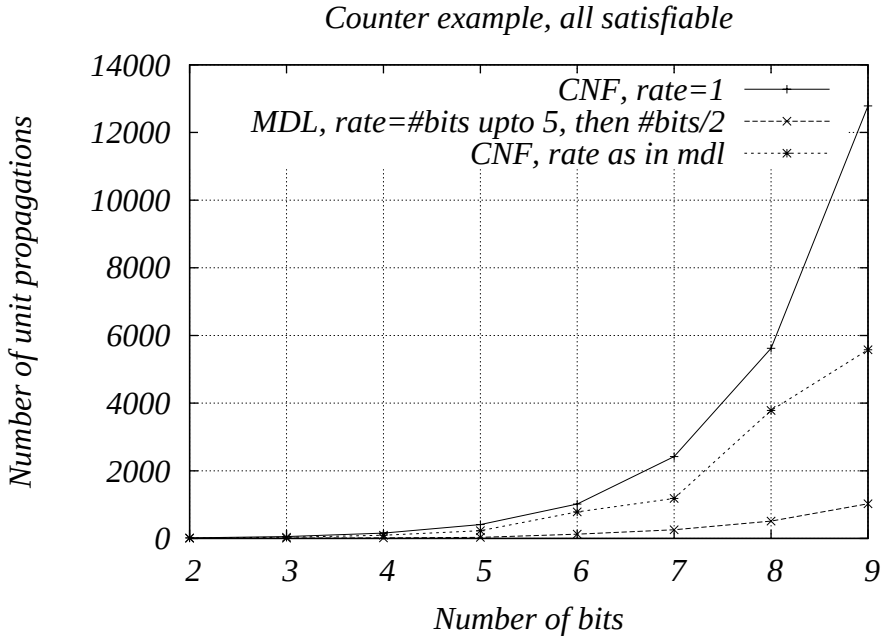


Figure 8: Number of unit propagation steps for the bit counter example, $k = 2^{\#bits} - 1$

6 Conclusions

The multi-domain approach to satisfiability solving represents an effective alternative to boolean SAT algorithms, especially for problems originating in bounded model checking. However, its possible superiority in efficiency remains doubtful in absence of more sophisticated theoretical and practical investigations.

We appreciate that the results obtained with this preliminary eager implementation constitute a good motivation for further refinements of the techniques and of the solving strategies.

Further work includes more comprehensive experiments with more classes of problems and with larger instances, as well as more sophisticated lazy implementations generalizing the current efficient methods used in SAT solvers.

In contrast to boolean logic, MDL presents more rich opportunities for theoretical insights and algorithmic developments regarding various aspects of the solving process: the preprocessing for static and for variable clustering, the choice of domain for splitting, the domain splitting strategies for branching, the strategies for the merging of variables, the representation of domains and of signs, as well as the data structures representing variables, clauses, etc.

References

[1] C. Ansotegui, J. Larrubia, Chu Min Li, and F. Manyà. Mv-Satz: A SAT solver for many-valued clausal forms. Proceedings of the 4th International Conference on Knowledge discovery and discrete mathematics, Metz, France, 2003, pp. 143–150.

[2] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Proc. Intl. Conf. on Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), Lecture Notes in Computer Science (LNCS), vol. 1579, Springer 1999.

[2] E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design (FMSD), vol. 19, number 1, Kluwer 2001.

[2] A. Biere. Bounded Model Checking. in *Handbook of Satisfiability,* A. Biere, M. Heule, H. Van Maaren, T. Walsh (eds.) IOS Press, 2009, pp. 457–481.

[2] B. Beckert, R. Hähnle, and F. Manyà. The SAT problem of signed CNF formulas. Labelled Deduction, volume 17 of Applied Logic Series, pp. 61–82. Kluwer, Dordrecht, May 2000.

[3] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. Communications of the ACM, 5:394–397, 1962.

[4] T. Jebelean, G. Kusper. Multi-Domain Logic and its Applications to SAT. Proceedings of SYNASC'08, IEEE Computer Society Press, ISBN 978-0-7695-3523-4, pp. 3–8, 2008.

[5] G. Kusper. Solving the SAT Problem by Hyper-Unit Propagation. RISC Technical Report 02-02, 1-18, University Linz, Austria, 2002.

[6] G. Kusper, L. Csőke. Better test results for the graph coloring and the Pigeonhole Problems using DPLL with k-literal representation. Proceedings of ICAI-2007, Volume II. 127-135, Eger, Hungary, January 2007.

[7] F. Manyà, R. Béjar, and G. Escalada-Imaz. The satisfiability problem in regular CNF-formulas. Soft Computing: A Fusion of Foundations, Methodologies and Applications, 2(3):116–123, 1998.

Predicate transformers in system verification

A.A. Letichevsky, O.A. Letichevskyi
Glushkov Institute of Cybernetics
Kiev, Ukraine
let@cyfra.net
lit@iss.org.ua
T.Weigert
Dept. of Comp. Sci., Missouri University of Science and Technology

Abstract

This paper demonstrates the use of first order logic in the verification of requirement specifications of reactive software systems. The presented concepts have been implemented in our VRS (Verification of Requirement Specifications) system. The key verification algorithms used in VRS are presented. We consider symbolic models of a specified system which are transition systems with symbolic states represented by formulae of first order logic. To compute transitions of such models, the basic elements of a specification, referred to as basic protocols, are interpreted as predicate transformers: For a given symbolic state of a system and a given basic protocol, the direct predicate transformer generates the next symbolic state as the strongest postcondition and the backward predicate transformer generates the previous symbolic state as the weakest precondition. To efficiently compute predicate transformers we use SMT (Satisfiability Modulo Theory) methods instead of logical inference in the corresponding calculi. This deductive system can be used for static requirement verification of a system by proving that a set of basic protocols specifies a deterministic system and is complete. Predicate transformers are further used for proving statically the invariance of safety conditions, for the dynamic verification of specifications by generating traces of a symbolic system, and for finding deadlocks or safety violations.

1 Introduction

This paper demonstrates the use of first order logic in the verification of requirement specifications of reactive software systems. The presented concepts have been implemented in our VRS (Verification of Requirement Specifications) system [5, 6, 7]. The key algorithms used in VRS for the verification of systems are presented. Specifications of software systems in VRS are represented by means of systems of basic protocols. A basic protocol is a formula of dynamic logic (Hoare triple) $\forall x(\alpha(x,r) \to \langle P(x,r)\rangle\beta(x,r))$ and describes the local properties of a system in terms of pre- and postconditions α and β. Both are formulae of first order multisorted logic interpreted on a data domain, P is a process, represented by means of an MSC diagram [1, 8], and describes the evolution of the specified multi-agent system when triggered by the precondition, x is a list of typed data variables, and r is a list of environment and agent attributes. All basic protocols in a system specification with true preconditions are performed in parallel.

Symbolic models of a specified system are transition systems with symbolic states represented by formulae of first order logic. To compute transitions of such models, basic protocols are interpreted as predicate transformers: for a given symbolic state of a system and a given basic protocol, the direct predicate transformer generates the next symbolic state as its strongest postcondition and the backward predicate transformer generates the previous symbolic state as its weakest precondition. To efficiently compute predicate transformers we use SMT (Satisfiability Modulo Theory) methods instead of logical inference in the corresponding calculi.

The deductive system of VRS supports the following data types: numeric (integer and real), symbolic (free terms), enumerated, functions (arrays are considered as functions with restricted domains), and lists.

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 118-130

This deductive system can be used for static requirement verification of a system by proving that a set of basic protocols specifies a deterministic system and is complete. Predicate transformers can also be used for proving statically the invariance of safety conditions, for dynamic verification of requirement specifications by generating traces of a symbolic system, and for finding deadlocks or safety violations.

In the following section, the specification of a system by means of basic protocols is formalized. We define transition systems explaining the semantics of such specifications for concrete and symbolic models. In section 3, we give an overview of the logic used to express such specifications in our VRS system. In the next two sections, we describe the two main algorithms VRS relies on during verification: (i) The algorithm for proving satisfiability of formulae, which integrate symbolic and numeric information with behaviors and functional data structures (which makes the solvability of the SAT problem far from obvious). (ii) The second algorithm computes transitions in symbolic models. and is based on the notion of a predicate transformer–i.e., a function that transforms the current state of the environment into a new state. The requirements on the backward predicate transformer are also discussed. Section 6 describes the use of these algorithms for verification in the VRS system. The conclusion briefly discusses related literature and highlights opportunities for further work.

2 Basic protocol specifications

A basic protocols specification $\Sigma = \langle B, E \rangle$ of a system consists of a set of basic protocols B together with an environment description E. E defines the signature of the language used to express pre- and postconditions of the basic protocols and the interpretation of this signature. We refer to the language defined by the environment description as the *base language* of a given specification; the details of the base language are presented in the next section. E also defines the set of attributes of the specified system (symbols that may take different values over time). Each attribute is of simple or functional type. If an attribute f has functional type $(\tau_1, \tau_2, \ldots) \to \tau$ then *attribute expressions* $f(t_1, t_2, \ldots)$ are available in basic protocols. Further, the types, attributes, and behaviors of agents inserted into an environment as well as the set of possible initial states of agents and the environment are defined in the environment description.

The general form of a basic protocol is

$$\forall x(\alpha(x,r) \to \langle P(x,r) \rangle \beta(x,r)) \tag{1}$$

The variables $x_1, x_2, \ldots$ in the list $x = (x_1, x_2, \ldots)$ are the *parameters* of a protocol and occur under the quantifier together with their types: $\forall x = \forall(x_1 : \tau_1, x_2 : \tau_2, \ldots)$. The list r used in the *precondition* $\alpha(x,r)$, the *process* $P(x,r)$, and the *postcondition* $\beta(x,r)$ of a basic protocol is the list of attribute expressions. The precondition is a formula of the base language, the process is an MSC diagram, and the postcondition is a formula of the base language extended by assignment statements (considered as temporal logic formula).

Each model of a specified system is a transition system with states represented in the form $s[m_1 : u_1, m_2 : u_2, \ldots]$ where s is the *kernel of the environment state*, $m_1, m_2, \ldots$ are the names of agents, and $u_1, u_2, \ldots$ are the behaviors of these agents considered as their states. All models are instances of a model of the interaction of agents and environments [2, 3], and a basic protocol specification can be interpreted as a method for defining the insertion function.

The actions of a model are instantiated basic protocols considered atomic (i.e., we do not consider the individual steps in the process of the basic protocol). Depending on how we represent the kernel of the environment state we can distinguish various kinds of models: *Concrete models* represent the state as mappings from the set of attribute expressions to their concrete values. *Symbolic models* represent the state as formulae of the base language. Symbolic models are further differentiated based on the condition

under which we consider a basic protocol applicable to a symbolic state $\gamma(r)$. For *universal models*, the applicability condition is the validity of implication $\gamma(r) \to \exists x \alpha(x,r)$. For *existential models*, a basic protocol is applicable if $\gamma(r) \wedge \exists x \alpha(x,r)$ is satisfiable. In this paper we consider existential symbolic models and their relationship to concrete models.

Concrete models. A concrete model C_Σ defined by environment description Σ is constructed as follows. A *constant attribute expression* is an attribute of a simple type or an attribute expression of the form $f(a_1,a_2,\ldots)$, where f is an attribute of a functional type and $a_1,a_2,\ldots$ are constant (ground) expressions of corresponding types. Let A_Σ be the set of all constant attribute expressions. The kernel of the state of a concrete model is a partial mapping $s : A_\Sigma \to D$ from the set of constant attribute expressions to the data domain D (its values). This mapping must preserve types: if $s(x)$ is defined then the type of $s(x)$ is equal to the type of x. The complete state of environment $s[m_1 : u_1, m_2 : u_2, \ldots]$ contains all agents inserted into the environment.

The mapping s is extended in a natural way to terms and formulae of a base language through iterative substitution of the values for attribute expressions, and we say that a formula γ is valid on the state s of a concrete model (denoted $s \models \gamma$) if $s(\gamma)$ is defined and valid.

Let $B = \forall x(\alpha(x,y) \to \langle P(x,r)\rangle \beta(x,r))$ be a basic protocol, $x = x_1, x_2, \ldots$ are the parameters of B and $a = a_1, a_2, \ldots$ is a list of values such that the type of a_i is contained in the type of parameter x_i (types are partially ordered). A formula $B(a) = (\alpha(a,r) \to \langle P(a,r)\rangle \beta(a,r))$ is called an instantiation of a protocol B. Let $s[m : u] = s[m_1 : u_1, m_2 : u_2, \ldots]$ and $s'[m'_1 : u'_1, m'_2 : u'_2, \ldots]$ be the states of a concrete model. Define the transition relation of a system C_Σ in such a way that

$$s[m : u] \overset{B(a)}{\to} s'[m : u'] \text{ iff}$$
$$(s[m : u] \models (m : u) \wedge \alpha(a,r)) \wedge (s'[m : u'] \models (m : u') \wedge \beta(a,r))$$

where $(m : u) = (m_1 : u_1) \wedge (m_2 : u_2) \wedge \ldots, (m : u') = (m_1 : u'_1) \wedge (m_2 : u'_2) \wedge \ldots$. An expression like $(m_i : u_i)$ asserts that an agent m_i is in a state u_i and is called *state assertion*.

A transition function defined in this manner may exhibit a high degree of non-determinism. For example, attributes that do not occur in a postcondition may possess arbitrary values and agents that do not occur in the state assumption of a postcondition can change their states arbitrarily. Such nondeterminism can be restricted in a usual way, by restricting that only attribute expressions occurring in postconditions of instantiated basic protocols can change their values and that only agents occurring in such postconditions can change their states (independency constraint). This is possible as in concrete models the values of attribute expressions and the states of agents are independent.

Symbolic models. The kernel of the environment state for an existential symbolic model is a formula of the base language. An environment state has the form $\gamma(r)[m : u] = \gamma(r)[m_1 : u_1, m_2 : u_2, \ldots]$ where $\gamma(r)$ is a formula, and $[m : u]$ defines the states of all agents inserted into the environment. The transition relation must satisfy the following minimal constraints, given $\gamma(r)[m : u] \overset{B}{\to} \gamma'(r)[m : u']$: (i) $\gamma(r) \wedge (m : u) \wedge \exists x \alpha(x,r)$ must be satisfiable (*applicability condition*) (ii) $\gamma'(r) \wedge (m : u') \to \exists x \beta(x,r)$ (*postcondition requirement*), and (iii) let $s[m : u] \overset{B(a)}{\to} s'[m : u']$ be an arbitrary transition of a concrete model C_Σ; if $s[m : u] \models \gamma(r) \wedge (m : u)$ and $\gamma(r)[m : u] \overset{B}{\to} \gamma'(r)[m : u']$ then $s'[m : u'] \models \gamma'(r) \wedge (m : u')$ (*simulation requirement*). Under these minimal restrictions, the system again may be highly non-deterministic. More deterministic means of defining the transition relation will be considered below in terms of predicate transformers.

3 Base language

The underlying language (base language) to capture specifications is a first order multisorted language with interpreted and uninterpreted functional symbols. The interpreted functional symbols characterize the environment and are used for the definition of transition functions, as discussed in the previous section. Uninterpreted symbols are used to represent the changing state of the environment and are identified with attributes of a system. All uninterpreted symbols have types, and thus their possible interpretations are restricted by definite domains and ranges of values. Functional symbols of arity 0 correspond to simple attributes, others correspond to the attributes of functional types or functional attributes.

We rely on the following types (sorts) of functional symbols: *integer*, *real*, *Boolean*, *symbolic*, and a set of *enumerated* data types are defined as simple types. For all types the equality predicate is defined. For numeric types the inequality relation is defined, but and only addition and multiplication by constants of the corresponding type are allowed as operations (interpreted functions). Consequentially, arithmetic is limited to linear arithmetic. The domain for a symbolic type is a set of free terms constructed from symbolic and numeric constants by means of a set of predefined constructors. Enumerated data types provide sets of possible values. For list types, access functions are defined, and lists can change their values by adding or removing elements to (from) head or tail only. (Thus, list types exhibit the behavior of queues.) Behavior types are used as agent states and are considered as elements of behavior algebra [3] (a kind of process algebra). This algebra has two operations: prefixing $a.u$ (a is an action, u is a behavior), and nondeterministic choice $u+v$. It has also three constants: dead lock 0, successful termination Δ and undefined behavior $\bot$. Behavior algebra is generated by constants, actions (arbitrary symbolic constant expressions can be used as actions) and predefined constant behaviors. The last are defined in environment description as minimal solutions of a system of equations in behavior algebra. Functional types are functions from simple types to simple types with arbitrary arity. Arrays are attributes of functional types with arguments (indexes) limited to enumerated types or integers.

All simple types may occur in symbolic data structures. Formulae describing precondition, postcondition and the kernel state of the environment for symbolic models may use only existential quantifiers in positive positions (when there is an even number of negations on any branch leading to the quantifier). Quantifiers can bind only variables of simple types. In preconditions, list data can be used only in access functions. The general form of a precondition is $\sigma(x,r) \wedge F(x,r)$, where

$$\sigma(x,r) = (m_1(x,r) : u_1(x,r)) \wedge (m_2(x,r) : u_2(x,r)) \wedge \ldots$$

is a conjunction of state assumptions and $F(x,r)$ does not contain state assumptions. A postcondition is a conjunction of arbitrary formulae of the base language, state assumptions, and assignments. A simple assignment has the form $x := y$, where x is an attribute expression and y is an expression of a type that is contained in the type of x and asserts that the new value of x is the old value of the expression y. A list assignment adds or removes from the head or tail of a list. The general form of the postcondition is $\sigma(x,r) \wedge R(x,r) \wedge L(x,r) \wedge F(x,r)$ where $\sigma(x,r)$ is a conjunction of state assumptions, $R(x,r)$ is a conjunction of simple assignments, $L(x,r)$ is a conjunction of list updating, and $F(x,r)$ is a formula of the base language without state assumptions.

Assignments are tied to two states: before the application of a basic protocol and after its application, and thus can be considered as temporal logic formulae. Simple assignments can be easily eliminated from basic protocols. For example, the basic protocol $\forall x(\alpha(x,r) \to \langle P(x,r)\rangle(u(v) := w) \wedge F)$ is equivalent to the basic protocol $\forall x \forall (y,z)(\alpha(x,r) \wedge (y = v) \wedge (z = w) \to \langle P(x,r)\rangle(u(y) = z) \wedge F$. List updating can also be eliminated by a small extension of the base language. (Thus, in the previous section we did not consider assignments as part of specification models.)

The general form of the kernel state of the environment is $\exists x(L(x) \wedge F(x))$, where $F(x)$ is a formula of the base language without list type expressions and $L(x)$ is a conjunction of equalities of the form $x = y$, where x is a constant attribute expression of a list type and y is a list expression.

4 Satisfiability

The main functionality of the deductive system of VRS is to check the satisfiability of formulae of the base language. This is used in determining the applicability of basic protocols and in computing predicate transformers.

The applicability of the basic protocol (1) in environment state $\gamma(r)[m:u] = \gamma(r)[m_1:u_1, m_2:u_2]$ is equivalent to the satisfiability of the formula

$$\gamma(r) \wedge (m:u) \wedge \exists x \alpha(x,r) \tag{2}$$

Checking satisfiability of this formula proceeds in four steps:

1. Elimination of state assumptions.
2. Elimination of list access functions.
3. Elimination of functional attributes
4. Proving closed formula without attribute expressions.

These steps are discussed below.

Elimination of state assumptions. Satisfiability of a state assumption given a precondition and a state of the environment is checked by means of matching concrete state assumptions of the environment state and state assumptions depending on parameters and attribute expressions, both considered as matching variables. Matching is performed modulo the following relations: $m:(u+v) = (m:u) \vee (m:v), (m:a.u = (n:b.v) \iff (m=n) \wedge (a=b) \wedge (u=v)$. The result will be a new constraint $\sigma'(x,r)$ that replaces state assumptions $\sigma(x,r)$ in precondition and environment state, after which the formula does not depend on state assumptions. Note that there can be multiple results depending on which state assumptions were matched, where each result gives explicit values for parameters and attribute expressions occurring in the precondition. To prove satisfiability we need only one solution, so these are considered one by one.

Elimination of list access functions. We compute the instantiations of the empty list predicate and the accessor functions. In both cases, a list is matched against $(u(x,r))$. The value of the empty list predicate is set to 0 or 1, depending on whether the match succeeded. For accessors, we either obtain a simple type expression a returned from an accessor, or a a new variable v of type τ (the type of the list expression) is generated and added to quantifier prefix (initially empty) of the formula (2). The accessor is set equal to either a or v. The matching of attribute expressions can have several results. Different results are represented as a disjunction of formulae.

Elimination of functional attributes. After step 2, we arrive at a formula D without state assumptions or list access functions. If all of attribute expressions that occur in D are simple attributes, for checking satisfiability it is sufficient to prove validity of the closed formula $\exists RD$, where R is the list of all simple attributes which occur in D considered as variables. In the general case, the Shostak method [30], adapted to a combination of numerical, symbolic, and enumerated data types is used. This method proceed as follows:

First, superpositions of functional expressions are eliminated by successive substitution of every innermost occurrence of $f(x)$ by a new variable y, bound with an existential quantifier, and adding the

formula $y = f(x)$. For example, formula $P(f(g(x)))$ is replaced by formula $\exists y((y = g(x)) \wedge P(f(y)))$. After all of such replacements, there will no more nested functional expressions. For every attribute expression f of array or functional type, all its occurrences $f(x_1), f(x_2), \ldots$ with different parameters $x_1, x_2, \ldots$ are considered: $f(x_i)$ is replaced by variable y_i, bound with an existential quantifier, and equations $(x_i = x_j) \rightarrow (y_i = y_j)$ are added. At this point, there will be only simple attributes and the method for simple attributes is applied. Elimination of functional expressions imposes restrictions on the range of values of arguments for the functional attributes of type array with integer or enumerated indexes. For example, if the attribute f has a type **a**rray(m, τ), where m is a number, and τ is an enumerated type with constants $a_1, a_2, \ldots$, then during elimination of functional expression $f(i,u)$, the generated formula will include conjunctive constraints $0 \leq i \wedge i \leq m-1 \wedge (u = a_1 \vee u = a_2 \vee \ldots)$.

The result is a closed formula (i.e. a formula not containing attributes) and all bound variables have types integer, real, or symbolic, or are enumerated types.

Proving a closed formula without attribute expressions. The deductive system of VRS contains three specialized provers: an integer prover for Pressburger arithmetic, the Furie-Motskin algorithm for linear arithmetic over reals, and a symbolic prover that includes an algorithm of finding the most general solution for a system of symbolic equations (modified Montanari-Rossi algorithm of unification integrated with numerical provers).

The method of proving closed formulae first reduces a formula to prenex normal form and tries to prove it by simple reductions and simplifications. If this is not successful, then the specialized provers are invoked. The first step of the reduction eliminates symbolic constructors. All symbolic equalities of the form $t_1 = t_2$ in which t_1 and t_2 are not variables are analyzed. If, for example, $t_1 = f_1(a_1, a_2), t_2 = (b_1, b_2)$, where f_1, f_2 are symbolic constructors such that $f_1 = f_2$, then this equality is replaced by the equivalent conjunction of equalities $a_1 = b_1 \wedge a_2 = b_2$. For different constructors, $t_1 = t_2$ is replaced with 0. After that all possible substitutions of variables are performed. Given a literal $v = e$, occurrences of variable v are replaced with e, if e does not depend on v and if the type of v includes the type of e. If a symbolic variable v occurs in expression e, the equality is impossible and is replaced by 0. Similarly, 0 is substituted for type mismatches.

Then enumerated types are eliminated. The naive method of replacing a subformula of a form $\exists x P(x)$ where x is a variable of enumerated type with values $a_1, a_2, \ldots$ by the disjunction $P(a_1) \vee P(a_2) \ldots$ is too inefficient due to the large expressions created. Instead, we rely on recursive elimination: After maximally narrowing the scopes of quantifiers, formulae are proved inside out, expanding the above existential quantification only when a recursive proof was successful.

To prove a closed formula without enumerated types it is reduced to a disjunction of conjunctions of literals bounded with existential quantifiers. It is now sufficient to prove one of these conjunctions. They are transformed to prenex normal form, and a proof search begins. All literals are divided into three groups: equalities, negation of equalities and numerical inequalities. At first, the system of symbolic-numerical equalities is solved. If this system is consistent, then the most general solution $v_1 = e_1 \wedge v_2 = e_2 \ldots$, where expressions $e_1, e_2 \ldots$ do not depend upon variables $v_1, v_2, \ldots$ is found. Variables $v_1, v_2, \ldots$ are eliminated by a substitution of $e_1, e_2 \ldots$ instead of the corresponding variables in a formula. Now there are only negations of equalities and numerical inequalities. Symbolic negations of equalities $p \neq q$ are eliminated by finding the most general solution of equality $p = q$. If this solution does not exist, then the negation is true and can be deleted. If a general solution $v = t$ for some symbolic variable v exists, the symbolic variables can take on infinitely many values, and therefore it is possible for this variable to take on a value which make equality $v = t$ false. Therefore a literal $p \neq q$ is solvable and can be removed. If a general solution contains numerical equalities only, we add the negation of this solution to the numerical inequalities. After the removal of all solved negations with symbolic variables, there remains a pure numerical formula which is solved by the appropriate numerical prover.

5 Predicate transformers

We define the algorithm of computing the transition

$$\gamma(r)[m:u] \xrightarrow{B} \gamma'(r)[m:u']$$

for basic protocol (1) in the case when the applicability condition is valid. The applicability condition is only a necessary condition for the existence of a transition that satisfies all other restrictions.

In a first step, the new states of the agents are computed. We force these new states to be concrete by assuming that each parameter and attribute expression from state assumptions in postconditions occur also in the state assumptions of preconditions. We now determine all solutions for $[m:u']$ and for each such solution add the corresponding constraints to the precondition of the basic protocol.

The second step will be computing $\gamma'(r)$ for each given solution for all states by applying the (direct) predicate transformer $\mathrm{pt}(D,\beta)$ that transforms a formula D of the base language to new formula of the base language. The final result of computing the transition is

$$\gamma'(r) = \exists x \mathrm{pt}(\gamma(r) \wedge \alpha(x,r) \wedge \sigma'(x,r), \beta(x,r))$$

Main requirements on pt. We shall consider only kernels of concrete states of the environment and formulae of the base language without state assumptions. Let s be a concrete state of the environment and D be a formula of the base language. D covers s if $s \models D$. Let $\mathrm{State}(D)$ be the set of all concrete states that are covered by the formula D. Let the post-condition of B be $\beta = R \wedge U \wedge F$, where $R = (r_1 := t_1 \wedge r_2 := t_2 \wedge \ldots)$ is a conjunction of simple assignments, L is conjunction of list update operators, and F is the formula part of the post-condition.

The formula of the post-condition determines a transition relation on the set of all concrete states. Let s be a concrete state. We will evaluate values on this state $u_1, u_2, \ldots$ of expressions $t_1, t_2, \ldots$ and new values $v_1, v_2, \ldots$ of updated lists $l_1, l_2, \ldots$. Consider a formula $(r_1 = u_1) \wedge (r_2 = u_2) \wedge \ldots \wedge (l_1 = v_1) \wedge (l_2 = v_2) \wedge \ldots \wedge F$. If s' is a concrete state covered by this formula, then s can transition to a new state $s' : s \xrightarrow{\beta} s'$. In the general case the transition relation, as determined by the post-condition β, is nondeterministic; it will be deterministic if $F = 1$. The transition relation $\xrightarrow{\beta}$ is defined on the sets of concrete states as follows:

$$S \xrightarrow{\beta} S' \Leftrightarrow S' = \{s' | \exists (s \in S')(s \xrightarrow{\beta} s')\} \wedge (S' \neq \emptyset)$$

This relation is deterministic. Let $\mathrm{Nstate}(S,\beta) = \{s' | \exists (s \in S)(s \xrightarrow{\beta} s')\}$. We impose the following requirements on a predicate transformer pt:

- $\mathrm{Nstate}(\mathrm{State}(D),\beta) \subseteq \mathrm{State}(\mathrm{pt}(D,\beta))$,
- $\mathrm{Nstate}(\mathrm{State}(D),\beta) = \emptyset \Rightarrow \mathrm{pt}(D,\beta) = 0$.

The first condition means that a new symbolic state of the environment must cover all concrete states which can be reached from the states covered by formula D and satisfy the simulation requirement. The second condition means that if from all states covered by the formula D no transitions are possible (i.e. all of them are deadlock states), then $\mathrm{pt}(D,\beta)$ must always be false (i.e. the symbolic state D is also a deadlock state). From the first condition the reverse of the second follows: if the predicate transformer returns an always-false formula, all states covered by D are deadlock states and $\mathrm{Nstate}(\mathrm{State}(D),\beta) = \emptyset$.

A predicate transformer is called *ideal*, if it satisfies

$$\mathrm{Nstate}(\mathrm{State}(D),\beta) = \mathrm{State}(\mathrm{pt}(D,\beta))$$

An ideal predicate transformer corresponds to the strongest postcondition. One of the possible predicate transformers, satisfying the main requirements, has been realized in VRS and is described below.

VRS predicate transformer. We will fix the input for the predicate transformer formula D and β. It will be assumed that the formula $(l_1 = L_1) \wedge (l_2 = L_2) \wedge \ldots$ expressing the states of a list expressions implicitly occurs in the formula D. Consider the set of all occurrences of functional expressions in these formulae, treating simple attributes as 0-ary functions. We call an occurrence of a functional expression as external (high-level), if it does not occur within other functional expressions. Consider three lists of functional expressions r, s and z. List $r = (r_1, r_2, \ldots)$ consists of the left hand sides of assignments and of other functional expressions that recursively depend on these left hand sides. List $s = (s_1, s_2, \ldots)$ consists of functional expressions which have external occurrences in the formula part F of the post-condition but do not coincide with expressions from list r. List $z = (z_1, z_2, \ldots)$ consists of functional expressions which have external occurrences in formula D, in right hand sides of assignments, and in list update functions, but are not included in the two other lists. Now, considering the formulae from which a post-condition and D are constructed as functions of external occurrences of elements of these lists we obtain a representation of the post-condition in the following form:

$$\beta(r,s,z) = (r_1(r,s,z) := t_1(r,s,z) \wedge r_2(r,s,z) := t_2(r,s,z) \wedge \ldots) \wedge U(r,s,z) \wedge F(r,s)$$

We write the formula D as $D(r,s,z)$ to make its dependence on the elements of the corresponding lists of functional expressions explicit.

The assignments and lists updates uniquely determine a correspondence between the old and new values of attribute expressions. As old values of attributes generally are not known, this correspondence for assignments (under restrictions discussed below), can be expressed by the following formula (assignment formula):

$$\exists(x,y)((r_1(x,y,z) = t_1(x,y,z)) \wedge (r_2(x,y,z) = t_2(x,y,z)) \wedge \ldots)$$

This formula expresses the following assumptions:

- Attributes from list s on which the formula part of the post-condition depends, can change the values arbitrarily provided that the formula $F(r,s)$ is true. Therefore, for these attributes their previous values, denoted by variables from the list y, are used in the assignment.
- Attributes from the list z preserve their values across the application of a basic protocol.
- Attributes from the list r change their values in accordance with the semantics of assignments, such that their previous values correspond to the elements of list x.

The formula for list updates is constructed in a similar way.

The predicate transformer is constructed as a strong formula from which the formula part of the post-condition and the results of assignments and list updates follows. In addition, such formula must take into account the substitutivity property for functional expressions and include the remaining accessible information from formula D of the previous state of the environment.

Predicate transformer pt is defined by the following formula:

$$\mathrm{pt}(D(r,s,z), \beta(r,s,z)) = q_1 \vee q_2 \vee \ldots$$

where

$$\begin{aligned} q_i &= \exists(x,y)(D(x,y,\xi_i) \wedge R_i(x,y,\xi_i) \wedge L_i(x,y,\xi_i) \wedge E_i(x,y,\xi_i)) \wedge F(r,s) \\ R_i(x,y,\xi_i) &= (r_1 = t_1(x,y,\xi_i)) \wedge (r_2 = t_2(x,y,\xi_i)) \wedge \ldots \\ L_i(x,y,\xi_i) &= (l_1 = L'_1(x,y,\xi_i)) \wedge (l_2 = L'_2(x,y,\xi_i)) \wedge \ldots \end{aligned}$$

ξ_i is z with some of functional expressions replaced with variables from lists x and y. Thus, $(r_1 = t_1(x,y,z)) \wedge (r_2 = t_2(x,y,z)) \wedge \ldots$ is a quantifier-free part of the assignment formula, and $(l_1 = L'_1(x,y,\xi_i)) \wedge (l_2 = L'_2(x,y,\xi_i)) \wedge \ldots$ is a quantifier-free part of the formula representing list updates.

Each of the disjuncts q_i corresponds to one of the possible means of identifying of functional expressions that occur in formulae D and β. We will consider the set M of all pairs of functional expressions of the form $(f(u), f(v)), u = (u_1, u_2, \ldots), v = (v_1, v_2, \ldots)$, where $f(u)$ is chosen from list z, and $f(v)$ from lists r and s. These functional expressions must be equal if their arguments were equal before application of the basic protocol.

For the construction of q_i we choose an arbitrary subset $N \subseteq M$ (including the empty set, where the different indexes i corresponds to different selections of the subset N, such that they cover all successful selections). For every pair $(f(u), f(v)) \in N$ we consider the conjunction of equalities $u = v, (u_1 = v_1 \vee u_2 = v_2 \vee \ldots)$. We join all such conjunctions and add to it the conjunct of the negations of all equalities which correspond to pairs not included in the set N. Call the formula $E_i(r,s,z)$. If this formula is satisfiable, then the choice is successful. $f(u)$ is not independent and its value is chosen such that $E_i(r,s,z)$ remains true. Thus, $f(u)$ changes its value corresponding to $f(v)$. In all formulae, the occurrence of functional expression $f(u)$ must be replaced with a variable which corresponds to the functional expression $f(v)$. If $f(u)$ coincides with several functional expressions, due to the transitivity of equality any legal value can be chosen, and all such replacements for the chosen set of pairs specify a substitution ξ_i.

The predicate transformer is the disjunction of formulae q_i for all successful selections of a set of pairs for the unified functional expressions. q_i may be unsatisfiable; therefore, before including it into the disjunction, its satisfiability is verified. Detailed proving of ideality of predicate transformer is considered in the paper [32].

6 Verification

The input language of VRS expresses requirements specifications for systems through basic protocols. The details of this language have been determined based on a large number of industrial projects aimed at the development of software for distributed reactive systems in various industries. These notations include only some abstractions necessary for the eventual description of algorithms and ignores details connected with the representation of the process part. Neither is efficiency a concern at this point. VRS allows the use of various representations of requirements, from tabular forms to scenario languages. These representations are translated into basic protocols by front-end tools.

VRS is comprised of two groups of tools. The first group is aimed at static requirement checking (SRC), the second is aimed at the dynamic verification of system behavior.

SRC is supported by the following tools: a consistency checker, a completeness checker, a safety checker, and a reachability checker. A basic protocol specification is consistent if under the same state assumptions there is only one basic protocol that can be applied. This condition ensures determinism of a specified system comprised of behaviors of agents that may be nondeterminate. Consistency is checked by proving the satisfiability of the preconditions of arbitrary two protocols. If their conjunction is not satisfiable they are consistent. Completeness is considered to be the validity of the disjunction of preconditions of basic protocols with the same state assumptions and is checked by means of a satisfiability algorithm, i.e., its negation must be unsatisfiable. Completeness is a sufficient condition for the absence of dead locks.

Both consistency and completeness conditions are only sufficient. If a violation has been found we must prove that such state is reachable from the initial states of a system. For this purpose the static reachability checker can be used which tries to prove the safety of the negation of such violation. The

reachability checker and the safety checker use the direct predicate transformer defined above. To prove that a given condition is a safety condition, the checker generates the following invariants: if a condition is valid before applying a basic protocol, then it will be valid after its application.

Dynamic verification tools generate traces of a model of the specified system. We provide two trace generators: The concrete trace generator (CTG) works with concrete models of a system and uses a restricted base language limiting assignment statements to the postconditions. The CTG is similar to a model checker and uses heuristics and abstractions to reduce the state space, decrease interleaving, etc. The symbolic trace generator (STG) imposes no restrictions on the base language and uses both direct and backward predicate transformers. The former is used for search for traces from initial to goal states, the latter is used for finding traces from a goal state to initial states. STG and CTG rely on a common generating engine and common tools for controling the search. Both generators can be controled by intermediate goal states and check the validity of given or predefined safety conditions, such as the absence of dead locks in the traversed state space.

7 Conclusions

We have outlined the key algorithms used in the VRS system for the symbolic verification of requirement specifications. VRS was successfully piloted in a number of industrial projects in a large corporation. The main application domains were telecommunication, automotive, and telematics. Industrial projects successively piloted using VRS consisted of up to 10,000 requirements formalized as basic protocols, with over thousand attributes. Substantial numbers of requirements defects were detected by applying the VRS tools to these industrial specifications.

The trace generators have also been leveraged for the generation of tests, and verified specifications have been used as the starting point for further product development.

The theoretical foundation of VRS is the theory of interaction of agents and environments [2, 3], now referred to as insertion modeling [31]. Traditional mathematical models for specifications of concurrent systems usually are based on process algebras (CSP [13], CCS [14], Lotos, ACP, μCRL, π-calculus, etc.), temporal and dynamic logics (LPTL, LTL, CTL, CTL*, PDL), and automata models.

Temporal logic is a formal specification language for the description of behavioral properties of non-terminating and interacting (reactive) systems. Traditional one distinguishes between safety (”something bad never happens”), liveness (”something good will eventually happen”), and various fairness properties. For example, Lamport’s TLA (Temporal Logic of Actions) [15, 16] is aimed at the description of such properties and is based on Pnueli’s temporal logic [17] with assignment, enriched signatures, and module specifications.

Many temporal logics are decidable and corresponding decision procedures exist for linear and branching time logics [18], propositional modal logic [19], and some variants of CTL* [20]. In such decision procedures techniques from automata theory, semantic tableaux, or binary decision diagrams (BDD) [21] have been be used. Typically, a system to be verified is modeled as a (finite) state transition graph, and the properties are formulated in an appropriate temporal logic. An efficient search procedure is then used to determine whether the state transition graph satisfies the temporal formula or not. This technique was first developed by Clarke and Emerson [22], and by Quielle and Sifakis [23] and extended later by Burch et.al. [24].

In the current version of our VRS system, temporal formulae are represented implicitly and are evaluated by checking algorithms. Enhancements are planned to allow the explicit representation of temporal formulae.

The most closely related verification tools to our approach are the SCR toolset [25] and the Action Language Verifier [26]. Different from these systems, VRS uses MSC [1, 8] as the language for capturing

the process part of system requirements. This choice of representation was guided by our experience in applying our tools in industrial projects. Powerful extensions to MSC have been proposed: Live Sequence Charts (LSC [27]), Triggered Message Sequence Charts (TMSC [28]), and Object Message Sequence Charts (OMSC [29]). We are investigating similar extensions to the representation of basic protocols as well as new algorithms for symbolic checking and invariant construction.

References

[1] International Telecommunications Union. ITU-T Recommendation Z.120: Message Sequence Charts. Geneva, ITU-T, 2002. http://www.itu.int/ITU-T/studygroups/com17/languages/Z.120AnnB-0498.pdf, last viewed June 2010, 2002-2010.

[2] A. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999. http://www.springerlink.com/content/2cratqgq5efr6vfh/, last viewed June 2010, 1999-2010.

[3] A. Letichevsky. Algebra of behavior transformations and its applications, in V.B.Kudryavtsev and I.G.Rosenberg eds. Structural theory of Automata, Semigroups, and Universal Algebra, NATO Science Series II. Mathematics, Physics and Chemistry - Vol. 207, pp. 241-272, Springer 2005. http://schum.kiev.ua/let/files/paper_on_behavior_transformations.pdf, last viewed June 2010, 2005-2010.

[4] S. Baranov, C. Jervis, V. Kotlyarov, A. Letichevsky, and T. Weigert. Leveraging UML to Deliver Correct Telecom Applications. In L. Lavagno, G. Martin, and B.Selic, editors. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Amsterdam, 2003. http://portal.acm.org/citation.cfm?id=886360, last viewed June 2010, 2003-2010.

[5] A. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 2005, 662-675. BasicProtocols,MessageSequenceCharts, andtheVerificationofRequirementsSpecifications, last viewed June 2010, 2005-2010.

[6] J. Kapitonova, A. Letichevsky, V. Volkov, and T. Weigert. Validation of Embedded Systems. In R. Zurawski, editor. The Embedded Systems Handbook. CRC Press, Miami, 2005. http://www.amazon.com/Embedded-Handbook-Industrial-Information-Technology/dp/0849328241, last viewed June 2010, 2005-2010.

[7] A. Letichevsky, J. Kapitonova, V. Volkov, A. Letichevsky, jr., S. Baranov, V. Kotlyarov, and T. Weigert. System Specification with Basic Protocols, Cybernetics and System Analyses, 4, 2005. http://portal.acm.org/citation.cfm?id=1103717, last viewed June 2010, 2005-2010.

[8] International Telecommunications Union. Recommendation Z.120 Annex B: Formal semantics of Message Sequence Charts, 1998. http://www.itu.int/ITU-T/studygroups/com17/languages/Z.120AnnB-0498.pdf, last viewed June 2010, 1998-2010.

[9] M. Reniers. Message Sequence Chart: Syntax and Semantics. Eindhoven, University of Technology, 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.8570, last viewed June 2010, 1998-2010.

[10] A. Letichevsky, J. Kapitonova, V. Kotlyarov, V. Volkov, A. Letichevsky Jr., and T. Weigert. Semantics of Message Sequence Charts, SDL Forum, 2005. http://www.springerlink.com/content/u0w44b0r00qmnujp/, last viewed June 2010, 2005-2010.

[11] A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, V. Volkov. Semantics of timed MSC language, Kibernetika and System Analysis, 2002. http://www.springerlink.com/content/q42280551554123x/, last viewed June 2010, 2002-2010.

[12] A. Degtyarev, J. Kapitonova, A. Letichevsky, A. Lyaletsky, and M. Morokhovets. Evidence algorithm and problems of representation and processing of computer mathematical knowledge. Kibernetika and System Analysis, (6):9-17, 1999. http://www.springerlink.com/content/q0540m70v17u7716/, last viewed June 2010, 1999-2010.

[13] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985. http://www.usingcsp.com/cspbook.pdf,last viewed June 2010, 1985-2010.

[14] R. Milner. Communication and Concurrency. Prentice Hall, 1989. http://portal.acm.org/citation.cfm?id=63446,last viewed June 2010, 1989-2010.

[15] L. Lamport. Introduction to TLA. SRC Technical Note 1994-001, 1994. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1994-001.html,last viewed June 2010, 1994-2010.

[16] L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 872-923, May 1994. http://research.microsoft.com/pubs/64074/lamport-actions.pdf,last viewed June 2010, 1994-2010.

[17] A. Pnueli. The temporal logic of programs. In: Proc. of the 18th Annual Symposium on the Foundations of Computer Science, 46-52, Nov. 1977. http://portal.acm.org/citation.cfm?id=1382534,last viewed June 2010, 1977-2010.

[18] E. Emerson and J. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. J. of Comp. and System Sci., 30(1):1-24, 1985. http://portal.acm.org/citation.cfm?id=800070.802190,last viewed June 2010, 1985-2010.

[19] M. Fisher and R. Ladner. Propositional modal logic of programs. In Proc. 9th ACM Ann. Symposium on Theory of Computing, 286-294, Boulder, Col., May 1977. http://portal.acm.org/citation.cfm?id=800105.803418,last viewed June 2010, 1977-2010.

[20] E. Emerson. Temporal and modal logic. In: J. van Leeuwen editor: Handbook of Theoretical Computer Science, Elsevier, (B):997-1072, 1990. http://www.cs.utexas.edu/users/emerson/Pubs/handbook3.ps,last viewed June 2010, 1990-2010.

[21] R. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, 35 (8): 677-691, 1986. http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf,last viewed June 2010, 1986-2010.

[22] E. Clarke and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In: The Workshop on Logic of Programs, 128-143, Lecture Notes in Computer Science, 131. Springer Verlag, 1981. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.2037,last viewed June 2010, 1981-2010.

[23] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In: Proc. 5th Intern. Symposium on Programming , 142-158, 1981. http://portal.acm.org/citation.cfm?id=647325.721668,last viewed June 2010, 1981-2010.

[24] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. Information and Computation, 98 (2): 142-170, 1992. http://portal.acm.org/citation.cfm?id=162046,last viewed June 2010, 1992-2010.

[25] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten, Computer Systems Science & Engineering, 1: 19-35, 2005. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.2130&rep=rep1&type=pdf,last viewed June 2010, 2005-2010.

[26] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In Proc. of ASE 2001, 382-386, November 2001. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.8434&rep=rep1&type=pdf,last viewed June 2010, 2001-2010.

[27] D. Harel, R. Marelly. Come, Let's Play: Scenario-Based programming Using LSCs and the Play-Engine. Springer 2003. http://portal.acm.org/citation.cfm?id=861638,last viewed June 2010, 2003-2010.

[28] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In Proceedings of SIGSOFT 2002/FSE-10, 167-176, ACM Press, 2002 http://portal.acm.org/citation.cfm?id=587077,last viewed June 2010, 2002-2010.

[29] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture-A System of Patterns. Wiley & Sons, New York, 1996. http://portal.acm.org/citation.cfm?id=249013,last viewed June 2010, 1996-2010.

[30] R. Shostak, A Practical Decision Procedure for Arithmetic with Function Symbols, Journal of the Assooatlon for Computing Machinery, Vol 26, No 2, April 1979, pp 351-360.

[31] A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, N. Nikitchenko, V. Volkov, and T. Weigert, Insertion modeling in distributed system design, Problems in Programming (ISSN 1727-4907), 4, 2008, 13-39. http://dspace.nbuv.gov.ua:8080/dspace/bitstream/handle/123456789/2599/03-Letichevsky.pdf?sequence=1,last viewed June 2010, 2008-2010.

[32] Letichevsky A.A., Godlevsky A.B., Letychevskyy A.A., Potiyenko S.V., Peschanenko V.S., Properties of a predicate transformer of VRS system. Kibernetika i sistemny analiz, 4, 2010, 3-16 (in russian). http://apsystem.org.ua/uploads/doc/ims/PPTVRS.pdf,last viewed June 2010, 2010-2010.

The RISC ProgramExplorer: Reasoning about Programs as State Relations (Extended Abstract)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

1 Introduction

We report on the formal framework underlying the *RISC ProgramExplorer* [12], a program reasoning environment which is under development at the Research Institute for Symbolic Computation (RISC) and which integrates the previously developed *RISC ProofNavigator* [14] as an interactive proving assistant. The current release of the software is a first demonstrator that incorporates the overall technological framework (including an elaborated graphical user interface) and language processors (for a simple subset of Java as a programming language and a formal specification language). Work is going on to provide this skeleton with the envisioned program reasoning capabilities. The goal of this presentation is to outline the formal basis underlying these capabilities and to explain the rationale for its particular design.

In the "Hoare Calculus" [5], a program command `x=x*x` is specified by a triple $\{x = a\}$`x=x*x`$\{x = a^2\}$ with two logical formulas $x = a$ and $x = a^2$ evaluated on the pre-state and on the post-state of the command. Here a "fresh" logical constant a is "pulled out of the hat" to let the post-condition refer to the value of the program variable `x` in the pre-state. In "Dynamic Logic" [3], the same command is specified by the formula $\forall a : \mathtt{x} = a \Rightarrow [\mathtt{x=x*x}]\mathtt{x} = a^2$ where a becomes a bound logical variable and the two formulas $\mathtt{x} = a$ and $\mathtt{x} = a^2$ separated by the modality `[x=x*x]` have to be interpreted over two different states, the pre-state of `x=x*x` and its post-state. Both formalisms suffer from the fact that they squeeze the formulation of a binary state *relation* (the relation between the pre- and the post-state of a command) into two unary state *conditions* and thus need logical constants/variables to "glue" them together. In both formalisms a layer of reasoning with unfamiliar rules (on Hoare triples respectively formulas with modalities) is required before ultimately the well-known layer of classical predicate logic is reached.

We have chosen another possibility described in [13], the *relational* view of program commands. This allows us to write a specification of form $\mathtt{x=x*x} : \mathtt{x}' = \mathtt{x}^2$ which associates to a program command `x=x*x` a single formula $\mathtt{x}' = \mathtt{x}^2$ in classical logic (the actual syntax described later is slightly different). This formula expresses a state relation with the unprimed variable `x` denoting the value of the corresponding program variable in the pre-state and the primed version $\mathtt{x}'$ its value in the post-state. A judgment $C : S$ can now be read as "program command C implements the specification S"; this also gives immediately rise to a notion of *refinement*, since $C : S'$ and $S' \Rightarrow S$ implies $C : S$. For verification, this calculus can be exploited in the following way: given a command C specified by state relation S, we first derive a judgment $C : S'$, i.e. we *translate* the command C into a logical formula S', and then prove $S' \Rightarrow S$ using the rules of classical logic.

However, to translate a program command to a state relation alone is not sufficient, because it only describes which state transitions are allowed but does not require that any state transition actually takes place. Thus for example the judgment `while(true){}` $: S$ is trivially satisfied for any specification S, since the non-terminating loop implements the empty state relation. Our solution to this problem is

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 131-143

straight-forward: we complement the specification of a command by a judgment $C \downarrow T$ where T is a logical formula that describes a state condition, namely a condition for the pre-state of C under which the command must terminate and produce a post-state. Here the refinement works in the other direction: $C \downarrow T'$ and $T \Rightarrow T'$ implies $C \downarrow T$. Both kinds of judgments $C : S$ and $C \downarrow T$ allow to specify a program command's safety (partial correctness) and liveness (total correctness).

Actually, a specification `x=x*x` : $\mathtt{x}' = \mathtt{x}^2$ is too weak since it does not state that every variable `y` different from `x` has the same value in the pre- and in the post-state (assuming that program variables are not aliased); also it does not allow to specify the effect of commands like `continue`, `break`, `return E` or `throw E` which change the state of a program, not by the modification of a user-visible variable, but by altering the flow of control. Our calculus is therefore based on a more general notion of state which also includes the current *execution mode* (executing normally, continuing a loop iteration, breaking from a loop, returning from a method, throwing an exception). Furthermore, it considers the *frame* of a command execution, i.e. the set of variables that may be modified. Judgments are thus actually of the form $C : [S]^V_m$ where V is the set of variables that may be modified by C and m constrains the execution mode of the post-state of C. We can thus write a specification `x=x*x` : $[\mathtt{x}' = \mathtt{x}^2]^{\{x\}}_{m_e}$ where m_e encodes the mode "normally executing"; another specification is `return x` : $[\text{value}(\text{next}) = x]^{\{\}}_{m_r}$ where 'next' is a logical constant representing the post-state and 'value' is a logical function that extracts the return value from the state.

The derivation of judgments is compositional such that e.g. the judgment `x=x+1;x=x*x` : $[S]^{\{x\}}_{m_e}$ with $S \equiv \exists x_0 : x_0 = x + 1 \wedge x' = x_0^2$ can be derived from the individual judgments for `x=x+1` and `x=x*x`. Further on the formula S can be logically simplified to $x' = (x+1)^2$ which concisely describes the *semantic essence* of command `x=x+1;x=x*x`. A core goal of the RISC ProgramExplorer is to make this computation of a program's semantic essence transparent to the programmer as a means to better understand the meaning of a program: program code is translated to a logical formula in a format that makes the relationship to the code explicit; after suitable simplification, the formula displays the code's semantic essence. Thus e.g. the RISC ProgramExplorer will be able to illustrate the relationship between the states at two control points in a declarative way.

Still there is a strong relation to the classical calculus, e.g. from a pre/post-condition pair $x = a$ and $x = a^2$, a state relation $\forall a : x = a \Rightarrow x' = a^2$ can be derived (which can then be further simplified to $x' = x^2$); a state relation $x' = x^2$ for command C can be immediately translated to a weakest precondition $\text{wp}(C, Q) \equiv \forall b : b = x^2 \Rightarrow Q[b/x]$ (which can be simplified to $Q[x^2/x]$) or to a strongest postcondition $\text{sp}(C, P) \equiv \exists a : P[a/x] \wedge x = a^2$. The RISC ProgramExplorer will provide these translations and make use of them, e.g. to describe the information known about the state at a particular control point or to derive the condition required to reach a subsequent control point.

In this extended abstract, we present only a simplified version of the calculus that does not include the execution modes sketched above and does correspondingly not handle statements that interrupt the control flow; also we do not discuss the treatment of global program variables, program methods, and recursion. See [13] for a treatment of these aspects. The organization of the presentation is as follows: in Section 2 we discuss the core calculus for deriving judgments $C : [F]^V$; in Section 3 we introduce auxiliary judgments for deriving pre- and post-conditions; in Section 4 we describe the judgment $C \downarrow T$. Section 5 discusses related work and Section 6 concludes.

2 The Core Calculus

Let *State* be the set of program states (to be defined below) and let $\mathit{StateRelation} := \mathbb{P}(\mathit{State} \times \mathit{State})$ be the set of binary relations on states. We interpret a state relation $R \in \mathit{StateRelation}$ as a set of possible state transitions: a pair of states $\langle s, s' \rangle \in R$ (i.e. two states s, s' such that the relation $R(s, s')$ holds) is

considered as a transition from pre-state s to post-state s' that is possible according to R.

We are now going to introduce a domain of (essentially classical) logical formulas such that every formula F is translated to a state relation $[\![F]\!] \in \mathit{StateRelation}$, i.e. as the set of transitions that are allowed by F. The formula domain is equipped with a (essentially classical) logical calculus such that, if the formula $F_1 \Rightarrow F_2$ can be proved, then $[\![F_1]\!] \subseteq [\![F_2]\!]$ holds, i.e. F_1 only allows those transitions that are also allowed by F_2. Likewise, we are going to introduce a domain of program commands and translate every command C to a state relation $[\![C]\!] \in \mathit{StateRelation}$, i.e. as the set of transitions that can be performed by the command (the relation $[\![C]\!]$ need not be a function, since C may be non-deterministic, or at least non-deterministically specified). The core of the calculus is a judgment $C : F$ that translates every command C to some formula F such that the soundness condition $[\![C]\!] \subseteq [\![F]\!]$ holds, i.e. every transition that can be performed by C is also allowed by F.

The domain of commands is defined by the grammar

$$C := I\texttt{=}E \mid \texttt{\{var } I\texttt{;}C\texttt{\}} \mid C_1\texttt{;}C_2 \mid \texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 \mid \texttt{while (}E\texttt{) } C^{F,T}$$

where E is an unspecified domain of program expressions denoting values. The programming language thus supports variable assignments, local variable declarations, command sequences, conditional commands, and loops (annotated with invariance formulas and termination terms). An invariance formula F describes not a condition on a single state but a relation between two states, the pre-state of the loop and the state before/after every loop iteration. On the other hand, the value of the termination term T is determined by a single state; this value must become smaller by every iteration according according to some well-founded ordering. The phrases F and T do not influence the execution of the program but aid reasoning about the execution.

For the semantic interpretation of this language, let $\mathit{State} := \mathit{Identifier} \to \mathit{Value}$ be the set of program states where $\mathit{Identifier} = \{I_1, \ldots, I_n\}$ is a finite set of identifiers and Value is a set of values; every state thus is just a mapping of program variables to values. For every command C, we are now going to define the semantics $[\![C]\!]$ by writing $[\![C]\!](s,s') :\Leftrightarrow \ldots$ to denote $[\![C]\!] := \{\langle s,s'\rangle : \ldots\}$.

$$
\begin{array}{l}
[\![I\texttt{=}E]\!](s,s') :\Leftrightarrow \\
\quad \forall J \in \mathit{Identifier} : \ \textsc{if}\ J = I\ \textsc{then}\ s'(I) = [\![E]\!](s)\ \textsc{else}\ s'(I) = s(I) \\
[\![\texttt{\{var } I\texttt{;}C\texttt{\}}]\!](s,s') :\Leftrightarrow \\
\quad \exists t,t' \in \mathit{State} : [\![C]\!](t,t') \wedge \forall J \in \mathit{Variable} : J \neq I \Rightarrow s(J) = t(J) \wedge t'(J) = s'(J) \\
[\![C_1\texttt{;}C_2]\!](s,s') :\Leftrightarrow \\
\quad \exists t \in \mathit{State} : [\![C_1]\!](s,t) \wedge [\![C_2]\!](t,s') \\
[\![\texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2]\!](s,s') :\Leftrightarrow \\
\quad \textsc{if}\ [\![E]\!](s) = \textsc{true}\ \textsc{then}\ [\![C_1]\!](s,s')\ \textsc{else}\ [\![C_2]\!](s,s') \\
[\![\texttt{while (}E\texttt{) } C^{F,T}]\!](s,s') :\Leftrightarrow \\
\quad \exists n \in \mathbb{N}, t \in \mathit{State}^* : \\
\qquad t(0) = s \wedge t(n) = s' \wedge [\![E]\!](s') = \textsc{false} \wedge \\
\qquad \forall i \in \mathbb{N} : i < n-1 \Rightarrow [\![E]\!](s) = \textsc{true} \ \wedge \ [\![C]\!](t(i),t(i+1))
\end{array}
$$

Most cases are self-evident; the semantics of a while loop is specified by the sequence t of states arising from the iteration of the loop. With the help of some auxiliary functions, the last three cases can be also written more concisely. Let $R_1 \circ R_2 := \{\langle s,s'\rangle : \exists t \in \mathit{State} : \langle s,t\rangle \in R_1 \wedge \langle t,s'\rangle \in R_2\}$ denote the composition of state relations R_1 and R_2. Let $R^* := \bigcup_{i\in\mathbb{N}} R^i$ with $R^0 := \{\langle s,s\rangle : s \in \mathit{State}\}$ and $R^{i+1} := R^i \cup R^i \circ R$ denote the reflexive transitive closure of state relation R. Let $S \rhd R := \{\langle s,s'\rangle \in R : s \in S\}$ denote the restriction of the domain of state relation R to state set (condition) S and let $R \lhd S := \{\langle s,s'\rangle \in R : s' \in R\}$ denote the restriction of the range of R to S. Let $f^{\mathbb{T}} := \{s \in \mathrm{dom}(f) : [\![f]\!](s) = \textsc{true}\}$ denote that subset of the domain of function f for which the value is TRUE, and let $f^{\mathbb{F}} := \mathrm{dom}(f) \setminus f^{\mathbb{T}}$ denote the complement of that set. Then we have

$[\![C_1 ; C_2]\!] = [\![C_1]\!] \circ [\![C_2]\!]$
$[\![\texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2]\!] = ([\![E]\!]^{\mathbb{T}} \rhd [\![C_1]\!]) \cup ([\![E]\!]^{\mathbb{F}} \rhd [\![C_2]\!])$
$[\![\texttt{while (}E\texttt{) } C^{F,T}]\!] = ([\![E]\!]^{\mathbb{T}} \rhd [\![C]\!])^* \lhd [\![E]\!]^{\mathbb{F}}$

While this "point-free" algebraic definition style has some appeal, the verbose original one makes the logic underlying the command semantics more explicit. In particular, for the more general notion of states described in [13] which supports commands that interrupt the control-flow, the underlying logic is considerably more complicated such that an algebraic definition would not really give more insight than the explicit logical characterization.

Next, we define the domain of formulas by the grammar

$F := p_n(T_1, \ldots, T_n) \mid \texttt{!}F \mid F_1 \text{ and } F_2 \mid F_1 \text{ or } F_2 \mid F_1 \texttt{ => } F_2 \mid \ldots \mid \text{forall } I\texttt{: } F \mid \text{exists } I\texttt{: } F$
$T := I \mid \text{var } I \mid \text{old } I \mid f_n(T_1, \ldots, T_n)$

where p_n stands for a family of n-ary predicate constants and f_n for a family of n-ary function constants. Let $\mathit{Environment} := \mathit{Identifier} \to \mathit{Value}$ be the set of formula environments that map logical variables to values. Then the formula semantics is defined as $[\![F]\!](s,s') :\Leftrightarrow \forall e \in \mathit{Environment} : [\![F]\!]^e(s,s')$ where $[\![F]\!]^e(s,s')$ is essentially the classical interpretation of formula F in environment e and $[\![T]\!]^e(s,s')$ is essentially the classical interpretation of term T in environment e. The only places where s and s' matter are in the rules

$[\![\text{var } I]\!]^e(s,s') := s'(I)$
$[\![\text{old } I]\!]^e(s,s') := s(I)$

i.e. var I denotes the value of the program-variable I in the post-state and old I denotes its value in the pre-state (we use this syntax rather than the syntax I' and I indicated in the introduction in order to differentiate between a reference var I respectively old I to a program variable I in a state and a reference I to a logical variable I in the environment). Furthermore, we define the the evaluation of a term $[\![T]\!]^e(s) := [\![T]\!]^e(s,s)$ respectively a formula $[\![F]\!](s) := [\![F]\!](s,s)$ on a single state (such that var I and old I have the same value) which will become handy later. Given a formula F and a set of program variables $\mathit{Is} \subseteq \mathit{Identifier}$ we also define a syntactic abbreviation

$[F]^{\mathit{Is}} \equiv F \text{ and var } I_1 \texttt{ = } \text{old } I_1 \text{ and } \ldots \text{ and var } I_n \texttt{ = } \text{old } I_n$

where $\{I_1, \ldots, I_n\} = \mathit{Identifier} \backslash \mathit{Is}$. In a formula $[F]^{\mathit{Is}}$, the set I_s defines the "frame" of formula F, i.e. the set of variables that may possibly be changed by a transition and whose values must therefore be described by F; all other variables remain unchanged.

The rules for deriving judgments of the form $C : [F]^{\mathit{Is}}$ are given in Figure 1.

- Rule *weaken* allows to weaken the formula derived from a judgment by logical implication. This rule makes use of a judgment of form which $\models_{\mathit{Is}} F$ which derives the classical validity of formula F where all occurrences of terms var I and old I are considered as (different) logical variables but an axiom var I = old I is added for every variable $I \notin \mathit{Is}$.

- Rule *frame* describes how to extend the frame of a rule by a new variable which has the same value in the pre-state as in the post-state.

- Rule *assign* defines the value of variable I in the post-state by a term T derived from the expression E by a judgment $E \simeq T$ which is sound if and only if $\forall s \in \mathit{State} : e \in \mathit{Environmente} : [\![E]\!](s) = [\![T]\!]^e(s)$, i.e. T is interpreted over a single state and has the same value as E. To simplify further substitutions, we assume that all references to program variables in T are of the form old I (not of the form var I).

$$(weaken)\ \frac{C : [F_1]^{I_s} \quad \models_{I_s} F_1 \Rightarrow F_2}{C : [F_2]^{I_s}}$$

$$(frame)\ \frac{C : [F]^{I_s}}{C : [\ F \text{ and var } I = \text{old } I\ ]^{I_s \cup \{I\}}}$$

$$(assign)\ \frac{E \simeq T}{I{=}E : [\ \text{var } I = T\ ]^{I}}$$

$$(var)\ \frac{\begin{array}{l} C : [F]^{Is} \\ I_o, I_v \text{ do not occur as terms in } F \end{array}}{\texttt{\{var } I\texttt{;} C\texttt{\}} : [\ \text{exists } I_o, I_v\text{: } F[I_o/\text{old } I, I_v/\text{var } I]\ ]^{Is \setminus \{I\}}}$$

$$(seq)\ \frac{\begin{array}{l} C_1 : [F_1]^{\{I_1,\ldots,I_n\}} \\ C_2 : [F_2]^{\{I_1,\ldots,I_n\}} \\ J_1, \ldots, J_n \text{ do not occur as terms in } F \end{array}}{\begin{array}{l} C_1\,;C_2 : [\ \text{exists } J_1, \ldots, J_n : F_1[J_1/\text{var } I_1, \ldots, J_n/\text{var } I_n] \wedge \\ \qquad\qquad F_2[J_1/\text{old } I_1, \ldots, J_n/\text{old } I_n]\ ]^{\{I_1,\ldots,I_n\}} \end{array}}$$

$$(if)\ \frac{\begin{array}{l} E \simeq F \\ C_1 : [F_1]^{Is} \\ C_2 : [F_2]^{Is} \end{array}}{\texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 : [\ \text{if } F \text{ then } F_1 \text{ else } F_2\ ]^{Is}}$$

$$(while)\ \frac{\begin{array}{l} E \simeq F_E \\ C : [F_C]^{I_1,\ldots,I_n} \\ \mathit{Invariant}(F, F_E, F_C)^{\{I_1,\ldots,I_n\}} \\ J_1, \ldots, J_n \text{ do not occur as terms in } F, F_E, F_C \end{array}}{\begin{array}{l} \texttt{while (}E\texttt{) } C^{F,T} : \\ \quad [\ !F_E[\text{var } I_1/\text{old } I_1, \ldots, \text{var } I_n/\text{old } I_n] \text{ and} \\ \qquad (F[\text{old } I_1/\text{var } I_1, \ldots, \text{old } I_n/\text{var } I_n] \text{ => } F)\ ]^{I_1,\ldots,I_n} \end{array}}$$

$$\begin{array}{l} \mathit{Invariant}(F, F_E, F_C)^{\{I_1,\ldots,I_n\}} \equiv \\ \quad \models_{\{I_1,\ldots,I_n\}} \\ \quad\ \text{forall } J_1, \ldots, J_n\text{:} \\ \qquad (F[J_1/\text{var } I_1, \ldots, J_n/\text{var } I_n] \text{ and} \\ \qquad F_E[J_1/\text{old } I_1, \ldots, J_n/\text{old } I_n] \text{ and } F_C[J_1/\text{old } I_1, \ldots, J_n/\text{old } I_n]) \text{ => } F \end{array}$$

Figure 1: Judgment $C : F$

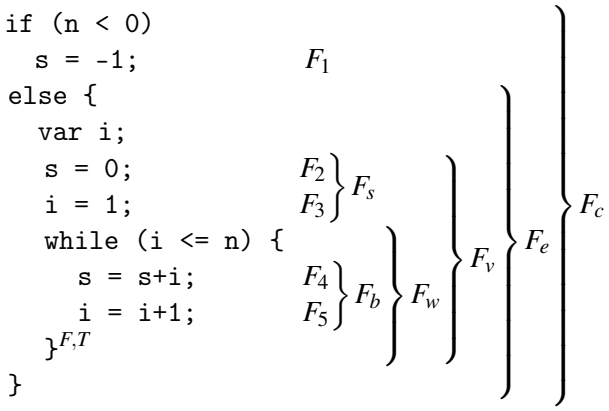


Figure 2: A Sample Program

- Rule *var* constructs from the formula F describing the behavior of the body of a block C the behavior of the block by existentially quantifying the value of the variable in the pre-state and in the post-state (the phrase $P[V/U]$ is identical to P except that V replaces every occurrence of U).
- Rule *seq* constructs the formula for a command sequence from the formulas of the individual commands with common frame by existentially quantifying the values of the variables in the intermediate state of the sequence (with the help of rule *frame* formulas can be always be extended to a common frame).
- Rule *if* constructs the formula for a conditional statement from the formulas of the individual statements with a common frame. The judgment $E \simeq F$ is sound if and only if $\forall s \in State : [\![E]\!](s) = \text{TRUE} \Leftrightarrow [\![F]\!](s)$, i.e. F is interpreted over a single state only. To simplify further substitutions, we assume that all references to program variables in F are of form old I (not of form var I).
- Rule *while* constructs the formula for a loop from the formulas F_E and F_C derived from the loop condition E and body C, respectively, and from a given loop invariant F. The proof obligation *Invariant*$(\ldots)$ states that F indeed remains invariant the execution of the loop body. The derived state relation formula expresses the fact that F_E does not hold in the post-state s' of the loop and that the invariant F holds on the post-state of the loop, provided that it also holds on the pre-state s (i.e. $[\![F]\!](s,s')$ holds, provided that $[\![F]\!](s,s)$ holds).

The rules satisfy the following theorem.

Theorem 1 (Soundness of Core Judgment)**.** *For every command C and formula F, if a judgment* $C : F$ *can be derived, then we have*

$$\forall s, s' \in State : [\![C]\!](s,s') \Rightarrow [\![F]\!](s,s')$$

In the following, we illustrate the derivation of a judgment $C : F_C$ for the program fragment C given in Figure 2 which sums the values from 1 to n. The core of the program is a while loop annotated by an invariant F and a termination term T. For the moment, only F is used, which constrains the state/before after every loop iteration (by references to program variables of the form var I); in general it may also

refer to the pre-state of the loop (by references of the form old I). We assume that $Value = \mathbb{Z}$, i.e. all variables denote integer numbers.

First we derive the formulas for the assignment statements:

$$\begin{array}{lcl} F_1 & \Leftrightarrow & [\text{var } s = -1]^{\{s\}} \\ F_2 & \Leftrightarrow & [\text{var } s = 0]^{\{s\}} \\ F_3 & \Leftrightarrow & [\text{var } i = 1]^{\{i\}} \\ F_4 & \Leftrightarrow & [\text{var } s = \text{old } s + \text{old } i]^{\{s\}} \\ F_5 & \Leftrightarrow & [\text{var } i = \text{old } i + 1]^{\{i\}} \end{array}$$

Next we extend F_2 and F_3 to a common frame and derive a judgment F_s for the sequence of the two assignment statements before the loop which is simplified to a logically equivalent formula:

$$\begin{array}{lcl} F_2 & \Leftrightarrow & [\text{var } s = 0 \text{ and var } i = \text{old } i]^{\{s,i\}} \\ F_3 & \Leftrightarrow & [\text{var } i = 1 \text{ and var } s = \text{old } s]^{\{s,i\}} \\ F_s & \Leftrightarrow & [\text{exists } v_s, v_i : v_s = 0 \text{ and } v_i = \text{old } i \text{ and var } i = 1 \text{ and var } s = v_s]^{\{s,i\}} \\ & \Leftrightarrow & [\text{var } i = 1 \text{ and var } s = 0]^{\{s,i\}} \end{array}$$

We do the same to derive a formula F_b for the sequence of assignment statements that represent the body of the loop.

$$\begin{array}{lcl} F_4 & \Leftrightarrow & [\text{var } s = \text{old } s + \text{old } i \text{ and var } i = \text{old } i]^{\{s,i\}} \\ F_5 & \Leftrightarrow & [\text{var } i = \text{old } i + 1 \text{ and var } s = \text{old } s]^{\{s,i\}} \\ F_b & \Leftrightarrow & [\text{exists } v_s, v_i : v_s = \text{old } s + \text{old } i \text{ and } v_i = \text{old } i \text{ and var } i = v_i + 1 \text{ and var } s = v_s]^{\{s,i\}} \\ & \Leftrightarrow & [\text{var } i = \text{old } i + 1 \text{ and var } s = \text{old } s + \text{old } i]^{\{s,i\}} \end{array}$$

From the loop invariant F, we construct the formula F_w for the while loop and simplify it logically:

$$\begin{array}{lcl} F_w & \Leftrightarrow & [!(\text{var } i <= \text{var } n) \text{ and} \\ & & \quad ((1 <= \text{old } i <= \text{old } n{+}1 \text{ and old } s = \sum_{j=1}^{\text{old } i - 1} j) => \\ & & \quad\quad (1 <= \text{var } i <= \text{var } n{+}1 \text{ and var } s = \sum_{j=1}^{\text{var } i - 1} j))]^{\{s,i\}} \\ & \Leftrightarrow & [(\text{old } n < \text{var } i) \text{ and} \\ & & \quad ((1 <= \text{old } i <= \text{old } n{+}1 \text{ and old } s = \sum_{j=1}^{\text{old } i - 1} j) => \\ & & \quad\quad (1 <= \text{var } i <= \text{old } n{+}1 \text{ and var } s = \sum_{j=1}^{\text{var } i - 1} j))]^{\{s,i\}} \\ & \Leftrightarrow & [(\text{old } n < \text{var } i) \text{ and} \\ & & \quad (1 <= \text{old } i <= \text{old } n{+}1 \text{ and old } s = \sum_{j=1}^{\text{old } i - 1} j) => \\ & & \quad\quad (\text{var } i = \text{old } n{+}1 \text{ and var } s = \sum_{j=1}^{\text{old } n} j)]^{\{s,i\}} \end{array}$$

To prove the invariance of F, we generate the following proof obligation

$$\begin{array}{l} \models_{\{s,i\}} \\ \quad \text{forall } v_s, v_i: \\ \quad\quad 1 <= v_i <= \text{var } n{+}1 \text{ and } v_s = \sum_{j=1}^{v_i - 1} j \text{ and} \\ \quad\quad v_i <= \text{var } n \text{ and var } i = v_i + 1 \text{ and var } s = v_s + v_i => \\ \quad\quad\quad 1 <= \text{var } i <= \text{var } n{+}1 \text{ and var } s = \sum_{j=1}^{\text{var } i - 1} j \end{array}$$

whose correctness can be easily established.

The formula F_w derived above contains contains an implication which expresses the truth of the invariant only under a condition on the range of the program variable i and the value of s in the pre-state of the loop. By combining this condition with the formula F_s that establishes the pre-state of the loop, the precondition is logically simplified to the condition that program variable n is not negative:

$$\begin{aligned} F_v \quad \Leftrightarrow \quad & [\text{exists } v_s, v_i : v_s = 0 \text{ and } v_i = 1 \text{ and} \\ & \quad \text{old } n < \text{var } i \text{ and} \\ & \quad (1 <= v_i <= \text{old } n{+}1 \text{ and } v_s = \textstyle\sum_{j=1}^{v_i - 1} j \Rightarrow \\ & \quad\quad \text{var } i = \text{old } n{+}1 \text{ and var } s = \textstyle\sum_{j=1}^{\text{old } n} j)]^{\{s,i\}} \\ \Leftrightarrow \quad & [\text{old } n < \text{var } i \text{ and} \\ & \quad (0 <= \text{old } n \Rightarrow \text{var } i = \text{old } n{+}1 \text{ and var } s = \textstyle\sum_{j=1}^{\text{old } n} j)]^{\{s,i\}} \end{aligned}$$

The program variable i is local to the else-branch of the conditional statement: its effect is thus captured by existential quantification; by logical simplification, the variable is removed from the formula F_e for this branch:

$$\begin{aligned} F_e \quad \Leftrightarrow \quad & [\text{exists } v_i : \text{old } n < v_i \text{ and } (0 <= \text{old } n \Rightarrow v_i = \text{old } n{+}1 \text{ and var } s = \textstyle\sum_{j=1}^{\text{old } n} j)]^{\{s\}} \\ \Leftrightarrow \quad & [0 <= \text{old } n \Rightarrow \text{var } s = \textstyle\sum_{j=1}^{\text{old } n} j]^{\{s\}} \end{aligned}$$

Finally, we combine the effect of both branches of the conditional statement and derive the formula for the whole command:

$$\begin{aligned} F_c \quad \Leftrightarrow \quad & [\text{if old } n < 0 \text{ then var } s = -1 \text{ else } 0 <= \text{old } n \Rightarrow \text{var } s = \textstyle\sum_{j=1}^{\text{old } n} j]^{\{s\}} \\ \Leftrightarrow \quad & [\text{if old } n < 0 \text{ then var } s = -1 \text{ else var } s = \textstyle\sum_{j=1}^{\text{old } n} j]^{\{s\}} \end{aligned}$$

We note that F_c describes the "semantic essence" of the program fragment: if the program variable n is initially negative, the program variable s has finally the value -1, and else the sum of all values from 1 to n. The calculation of this semantic essence is the core of the calculus: it proceeds "inside-out" from atomic commands to composed commands and may apply logic to simplify formulas; its soundness depends on the generated proof obligation that the formula annotating the loop is indeed invariant invariant with respect to the execution of the loop body.

3 Pre- and Post-Conditions

From the judgment of the previous section, the auxiliary judgments presented in Figure 3 can be derived:

- $\text{PRE}(C,Q) = P$: Given a state formula Q and a command C, this judgment determines a state formula P, such that if P holds in the state in which C is executed, Q holds afterward. P is thus a pre-condition of C with respect to post-condition Q.
- $\text{POST}(C,P) = Q$: Given a state formula P and a command C, this judgment determines a state formula Q, such that if P holds in the state in which C is executed, Q holds afterwords. Q is thus a post-condition of C with respect to pre-condition P.

Formulas P and Q represent state conditions, i.e. are evaluated over single states; to simplify further substitutions, we assume that they refer to program variables in the form old I (and not var I); the generated pre-/post-conditions preserve this property.

In general, the judgments do not always determine the weakest pre- respectively strongest post-condition, because the reasoning about loops is based on externally provided invariants, which are not necessarily the strongest possible ones. Still they preserve the following constraint.

Theorem 2 (Soundness of Pre/Post-Conditions)**.** *For all commands C and state formulas P,Q, if the judgment* $\text{PRE}(C,Q) = P$ *or the judgment* $\text{PRE}(C,P) = Q$ *can be derived, we have:*

$$\forall s, s' \in \mathit{State} : [\![P]\!](s) \wedge [\![C]\!](s,s') \Rightarrow [\![Q]\!](s')$$

$$(pre)\ \frac{\begin{array}{l} C : [F]^{\{I_1,\ldots,I_n\}} \\ \texttt{var } \llcorner \text{ does not occur in } Q \\ J_1,\ldots,J_n \text{ do not occur as terms in } F,Q \end{array}}{\begin{array}{l} \textsc{pre}(C,Q) = \\ \quad \texttt{forall } J_1,\ldots,J_n\texttt{: } F[J_1/\texttt{var } I_1,\ldots,J_n/\texttt{var } I_n] \texttt{ =>} \\ \quad\quad Q[J_1/\texttt{old } I_1,\ldots,J_n/\texttt{old } I_n] \end{array}}$$

$$(pre)\ \frac{\begin{array}{l} C : [F]^{\{I_1,\ldots,I_n\}} \\ \texttt{var } \llcorner \text{ does not occur in } P \\ J_1,\ldots,J_n \text{ do not occur as terms in } F,P \end{array}}{\begin{array}{l} \textsc{post}(C,P) = \\ \quad \texttt{exists } J_1,\ldots,J_n\texttt{: } P[J_1/\texttt{old } I_1,\ldots,J_n/\texttt{old } I_n] \texttt{ and} \\ \quad\quad F[J_1/\texttt{old } I_1,\ldots,J_n/\texttt{old } I_n,\texttt{old } I_1/\texttt{var } I_1,\ldots,\texttt{old } I_n/\texttt{var } I_n] \end{array}}$$

Figure 3: Judgments $\textsc{pre}(C,Q) = P$ and $\textsc{post}(C,P) = Q$

As an example, for command $C = \texttt{x=x+1}$ with state relation $[\texttt{var } x = \texttt{old } x + 1]^{\{x\}}$, we get

$$\begin{array}{rcl} \textsc{pre}(C,Q) & = & \texttt{forall } v_x\texttt{: } v_x = \texttt{old } x + 1 \texttt{ =>} Q[v_x/\texttt{old } x] \\ & = & Q[\texttt{old } x + 1/\texttt{old } x] \end{array}$$

$$\textsc{post}(C,P) = \texttt{exists } v_x\texttt{: } P[v_x/\texttt{old } x] \texttt{ and old } x = v_x + 1$$

The computation of pre- and post-conditions has various applications; one of them is in the termination calculus presented in the following section.

4 Termination

The judgments presented up to now are only concerned with the partial correctness of a command: they constrain which transitions may take place but do not demand that any transition actually must take place, i.e. a command may also block respectively not terminate. To support also total correctness arguments, we are now going to capture the termination of a command C by an interpretation $\langle\!\langle C \rangle\!\rangle \in \mathit{StateCondition} := \mathbb{P}(\mathit{State})$ such that $\langle\!\langle C \rangle\!\rangle(s)$ (i.e. $s \in \langle\!\langle C \rangle\!\rangle$) holds if the execution of C in state s terminates. For our programming language, the definition of $\langle\!\langle C \rangle\!\rangle$ is as follows:

$$\begin{array}{l} \langle\!\langle \texttt{I=E} \rangle\!\rangle(s) :\Leftrightarrow \textsc{true} \\ \langle\!\langle \texttt{\{var } I\texttt{; } C\texttt{\}} \rangle\!\rangle(s) :\Leftrightarrow \forall s' \in \mathit{State} : (\forall J \in \mathit{Identifier} : J \neq I \Rightarrow s(J) = s'(J)) \Rightarrow \langle\!\langle C \rangle\!\rangle(s') \\ \langle\!\langle C_1\texttt{; } C_2 \rangle\!\rangle(s) :\Leftrightarrow \langle\!\langle C_1 \rangle\!\rangle(s) \wedge \forall s' \in \mathit{State} : [\![C]\!](s,s') \Rightarrow \langle\!\langle C_2 \rangle\!\rangle(s) \\ \langle\!\langle \texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 \rangle\!\rangle(s) :\Leftrightarrow \\ \quad \textsc{if } [\![E]\!](s) = \textsc{true } \textsc{then } \langle\!\langle C_1 \rangle\!\rangle(s) \textsc{ else } \langle\!\langle C_2 \rangle\!\rangle(s) \\ \langle\!\langle \texttt{while (}E\texttt{) } C^{F,T} \rangle\!\rangle(s) :\Leftrightarrow \\ \quad (\forall n \in \mathbb{N}, t \in \mathit{State}^* : \\ \quad\quad t(0) = s \wedge (\forall i \in \mathbb{N} : i < n-1 \Rightarrow [\![E]\!](s) = \textsc{true } \wedge [\![C]\!](t(i),t(i+1))) \Rightarrow \langle\!\langle C \rangle\!\rangle(t(n))) \wedge \\ \quad (\neg \exists t \in \mathit{State}^\omega : \\ \quad\quad t(0) = s \wedge \forall i \in \mathbb{N} : [\![E]\!](s) = \textsc{true } \wedge [\![C]\!](t(i),t(i+1))) \end{array}$$

Most cases are self-evident; the termination semantics for while loops demands that the execution of each loop iteration must terminate and that there are not infinitely many such iterations. The relationship between both semantic interpretations of commands is determined by the following constraint.

$$(strengthenT)\ \frac{C \downarrow F_1 \quad \models_\emptyset F_2 \Rightarrow F_1}{C \downarrow F_2}$$

$$(assignT)\ I{=}E \downarrow \mathsf{true}$$

$$(varT)\ \frac{C \downarrow F \quad J \text{ does not occur as a term in } F}{\texttt{\{var } I\texttt{;} C\texttt{\}} \downarrow \mathsf{forall}\ J\colon F[J/\mathsf{old}\ I]}$$

$$(seqT)\ \frac{C_1 \downarrow F_1 \quad C_2 \downarrow F_2 \quad \mathrm{PRE}(C_1, F_2) = F_3}{C_1\texttt{;}C_2 \downarrow F_1 \text{ and } F_3}$$

$$(ifT)\ \frac{E \simeq F \quad C_1 \downarrow F_1 \quad C_2 \downarrow F_2}{\texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 \downarrow \mathsf{if}\ F\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2}$$

$$(whileT)\ \frac{\begin{array}{l} E \simeq F_E \\ C : [F_C]^{I_1,\ldots,I_n} \\ \mathit{Invariant}(F, F_E, F_C)^{\{I_1,\ldots,I_n\}} \\ C \downarrow F_T \\ J_1,\ldots,J_n \text{ do not occur as terms in } F, F_E, F_C, F_T, T \\ \models_{\{I_1,\ldots,I_n\}} \\ \quad \mathsf{forall}\ J_1,\ldots,J_n\colon \\ \qquad F[J_1/\mathsf{old}\ I_1,\ldots,J_n/\mathsf{old}\ I_n, \mathsf{old}\ I_1/\mathsf{var}\ I_1,\ldots,\mathsf{old}\ I_n/\mathsf{var}\ I_n] \text{ and} \\ \qquad F_E \text{ and } F_C \Rightarrow \\ \qquad\quad F_T \text{ and } T < T[\mathsf{var}\ I_1/\mathsf{old}\ I_1,\ldots,\mathsf{var}\ I_n/\mathsf{old\_}I_n] \\ < \text{ represents a well-founded ordering} \end{array}}{\texttt{while (}E\texttt{) } C^{F,T} \downarrow F[\mathsf{old}\ I_1/\mathsf{var}\ I_1,\ldots,\mathsf{old}\ I_n/\mathsf{var}\ I_n]}$$

Figure 4: Judgment $C \downarrow F$

Theorem 3 (Termination Semantics)**.** *For every command C, we have*

$$\forall s \in \mathit{State} : \langle\!\langle C \rangle\!\rangle(s) \Rightarrow \exists s' \in \mathit{State} : [\![C]\!](s, s')$$

i.e. the state condition $\langle\!\langle C \rangle\!\rangle$ *represents a (possibly proper) subset of the domain of the state relation* $[\![C]\!]$.

In the following, we derive judgments of the form $C \downarrow F$ where F is represents a state condition (i.e. F is a formula evaluated over a single state) such that $[\![F]\!]$ describes those pre-states for which the execution of command C must terminate; F is thus called a termination condition of C. Figure 4 gives the rules for deriving termination conditions. Rule *strengthenT* shows that a termination condition can be strengthened by logical implication. Most other rules are pretty self-evident, with the exception of (*whileT*): this rule claims the termination of a loop in a state in which the loop invariant F holds provided that it can be proved that F is indeed an invariant and that in every iteration the execution of the loop

body terminates and the termination term decreases according to some well-founded ordering <. For this proof, the pre-state of the loop iteration may be assumed to satisfy the invariant and the loop condition.

The termination calculus meets the following soundness constraint.

Theorem 4 (Soundness of Termination Judgment)**.** *For all commands C and state formulas F, if the judgment $C \downarrow F$ can be derived, we have*

$$\forall s \in State : [\![F]\!](s) \Rightarrow \langle\!\langle C \rangle\!\rangle(s)$$

As an example, for the sample program C given in Figure 2, we can derive $C \downarrow \mathsf{true}$ where, in addition to the obligation of proving the correctness of the loop invariant, the following proof obligation is generated (after some logical simplification):

$\models_{s,i}$
forall v_s, v_i :
1 <= old i <= old n+1 and old s = $\sum_{j=1}^{\text{old } i\text{-}1} j$ and
old i <= old n and var s = old s + old i and var i = old i + 1 =>
old n - old i + 1 <= var n - var i + 1

From the last two lines and the fact that in frame $\{s,i\}$ we have axiom var n = old n, this condition is clearly valid.

5 Related Work

The idea of "programs as state relations" is not new; it has been variously advocated, e.g. in Lamport's "Temporal Logic of Actions" [7], Boute's "Calculational Semantics" [2], Hehner's "Practical Theory of Programming" [4], Hoare and Jifeng's "Unifying Theory of Programming" [6], Morgan's calculus for "Programming from Specifications" [9], and the "Refinement Calculus" of Back and von Wright [1] respectively Morris [10]. Still most program reasoning texts and tools are due to historical reasons dominated by the pre/post-condition view; this has been criticized in [11].

As a concrete example, the recent calculus of Mili et al [8] describes a loop by a reflexive and transitive state relation that is determined by the loop invariant; if the relation (that characterizes the relationship between two states that are separated by an arbitrary number of loop iterations) is also symmetric, (a lower bound approximation of) the state function implemented by the loop can be automatically derived. This relational framework differs from our in various respects:

- The interpretation of a relation $R \subseteq State \times State$ as a program specification is that, for every pre-state $s \in \mathrm{dom}(R)$, a program described by R *must* terminate with some post-state s' such that $\langle s, s' \rangle \in R$. This gives rise to the notion that a relation R_1 refines a relation R_2 if and only if $(R_1 \circ S) \cap (R_2 \circ S) \cap (R_1 \cup R_2) = R_2$ where $S := State \times State$ is the universal relation.

  In our framework, a program is characterized by a pair of a state relation $R \subseteq State \times State$ and a termination condition $T \subseteq State$ such that $T \subseteq \mathrm{dom}(R)$; here R describes only the space of possible transitions while T describes those pre-states for which a transition must take place. A specification $\langle R_1, T_1 \rangle = \langle [\![F_1]\!], [\![G_1]\!] \rangle$ refines a specification $\langle R_2, T_2 \rangle = \langle [\![F_1]\!], [\![G_1]\!] \rangle$ if and only if $R_1 \subseteq R_2$ and $T_2 \subseteq T1$, i.e. if $F_1 \Rightarrow F_2$ and $G_2 \Rightarrow G_1$ are valid formulas. Our framework thus provides a simple logical characterization of the semantics of a program and the notions of program specification and specification refinement.

- An invariant F is required to be a reflexive and transitive state relation, i.e. $[\![F]\!](s,s)$ must hold for every state s independently of the program context into which the loop is embedded, i.e., whether s is a possible pre-state of the loop or not.

  In our framework, F is need not be reflexive. The relation $[\![F]\!](s,s)$ must only hold for every state s that is a possible pre-state for the context in which the loop is embedded (the relation derived from the loop has a corresponding pre-state condition as the antecedent of an implication that has the invariant as its consequent). The proof of invariance also does not establish general transitivity of the relation but only that the invariance (with respect to the pre-state of the loop) is preserved by every iteration. While it would be also possible to strengthen the proof obligation for loops to check the reflexivity and transitivity of F, our framework thus allows to describe the invariance of a loop with respect to the actual context in which it is executed.

- Our framework uses a more general notion of state than described in this extended abstract. A state not only contains the values of program variables but also execution modes that arise from executing statements that interrupt the control-flow (loop continuations and breaks, function returns, exception throws). These extensions give a rise to an essentially more complex execution behavior, in particular with respect to loops, that can be nevertheless characterized in a relational framework and handled by the judgments introduced in the previous sections. The framework also considers global program variables, modular reasoning about methods specified by contracts, and (direct and indirect) recursive method invocations. Details can be found in [13].

Similar differences hold for most of the calculi cited above.

6 Conclusions

The "state relation" view by itself does not give rise to bigger automation, in particular the derivation of a loop's state relation still depends on an externally provided invariant. Nevertheless, we believe that switching from state conditions to state relations substantially alters one's mind set and is useful beyond the mere purpose of verification: with appropriate tool support (automatic derivation and simplification of formulas characterizing commands and program fragments) it is ultimately able to give humans *insight* into the programs they write.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, New York, 1998.

[2] Raymond T. Boute. Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus. *ACM Transactions on Programming Languages and Systems*, 28(4):747–793, July 2006.

[3] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[4] Eric C.R. Hehner. *A Practical Theory of Programming*. Springer, New York, 2006. `http://www.cs.utoronto.ca/~hehner/aPToP`.

[5] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[6] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, London, UK, 1998. `http://www.unifyingtheories.org`.

[7] Leslie Lamport. *Specifying Systems; The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. `http://research.microsoft.com/users/lamport/tla/book.html`.

[8] Ali Mili, Shir Aharon, Chaitanya Nadkarni, Lamia Labed Jilani, Asma Louhichi, and Olfa Mraihi. Reflexive transitive invariant relations: A basis for computing loop functions. *Journal of Symbolic Computation*, in press:DOI: 10.1016/j.jsc.2008.11.007, 2010.

[9] Carroll Morgan. *Programming from Specifications*. Prentice Hall, London, UK, 2nd edition, 1998. http://web2.comlab.ox.ac.uk/oucl/publications/books/PfS.

[10] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[11] David Lorge Parnas. Really Rethinking 'Formal Methods'. *Computer*, 43(1):28–34, 2010.

[12] The RISC ProgramExplorer, 2010. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, http://www.risc.jku.at/research/formal/software/ProgramExplorer.

[13] Wolfgang Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008. http://www.risc.jku.at/people/schreine/papers/ProgramCalculus2008.pdf.

[14] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 21(3):277–291, 2009.

Extended Web Services for Computational Origami

Asem Kasem, Tetsuo Ida
University of Tsukuba
Tsukuba 305-8573, Japan
kasem@score.cs.tsukuba.ac.jp ida@score.cs.tsukuba.ac.jp

1 Introduction

Origami, or paper folding, is an interdisciplinary subject spanning science, technology, art and education. It is a geometric tool that allows the construction of shapes by hands, and its capabilities go beyond what is constructible using classical tools of Euclidean geometry [14].

In computational origami, we study the computer-aided construction and visualization of origami, as well as its geometrical aspects. As part of our study, we developed a system called EOS (E-origami system) [6, 7], which has capabilities of symbolic and numeric constraint solving, visualization of origami constructions, and also assists the user in proving geometric theorems about constructed origamis using methods of computer algebra. Users of EOS can build an origami through a process of function calls, which fold, unfold, mark points, etc. Using a many-sorted first order logic language, users can formulate conclusion properties which they want to prove as a consequence of their construction [1]. EOS can then generate a system of polynomial equalities and/or inequalities that represent a theorem to be proved.

In the past few years, we have been working on exposing this process and experience of computational origami to interested researchers and origamists through the internet. Therefore, we developed a web interface for origami construction and reasoning, called WEBEOS [12, 8]. We have published WEBEOS development progress and examples in SCSS workshop in 2008 [13], where we showed a near-fully integrated environment which allows web origami construction, and provides access to symbolic computation software.

In this paper, we present a newer version of WEBEOS system, and discuss the various improvements over the pervious publications. We will present the outcome of our efforts to improve the web interface, and to make its usage as close as possible to the experience which we have with EOS running on a local computer. Users of WEBEOS can now interactively construct origami, formulate a geometric theorem, obtain polynomial output representing the theorem, and submit requests for symbolic computation services, all from a web browser. Description of WEBEOS system and aspects of accomplished improvements are explained in Section 2, and the main new features are presented in Section 3.

2 Background of WEBEOS

WEBEOS is the web frontend of EOS system for interactive origami construction and reasoning. It is developed based on Ajax technology to make rich and natural interface for constructing origami, and allows asynchronous access to various functions of EOS. It facilitates dynamic origami constructing using a graphical user interface running in a web browser, which is all what interested users need to have in order to use the system. We use Google Web Toolkit (GWT) [4] as a development tool, in addition to other tools and libraries, to obtain high level of cross browser compatibility. WEBEOS aims at reaching a wide community of users interested in origami, demonstrating interesting examples and problems, and sharing them publicly on the web. Further details about the older version of WEBEOS system can be found in [11, 13].

T. Jebelean, M. Mosbah, N. Popov (eds.): SCSS 2010, volume 1, issue: 1, pp. 144-154

EOS system, which is developed using *Mathematica* software, is the back-end which performs the actual computations for origami operations, visualization, and proving. Figure 1 shows how different users access EOS and WEBEOS systems. On a local machine having *Mathematica*, a user directly loads EOS packages and makes function calls through *Mathematica*'s notebook frontend. Web users, however, use their web browsers to send requests to WEBEOS, which in turn connects to EOS and responds to these requests.

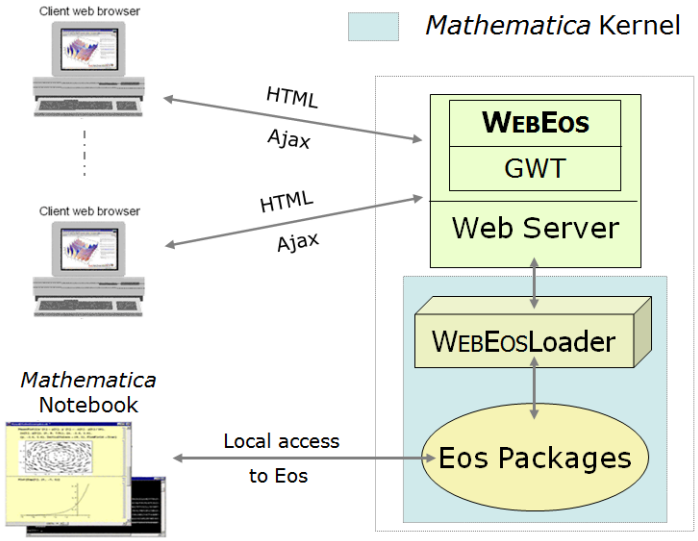


Figure 1: Access overview to EOS and WEBEOS systems

The implementation of EOS underwent various modifications, and EOS was upgraded to use the latest version of *Mathematica*, i.e. version 7.0. This upgrade resulted in various computation speedups compared with the older implementation that used *Mathematica 5.2*. Besides, the continued work on EOS led to more stability and improvements, introduction of new features, reduction in the number of generated polynomials, and a suggestion of variables order to compute Gröbner basis with, which we found very practical in proving origami theorems using our system [9]. Since these enhancements were only available in the latest version of EOS and *Mathematica*, the older version of WEBEOS could not exploit them because it was compatible only with *Mathematica 5.2*. This compatibility issue was mainly due to the use of *webMathematica2* [17] in WEBEOS, a product for empowering websites with *Mathematica* computations. *webMathematica2* was only compatible with *Mathematica 5.2* at the time of development, until version 3 was released recently. Another reason was the numerous changes that happened in EOS itself.

Therefore, we re-designed and developed a *Mathematica* package called *WebEosLoader*, to link WEBEOS with the latest version of EOS system. We also abandoned *webMathematica2* and implemented a way, using *WebEosLoader*, to substitute the features of *webMathematica2* that we used before. As a result, WEBEOS now benefits from the improvements of latest version of EOS, and has access to its larger set of functionalities, compared with the older version.

WebEosLoader Layer

In addition to linking WEBEOS with *Mathematica 7*, *WebEosLoader* layer allows and manages concurrent users access to WEBEOS. When a user performs an origami fold, or any other operation, a call to

WEBEOS server side is made, the operation is executed, and proper output is generated. This happens by accessing an instance of *Mathematica*'s kernel on the server, loaded with EOS packages. Since multiple users might access the system at the same time, each operation in the kernel must be carried out based on user's specific origami data structures and system options. This scenario is different from the normal case when EOS is running on a local machine by one user only. This sensitive data is considered as session data inside *Mathematica*, in addition to the usual session information stored in the web server.

To achieve this separation of data among users, we consider the following two possible approaches to preserve session data from being mixed up among different users:

1. Using *Mathematica*'s concept of contexts to assign session data of a user to his/her own context. In *Mathematica*, any symbol or variable belongs to a certain context, and different symbols having the same name can exist if they belong to different contexts. We use this concept to make each variable of session data exist in multiple contexts, based on the user to whom this data belongs. When a user acquires the kernel[1] to execute an origami operation, we activate the corresponding context of data on which the code of EOS will operate. When it is time for the execution of another user, his/her own context will be activated, and thus the new operation will run on the correct session data. This approach was used in the older version of WEBEOS, and it gave us a safe and fast solution. It was also transparent and did not require serious modifications in the code of EOS. The downside of this solution is the need to keep all the contexts of currently logged-in web users available in the memory space of *Mathematica*. Since the origami session data is relatively large, if many users access the system at the same time, *Mathematica* could run out of memory.

2. Saving session data updates into a file, and loading it when necessary before executing EOS code on behalf of a user. When a user wants to execute an origami operation, the data is loaded from his/her own saved file, and other users' data is cleared from the memory. In this way, *Mathematica*'s memory space contains the data of only one user at a single moment, compared with big memory usage in the first solution. The downside of this approach is the time delay for saving and loading session data from the hard disk, which is necessary before any execution of origami operation. To reduce this burden, the layer avoids unnecessary saving and loading of session data when it finds that the same user is running consecutive origami operations. We decided to use this approach in the new version of WEBEOS.

3 Features of WEBEOS

WEBEOS is published at the following URL: `http://webeos.score.cs.tsukuba.ac.jp` .

In addition to the enhancements mentioned in the previous section, we present the main features available in the current version.

3.1 Folding Operations

Mainly, WEBEOS users fold origami using 7 basic folding operations, known as Huzita's axioms [5]. These axioms are declarative statements involving points and lines, and define the lines along which we can make a fold. For origamists, finding these fold lines is possible using hands only, without additional means or tools. WEBEOS interface shows an explanation of each axiom and asks for required parameters. In addition to the folding operations, there is an auxiliary set of operations which users can use while folding origami, such as unfolding, turning origami over, creating and duplicating points after folding,

[1] Once acquired, the kernel is used until the end of computation, and other users' execution requests wait until the current operation finishes, and then compete to acquire the kernel.

etc. Although this feature is not new in WEBEOS, we present it for the clarity of the rest of the paper, and because the implementation behind the scene is different from the older version, and went through revisions and improvements. Besides, the graphical interface of the whole system has been updated, and we use new graphical components such as animated panels and color pallets.

Figure 2 shows a snapshot of WEBEOS page while trisecting an arbitrary angle using origami folds. We show later how proving the correctness of this construction can be performed automatically using WEBEOS.

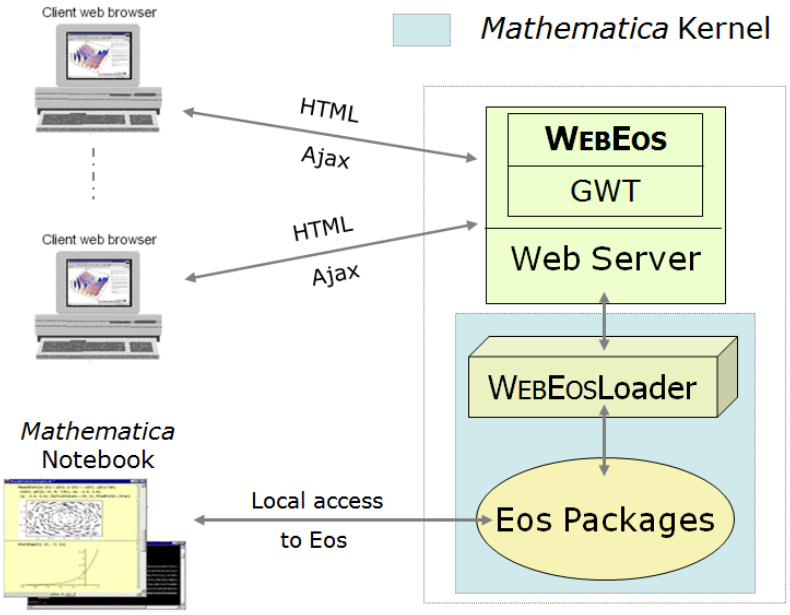


Figure 2: Folding origami to trisect angle $\measuredangle EAB$, following a method attributed to Hisashi Abe

3.2 3D Visualization

In WEBEOS, users can now modify some of the viewing options of origami output. For example, they can get a 3D image view of origami by changing the camera location to have a 3D perspective, compared to the previous 2D perspective where the camera is looking down vertically on origami plane. In our system, origami is represented as a set of faces. When folded, faces are split by the fold line, and new faces are generated. We distinguish each face by assigning a unique ID to it, and it is now possible to choose an option that shows or hides these faces IDs. Figure 3.(a) shows an example of how the output looks like.

If the browser is Java enabled, users can even visualize origami completely as a Java3D object, where they can use the mouse to scale, translate, and rotate origami. When folded origami is shown in 3D mode, whether as 3D image or as Java3D object, it makes sense to allow a non-zero gap distance between folded faces to illustrate their overlapping relation. Figure 3.(b) demonstrates this viewing option.

The visualization of Java3D origami is realized using *JavaView*[15] library. *JavaView* is a 3D geometry viewer and mathematical visualization software. It displays interactive 3D geometries and enables a smooth integration with commercial software like *Mathematica* and *Maple*. We use its light version *JavaView-Lite*, which is optimized for fast download and contains the viewer module only. *JavaView* understands the graphics directives of origami output generated by *Mathematica*, which made it a con-

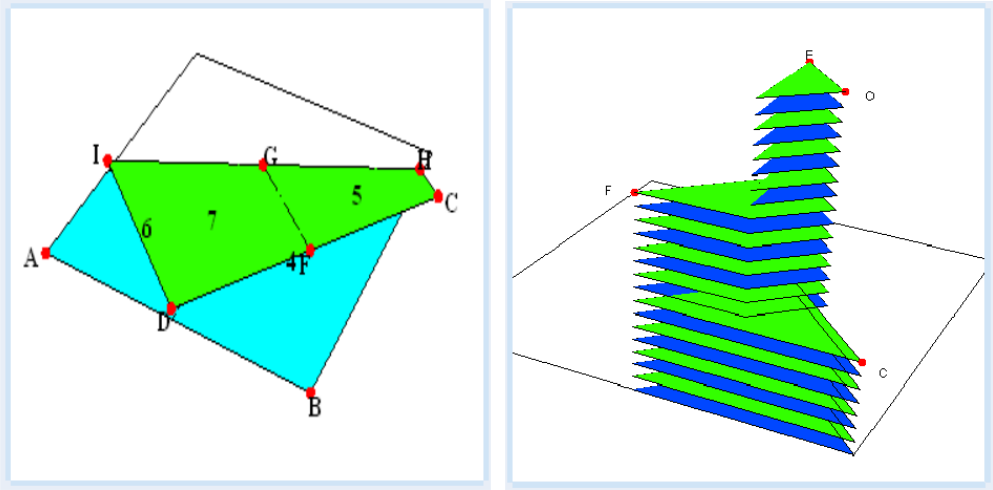


Figure 3: (a) 3D image view of origami showing faces IDs. (b) Java3D view with gap between faces

venient choice to use in WEBEOS. We had to make small tweaks to make it work with the output of *Mathematica* version 7 though.

3.3 Formulating Theorems

After a user constructs an origami using the operations provided in WEBEOS, he/she might be interested in proving some properties of the construction. For instance, at the beginning of this section, we showed in Figure 2 an angle trisection example by following folding steps introduced by Abe. How can we automatically prove that these folding steps construct the trisectors that correctly trisect the angle?

Stated formally, users might be interested to decide whether the formula $\mathscr{L} : \mathscr{K} \Rightarrow \mathscr{C}$ holds, where $\mathscr{K}$ represents the geometrical constraints accumulated during the construction, and $\mathscr{C}$ is a desired target property that they hope to prove. In the trisection example, $\mathscr{C}$ would be a property stating that the obtained angle is one third of the given angle.

WEBEOS allows users to formulate geometrical properties described in a first-order logic language [1], consisting of a set of predicate and function symbols, with sets of points and lines as parameters. We express geometric properties by writing formulae of the language. WEBEOS provides a graphical interface to build a formula tree of this language. Figure 4 shows how the interface is used to formulate the conclusion that we try to prove in the angle trisection example.

The conclusion that we want to prove in this example is that $\measuredangle EAK$ is equal to $\measuredangle KAJ$, and that $\measuredangle KAJ$ is equal to $\measuredangle JAB$ (refer to Figure 2). We build a formula tree representing this property by using the pallets of operators and available predicates and functions as shown in Figure4. We construct the formula in the way of in-order tree walk. We first create an `And` node as the root of the formula tree, and it automatically comes up with 2 empty child nodes to represent the operands of the `And` operator. In the place of the first empty node, we create an equal node, `==`, to indicate the equality of two terms. It also generates 2 new empty nodes that should represent the two angles we are trying to prove equal. To refer to the angles, we use `Spread` function that measures the separation of the lines defining an angle in terms of rational trigonometry [19]. We continue in the same way to fill all the empty nodes of the formula tree.

There are several predicates and functions made available in building the formula tree. Predicates are: `OnLine`, `OnSegment`, and `Collinear`. Functions are: `Distance`, `Spread`, and `ToTangent`. Clicking

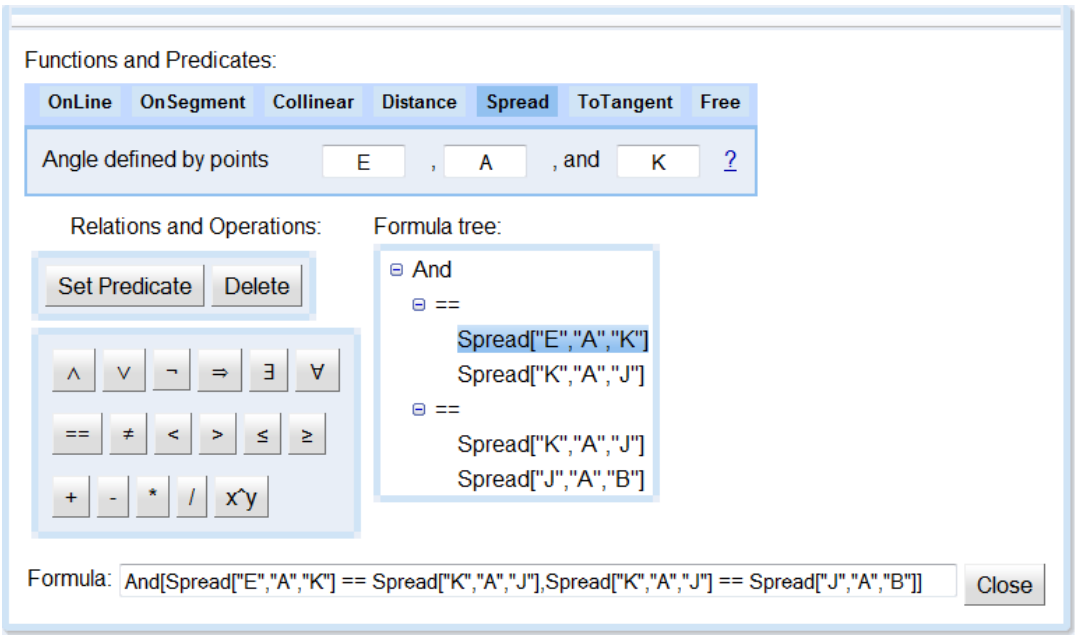


Figure 4: Formulating a conclusion using WEBEOS interface

on any of these functions or predicates shows a list of their required parameters. There is also a '?' link next to the parameters to show an explanation and related information, like the ones shown in Figure 5.

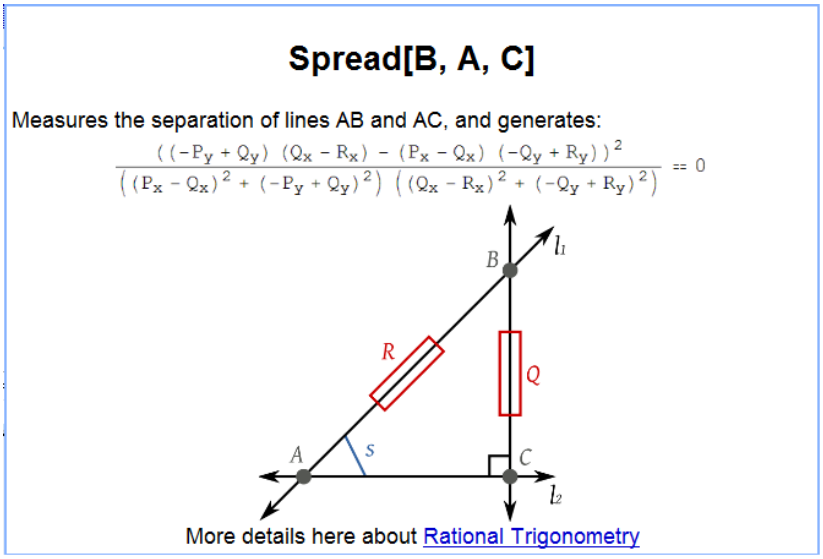


Figure 5: Explanation of `Spread` function

In some of origami theorem proving examples, we can not prove a theorem because of a degenerate case in the theorem. For the automatic proof to work, we need to impose certain properties, which we call the additional premisses $\mathscr{X}$, to filter out such cases. These premisses are added to the accumulated construction properties. An example which we investigated thoroughly is the origami construction and

automated proof of Morley's triangle [9], where we show why such extra premisses are necessary to prove the theorem in its general form. The formula representing the theorem becomes:

$$\mathscr{L}: \ \mathscr{K} \wedge \mathscr{X} \Rightarrow \mathscr{C} \tag{1}$$

In such cases, users of WEBEOS can use the formula tree interface to also formulate the additional premisses $\mathscr{X}$.

3.4 Hand-Written Mathematical Symbols Recognition

While building a formula tree, sometimes we need to refer to symbols names, numeric values, or to type a predicate or function that is not available. To cover such possibility, WEBEOS allows the creation of a free node in the formula tree, that is filled by user's textual input. To use a node whose value is a number, say 4, or a variable, say X, the user creates a free node and types the desired value.

However, we realized that it's important sometimes to allow the input of various mathematical symbols that are unavailable on user's keyboard, such as the symbols α or $\in$. One solution to allow such input was to provide a pallet or a list of the most common ones, and the user selects from them using mouse clicks. However, this limits the number of symbols which we can show and accept. Therefore, we decided to implement an interface to recognize handwritten symbols. We allow users to draw symbols using a mouse or a pen-input device, and then we recognize these symbols and insert them in the textual context of the free node.

To achieve this, we used an implementation of a distance-based classification algorithm for recognizing handwritten symbols with orthogonal series, introduced by Oleg Golubitsky and Stephen Watt [2, 3]. A user draws a multi-stroke symbol on a WEBEOS canvas in the web browser. We compute coefficients of the truncated Legendre-Sobolev expansions of the coordinates x and y, which give us a representation of the drawn curves. After that, the coefficients are sent to the server side, where they get compared to a database of coefficients of handwritten symbols. An ordered list of the closest recognized symbols is returned back to the client side, where the user can select the desired one. Figure 6 shows a snapshot of recognizing the symbol α and inserting it to get the expression $\alpha \in Alg$.

Figure 6: Recognizing mathematical symbol α

3.5 Polynomials and Variables generation

After formulating the origami theorem, WEBEOS generates a set of polynomials that can be used to prove the theorem.

Considering the formula in (1), we let $P_{\mathscr{L}}$ be the set of polynomials representing the formula $\mathscr{L}$, and let Ideal(M) be the ideal generated by the set of polynomials M. Formula $\mathscr{L}$ is true if $1 \in \text{Ideal}(P_{\neg\mathscr{L}})$. The ideal membership problem $1 \in \text{Ideal}(M)$ can be solved constructively by computing the Gröbner bases of M. Namely, the formula $\mathscr{L}$ is true iff the reduced Gröbner basis of $P_{\neg\mathscr{L}}$ is $\{1\}$, and WEBEOS generates the set of polynomials $P_{\neg\mathscr{L}}$ for the user.

The computation of a Gröbner basis is well known to be significantly affected by the used order of the variables of polynomials. A practical suggested ordering is discussed in [9], and WEBEOS uses it to generate the sets of polynomials and variables shown in Figure 7. WEBEOS actually generates the standard *MathML* [16] representation of the set of polynomials, which is shown in Figure 8, and the graphical output of the polynomials is rendered using *WebEQ* [10] product, a Java Applets library to visualize and input mathematical expressions on the web.

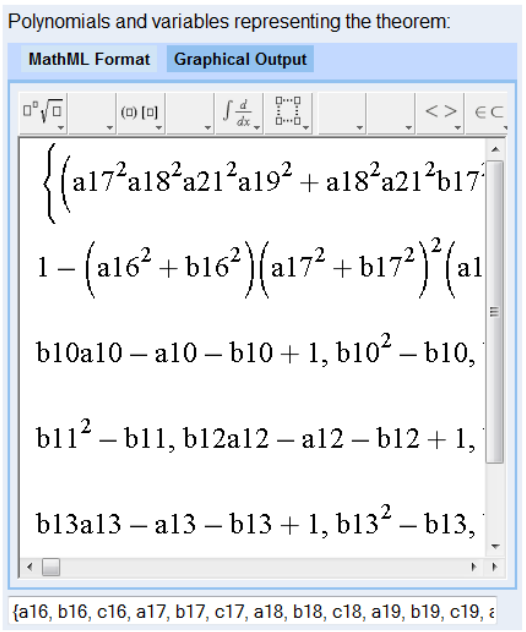


Figure 7: Sets of generated polynomials and ordered variables

3.6 Symbolic Computation Services

The remaining step to prove an origami theorem is to compute Gröbner basis of the generated polynomials, which requires access to symbolic computation software that can perform such computation. We have showed how to provide web access to *Mathematica*'s implementation of Gröbner bases computation through another web system called SCORUM [13]. After the generation of polynomials and variables, WEBEOS users can click on the button named "Submit to SCORUM" to send the generated data to SCORUM and schedule a request to compute Gröbner basis. SCORUM is still under development and not yet made accessible from outside our local network. Therefore, in the current version of WEBEOS, we don't redirect users to SCORUM and just show a message explaining that.

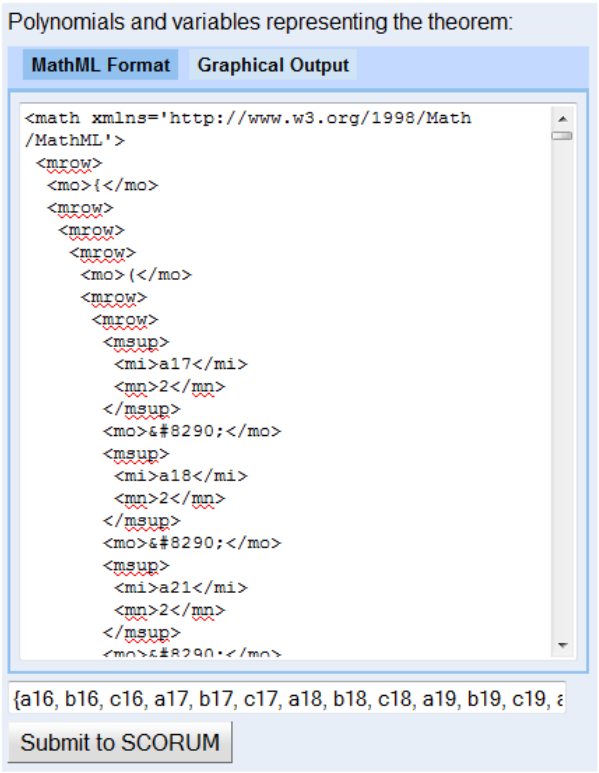


Figure 8: *MathML* representation of the polynomials

3.7 Origami Examples

We also enriched WEBEOS available examples which a user can preview and dynamically learn how to construct and prove. We have completed the theorem formulation part of previously available examples, and added new examples such as angle trisection, regular pentagon, and a couple of artistic origami pieces.

4 Conclusion and Future Work

We have presented the latest version of WEBEOS, a web system for interactive origami construction and theorem proving. We have discussed the various improvements compared with earlier versions, along with a set of new features. WEBEOS enables a web experience of origami theorem proving that is analogous to the experience one would have using the origami system EOS, which runs on a local machine, and requires the installation of expensive software. Yet, in WEBEOS, construction and theorem formulation is more interactive and user-friendly.

For future work, several things can be done to improve the system. For example, we can extend the system to use multiple *Mathematica* kernels, possibly running on distributed servers, and manage load balancing to allow efficient handling of computation requests on times of intensive access. Another possibility to improve efficiency lies in the *WebEosLoader* layer. We can merge the two solutions presented in Section 2 to separate users contexts in the memory space of *Mathematica*, and start using files to store session's data only when the memory is overloaded.

We also need to finish implementing the link to SCORUM system, and provide access to more symbolic computation software systems that compute Gröbner basis, especially free ones such as *Sage* [18].

Acknowledgment

The first author has carried out an internship program in ORCCA, Ontario Research Centre for Computer Algebra, in the University of Western Ontario, Canada. As part of the research activities in this internship, a Java implementation of an algorithm for recognizing hand-written mathematical characters was implemented. The implementation was a joint work with Oleg Golubitsky, and supervised by Stephen Watt. We thank both of them for giving us the permission to reuse parts of the implementation in our system.

This research is supported by the JSPS Grants-in-Aid for Exploratory Research No. 22650001 and Scientific Research (B) No. 20300001.

References

[1] Fadoua Ghourabi, Tetsuo Ida, Hidekazu Takahashi, Mircea Marin, and Asem Kasem. Logical and algebraic view of huzita's origami axioms with applications to computational origami. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, pages 767–772. ACM Press, New York, 2007.

[2] Oleg Golubitsky and Stephen M. Watt. Distance-based classification of handwritten symbols. *International Journal on Document Analysis and Recognition*, 2009.

[3] Oleg Golubitsky and Stephen M. Watt. Online recognition of multi-stroke symbols with orthogonal series. In *ICDAR '09: Proceedings of the 2009 10th International Conference on Document Analysis and Recognition*, pages 1265–1269, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Google. Google Web Toolkit.
http://code.google.com/webtoolkit/.

[5] H. Huzita. Axiomatic Development of Origami Geometry. In H. Huzita ed., editor, *Proceedings of the First International Meeting of Origami Science and Technology*, pages 143–158, 1989.

[6] T. Ida, D. Ţepeneu, B. Buchberger, and J. Robu. Proving and Constraint Solving in Computational Origami. In *Proceedings of the 7th International Symposium on Artificial Intelligence and Symbolic Computation (AISC 2004)*, volume 3249 of *LNAI*, pages 132–142, 2004.

[7] T. Ida, H. Takahashi, M. Marin, A. Kasem, and F. Ghourabi. Computational Origami System Eos. In *Proceedings of 4th International Conference on Origami, Science, Mathematics and Education*, page 69. 4OSME, 2006. Caltech, Pasadena CA.

[8] Tetsuo Ida and Asem Kasem. WEBEOS: A system for origami construction and proving on the web. In George E.LASKER and Jochen PFALZGRAF, editors, *ADVANCES in MULTIAGENT SYSTEMS, ROBOTICS and CYBERNETICS: Theory and Practice*, volume II. The INTERNATIONAL INSTITUTE for ADVANCED STUDIES in SYSTEMS RESEARCH and CYBERNETICS (Tecumseh, Canada), 2008. ISBN 978-1-897233-61-0.

[9] Tetsuo Ida, Asem Kasem, Fadoua Ghourabi, and Hidekazu Takahashi. Morley's theorem revisited: Origami construction and automated proof. *Journal of Symbolic computation*, 2010. (to appear).

[10] Design Science Inc. WebEQ developers suite.
http://www.dessci.com/en/products/webeq.

[11] A. Kasem, H. Takahashi, M. Marin, and T. Ida. weborigami2 : A system for origami construction and proving using web 2.0 technologies. In *Proceedings of the Annual Symposium of Japan Society for Software Science and Technology*, Nara, Japan, 2007. JSSST.

[12] Asem Kasem and Tetsuo Ida. Computational origami environment on the web. *Frontiers of Computer Science in China*, 2(1), 2008. To be published.

[13] Asem Kasem and Tetsuo Ida. Experiences with web environment origamium: Examples and applications. In *SCSS-2008*, RISC Technical Report Series No. 08-08, pages 109–122, Hagenberg, Austria, July 12-13 2008.

[14] R. J. Lang. Origami and geometric constructions.
http://www.langorigami.com/science/hha/origami_constructions.pdf.

[15] Konrad Polthier. The JavaView Project. `http://www.javaview.de`. Copyright 1999-2006.

[16] W3C Recommendation.
Mathematical Markup Language (MathML) Version 2.0 (Second Edition).
http://www.w3.org/TR/2003/REC-MathML2-20031021/, October 2003.

[17] Wolfram Research. webMathematica 2.
http://www.wolfram.com/products/webmathematica.

[18] W. A. Stein et al. *Sage Mathematics Software (Version 4.3.5)*. The Sage Development Team, 2010. `http://www.sagemath.org`.

[19] N. J. Wildberger. *Divine Proportions*. Wild Egg Pty Ltd, 2005.