Preface

This volume contains the proceedings of PDPAR'03, the first workshop on *Pragmatics of Decision Procedures in Automated Reasoning*, held July 29, 2003, in Miami, Florida (USA). The main goal of PDPAR'03 was to bring together researchers interested in the pragmatic aspects of decision procedures in automated reasoning, giving them a forum for presenting and discussing implementation and evaluation techniques. Another goal of the workshop was to provide the occasion to discuss the future of SMT-LIB, a research initiative aimed at establishing a common standard for the specification of benchmarks and background theories for satisfiability modulo theories, and at creating a repository of such benchmarks. (See `http://combination.cs.uiowa.edu/smtlib/` for more information.) To this end, the workshop hosted a panel discussion on the SMT-LIB format.

These proceedings contain seven regular submission papers, each of which was reviewed by at least two members of the Program Committee. The papers present and discuss: decision procedures for various extensions of a decidable fragment of set theory; a programming language for implementing (combination of) decision procedures; proof production for a combination of SAT solving and first-order decision procedures; abstraction techniques for efficiently handling fragments of the theory of reals; a combination of BDDs and first-order decision procedures; an application of a combination of SAT solving and decision procedures for fragments of arithmetics to the verification of hybrid systems; and a description of a collection of benchmarks for first-order decision procedures extracted from the verification of microprocessors.

We welcomed one invited lecture by Mark Stickel on "Specialized Reasoning in SNARK". A short abstract of Stickel's talk is included in this volume. Finally, also included in this volume is a white paper by these organisers on a common format for SMT-LIB. The paper was used as the starting point of a panel discussion among the following panelists: Aaron Stump, Clark Barrett, Greg Nelson, Roberto Sebastiani, Geoff Sutcliffe, and ourselves.

We would like to thank the members of the program committee and the two external referees for their care and time in reviewing the submitted papers. We also thank the members of the organising committee of CADE-19 (to which this workshop is affiliated) for their help in the practical organisation of the workshop; and the institutions that supported the workshop: CADE Inc. and INRIA.

Silvio Ranise and Cesare Tinelli
PDPAR co-chairs
Nancy and Iowa City, August 2003

Organization

PDPAR'03 is a workshop affiliated to CADE 19 (Miami, USA).

Program Commitee

Alessandro Armando (University of Genova, Italy)	
Clark Barrett (New York University, USA)	
Sergey Berezin (Stanford University, Stanford, California, USA)	
Alessandro Cimatti (IRST-ITC, Trento, Italy)	
Deepak Kapur (University of New Mexico, New Mexico, USA)	
Predrag Janicic (University of Belgrade, Belgrade, Yugoslavia)	
Greg Nelson (Compaq, Palo Alto, California, USA)	
Silvio Ranise (LORIA & INRIA-Lorraine, France)	*Program Co-chair*
Christophe Ringeissen (LORIA & INRIA-Lorraine, France)	
Harald Ruess (SRI, USA)	
Roberto Sebastiani (University of Trento, Trento, Italy)	
Ofer Strichman (Carnegie-Mellon University, USA)	
Aaron Stump (Washington University, USA)	
Cesare Tinelli (University of Iowa, USA)	*Program Co-chair*
Ashish Tiwari (SRI, USA)	
Miroslav Velev (Georgia Institute of Technology, USA)	

Additional Referees

Joel Ouaknine
Michael Theobald

Sponsoring Institutions

CADE Inc.
INRIA (Institut National de Recherche en Informatique et en Automatique)

Table of Contents

Invited Contribution

Specialized Reasoning in SNARK 1
Mark Stickel

Regular Contributions

Various commonly occurring decidable extensions of multi-level syllogistic 2
D. Cantone, A. Formisano, E. G. Omodeo, J. T. Schwartz

Rogue Decision Procedures 15
Aaron Stump, Arumugam Deivanayagam, Spencer Kathol, Dylan Lingelbach, and Daniel Schobel

A Proof-Producing Boolean Search Engine 25
Clark Barrett, Sergey Berezin

Abstraction Based Theorem Proving: An example from the theory of Reals 40
Ashish Tiwari

Simplifying OBDDs in Decidable Theories —Extended Abstract— 53
Alessandro Armando

Verifying Industrial Hybrid Systems with MathSAT 62
Gilles Audemard, Marco Bozzano, Alessandro Cimatti, Roberto Sebastiani

Collection of EUFM Benchmark Suites from Formal Verification of Microprocessors 76
M. Velev

SMT-LIB Panel

The SMT-LIB Format: An Initial Proposal 94
Silvio Ranise, Cesare Tinelli

Author Index 112

Specialized Reasoning in SNARK

Mark Stickel

Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
stickel@ai.sri.com

Abstract. SNARK is an automated theorem-proving program whose principal inference rules are resolution and paramodulation. Its basic approach to deduction is similar to Otter's but SNARK is distinctive for its diverse capabilities for extending the basic inference rules by specialized reasoning, including support for special unification, sorts, procedural attachment, and constraint reasoning. This talk will discuss approaches and problems of incorporating specialized reasoning into systems like SNARK.

Various commonly occurring decidable extensions of multi-level syllogistic⋆

D. Cantone[1], A. Formisano[2], E. G. Omodeo[3], J. T. Schwartz[4]

[1] Università di Catania, Dipartimento di Matematica e Informatica; email: `cantone@dmi.unict.it`
[2] Università di Perugia, Dip. di Matematica e Informatica; email: `formis@dipmat.unipg.it`
[3] Università di L'Aquila, Dipartimento di Informatica; email: `omodeo@di.univaq.it`
[4] New York University, Department of Computer Science, Courant Institute of Mathematical Sciences; email: `schwartz@cs.nyu.edu`

Abstract. The paper focuses on extending existing decision procedures for set theory and related theories commonly used in mathematics to handle such notions as monotonicity, ordering, inverse functions, etc. The proposed technique is based on a syntactic translation of formulas with such special function and predicate symbols into the pure set-theoretic logic decidable by the existing decision procedures. The result can be quite useful for tool developers who aim at common mathematical reasoning.
Key words: Satisfiability decision procedures, syllogistic, proof verification, set theory.

Introduction

Engaging formal proofs, such as those founding the field of mathematical analysis, very often rely on routine forms of set-theoretical reasoning which a human expert exploits almost unconsciously and a computerized proof-checker must encompass as basic inferential services. This paper focuses on situations when such reasoning services are implemented as decision algorithms for fragments of set theory. We will start with a specific satisfiability test which, despite being rather limited in the constructs it can deal with, proved to be very useful in practice (whereas decision algorithms for more expressive sub-languages of set theory tend to be utterly unmanageable due to their high complexity). We will address the issue: how can we extend the realm of applicability of this decision algorithm without substantial reworking of its internals, but rather via systematic preprocessing techniques? The method which we will propose as an answer is already at work in the proof-verification system described in [7, 13].

The ideas reported in this paper have connections with the work of Armando *et al.* (cf. [1]).

⋆ Work partially supported by MURST/MIUR Grant prot. 2001017741 under project *"Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps."*

1 Multi-level syllogistic

MLSS (*multilevel syllogistic with singleton*) is the unquantified language of set theory consisting of a denumerable infinity $u, v, w, x, y, z, \ldots$ of set variables, the 'null set' constant $\emptyset$, the set operators $\cdot \cap \cdot$, $\cdot \setminus \cdot$, $\cdot \cup \cdot$, $\{\cdot, \ldots, \cdot\}$, the set predicates $\cdot \in \cdot$, $\cdot = \cdot$, $\cdot \subseteq \cdot$, and propositional connectives.

The semantics of **MLSS** is based upon the von Neumann cumulative hierarchy $\mathcal{V}$ defined as follows (where $\mathcal{O}rd$ and $\mathcal{P}(X)$ designate the class of all ordinals and the power-set of X):

$$
\begin{aligned}
\mathcal{V}_0 &=_{\text{Def}} \emptyset\,; \\
\mathcal{V}_{\alpha+1} &=_{\text{Def}} \mathcal{P}(\mathcal{V}_\alpha)\,; \text{for each ordinal } \alpha\,; \\
\mathcal{V}_\lambda &=_{\text{Def}} \textstyle\bigcup_{\mu<\lambda} \mathcal{V}_\mu\,, \text{for each limit ordinal } \lambda\,; \\
\mathcal{V} &=_{\text{Def}} \textstyle\bigcup_{\alpha\in\mathcal{O}rd} \mathcal{V}_\alpha\,.
\end{aligned}
$$

An *assignment* $\mathcal{M}$ over a collection of all variables V is any map from V into $\mathcal{V}$. Let φ be an **MLSS**-formula over a collection V of variables, and let $\mathcal{M}$ be an assignment over V. By $\varphi^{\mathcal{M}}$ we denote the truth-value of φ obtained by interpreting each variable $x \in V$ with the set $x^{\mathcal{M}}$ and the set operators and propositional connectives according to their standard meanings. Such a φ is said to be *satisfiable* if it has a *model*, namely, an assignment $\mathcal{M}$ making $\varphi^{\mathcal{M}}$ true.

The satisfiability problem for **MLSS** is the problem of determining whether or not any given **MLSS**-formula φ is satisfiable. It was first solved in [11]. Subsequently, it was shown that the satisfiability problem for conjunctions of 'flat' **MLSS**-literals of the forms

$$
x = y\,, \quad x \neq y\,, \quad x \in y\,, \quad x \notin y\,, \quad x = y \cup z\,, \quad x = y \setminus z\,, \quad x = \{y\}\,, \qquad (1)
$$

to be called *normalized* **MLSS**-conjunctions, is *NP*-complete (cf. [5]); more recently, its decision procedure was optimized in [6, 9] by means of semantic tableaux. For the reader's convenience, we sketch a decision procedure for normalized **MLSS**-conjunctions based on semantic tableaux.

Table 1 lists the rules of a tableau calculus for **MLSS**. Notice that the rules (2), (5), and (9) cause branch splits.

Next we define **MLSS**-tableaux (for general notions on tableaux, the reader is referred to [12]).

Let $\mathcal{S}$ be a finite collection of flat **MLSS**-literals of the form (1). An INITIAL **MLSS**-TABLEAU for $\mathcal{S}$ is a one-branch tree whose nodes are labeled by the literals in $\mathcal{S}$.

An **MLSS**-TABLEAU for $\mathcal{S}$ is a tableau labeled with **MLSS**-literals which can be constructed from the initial tableau for $\mathcal{S}$ by a finite number of applications of the rules (1)–(11) of Table 1.

Let $\mathcal{T}$ be an **MLSS**-tableau for $\mathcal{S}$. A branch ϑ of $\mathcal{T}$ is said to be

- STRICT, if no rule has been applied more than once on ϑ to the same literal occurrences;
- SATURATED, if each of the tableaux rules (1)–(11) has been applied at least once on each instance of its premises on ϑ;

$$\frac{\begin{array}{c} x = y_1 \cup y_2 \\ z \in y_i \end{array}}{z \in x}\ (1) \qquad \frac{\begin{array}{c} x = y_1 \cup y_2 \\ z \in x \end{array}}{z \in y_1 \mid z \in y_2}\ (2) \qquad \frac{\begin{array}{c} x = y_1 \setminus y_2 \\ z \in x \end{array}}{\begin{array}{c} z \in y_1 \\ z \notin y_2 \end{array}}\ (3) \qquad \frac{\begin{array}{c} x = y_1 \setminus y_2 \\ z \in y_1 \\ z \notin y_2 \end{array}}{z \in x}\ (4)$$

$$\frac{\begin{array}{c} x = y_1 \setminus y_2 \\ z \in y_1 \end{array}}{z \in y_2 \mid z \notin y_2}\ (5) \qquad \frac{x = \{y\}}{y \in x}\ (6) \qquad \frac{\begin{array}{c} x = \{y\} \\ z \in x \end{array}}{z = y}\ (7) \qquad \frac{\begin{array}{c} y_1 \in x \\ y_2 \notin x \end{array}}{y_1 \neq y_2}\ (8)$$

$$\frac{x \neq y}{\begin{array}{c|c} w \in x & w \notin x \\ w \notin y & w \in y \end{array}}\ (9)^a \qquad \frac{\begin{array}{c} x = y \\ \phi \end{array}}{\phi^x_y}\ (10)^b \qquad \frac{\begin{array}{c} y = x \\ \phi \end{array}}{\phi^x_y}\ (11)^b$$

[a] w must be a new variable not occurring on the branch to which the rule is applied.
[b] By ϕ^x_y we denote the formula resulting by substituting in ϕ each occurrence of x with y.

Table 1. Tableaux rules for **MLSS**

- CLOSED, if either ϑ contains a set of literals of the form $x \in x_1$, $x_1 \in x_2, \ldots,$ $x_{n-1} \in x_n$, $x_n \in x$, for some variables $x, x_1, \ldots, x_n$ with $n \geqslant 0$, or it contains a pair of complementary literals $X, \neg X$;
- OPEN, if it is not closed;
- SATISFIABLE, if there exists a set model for the literals occurring on ϑ.

A tableau $\mathcal{T}$ is said to be

- STRICT, or SATURATED, or CLOSED, if such are all of its branches;
- SATISFIABLE, or OPEN, if such is at least one of its branches.

Notice that according to the above definition, any closed branch, and therefore any closed **MLSS**-tableau, is unsatisfiable.

The system of rules (1)–(11) is plainly *sound*, namely any **MLSS**-tableau for a satisfiable normalized **MLSS**-conjunction must be satisfiable, and therefore must be open.

In addition, the tableau calculus in Table 1 is *complete*, namely any unsatisfiable normalized **MLSS**-conjunction has a closed **MLSS**-tableau. What is important for our decidability purposes is that completeness is not disrupted even when the tableau rules are subject to the following restrictions, which guarantee termination:

R1. all applications of tableau rules are strict;
R2. rule (9) is applied only to literals of the form $x \neq y$, with x and y occurring in the initial collection of **MLSS**-literals.

It can easily be seen that starting with an initial collection $\mathcal{S}$ of flat **MLSS**-literals, any tableau construction rule subject to the above restrictions R1 and R2 must terminate

in a finite number of steps, generating a saturated tableau $\mathcal{T}_{\mathcal{S}}$ for $\mathcal{S}$. Then the decidability of **MLSS** follows from the fact that $\mathcal{S}$ is satisfiable if and only if the tableau $\mathcal{T}_{\mathcal{S}}$ is open.

From the soundness of rules (1)–(11), one only needs to check that if $\mathcal{T}_{\mathcal{S}}$ is open then $\mathcal{S}$ is satisfiable. Thus, let us assume that $\mathcal{T}_{\mathcal{S}}$ is open and let ϑ be an open (saturated) branch of $\mathcal{T}_{\mathcal{S}}$. Let

$V_{\mathcal{S}}$ be the collection of variables occurring in $\mathcal{S}$;
T be the collection of variables occurring on ϑ other than $V_{\mathcal{S}}$;
$\sim_{\mathcal{S}}$ be the equivalence relation induced on $V_{\mathcal{S}} \cup T$ by equality literals $x = y$ in ϑ;
T' be the set $\{t \in T : t \not\sim_{\mathcal{S}} x \text{ , for all } x \in V_{\mathcal{S}}\}$;
V' be the set $(V_{\mathcal{S}} \cup T) \setminus T'$;
$\widehat{\in}_{\vartheta}$ be the dyadic relation on $V' \cup T'$ defined as follows:

$$x \widehat{\in}_{\vartheta} y \quad \text{iff} \quad \text{the literal } x \in y \text{ is in } \vartheta.$$

In addition, for each $t \in T'$, let $\boldsymbol{u}_t$ be an assigned set.

Since the branch ϑ is not closed, the relation $\widehat{\in}_{\vartheta}$ is acyclic. Therefore we can recursively define the following assignment, called the *realization* of the branch ϑ relative to $\mathcal{S}$ and the sets $\boldsymbol{u}_t$, for $t \in T'$:

$$\begin{aligned} R_{\vartheta} x &= \{R_{\vartheta} y \mid y \widehat{\in}_{\vartheta} x\}\,, && \text{if } x \in V' \\ R_{\vartheta} t &= \boldsymbol{u}_t\,, && \text{if } t \in T'\,. \end{aligned}$$

It can be checked that if the sets $\boldsymbol{u}_t$ satisfy the conditions

(a) $\boldsymbol{u}_{t_1} \neq \boldsymbol{u}_{t_2}$, for every pair of distinct $t_1, t_2 \in T'$,
(b) $\boldsymbol{u}_t \neq R_{\vartheta} x$, for all $t \in T'$ and $x \in V'$,

then the realization R_{ϑ} is a model for ϑ, and in turn for $\mathcal{S}$. Since conditions (a) and (b) can always be enforced, for instance by choosing $|T'|$ distinct sets $\boldsymbol{u}_t$ of large enough cardinalities, we have the completeness of our tableau calculus.

It is also interesting to note that the realization R_{ϑ} can be used on open non-saturated branches to guide the saturation process, as discussed in [3].

Figure 1 contains a closed **MLSS**-tableau for the collection

$$\mathcal{S} = \{x = \{y\},\ x = z \cup w,\ y \notin z,\ x \neq w\}$$

of flat **MLSS**-literals.

Notice that in the above **MLSS**-tableau

- literals 1–4 form the initial tableau for $\mathcal{S}$;
- literal 5 has been added by rule (6);
- literals 6 and 7 have been added by rule (2);
- literals 8–11 have been added by rule (9);
- literal 12 has been added by rule (7);
- literal 13 has been added by rule (10);
- literal 14 has been added by rule (1).

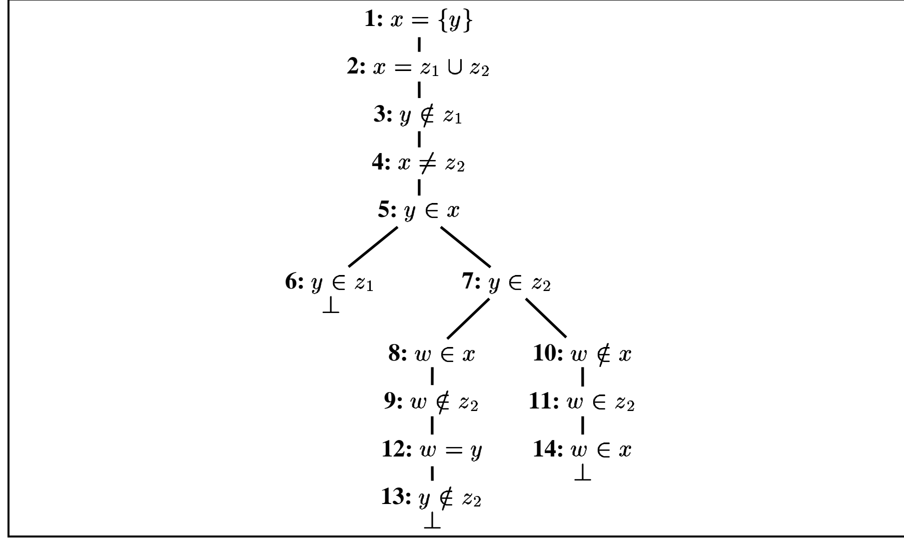


Fig. 1. A closed **MLSS**-tableau.

2 Extensions of multi-level syllogistic

The satisfiability algorithm for **MLSS** can be extended in useful ways by allowing otherwise uninterpreted function symbols subject to certain universally quantified statements to be intermixed with the other operators of **MLSS**. Note however that the statements decided by the method to be described remain unquantified; the quantified statements to which we refer appear only as implicit 'side conditions'.

The pairing operator cons and the two associated component extraction operators car and cdr exemplify the operator families to which our extension technique is applicable. Assume that an 'arbitrary selection' operator arb is available which meets the statement

$$\left[\forall x \mid \mathsf{arb}(x) \in x \cup \{x\} \,\&\, \mathsf{arb}(x) \cap x = \emptyset\right].$$

(That is, $\mathsf{arb}(x)$ returns an element of x which—as a set—does not have elements in common with x, *except* when $x = \emptyset$, in which case $\mathsf{arb}(x) = \emptyset$.)[5] Then cons, car, and cdr can be given the following formal set-theoretic definitions:

$$\begin{aligned} \mathsf{cons}(x, y) &=_{\text{Def}} \Big\{\{x\}, \Big\{\{x\}, \{\{y\}, y\}\Big\}\Big\}, \\ \mathsf{car}(p) &=_{\text{Def}} \mathsf{arb}\big(\mathsf{arb}(p)\big), \\ \mathsf{cdr}(p) &=_{\text{Def}} \mathsf{arb}\Big(\mathsf{arb}\big(\mathsf{arb}\big(\mathsf{arb}\big(p \setminus \{\mathsf{arb}(p)\}\big) \setminus \{\mathsf{arb}(p)\}\big)\big)\Big). \end{aligned}$$

However, in most settings, the details of these definitions are irrelevant. Only the following properties of these operators matter:

[5] Note, in passing, that a choice set for any family $\mathcal{F}$ of disjoint non-null sets can be formed as $\{\,\mathsf{arb}(x) \;:\; x \in \mathcal{F}\,\}$; hence, the assumed availability of arb, jointly with the replacement axiom of set theory, yields as a consequence the somewhat controversial *postulate of choice*.

- The object $\mathsf{cons}(x, y)$ can be formed for any two sets x, y.
- Both of the sets x, y from which $\mathsf{cons}(x, y)$ is formed can be recovered uniquely from the single object $\mathsf{cons}(x, y)$, since $\mathsf{car}(\mathsf{cons}(x, y)) = x$ and $\mathsf{cdr}(\mathsf{cons}(x, y)) = y$.

Almost all proofs in which the operators cons, car, and cdr appear use only these facts about this triple of operators. That is, they implicitly treat these operators as a family of three otherwise uninterpreted operators, subject only to the conditions

$$\big[\forall\, x,\ y \mid \mathsf{car}(\mathsf{cons}(x, y)) = x\big] \;\&\; \big[\forall\, x,\ y \mid \mathsf{cdr}(\mathsf{cons}(x, y)) = y\big]\ .$$

The treatment indicated throws away information about these operators (e.g. it hides that $\mathsf{car}(x)$ is always a member of a member of x) that may become relevant only in quite unusual situations.

Even the more fundamental issue of extending **MLSS** with the arb operator can be tackled in this frame of mind. It was shown in [10] how to constrain the semantics of arb so strongly as to take a commitment concerning the truth value of the existential closure of *any* conjunction φ of **MLSS** extended with arb. On the other hand, it is more in the spirit of this paper to remain neutral concerning infinitely many such statements involving arb; by leaving, e.g., the value of $\big[\exists\, x,\ y \mid \mathsf{arb}(\{\{x\}, \{x, y\}\}) \neq \{x\}\big]$ undefined.

Similar remarks apply to many important families of operators. We list some of these, along with their associated universally quantified statements:

(i) monotone functions:

$$\big[\forall x,\ y \mid x \supseteq y \Rightarrow f(x) \supseteq f(y)\big]\ ;$$

(ii) monotone functions having a known order relationship:

$$\big[\forall x,\ y \mid x \supseteq y \Rightarrow f(x) \supseteq f(y)\big] \;\&\; \big[\forall x,\ y \mid x \supseteq y \Rightarrow g(x) \supseteq g(y)\big] \;\&\; \big[\forall x \mid f(x) \supseteq g(x)\big]\ ;$$

(iii) monotone functions of several variables:

$$\big[\forall x,\ y,\ u,\ v \mid (x \supseteq y \,\&\, u \supseteq v) \Rightarrow f(x, u) \supseteq f(y, v)\big]\ ;$$

(iv) idempotent functions on a set w:

$$\big[\forall x \in w \mid f(x) \in w \,\&\, f\big(f(x)\big) = f(x)\big]\ ;$$

(v) total ordering relationships on a set:

$$\big[\forall x \in w,\ y \in w \mid R(x, y) \vee R(y, x)\big] \;\&\; \big[\forall x \in w,\ y \in w,\ z \in w \mid R(x, y) \,\&\, R(y, z) \Rightarrow R(x, z)\big]\ ;$$

(vi) (multiple) functions with known ranges w_j and domains v_j:

$$\big[\forall x \in v_j \mid f_j(x) \in w_j\big]\ ;$$

(vii) pairs of mutually inverse functions on a set:

$$\big[\forall x \in w \mid f(x) \in w \,\&\, g(x) \in w \,\&\, f\big(g(x)\big) = x \,\&\, g\big(f(x)\big) = x\big]\ .$$

These are all mathematically significant relationships, as the existence of names associated with them attests.

These cases can all be handled by a common method under the following conditions. Suppose that we are given an unquantified collection C of statements involving the operators of **MLSS** plus certain other function symbols f, g of various numbers of arguments. We can suppose that all occurrences of these additional symbols are in simple flat statements of forms like $y = f(x)$, $y = g(x, z)$, etc. From these initially given

statements we must be able to draw a 'complete' collection S of consequences involving the variables which appear in them, along with some finite number of additional variables that it may be necessary to introduce. The translated formula, comprising S and some residue of the original C, will be entirely within the language of **MLSS**. 'Completeness' means that we can be sure that any model $\mathcal{M}$ of the translated formula can be extended to include the original function symbols f in such a way that their interpretation $f^{\mathcal{M}}$ actually satisfies the desired properties (monotonicity, etc.).

Call these added statements S the *extension conditions* for the given set of functions: e.g., in most of the cases listed above, S will comprise *single-valuedness conditions* $x = u \Rightarrow y = v$ for all pairs $y = f(x)$, $v = f(u)$ originally present in C. If we can find them, the extension conditions will encapsulate everything which the appearance of the functions in question tells us about the set variables which also appear. Hence satisfiability can be determined by replacing all the statements $y = f(x)$, $y = g(x, z)$ in our original collection by the extension conditions derived from them.

This gives us a systematic way of reducing various languages extending **MLSS** to pure **MLSS**. As we will see, this approach can be exploited, to some extent, with predicates too, thanks to the fact that certain properties of predicates can be rendered via representing functions.

Note that the 'extension conditions' technique can be applied even if the recipe for removing function or predicate occurrences by adding compensating extension clauses is not complete, as long as it is sound, i.e. all the clauses added do follow from known properties of the functions or predicates removed.

Take Case (i) above (the 'monotone functions' case) as an example. Here the extension conditions can be derived as follows. Let the function symbols known to designate monotone functions be f, etc. Replace all the statements $y = f(x)$, $v = f(u)$ originally present by clauses of the form

$$x \supseteq u \Rightarrow y \supseteq v. \tag{2}$$

(Note that this implies the single-valuedness conditions for f.) The added clauses ensure that if a model exists, the set of pairs $\langle x^{\text{Model}}, y^{\text{Model}} \rangle$, formed for all the x and y initially appearing in clauses $y = f(x)$, defines a function F which is monotone on its domain. This can be extended to a function F' defined everywhere by defining $F'(s)$ as the union of all the $F(t)$, extended over all the elements t of the domain of F for which $s \supseteq t$. It is clear that the F' defined in this way is also monotone and extends F. This proves that the clauses (2) express the proper extension condition in Case (i). Note that the number of clauses (2) required is roughly as large as the square of the number of clauses $y = f(x)$ originally present.

To make this method of proof entirely clear we give an example. Suppose that we need to prove the implication

$$f\big(f(x \cup y)\big) \supseteq f\big(f(x)\big) \tag{3}$$

under the assumption that the function f is monotone. By flattening the compound terms which appear in this statement, we get the collection

$$z = x \cup y,\;\; u = f(z),\;\; w = f(u),\;\; u_1 = f(x),\;\; v_1 = f(u_1),\;\; \neg(w \supseteq v_1)\;,$$

which we must prove to be unsatisfiable. The four statements

$$u = f(z),\;\; w = f(u),\;\; u_1 = f(x),\;\; v_1 = f(u_1)$$

in this collection give rise to the 12 extension conditions

$$\begin{array}{lll} z \supseteq u \Rightarrow u \supseteq w, & z \supseteq x \Rightarrow u \supseteq u_1, & z \supseteq u_1 \Rightarrow u \supseteq v_1, \\ u \supseteq z \Rightarrow w \supseteq u, & u \supseteq x \Rightarrow w \supseteq u_1, & u \supseteq u_1 \Rightarrow w \supseteq v_1, \\ x \supseteq z \Rightarrow u_1 \supseteq u, & x \supseteq u \Rightarrow u_1 \supseteq w, & x \supseteq u_1 \Rightarrow u_1 \supseteq v_1, \\ u_1 \supseteq z \Rightarrow v_1 \supseteq u, & u_1 \supseteq u \Rightarrow v_1 \supseteq w, & u_1 \supseteq x \Rightarrow v_1 \supseteq u_1, \end{array}$$

which replace the four initial statements. It now becomes possible to see that

$$z = x \cup y,\;\; z \supseteq x \Rightarrow u \supseteq u_1\,,\;\; u \supseteq u_1 \Rightarrow w \supseteq v_1,\;\; \neg(w \supseteq v_1)$$

is an unsatisfiable conjunction, proving the validity of (3).

Without entering into much detail, let us observe that to seek a model for a collection of **MLSS** clauses, plus statements of the form $w = \mathsf{arb}(z)$, we could proceed in analogy with Case (i), by replacing all the $y = \mathsf{arb}(x)$, $v = \mathsf{arb}(u)$ originally given by the corresponding extension condition

$$\big((x = \emptyset \,\&\, y = \emptyset) \vee (y \in x \,\&\, y \cap x = \emptyset)\big) \,\&\, (x = u \Rightarrow y = v).$$

Case (ii) (monotone functions having a known order relationship) and Case (iii) (monotone functions of several variables) can be treated in much the same way as the somewhat simpler Case (i). In Case (ii), for example, given monotone f, g, where it is known that $f(x) \supseteq g(x)$ is universally true, first force the known part of their domains to be equal by introducing a u satisfying $g(x) = u$ for each initially given clause $y = f(x)$ and vice versa. Then proceed as in Case (i), but now add inclusions $x = v \Rightarrow y \supseteq u$ for every pair $g(v) = u$, $f(x) = y$ of clauses originally present. It is clear that the extensions of g and f eventually satisfied stand in the proper ordering relationship.

The related case of *additive functions* of a set variable can also be treated in the way which we will now explain (but the very many clauses which this technique introduces hints that 'additivity' is a significantly harder case than 'monotonicity'). A set-valued function f of sets is called 'additive' if $f(x \cup y) = f(x) \cup f(y)$ for all x and y. Given an otherwise uninterpreted function f which is supposed to be additive, and clauses $y = f(x)$, introduce all the 'atomic parts' of all the variables x which appear in such clauses. These are variables, named a_j in what follows, representing all the intersections of some of the sets represented by variables occurring in the original clauses with the complements of the other similar sets. In terms of these intersections, which clearly are all disjoint, express each x in terms of its atomic parts a_j, namely as $x = a_{j_1} \cup \cdots \cup a_{j_k}$. Likewise, after introducing clauses $b_j = f(a_j)$ giving names to the range elements $f(a_j)$, write out all the disjoint union relationships $y = b_{j_1} \cup \cdots \cup b_{j_k}$ that derive from clauses $y = f(x)$. Finally, add statements $a_j \cap a_i = \emptyset$ which express the disjointness of distinct sets a_j, and implications $a_j = \emptyset \Rightarrow b_j = \emptyset$ which state that the range of f on any null set must also be null. Now suppose that the set of clauses we have written has a model in which the a_j, b_j, x, y, etc. appearing above are represented by sets a'_j, b'_j, x', y', etc. and define the set-valued function $F(z')$, for each z', to be the

union of all the sets b'_j for which z' intersects a'_j. The function F defined in this way is clearly additive. It is also clear that if a clause $y = f(x)$ is present in our initial collection, and the variables x and y are represented by sets x' and y', then $y' = F(x')$. Hence F can represent f in the model we have constructed, so f can be represented by an additive function, proving that the clauses we have added to our original collection are the appropriate extension conditions. A more formal treatment of the extension of **MLSS** with additive functions and other constructs can be found in [8].

Case (iv) (idempotent functions on a set) is also easy. We can proceed as before, adding a clause $y = f(y)$ whenever a clause $y = f(x)$ is present. Then we add implications $u = x \Rightarrow z = y$ whenever two clauses $y = f(x)$, $z = f(u)$ are present, and remove all the clauses $y = f(x)$. The added clauses ensure that if a model exists, the mapping F which sends x^{Model} to y^{Model} for each clause $y = f(x)$ initially present is single-valued; moreover, thanks to the added clauses $y = f(y)$, this mapping is clearly idempotent where defined. It can be extended by mapping all elements not in the domain of F to any selected element of the range of F.

The 'self-inverse' function case

$$\big[\forall x \in w \mid f(x) \in w \,\&\, f\big(f(x)\big) = x\big]$$

can be handled in much the same way, but we omit the details since this can also be viewed as a special subcase of Case (vii) which we will discuss later.

Predicates representable by functions in one of the above classes can be removed automatically by first replacing them by the functions that represent them, and then removing these functions by writing the appropriate extension conditions. For example, equivalence relationships $R(x, y)$ can be written using a representing function f as $f(x) = f(y)$; f only needs to be single-valued. Any partial ordering relationship $R(x, y)$ can be written as $f(x) \supseteq f(y)$, where f only needs to be single-valued, and f is monotone if and only if the ordering relationship $R(x, y)$ is compatible with inclusion in the sense that

$$\big[\forall x,\, y \mid x \supseteq y \Rightarrow R(x, y)\big]\,.$$

Monadic predicates $P(x)$ satisfying the condition

$$\big[\forall\, x,\, y \mid P(x) \,\&\, P(y) \Rightarrow P(x \cup y)\big] \,\&\, \big[\forall\, x,\, y \mid P(x) \,\&\, x \supseteq y \Rightarrow P(y)\big]$$

can be written in the form $P(x) \;\Leftrightarrow_{\text{Def}} p \supseteq f(x)$, where f is additive. The predicates $Finite(x)$, $Countable(x)$, and $Is_map(x)$ (where $Is_map(s) \;\Leftrightarrow_{\text{Def}} \big[\forall\, x \in s \mid [\exists\, u,\, v \mid x = \mathsf{cons}(u, v)]\big]$) illustrate this remark.

Case (v) (total ordering relationships on a set) can be viewed in the following way deriving from the immediately preceding remarks. Let R be such a relationship. Introduce the representing function f for it, i.e. $f(x) \supseteq f(y) \Leftrightarrow R(x, y)$. Then R is a linear ordering if and only if the range elements $f(x)$ all belong to a collection of sets linearly ordered by inclusion. So write a clause $y \supseteq v \vee v \supseteq y$ for each pair of clauses $y = f(x)$, $v = f(u)$, and also write the conditions needed to ensure that f is single-valued. In the resulting model f plainly maps its domain into a collection of sets linearly ordered by inclusion, and then f can be extended to all other sets by sending them to $\emptyset$.

Case (vi) (multiple functions with known ranges and domains) is also very easy. Given a collection of such functions $f_j(x)$ and the associated sets v_j and w_j, plus some collection of clauses, simply write the usual single-valuedness conditions $x = u \Rightarrow f_j(x) = f_j(u)$, and add conditions $x \in v_j \Rightarrow y \in w_j$ derived from each of the clauses $y = f_j(x)$ originally present. Then in the model constructed each f_j is clearly modeled by a function which maps the intersection of its domain with v_j into w_j, and this can be extended by mapping all other elements to the null set.

Extension conditions for Case (vii) (pairs of mutually inverse functions f, g on a set) can be formulated as follows. Write the clauses that force f and g to be single-valued. To these, add clauses $y = v \Rightarrow x = u$ derived from all the given statements $y = f(x)$, $v = f(u)$, that force f to be 1-1 on its domain, and similarly for g. (Note that this much also handles the case of functions known to be 1-1.) Next add clauses $y = u \Rightarrow x = v$ derived from all the statement pairs $y = f(x)$, $v = g(u)$. Then, in the resulting model, the model functions F and G of f and g must both be 1-1, and G must be the inverse of F on $\mathsf{domain}(G) \cap \mathsf{range}(F)$. Since G is 1-1, it follows that the range of G on $\mathsf{domain}(G) \setminus \mathsf{range}(F)$ must be disjoint from $\mathsf{domain}(F)$, and similarly the range of F on $\mathsf{domain}(F) \setminus \mathsf{range}(G)$ must be disjoint from $\mathsf{domain}(G)$. Therefore F can be extended to

$$\mathsf{range}(G|_{\mathsf{domain}(G)\setminus\mathsf{range}(F)}) \qquad \text{(the range on the restriction)}$$

as the inverse of G, and similarly G extended to $\mathsf{range}(F|_{\mathsf{domain}(F)\setminus\mathsf{range}(G)})$ as the inverse of F. Let F' and G' be these extensions. Then plainly $\mathsf{domain}(F') = \mathsf{domain}(F) \cup \mathsf{range}(G)$, and so $\mathsf{range}(G') = \mathsf{range}(G) \cup \mathsf{domain}(F) = \mathsf{domain}(F')$ and vice versa. Hence the extensions F' and G' are mutually inverse with $\mathsf{domain}(F') = \mathsf{range}(G')$ and vice versa. F' and G' can now be extended to mutually inverse maps defined everywhere by using any 1-1 map of the complement of $\mathsf{domain}(F')$ onto the complement of $\mathsf{range}(F')$. This shows that the clauses listed above are the correct extension conditions for Case (vii).

The extension conditions for the important car, cdr, and cons case can be worked out in somewhat similar fashion as follows. Regard $\mathsf{cons}(x, y)$ as a family of one parameter functions $\mathsf{cons}_x(y)$ dependent on the subsidiary parameter x. The ranges of all the functions cons_x in the family are disjoint (since $\mathsf{cons}(x, y)$ can never equal $\mathsf{cons}(u, v)$ if $x \neq u$). For the same reason, each cons_x is 1-1, and cdr is its (left) inverse, i.e. $\mathsf{cdr}(\mathsf{cons}_x(y)) = y$. Also, $\mathsf{car}(\mathsf{cons}_x(y)) = x$ everywhere. The extension conditions needed can then be stated as follows:

(A) cons must be 'doubly 1-1' and well defined: add clauses
$$(x \neq u \vee y \neq v) \Rightarrow z \neq w \qquad \text{and} \qquad (x = u \,\&\, y = v) \Rightarrow z = w$$
derived from all pairs of initial clauses $z = \mathsf{cons}(x, y)$, $w = \mathsf{cons}(u, v)$;

(B) car and cdr must stand in the proper inverse relationship to cons: add clauses $u = z \Rightarrow x = v$ derived from all pairs $z = \mathsf{cons}(x, y)$, $v = \mathsf{car}(u)$, and clauses $u = z \Rightarrow y = v$ derived from all pairs $z = \mathsf{cons}(x, y)$, $v = \mathsf{cdr}(u)$ of initial statements.

We note a few more cases, whose details we omit, which can be handled by the 'extension conditions' technique. Uninterpreted commutative functions of two variables,

having the property

$$[\forall x,\, y \mid f(x,y) = f(y,x)],$$

can readily be handled by methods like those shown above. It might be possible to treat associativity also, possibly based an a prior **MLSS**-like theory of the concatenation operator. A further known case (cf. [2] and [4, Chapter 7]) is that in which we allow the special constant symbols $\mathbb{N}$ and $\mathcal{O}rd$ for the set of non-negative integers and for the class of all ordinals, respectively.

3 Using extensions of multi-level syllogistic in a proof-verifier

The proof-checker which we are currently developing invokes the **MLSS**-decider whenever the directive 'ELEM' is supplied as the justification of a statement ϑ during the construction of a proof. All steps which appear before ϑ in the proof (or part of them, if the user instructs the system to do so) are then collected together to form a set Ψ; the essential structure of the implication $(\&\,\Psi) \Rightarrow \vartheta$ is retained by a special 'blobbing' operation which reduces it to a formula φ of **MLSS** (or of an extension of **MLSS** treatable by the method discussed in this paper): this is the input passed to the decider.

Because of their special importance, the treatment of arb and of the cons-car-cdr group is built into ELEM. The use of supplementary proof mechanisms for handling other extended ELEM deductions like those described above can be switched on in the following way. Each of the cases discussed above is given a name, specifically MONOTONE_FCN (i), MONOTONE_GROUP (ii), MONOTONE_MULTIVAR (iii), IDEMPOTENT (iv), TOTAL_ORDERING (v), INVERSE_PAIR (vii), SELF_INVERSE, RANGE_AND_DOMAIN, etc. To enable the use of supplementary inferencing for a particular operator belonging to one of these named classes, one writes a verifier command of a form like

ENABLE_ELEM(*class_name*; *operator_list*),

where *class_name* is one of the names in the preceding list, and *operator_list* lists the operator symbols for which the designated style of inferencing is to be applied. An example is

ENABLE_ELEM(MONOTONE_FCN; $\bigcup$),

which states that during ELEM inferencing the 'union of elements' operator $\bigcup$ is to be treated as an otherwise uninterpreted symbol for a monotone increasing set operator. The *operator_list* parameter in each ENABLE_ELEM command must consist of the number of operators appropriate to the *class_name* used; e.g., MONOTONE_GROUP and INVERSE_PAIR each call for two operators f, g.

The ENABLE_ELEM command scans the list of all currently available theorems for theorems of form suitable to the type of inference defined by the *class_name* parameter. For example, SELF_INVERSE calls for a theorem of the form

$$[\forall x \in w \mid f(x) \in w \,\&\, f\big(f(x)\big) = x],$$

where f is the function symbol that appears as *operator_list* in this case; and our earlier command example ENABLE_ELEM(MONOTONE_FCN; $\bigcup$) calls for the theorem

$$\left[\forall x,\, y \mid x \supseteq y \Rightarrow \bigcup x \supseteq \bigcup y\right].$$

Cardinality is another similar example; the command

ENABLE_ELEM(MONOTONE_FCN; $\#$)

calls for the theorem

$$\left[\forall x,\, y \mid x \supseteq y \Rightarrow \#x \supseteq \#y\right].$$

Such presupposed theorems must have already been fed into the system along with their proofs, and should have been successfully verified: when the required theorem is found, the declared style of inferencing becomes available for the operator or operators listed; otherwise an error message is issued.

Since extensions of ELEM inferencing is not without its efficiency costs, one may wish to switch it on and off selectively, dynamically changing the collection of constructs and special properties enjoyed by them which the proof-checker must take into account. To switch off extended ELEM inferencing of a specified kind for specified operators one uses a command

DISABLE_ELEM(*class_name*; *operator_list*)

with the same parameters as the corresponding ENABLE_ELEM command.

The ELEM inferencing mechanism subroutine will handle such DISABLE_ELEM commands by comparing the theorem supplied to each of its stock of templates, and, if it matches, putting the named operator on record as having a named property (but ordinarily not using this fact). However, individual subsequent ELEM proof steps which carry a special 'use properties' indication, e.g. ELEM($\#$), ELEM($\bigcup$), or ELEM($\#, \bigcup$) can then leave the indicated functions unblobbed but replace occurrences of them in the manner indicated above.

References

1. A. Armando, S. Ranise, and M. Rusinowitch. Uniform derivation of decision procedures by superposition. In L. Fribourg (Ed.), *Proc. of Computer Science Logic 2001*, volume 2142 of *LNCS*, pp. 513–527, 2001.
2. M. Breban, A. Ferro, E. G. Omodeo, and J. T. Schwartz. Decision procedures for elementary sublanguages of set theory. II. Formulas involving restricted quantifiers, together with ordinal, integer, map, and domain notions. *Comm. Pure Applied Math.*, 34:177–195, 1981.
3. D. Cantone. A fast saturation strategy for set-theoretic tableaux. In D. Galmiche (Ed.), *Proc. of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1227 of *LNAI*, pp. 122–137. Springer-Verlag, 1997.
4. D. Cantone, A. Ferro, and E. G. Omodeo. *Computable Set Theory*. Vol. 6 Oxford Science Publications of *International Series of Monographs in Computer Science*. Clarendon Press, Oxford, 1989.
5. D. Cantone, E. G. Omodeo, and A. Policriti. The automation of syllogistic. II. Optimization and complexity issues. *J. Automated Reasoning*, 6(2):173–187, 1990.

6. D. Cantone, E. G. Omodeo, and A. Policriti. *Set Theory for Computing. From Decision Procedures to Declarative Programming with Sets*. Monographs in Computer Science. Springer-Verlag, New York, 2001.
7. D. Cantone, E. G. Omodeo, J. T. Schwartz, and P. Ursino. Notes from the logbook of a proof-checker's project. *Proc. of the International Symposium on Verification: theory and practice*, N. Dershowitz ed., LNCS, Springer-Verlag, to appear.
8. D. Cantone, J. T. Schwartz, and C. G. Zarba. A Decision Procedure for a Sublanguage of Set Theory Involving Monotone, Additive, and Multiplicative Functions. In I. Dahn and L. Vigneron (Eds.) *Proc. of FTP'2003*, Electronic Notes in Theoretical Computer Science, Vol. 86(1), Elsevier, 2003.
9. D. Cantone and C. G. Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In R. Caferra, G. Salzer (Eds.), *Proc. of the International Workshop on First-Order Theorem Proving (FTP'98)*, LNAI 1761, pp. 126–136. Springer-Verlag, 2000.
10. A. Ferro, E. G. Omodeo. Decision procedures for elementary sublanguages of set theory. VII. Validity in set theory when a choice operator is present. *Comm. Pure Applied Math.*, 40:265–280, 1987.
11. A. Ferro, E. G. Omodeo, and J. T. Schwartz. Decision procedures for elementary sublanguages of set theory. I. Multi-level syllogistic and some extensions. *Comm. Pure Applied Math.*, 33(5):599–608, 1980.
12. M. C. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 2nd edition, 1996. 1st ed., 1990.
13. E. G. Omodeo, and J. T. Schwartz. A 'Theory' mechanism for a proof-verifier based on first-order set theory. In A. Kakas and F. Sadri (Eds.), *Computational Logic: Logic Programming and beyond, Essays in honour of Robert Kowalski*, part II, LNAI 2408, pp. 214-230. Springer-Verlag, 2002.

Rogue Decision Procedures

Aaron Stump, Arumugam Deivanayagam, Spencer Kathol,
Dylan Lingelbach, and Daniel Schobel

Dept. of Computer Science and Engineering
Washington University in St. Louis
One Brookings Drive, Campus Box 1045
St. Louis, MO 63130-4899

Corresponding email: stump@cs.wustl.edu

Abstract. Efficient decision procedures require a substantial engineering effort to implement in mainstream languages. This paper proposes a new programming language called Rogue for implementing decision procedures. Rogue is a rewriting language with backtrackable mutable expression attributes and an interface to a fast SAT solver. Work in progress on implementing a Nelson-Oppen style cooperating validity checker in Rogue is also briefly described.

1 Introduction

High-performance decision procedures (DPs) for decidable logical theories have enjoyed significant recent research and development activity, largely due to their usefulness for system verification. But developing high-performance DPs, particularly the combinations of cooperating DPs most useful for verification, is a significant engineering challenge. This is largely because of the following common characteristics of most DPs:

symbolic computation Most DPs involve some form of manipulation of expressions. This often takes the form of rewriting. For example, all DPs which follow the Shostak approach [13] of canonizing terms and solving equations are based on rewriting expressions.

imperative data structures DPs usually employ special-purpose data structures for efficiency. Implementing such data structures in a programming language is most natural if the language has mutable references or some other imperative construct.

interface to SAT solver For many DPs, it is important to be able to take advantage of the great advances that have been made in the last decade in the performance of propositional SAT solvers. Early empirical results suggest that for highest performance, a tight coupling of the SAT solver and the rest of the DP may be necessary [3, 9]. Hence, some kind of interface to a fast SAT solver is needed.

backtracking Integrating DPs with a SAT solver then requires that the imperative data structures be backtracked to stay in sync with the SAT solver, which backtracks while searching for a satisfying propositional assignment.

How well do available programming languages and environments meet these needs?

industrial languages Industrial programming languages like Java, C, and C++ are widely used for implementing DPs. These languages are mature, efficient, and widely used; and they are well suited to implementing imperative data structures. Unfortunately, they lack native support for symbolic computation. Backtracking and an interface to a SAT solver can be built in these languages, but in the absence of a standard library for such things, a new DP will need to implement them from scratch. Systems like SVC (around 22K lines of C++) [2], Simplify (around 26K lines of Modula-3) [6], Verifun (around 10.5K lines of Java and 800 lines of C, not counting the SAT solver) [9], and CVC (150K lines of C++, including the SAT solver) [14] are written in these sorts of languages. The fairly hefty line counts are a tribute to the engineering effort required for these systems!

logic programming languages Logic programming languages are good at symbolic manipulation of expressions and backtracking. With some ingenuity, efficient data structures can be implemented in them, particularly if dynamic clauses are supported [1]. Rewriting can also be performed efficiently. It is not clear how the backtracking of a logic programming environment would be integrated with that of an existing SAT solver. This is because the SAT solver and the logic programming engine are both designed to control when backtracking occurs, and it is not clear how to make them cooperate.

ML and variants Languages like ML and its variants are actually probably the best fit for implementing DPs. Expressions can be implemented as a datatype, and the pattern-matching features of the language can then be conveniently used to rewrite expressions. Mutable references can be used for implementing imperative data structures. Backtracking and interfacing to the SAT solver can be done just as for mainstream languages, with the same engineering cost required. The ICS system (around 5K lines of Ocaml for the logical core, and 6K other lines) [8] and the Vampyre DP from Berkeley (around 10K lines of Ocaml; see http://www-cad.eecs.berkeley.edu/˜rupak/Vampyre/) are examples.

rewriting languages Pure rewriting languages like Maude [5] support extraordinarily efficient rewriting, but since they lack mutable references, they do not support efficient imperative data structures.

This paper proposes implementing decision procedures in a new programming language, called Rogue, under development by the authors. Rogue is based on an untyped version of the Rewriting Calculus of Cirstea and Kirchner [4]. Rogue adds backtrackable mutable expression attributes and fixes a particular evaluation order of expressions (which the Rewriting Calculus leaves unspecified). An interface to the Chaff SAT solver [10] is also included. The paper describes work in progress to implement a Rogue Validity Checker (RVC) in Rogue. Because Rogue is such a good fit for implementing decision procedures, RVC is on track to be implemented in much fewer lines of code than decision procedures based on mainstream languages. The prediction is that not counting the size of the Rogue interpreter and the SAT solver, the system can easily be implemented in less than 2000 lines of Rogue code.

2 Rogue

This section begins with some example pieces of Rogue code, and an informal explanation of attributes. It then defines the syntax and gives a big-step operational semantics for the Rogue language. It concludes with a description of the backtracking and SAT interfaces.

2.1 Examples

The code in Figure 1 gives a recursive definition of a function `prop.nnfer`. This function performs the familiar operation of converting a propositional formula built using `NOT`, `AND`, and `OR` from propositional variables to negation normal form (NNF).

```
prop.nnfer :=
 (NOT(NOT('f1))     -> prop.nnfer @ 'f1 |
  NOT(AND('f1,'f2)) -> OR(prop.nnfer @ NOT( 'f1),
                          prop.nnfer @ NOT('f2)) |
  NOT(OR('f1,'f2))  -> AND(prop.nnfer @ NOT('f1),
                           prop.nnfer @ NOT('f2)) |
  AND('f1,'f2)      -> AND(prop.nnfer @ 'f1, prop.nnfer @ 'f2) |
  OR('f1,'f2)       -> OR(prop.nnfer @ 'f1, prop.nnfer @ 'f2) |
  'x -> 'x);
```

Fig. 1. Rogue code for conversion to NNF

The definition consists of a collection of rewrite rules, separated by |. When `prop.nnfer` is applied to an expression E, the rules are tried in order. Applications are written using @. A rule `L -> R` matches iff there is a substitution σ for the variables of L such that $\sigma(L) \equiv E$. Variables are indicated with a single quotation mark. The last rule accepts any expression and just returns it. This default will be used here to translate propositional variables and negations thereof to themselves. The two rules just before the last one are congruence rules, which will apply to binary applications of `AND` and `OR` to transform their immediate subexpressions.

As a second example, here is a definition of a non-strict if-then-else operator:

```
u.ite := ('cond -> (1 -> 'then) -> (1 -> 'else) ->
             ((1    -> 'then) |
              (null -> 'else)) @ 'cond)
```

The definition makes `u.ite` take in three arguments: a condition (`'cond`), a then part (`1 -> 'then`) and an else part (`1 -> 'else`). We rely on the fact that in Rogue, evaluation does not occur on the right hand side of an arrow expression. This means that we can delay evaluation of then and else parts by putting them on the right hand sides of arrow expressions. The choice of 1 for the left hand sides of those arrow

expressions is arbitrary. After taking in the three arguments, the definition says that depending on whether the condition has evaluated to `1` or `null`, an application of `u.ite` will return either `'then` or `'else`. Notice that whichever of these is returned, it will then be evaluated, because it is no longer on the right hand side of an arrow.

Finally, here is a definition of a `let` operator for local definitions:

```
u.let := ((1 -> ('X,'E1,'E2)) -> ('X -> 'E2) @ 'E1)
```

Evaluating `u.let @ (1 -> ('q,33,'q+'q))` gives 66. This definition of `u.let` is based on the standard translation of let expressions into applications of functions. We use the same trick as for `u.ite` of hiding things on the right hand side of expressions to delay their evaluation. This is necessary to prevent the body of the let expression (`'E2`) from being evaluated before the value of `'E1` has been substituted for `'X`. Our definition is not entirely perfect, however. The standard translation of let expressions is applied as a pre-processing step before evaluation. We do not have such fine-grained control of evaluation here that we can specify such a requirement. Hence, `u.let` expressions will be translated as part of evaluation. This means that `u.let` expressions binding the same variable should not be nested. Nevertheless, `u.let` is still quite useful for making local definitions more readable.

2.2 Attributes

Rogue supports reading and writing attributes of expressions. For example,

```
a.b := c
```

sets the `b` attribute of expression `a` to hold expression `c` as its value. Evaluating `a.b` subsequently (with no intervening writes to `a.b`) returns `c`. We view `X.Y := Z` as an application of a single ternary operator for updating the value of attribute `Y` of expression `X` to store value `Z`. Occurrences of `X.Y` in any other position than immediately on the left of `:=` are parsed as applications of a binary attribute lookup operator.

Any expression in normal form can be used as an attribute of any other such expression. So things like

```
(a -> a).(b -> b) := (c -> c)
```

are allowed. Evaluating $E_1.E_2$ when the E_2 attribute of E_1 has not been set to hold anything returns `null`. The definitions given in the previous section are all just writes to attributes; for example, the `let` attribute of expression `u`. Recursion is supported, since evaluation is not performed on the right hand side of an arrow expression. Thus, if we have

```
foo.bar := ('x -> foo.bar @ 2*'x)
```

then an application like `foo.bar @ 3` will partially evaluate to `foo.bar @ 6`. Even though `foo.bar` was not defined at the point when its definition was being evaluated, it is defined now. So `foo.bar @ 6` will partially evaluate to `foo.bar @ 12`, etc. If evaluation occurred on the right hand sides of arrow expressions, then the definition of `foo.bar` would partially evaluate to

```
foo.bar := ('x -> null @ 2*'x)
```

which would evaluate to

```
foo.bar := ('x -> null)
```

which is not what we want.

As a final note, the mapping from attribute lookup expressions $E_1.E_2$ to stored values is implemented with a hash table, so looking up an attribute takes expected constant time.

2.3 Syntax and Operational Semantics

Rogue has the following syntax. The set of expressions is defined to be the least set containing constant expressions (identifiers and a special constant `null`) and variables (written with a preceding single quote, like `'x`), and closed under forming expressions with binary operators "@" (application), "$\rightarrow$" (formation of a rewrite rule; rewrite rules can alternatively just be viewed as functions), "|" (disjunctive sequencing of rewrite rules), "," (conjunctive sequencing), and "." (attribute lookup); and ternary operator ":=" (attribute update). Numerals of unbounded length and applications of the standard arithmetic operators (written in infix notation) are also expressions.

Figure 2 gives the operational semantics. In order to model the reading and writing of attributes, we make use of stores, which map attribute lookup expressions $E_1.E_2$ to stored values for the attributes. We derive sequents of the form $S :: X \Longrightarrow S' :: X'$, with the intended meaning that if X is evaluated in store S, it evaluates to X' and the store becomes S'. The rules (null-apply), (|-apply-1), (|-apply-2), (,-apply), ($\rightarrow$-apply), and (apply-cong) tell how to evaluate applications. The rules (attr-write) and (attr-read) are for evaluating writes and reads of attributes.

The helper functions used in Figure 2 are defined as follows. Given input E_1, E_2, the function *dropnull* returns E_i if E_{3-i} is `null`, and E_1, E_2 otherwise. Given inputs X, Y, Z, the function *trans* returns $\sigma(Y)$ if there exists a substitution σ for the variables of X such that $\sigma(X) \equiv Z$; and `null` if there is no such substitution. If S is a store, then $S[Q \mapsto R]$ is the store that is the same as S except that expression Q is mapped to expression R. Given inputs S a store and Q an expression, the function *lookup* is defined to return R if S maps Q to R, and `null` otherwise. We omit the operational semantics for the arithmetic operations. The implementation uses the GNU MP library to perform arbitrary-precision arithmetic.

2.4 Backtracking

We expose an interface in the Rogue language to backtrack the store assigning values to attributes. Rogue programs make calls to this interface explicitly to do backtracking. Multiple backtracked stores can exist independently in what we call *contexts*. The interface supports switching between contexts, which takes constant time in the current implementation. We give just the following informal description of the interface. For each context, a history of the store is maintained, organized into scopes. When the Rogue interpreter starts, it is in an initial context named `default`, whose scope level is 1. Below, v is a number greater than 0, and C is a constant expression :

$$\text{(base)}\ \frac{}{S :: X \Longrightarrow S :: X}\quad X \text{ a variable or constant}$$

$$\text{(,-cong)}\ \frac{S :: X \Longrightarrow S_1 :: X' \quad S_1 :: Y \Longrightarrow S_2 :: Y'}{S :: X, Y \Longrightarrow S_2 :: \mathit{dropnull}(X', Y')}$$

$$\text{(|-cong)}\ \frac{S :: X \Longrightarrow S_1 :: X' \quad S_1 :: Y \Longrightarrow S_2 :: Y'}{S :: X|Y \Longrightarrow S_2 :: X'|Y'}$$

$$\text{($\rightarrow$-cong)}\ \frac{S :: X \Longrightarrow S' :: X'}{S :: X \rightarrow Y \Longrightarrow S' :: X' \rightarrow Y}$$

$$\text{(apply-cong)}\ \frac{S :: F \Longrightarrow S_1 :: F_1 \quad S_1 :: Z \Longrightarrow S_2 :: Z_1}{S :: F@Z \Longrightarrow S_2 :: F_1@Z_1}\quad \begin{array}{l} F_1 \text{ not a ``}\rightarrow\text{'',} \\ \text{``|'', or ``,'' expr.} \end{array}$$

$$\text{(null-apply)}\ \frac{S :: F \Longrightarrow S_1 :: \mathit{null} \quad S_1 :: Z \Longrightarrow S_2 :: Z'}{S :: F@Z \Longrightarrow S_2 :: \mathit{null}}$$

$$\text{(|-apply-1)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X|Y \quad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: X@Z_1 \Longrightarrow S_3 :: Z_2 \end{array}}{S :: F@Z \Longrightarrow S_3 :: Z_2}\quad Z_2 \not\equiv \texttt{null}$$

$$\text{(|-apply-2)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X|Y \quad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: X@Z_1 \Longrightarrow S_3 :: \texttt{null} \quad S_3 :: Y@Z_1 \Longrightarrow S_4 :: Z_2 \end{array}}{S :: F@Z \Longrightarrow S_4 :: Z_2}$$

$$\text{(,-apply)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X, Y \quad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: X@Z_1 \Longrightarrow S_3 :: Q_1 \quad S_3 :: Y@Z_1 \Longrightarrow S_4 :: Q_2 \end{array}}{S :: F@Z \Longrightarrow S_4 :: \mathit{dropnull}(Q_1, Q_2)}$$

$$\text{($\rightarrow$-apply)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X \rightarrow Y \quad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: \mathit{trans}(X, Y, Z_1) \Longrightarrow S_3 :: R \end{array}}{S :: F@Z \Longrightarrow S_3 :: R}$$

$$\text{(attr-write)}\ \frac{\begin{array}{l} S :: X \Longrightarrow S_1 :: X' \quad S_1 :: Y \Longrightarrow S_2 :: Y' \\ S_2 :: Z \Longrightarrow S_3 :: Z' \end{array}}{S :: X.Y := Z \Longrightarrow S_3[X'.Y' \mapsto Z'] :: Z'}$$

$$\text{(attr-read)}\ \frac{S :: X \Longrightarrow S_1 :: X' \quad S_1 :: Y \Longrightarrow S_2 :: Y'}{S :: X.Y \Longrightarrow S_2 :: \mathit{lookup}(S_2, X'.Y')}$$

Fig. 2. Operational semantics of Rogue

`CM @ incsl(v)`: Increment the scope level of the current context by v.
`CM @ decsl(v)`: Let l be the scope level of the current context C, and let l' be $l - v$. If l' is greater than 0, set C's scope level to be l', and revert the store to the last form it had when its scope level was most recently l'. If l' is less than 1, do nothing.
`CM @ sl`: Return the scope level of the current context.
`CM @ cur`: Return the name of the current context.
`CM @ switch(C)`: Switch to the context named C. If there is no such context, create one, set its scope level to 1, and switch to it.

2.5 SAT Interface

Rogue has an interface to a single instance of the Chaff SAT solver. This Rogue interface is built using Clark Barrett's SAT API. We have exposed just the bare minimum of this interface needed to implement RVC:

SATSOLV @ AddVar: Return a fresh propositional variable
SATSOLV @ AddClause(L): Add the clause corresponding to L to the clause database. L is a comma-separated list of literals, where a literal is either a variable `v` returned by SATSOLV @ AddVar, or else a Rogue expression of the form `NOT(v)` for such a `v`.
SATSOLV @ AssignHook(E): Whenever the SAT solver sets a propositional variable x to have a boolean value v, the Rogue expression $(E @ x @ v)$ will be evaluated by the Rogue interpreter. This allows Rogue programs to take some action whenever the SAT solver sets a variable.
SATSOLV @ LevelHook(E): Whenever the SAT solver changes its scope level by some integer i, the Rogue expression $(E @ i)$ will be evaluated. This allows Rogue programs to take some action in this case, like calling one of the CM methods described above.
SATSOLV @ Satisfiable: Return SATISFIABLE or UNSATISFIABLE depending on whether or not the SAT solver finds a satisfying assignment for its clause database.

3 RVC

This section briefly describes the work in progress to implement a Nelson-Oppen combination of decision procedures in Rogue. To check a quantifier-free first-order formula F for satisfiability with respect to a combination of background theories, F is first desugared to remove all propositional operators other than negation, conjunction, and disjunction. It is then *atomified* (cf. [3]). Atomification replaces each distinct atomic subformula ϕ of F by a new propositional variable P_ϕ. A mapping is maintained from ϕ to P_ϕ and vice versa using an attribute called `expr`. One direction of this mapping makes it easy to replace identical subformulas by the same propositional variable. The use of the other is described next.

The resulting abstraction of F is put into CNF in a standard way. Its clauses are then given to Chaff via the `SATSOLV @ AddClause` method. `SATSOLV @ Satisfiable`

is called to check the satisfiability of the clauses. Rogue callbacks are registered with Chaff using `SATSOLV @ AssignHook` and `SATSOLV @ LevelHook`. Whenever a propositional variable P_ϕ is assigned a value, the registered AssignHook callback looks up ϕ from P_ϕ using the `expr` attribute. It then asserts ϕ to the decision procedures. If the decision procedures discover that the current set of asserted atomic formulas is unsatisfiable, they communicate this back to Chaff by asserting the disjunction of the negations of the corresponding propositional variables. This causes Chaff to backtrack. Rogue code for desugaring, atomifying, converting to CNF, and making use of the Chaff interface is complete. It is less than 100 lines long.

An online version of the Downey-Sethi-Tarjan [7] congruence closure algorithm is complete. It is about 300 lines of Rogue code, including the code in Figure 3 for the union-find data structure. This code implements path compression and union by rank to get the almost constant amortized running time. Union by rank is implemented even while ensuring that the representative of `union`'s first argument (`'y`) becomes the new representative. This is done using a `rep` attribute. Attributes are also used to hold the ranks and to implement the find pointers. The rest of the congruence closure algorithm uses attributes to hold signatures and signature representatives.

```
uf.union :=
('y -> 'x ->
  u.let @ (1 -> ('fx, (uf.find_top @ 'x),
  u.let @ (1 -> ('fy, (uf.find_top @ 'y),
  u.let @ (1 -> ('rfx, u.if @ (u.not @ 'fx.rank) @
                         (1 -> 'fx.rank := 0),
  u.let @ (1 -> ('rfy, u.if @ (u.not @ 'fy.rank) @
                         (1 -> 'fy.rank := 0),
                 u.ite @ ('rfy < 'rfx) @
                   (1 -> ('fy.find := 'fx, 'fx.rep := 'fy.rep)) @
                   (1 -> ('fx.find := 'fy,
                          u.if @ (u.eq @ 'rfx @ 'rfy) @
                           (1 -> 'fy.rank := 'rfy + 1)))))))))))),

uf.find_top  :=
('x ->
  u.let @ (1 -> ('top, (u.ite @ (u.not @ 'x.find) @
                          (1 -> 'x) @
                          (1 -> 'x.find := (uf.find_top @ 'x.find))),
  u.let @ (1 -> ('tmp, u.if @ (u.not @ 'top.rep) @
                         (1 -> 'top.rep := 'top),
                 'top))))),

uf.find := ('x -> (uf.find_top @ 'x).rep);
```

Fig. 3. Implementation of union-find in Rogue

A decision procedure for linear real arithmetic is about 75% complete. It currently stands at a bit under 250 lines of Rogue code implementing canonization of arithmetic expressions and Fourier-Motzkin variable elimination. Attributes can be used for things like mapping an arithmetic variable to the lists of its upper and lower bounding inequalities.

4 Future Work

One of the biggest concerns with the proposed approach is whether or not the Rogue code can be executed efficiently enough, both in terms of time and memory. Memory is an issue, because expressions with attributes implementing imperative data structures should not be garbage-collected. Currently, this constraint is observed simply by not garbage collecting any expressions, but this clearly will waste unacceptable amounts of memory on long runs. Also, the Rogue interpreter is currently implemented using infrastructure from the CVC system. It should be possible to implement a much smaller and perhaps more efficient interpreter by specializing various data structures to just what is needed for Rogue. It should be possible to use term indexing [12] to improve the performance of evaluating applications of sets of rules connected by $|$.

References

1. A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 2002. to appear.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, 1996.
3. C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.
4. H. Cirstea and C. Kirchner. The Rewriting Calculus - Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:363–399, May 2001. Also available as Technical Report A01-R-203, LORIA, Nancy (France).
5. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.
6. D. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. *Draft Paper*, 1996.
7. P. Downey, R. Sethi, and R. Tarjan. Variations on the Common Subexpression Problem. *Journal of the Association for Computing Machinery*, 27(4):758–71, 1980.
8. J. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *13th International Conference on Computer-Aided Verification*, 2001.
9. C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem Proving using Lazy Proof Explication. In W. Hunt and F. Somenzi, editors, *15th International Conference on Computer-Aided Verification*, 2003.
10. M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

11. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.
12. R. Sekar, I. Ramakrishnan, and A. Voronkov. *Term Indexing*, chapter 26. Volume 2 of Robinson and Voronkov [11], 2001.
13. R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.
14. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

A Proof-Producing Boolean Search Engine*

Clark Barrett[1] and Sergey Berezin[2]

[1] New York University, barrett@cs.nyu.edu
[2] Stanford University, berezin@stanford.edu

Abstract. We present a proof-producing search engine for solving the Boolean satisfiability problem. We show how the proof-producing infrastructure can be used to track the dependency information needed to implement important optimizations found in modern SAT solvers. We also describe how the same search engine can be extended to work with decision procedures for quantifier-free first-order logic. Initial results indicate that it is possible to extend a state-of-the-art SAT solver with proof production in a way that both preserves the algorithmic performance (e.g. the number of decisions to solve a problem) and does not incur unreasonable overhead for the proofs.

1 Introduction

Decision procedures for domain-specific first-order theories have become important tools for many verification applications. Two of the primary challenges in creating a practical implementation of such decision procedures are ensuring correctness and achieving adequate performance. The addition of proof production can help accomplish both of these goals.

Many arguments have been made for adding proof production to automated theorem provers. For example, proofs provide additional reliability and the ability to check a result independently using a trusted proof-checker. We advocate proof production for an additional reason: the proof infrastructure tracks dependencies among assumed and derived facts during the proof search. These dependencies capture exactly the information that is needed to to determine the cause of each conflict during the proof search, making it easy to generate *conflict clauses*. As described in Section 3, conflict clauses are an essential ingredient of efficient SAT algorithms.

Although other methods exist for generating conflict clauses when the input to the SAT solver is a Boolean formula, our algorithm can produce conflict clauses when extended to quantifier-free first order logic.

The paper is organized as follows. Following a survey of related work, Section 2 describes our proof system and a simple proof-producing SAT solver. Then, in Section 3, we give an overview of the performance enhancements used in modern SAT solvers. Section 4 gives a detailed implementation of an efficient proof-producing SAT solver, and Section 5 discusses the extension to quantifier-free first-order logic. Section 6 concludes.

---

* This research was supported by GSRC contract DABT63-96-C-0097-P00005, by National Science Foundation CCR-0121403, and by a grant from Intel Corporation. The content of this paper does not necessarily reflect the position or the policy of GSRC, NSF, Intel, or the Government, and no official endorsement should be inferred.

1.1 Related Work

In the past few years, there has been significant interest in combining decision procedures for first-order theories with SAT [1–3, 7, 8, 16]. The implementations described in these approaches treat the SAT solver more or less as a black box. In particular, the first-order problem must first be translated into a purely Boolean problem. Often, valuable structural information is lost during such a translation. In contrast, our approach integrates the SAT solver with the first-order decision procedures, allowing structural information to be preserved. In addition, hybrid systems which use available SAT solvers are unable to produce proofs. Our integrated solver does produce proofs.

Recently, there has been some work done on proof-producing SAT solvers [10, 17]. However, the "proof" produced by these solvers is really just a script which enables another, presumably trusted, solver to duplicate the steps taken by the original solver. Though this does increase confidence in the solution, it does not actually produce a proof object which can be checked by a theorem prover. In our approach, an actual proof object can be produced. This proof can then be checked by a small trusted theorem prover which does not need to include a SAT solver.

Previous work at Stanford on proof-producing decision procedures culminated with the proof-producing tool CVC [14, 15]. CVC includes two options for solving the Boolean part of the problem: a slow SAT solver that produces proofs and a fast SAT solver that does not produce proofs. The current research aims to combine these approaches, resulting in solver which is both fast and able to produce proofs.

2 A Simple Proof-Producing SAT solver

Consider the simple propositional logic described in Fig. 1. A propositional formula is built from the constant formulas $\top$ (always true) and $\bot$ (always false), propositional variables (i.e. variables that can either be assigned true or false), and Boolean operators $(\wedge, \vee, \neg)$. A *literal* is a propositional variable or the negation of a propositional variable. We also define $\phi_1 \rightarrow \phi_2$ to be an abbreviation for $\neg\phi_1 \vee \phi_2$, and $\phi_1 \leftrightarrow \phi_2$ to be an abbreviation for $(\phi_1 \rightarrow \phi_2) \wedge (\phi_1 \rightarrow \phi_2)$.

Given such a formula, the goal of SAT is to find an assignment of true or false to each variable such that the formula evaluates (under the obvious standard semantics) to true.

```
propositional formula ::= ⊤ | ⊥ | propositional variable
        | propositional formula ∧ propositional formula
        | propositional formula ∨ propositional formula
        | ¬propositional formula
```

Fig. 1. Propositional logic

If no satisfying assignment exists, a formula is said to be *unsatisfiable*. Given an unsatisfiable propositional formula ϕ, a proof system should be able to produce a proof

of $\neg\phi$. We now introduce a proof system for accomplishing this task which is based on natural deduction.

2.1 Proof System

A proof is a tree, each of whose nodes is labeled with a formula. The formulas at the leaves of the tree are called *assumptions*. Assumptions may be designated as *open* or *closed*.

A *sequent* is a pair $\Gamma \vdash \phi$, where Γ is a set of formulas and ϕ is a formula. Since we are often interested only in the assumptions and the conclusion (the formula labeling the root) of a proof, the sequent $\Gamma \vdash \phi$ is used to represent any proof whose open assumptions are among Γ and whose conclusion is ϕ.

A *proof rule* or *inference rule* is a function which takes one or more proofs (called premises) and returns a new proof whose root node has each of the input proofs as its successors. A proof rule specifies the formula which should label the new root node and may also change the designation of one or more assumptions from open to closed.

Proof rules depend only on the assumptions and conclusions of their premises and can thus be described using sequents. We denote a proof rule as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the P_i's are sequents for the premises and C is a sequent representing the new proof tree. The proof rule takes any set of proofs which match the P_i's and returns a new proof whose root is labeled by the right-hand side of C. If an assumption appears in some P_i but not in C, then that assumption is closed in the proof tree constructed by the proof rule. If there are no premises, the rule is called an *axiom*. We will describe the proof rules of our system below.

A sequent $\Gamma \vdash \phi$ is *valid* if the conjunction of the assumptions in Γ implies ϕ. A proof rule is *sound* if the validity of all its premises implies the validity of the conclusion. The set of *valid* proofs are those which can be constructed using the proof rules.

Though we will not prove it here, it is straightforward to show that if all the proof rules are sound, then the sequent for a valid proof is indeed valid.

2.2 Proof Rules

To simplify the notation, we write Γ, α to denote $\Gamma \cup \{\alpha\}$ for the assumptions. This notation also implies that $\alpha \notin \Gamma$.

The most basic rule is the assumption axiom. This and other rules needed for a simple proof-producing SAT solver are shown below.

$$\frac{}{\phi \vdash \phi}\text{ assume} \qquad \frac{\Gamma_1, \alpha \vdash \phi \quad \Gamma_2, \neg\alpha \vdash \phi}{\Gamma_1 \cup \Gamma_2 \vdash \phi}\text{ caseSplit}$$

$$\frac{\Gamma \vdash \phi \leftrightarrow \top}{\Gamma \vdash \phi}\text{ iffTrueElim} \qquad \frac{\Gamma \vdash \phi \leftrightarrow \bot}{\Gamma \vdash \neg\phi}\text{ iffFalseElim}$$

$$\frac{\Gamma_0 \vdash \alpha_0 \quad \Gamma_1 \vdash \alpha_1 \quad \dots \quad \Gamma_n \vdash \alpha_n}{\Gamma_0 \cup \Gamma_1, \dots, \Gamma_n \vdash \phi \leftrightarrow \phi'}\ \text{simplify}$$

$$\frac{\Gamma_1 \vdash \phi \leftrightarrow \psi \quad \Gamma_2 \vdash \psi \leftrightarrow \theta}{\Gamma_1 \cup \Gamma_2 \vdash \phi \leftrightarrow \theta}\ \text{trans}$$

In the pseudo-code, these rules are used as function calls which take the premises and, possibly, some additional parameters, and return the conclusion sequent. For instance, $\texttt{assume}(\phi)$ takes ϕ as an argument and returns the sequent $\phi \vdash \phi$ as the result. Similarly, $\texttt{caseSplit}(s_1,\ s_2,\ \alpha)$ is an application of the caseSplit rule with premises s_1 and s_2. The parameter α identifies which assumptions to eliminate from s_1 and s_2. A call to $\texttt{simplify}(\Delta,\ \phi)$ takes the set of premises $\Delta = \{\Gamma_i \vdash \alpha_i \mid i \in \{0 \dots n\}\}$ as its first parameter, and the formula ϕ to be simplified as its second parameter. It returns a sequent for $\phi \leftrightarrow \phi'$ where ϕ' is obtained by replacing all instances of the literals in Δ by true (and their negations by false) and applying obvious Boolean simplifications to the result.

Note that the premises must be of a certain form in order for the rule to be sound. Our implementation includes an option which causes each of these rules to verify at run-time that its arguments are of the right form and generates an error if the check fails. This provides a very efficient "on-the-fly" internal soundness check in the tool which can often be used to detect soundness bugs without the need for an external proof checker.

2.3 Naïve SAT solver with proof production.

A simple SAT solver can be constructed using an algorithm which first picks a propositional variable α called a *splitter*, assigns it true or false, and then calls itself recursively until the formula evaluates to true or false.

A simple proof-producing SAT solver using the proof rules just described is shown in Fig. 2. The procedure `checkSAT` takes as input a formula ϕ and returns either a theorem of the form $\vdash \neg\phi$ or a theorem of the form $\Gamma \vdash \phi$, where Γ is a set of literals appearing in ϕ.

The `checkSAT` procedure calls $\texttt{checkSAT}_r$, a recursive procedure which takes as input a set of assumptions Δ (theorems of the form $\alpha \vdash \alpha$ where α is a literal), and a formula ϕ, and returns either a theorem of the form $\Gamma \vdash \phi \leftrightarrow \top$ or $\Gamma \vdash \phi \leftrightarrow \bot$.

Both procedures use a helper function called `getRHS`. This function takes a proof of $\phi \leftrightarrow \phi'$ as input (for some ϕ and ϕ') and returns the formula ϕ'. The $\texttt{checkSAT}_r$ procedure also makes use of the `findSplitter` function which takes a formula and returns a propositional variable appearing in the formula.

3 Efficient SAT algorithms

The algorithm in the previous section is essentially an implementation of the standard Davis-Putnam-Logemann-Loveland (DPLL) algorithm [5, 6]. Modern SAT solvers like GRASP [12] and Chaff [13] are also based on this same fundamental algorithm, but include significant refinements and optimizations.

```
checkSAT(φ) {
  s := checkSAT_r(∅, φ);
  φ' := getRHS(s);
  if (φ' = ⊤) return iffTrueElim(s);
  return iffFalseElim(s);
}

checkSAT_r(Δ, φ) {
  s_0 := simplify(Δ, φ);
  φ' := getRHS(s_0);
  if (φ' ∈ {⊤,⊥}) return s_0;
  α := findSplitter(φ');
  s_1 := trans(s_0,checkSAT_r(Δ ∪ assume(α),φ'));
  if (getRHS(s_1)= ⊤) return s_1;
  s_2 := trans(s_0,checkSAT_r(Δ ∪ assume(¬α),φ'));
  if (getRHS(s_2)= ⊤) return s_2;
  return caseSplit(s_1,s_2,α);
}
```

Fig. 2. Naïve SAT solver.

Efficient SAT algorithms are based on fast manipulation of *clauses*. A clause is a disjunction of one or more literals. Most SAT solvers assume that the formula to be checked is given in *Conjunctive Normal Form* (CNF), that is, as a conjunction of clauses.

Fig. 3 shows pseudo-code for an enhanced SAT solver (without proofs). It is similar to the algorithms in [12, 13], but is organized slightly differently. The `checkSAT` procedure takes as input a formula ϕ and returns either $\emptyset$, indicating that the formula is unsatisfiable, or a satisfying assignment. The satisfying assignment is represented as a set of literals θ with the property that if each literal in θ is true, then the formula ϕ is also true. The formula to be checked is first converted to CNF. We do not address how to do this here, but the conversion is straightforward and discussed in other papers, such as [3, 11]. The CNF clauses are stored in the global variable Φ, and then the recursive procedure $\texttt{checkSAT}_r$ is called.

$\texttt{checkSAT}_r$ takes as input a partial assignment (an assignment to some subset of the variables appearing in Φ) again represented as a set θ of literals. It returns an assignment θ' which extends θ if there exists such an assignment satisfying Φ. Otherwise, it returns $\emptyset$. The first step in $\texttt{checkSAT}_r$ is *Boolean Constraint Propagation* (BCP). BCP uses the structure of the clauses in Φ to deduce additional assignments that must hold in order to obtain a satisfying assignment for Φ, and is described in more detail below. It returns a new partial assignment. BCP may discover that some variable v is required to take on two different values by different clauses in Φ. In this case, the returned partial assignment has both v and $\neg v$ and is said to be *inconsistent*.

If BCP discovers a conflict, the conflict is analyzed to produce a new *conflict clause*. This clause identifies a subset of θ which is responsible for the conflict and is (permanently) added to Φ, ensuring that the conflict will not reoccur. We discuss this more

below. When a conflict is discovered, $\texttt{checkSAT}_r$ returns $\emptyset$, indicating that the given partial assignment θ cannot be extended to a satisfying assignment.

If BCP does not discover a conflict, then the search for a satisfying assignment can continue. $\texttt{checkSAT}_r$ calls `findSplitter` which searches Φ for a variable not already assigned by θ. If such a variable is found, it is returned in α. Otherwise, $\emptyset$ is returned, in which case all variables are assigned, so the current assignment is a satisfying assignment.

If a splitter is found, it is added to the current assignment. Then $\texttt{checkSAT}_r$ is called recursively with the current assignment. The result is either a satisfying assignment which is then returned, or $\emptyset$, indicating that the current assignment does not have a satisfying assignment. In the latter case, execution continues at the top of the loop where BCP is called again. Because of the conflict clause just added by the most recent call to $\texttt{checkSAT}_r$, we are guaranteed that this call to BCP will detect an inconsistency.

```
checkSAT(φ) {
  Φ := convertToCNF(φ);
  return checkSAT_r(∅);
}

checkSAT_r(θ) {
  while (true) {
    θ := BCP(Φ,θ);
    if (inconsistent(θ)) {
      conflictClause := analyzeConflict(Φ,θ);
      Φ := Φ ∪ {conflictClause};
      return ∅;
    }
    α := findSplitter(Φ,θ);
    if (α = ∅) return θ;
    θ := θ ∪ α;
    θ' := checkSAT_r(θ);
    if (θ' ≠ ∅) return θ';
  }
}
```

Fig. 3. Enhanced SAT solver.

We now discuss the importance and implementation of several components of enhanced SAT algorithm shown in Fig. 3: fast BCP, conflict clauses, so called *non-chronological backtracking*, or *intelligent backjumping*, and finally, good *decision heuristics* for picking splitters.

3.1 Fast Boolean Constraint Propagation (BCP).

Intuitively, the purpose of BCP is to derive all the assignments to variables that logically follow from the current assignments without having to split on any variable. The

complexity of BCP is polynomial, and can be implemented very efficiently. Since the worst-case complexity of SAT is exponential in the number of variables, reducing the number of variables by BCP is one of the most important aspects of the algorithm.

When the formula is represented as a set of clauses, BCP amounts to finding *unit clauses*, those that have exactly one literal unassigned, and assigning the corresponding variable to make the literal true. This assignment may result in more unit clauses, and the process continues until either a contradiction is detected (one of the clauses gets all of its literals assigned to false), or no more unit clauses remain.

GRASP and Chaff implement unit clause detection by having two *watched literals* in each clause. As long as both literals stay unassigned, the clause is guaranteed not to become a unit clause. If either of the watched literals gets assigned, the clause is searched for another unassigned literal, and if one is found, it becomes the new watched literal. Each variable maintains two lists of clauses in which the variable appears as a positive or negative literal respectively. So, for each variable assignment, only those clauses are processed in which a watched literal becomes false.

Learning the Conflict Clauses. When a SAT solver detects a conflict, it is often the case that only a small subset of the variable assignments is responsible for the contradiction, and therefore, the same assignment will appear in many branches of the decision tree. A SAT solver takes advantage of this fact by learning such *conflict assignments*, so that when they show up again, it immediately backtracks without having to derive the contradiction again.

Typically, a conflict assignment contains exactly one variable assigned at the last level of recursion (often the most recent splitter). Other variables may be either previous splitters themselves, or assignments derived from those splitters by BCP.

A conflict assignment can be expressed as a formula $c \equiv \ell_1 \wedge \cdots \wedge \ell_n$, where ℓ_i's are literals. Since we know that the original problem Φ is unsatisfiable when c is true, we can state that Φ is satisfiable only if $\overline{c} \equiv \overline{\ell}_1 \vee \cdots \vee \overline{\ell}_n$ is true.

Notice that $\overline{c}$ has the syntax of a clause, so it can simply be added to Φ. When the same assignment is made again, the conflict will be immediately detected by BCP due to the newly added clause $\overline{c}$, which we call a *conflict clause*.

Intelligent Backjumping. Each variable assignment has an associated *decision level*, which is the corresponding depth in the decision tree where the assignment is made.

When a conflict occurs, the SAT solver returns to the previous decision level, effectively undoing all the assignments made at the current decision level. Notice that if the conflict clause includes the most recent splitter, then the negation of the splitter will be derived by BCP from the conflict clause. Therefore, there is no need to consider the opposite assignment of the splitter explicitly, it will happen automatically.

In some cases, the SAT solver can backtrack beyond the previous decision level. If the conflict clause does not include any variables from the previous decision level, then the negation of the splitter is still implied in the decision level before that. Thus, we can backtrack to the most recent decision level in which a variable from the conflict clause is assigned.

Decision Heuristics. It is well-known that the order in which the splitters are chosen can dramatically affect the performance of the SAT algorithm. Modern SAT solvers have developed sophisticated splitter heuristics that work amazingly well in practice. Examples of these heuristics are detailed in [9, 13].

4 An Efficient Proof-Producing SAT solver

We now give a relatively detailed description of a proof-producing SAT solver with the enhancements described above. Before describing the algorithm, we describe the data structures used in the algorithm. Some additional proof rules used by the algorithm are shown in Fig. 4.

4.1 Basic Data Structures

Expressions All formulas and terms are represented as DAGs with maximal sharing of subexpressions. That is, if two expressions e_1 and e_2 are syntactically the same, then they are physically stored in the same location. In particular, checking expressions for (syntactic) equality is a constant time operation (comparison of pointers).

Theorems Theorems hold a sequent $\Gamma \vdash \phi$, where Γ is a set of formula expressions and ϕ is a formula expression. Besides the sequent, theorems may also carry the actual proof tree corresponding to the sequent (as a special *proof term* expression, not discussed here). The proofs are only generated when the tool is requested to produce an externally checkable proof. Otherwise, the sequents are sufficient for the functionality of the algorithm. The metavariable s (for "sequent") is used to represent theorems. If s is a theorem whose sequent is $\Gamma \vdash \phi$, then `getAssumptions(`s`)` returns the set of formulas in Γ, and `getConc(`s`)` returns the formula ϕ. To simplify the algorithm, we also allow a special "NULL" theorem, without a sequent or a proof, denoted by $\emptyset$.

4.2 Program State

The state of the SAT solver consists of the following components. Some components are *backtracked*, meaning that when `pop()` is called, they revert to the value they had when the corresponding call to `push()` was made. Backtracked components are marked with the † sign.

†**Assumptions:** Δ is a set of theorems called *assumptions*, each of whose conclusions is a literal. These literals (and the corresponding variables) are said to be *assigned*. All other variables and literals are *unassigned*. Δ corresponds to decisions and derived literals.

†**Literals:** `literals` is a queue containing theorems whose conclusions are literals (which are waiting to be added as assumptions). The function call `pushBack(literals,`s`)` inserts s into the queue and the corresponding call to `popFront(literals)` returns and removes the first item in the queue. Initially, `literals` is empty.

$$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \wedge_E \text{-left} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \wedge_E \text{-right} \qquad \frac{\Gamma, \alpha \vdash \bot}{\Gamma \vdash \neg\alpha} \neg_I$$

$$\frac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \leftrightarrow \psi}{\Gamma_1 \cup \Gamma_2 \vdash \psi} \text{iffMP} \qquad \frac{(\Gamma_i \vdash \neg\alpha_i)_{i \neq j} \quad \Gamma \vdash \bigvee_i (\alpha_i)}{\Gamma \cup \bigcup_{i \neq j} (\Gamma_i) \vdash \alpha_j} \text{unitProp}$$

$$\frac{\Gamma, \alpha_1, \ldots, \alpha_n \vdash \bot}{\Gamma \vdash \neg\alpha_1 \vee \cdots \vee \neg\alpha_n} \text{conflictClause}$$

Fig. 4. Additional Proof Rules.

Clauses: Φ is a set of theorems, each of whose conclusions is a clause. Φ contains clauses that are part of the original formula to be checked for satisfiability as well as derived conflict clauses. Given a clause expression c, the function `setupWatchPointers(c)` selects two of its literals to be *watched literals* and associates with c two *watch pointers* indexed by $i \in \{0, 1\}$ which point to these literals. Given a clause c and an index $i \in \{0, 1\}$, the function `updateWatchPointer(c,i)` is called when the ith watch pointer in c is assigned. It searches for an unassigned literal in c and updates the ith watch pointer to point to the new literal.

Non-clauses: Θ is a set of theorems, each of whose conclusions is neither a literal nor a clause. It contains parts of the original formula which are in non-clausal form.

Watch lists: Associated with each literal is a list of the clauses where the literal is being watched. For a literal l, the function `getWatchPointers(l)` returns a set of pairs (c, i) where c is a clause in which l is being watched and $i \in \{0, 1\}$ is the index corresponding to the watched literal l in c. Initially these lists are empty.

4.3 The Algorithm

The code for an efficient proof-producing SAT solver is shown in Fig. 5 and Fig. 6.

checkSAT. The main function in the SAT checking algorithm is `checkSAT` which, as in the algorithm of Fig. 2, takes as input a formula ϕ and returns either a theorem of the form $\vdash \neg\phi$ or a theorem of the form $\Gamma \vdash \phi$, where Γ is a set of literals appearing in ϕ.

`checkSAT` begins by initializing the theorem sets to be empty. It then takes the formula to be checked and passes it as a trivial theorem (created using the `assume` rule) to `addFact`, which partitions ϕ into literals, clauses, and non-clauses. Next, `checkSAT` calls `checkSAT`$_r$ which does the main recursive search. If `checkSAT`$_r$ returns $\emptyset$, then this means that ϕ is satisfiable under the set of assumptions contained in Δ, so the `simplify` rule (followed by `iffTrueElim`) can be used to get a theorem of the appropriate form. If `checkSAT`$_r$ does not return $\emptyset$, then it must return a theorem whose conclusion is $\bot$. Since ϕ is assumed in the first call to `addFact`, any derivation of $\bot$ returned by `checkSAT`$_r$ will also contain ϕ as an assumption, so we can use the `notIntro` rule to derive a theorem whose conclusion is $\neg\phi$.

```
checkSAT(φ) {
  Θ := Φ := Δ := ∅;
  addFact(assume(φ));
  s := checkSAT_r();
  if (s = ∅) return iffTrueElim(simplify(Δ,φ));
  return notIntro(s,φ);
}

addFact(s) {
  φ := getConc(s);
  if (isConjunction(φ)) {
    addFact(andElimLeft(s));
    addFact(andElimRight(s));
  }
  else if (isLiteral(φ)) pushBack(literals, s);
  else if (isClause(φ)) {
    Φ := Φ ∪ {s};
    setupWatchPointers(getConc(s));
  }
  else Θ := Θ ∪ {s};
}

checkSAT_r() {
  while (true) {
    s := BCP();
    if (s ≠ ∅) return s;
    α := findSplitter();
    if (α = ∅) return ∅;
    push();
    addFact(assume(α));
    s := checkSAT_r();
    if (s = ∅) return s;
    pop();
    if (backJump(s)) return s;
    addFact(notIntro(s,getLastAssumption(s)));
  }
}
```

Fig. 5. Enhanced SAT solver with proofs.

addFact. `addFact` takes as input a theorem s and figures out where to put it. It first assigns ϕ to be the conclusion of the theorem. If ϕ is a conjunction, then s is split using the conjunction elimination rules and `addFact` calls itself recursively. If ϕ is a literal, s gets pushed onto the literal queue. If ϕ is a clause, s gets added to the set Φ of clauses. Otherwise, s is added to the set Θ of non-clauses.

checkSAT$_r$. checkSAT$_r$ is similar to the procedure of the same name in Fig. 3. It starts by calling BCP which figures out additional assignments which are implied by the current set of assignments. If BCP does not return $\emptyset$, it means that an inconsistency was detected and the return value is a proof of $\bot$ from the current set of assumptions. In this case, the current context is inconsistent, so there is no need to search further along the current branch. The theorem proving $\bot$ is returned.

If BCP does not detect an inconsistency, then we continue the search for a satisfying assignment by finding a splitter. The function `findSplitter` can look for a splitter in either the set of clauses Φ or the set of non-clauses Θ. In our implementation, find-Splitter first uses a depth-first search to find splitters from Θ. When all the literals in Θ have been assigned, it then uses a Chaff-like heuristic to pick splitters from Φ. There are certainly more sophisticated heuristics that could be used and investigating these is part of our current research.

If no splitter can be found, checkSAT$_r$ returns $\emptyset$ to indicate that the current set of assumptions constitutes a satisfying assignment. Otherwise, the backtracked state (Δ and `literals`) is saved by calling `push()`. Then, the splitter α is added to the set of assigned literals by calling `addFact` and checkSAT$_r$ is called recursively. If the result is $\emptyset$ indicating that a satisfying assignment was found, then checkSAT$_r$ returns without calling `pop()` to preserve the satisfying assignment.

The other possibility is that the recursive call results in a proof of $\bot$ from some subset of the current assumptions. In this case, we first check for the possibility of intelligent backjumping. The function backJump(s) returns true if none of the assumptions in s (excluding the most recently assigned assumption) were assigned in the current recursion level (since the most recent active call to `push`). In this case, the theorem s can be used to derive a literal in the previous recursion level, and is thus returned.

If backJump(s) returns false, then the `notIntro` rule can be used to derive the negation of the most recent assumption in s from the others, and checkSAT$_r$ starts over at the top of the loop.

BCP. The last part of the SAT solver is the BCP code. BCP begins by processing the queue of literals. Each theorem in the queue is added to the set of assumptions. Then the watch pointers for all clauses which are watching the literal are updated. If any of these clauses become unsatisfiable (all its literals are assigned false), the `inconsistent` flag is set and the inconsistent clause is stored in `unsatClause`. If any of these clauses becomes a unit clause, then the `unitProp` rule is used to derive the remaining unassigned literal.

Once all of the literals have been processed, if no inconsistency has been detected, then the non-clauses are processed. This is done by checking if any of them simplify

```
BCP() {
  inconsistent := false;
  while (¬inconsistent) {
    s := popFront(literals);
    Δ := Δ ∪ {s};
    l := getConc(s);
    w := getWatchPointers(l);
    foreach (c,i)∈ w {
      updateWatchPointers(c,i);
      if (isUnsat(c)) {
        inconsistent := true;
        unsatClause := c;
      }
      else if (isUnit(c)) addFact(unitProp(c));
    }
  }
  if (¬inconsistent) {
    foreach sθ ∈ Θ {
      s := iffMP(sθ,simplify(Δ,getConc(sθ)));
      if (getConc(s)= ⊥) return s;
    }
    return ∅;
  }
  conflict = processImplGraph(unsatClause);
  addFact(conflictClause(conflict));
  return conflict;
}
```

Fig. 6. Enhanced BCP with proofs.

to $\bot$. If so, a theorem deriving $\bot$ is returned. Otherwise, $\emptyset$ is returned indicating no inconsistency.

If an inconsistent clause is detected, then the so-called "implication graph" can be used to generate a conflict clause. The implication graph is stored implicitly by the theorems in Φ. We start by looking up the theorem for the inconsistent clause in Φ and collecting all of its assumptions. We then look in Δ for the theorems justifying these assumptions. This process is continued until we have a set of assumptions, at most one of which was assigned at the current recursion level (since the last active call to `push`). The call to `processImplGraph` returns a theorem which derives $\bot$ from these assumptions.

Finally, the `conflictClause` rule is used to turn the conflict theorem into a conflict clause, which is then added to the set of clauses (implementing conflict clause learning), and the conflict is returned.

5 Extension to Quantifier-Free First-Order Logic

Although the algorithm just presented only handles Boolean logic, it can easily be extended to cooperate with quantifier-free first-order decision procedures. In first-order logic, the atomic formulas are no longer required to be propositional variables, but can also be predicates applied to terms over object constants and variables. Therefore, we redefine the notion of a *literal* to be an atomic formula (a predicate or a propositional variable) or its negation.

Since the atomic formulas are no longer independent from each other, purely Boolean constraint propagation is not sufficient. To overcome this problem, we extend BCP to *first-order constraint propagation* (FOCP) as follows. Every time a new literal ϕ is added to Δ, it is also submitted to the first-order decision procedure. Similarly, whenever a decision procedure derives a new literal l, `addFact(`l`)` is called. In fact, if a decision procedure derives non-literals, these can also be handled just by calling `addFact`. However, in this case the additional clauses or non-clauses will have to be backtracked.

Additionally, after each literal is submitted to the first-order decision procedure, the first-order decision procedure may report an inconsistency. If the first-order decision procedure is instrumented with proof production, then it can return a theorem deriving $\bot$ from some subset of the current assumptions. This theorem can be then be used to generate a conflict clause just as with purely Boolean conflicts.

The ability to produce proofs from the first-order decision procedure is thus essential for enabling the efficiency gains that come from learning conflict clauses.

6 Conclusion

We have implemented a prototype of our algorithm in an emerging tool called *CVC Lite*. CVC Lite is a smaller, more light-weight implementation of CVC, designed for experimentation and rapid prototyping.

For our experiments, we adopted the philosophy of using families of benchmarks as advocated in [4]. We compared the average number of decisions required to solve the

Boolean benchmarks in the *PC* families (each containing 33 benchmarks) of the *hole4* and *hole5* benchmarks [4]. We compared zchaff, CVC (in its standard proof-producing mode), and CVC Lite. Though CVC has a fast SAT-based mode which is essentially equivalent to zchaff, this mode cannot produce proofs.

benchmark	zchaff	CVC Lite	CVC
hole4	30	31	800
hole5	149	163	25832

Fig. 7. Comparison of the number of decisions.

Clearly, CVC Lite does not quite capture all of the intelligence in zchaff, but it is close. When proofs are enabled, CVC does a poor job on Boolean examples. CVC Lite demonstrates that it is possible to implement a proof-producing Boolean solver whose algorithmic performance is close to that of a highly tuned non-proof-producing solver.

Because our prototype implementation has not been tuned for run-time performance, we did not compare run-times. However based on [17] as well as our previous experience with CVC, proof-production only adds a small constant overhead to the underlying algorithm (the reason CVC is so much slower in proof-production mode is that it is not using the efficient SAT algorithms, not because proof-production adds a lot of overhead). Also, the optimizations enabled by proof production typically lead to an exponential speedup, so a small amount of overhead is reasonable.

As already mentioned, one important area of ongoing research is investigating decision heuristics for combined clausal and non-clausal formulas. An obvious question is: why not just convert everything to CNF? Although this is possible for purely Boolean formulas, it often degrades performance of some non-Boolean testcases (this problem is also mentioned in [3]). We are working on decision heuristics which perform well on both Boolean and non-Boolean testcases.

References

1. Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-Based Procedures for Temporal Reasoning. In S. Biundo and M. Fox, editors, *Proceedings of the 5th European Conference on Planning (Durham, UK)*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2000.
2. Gilles Audemard, Piergiorgio Bertoli, and Alessandro Cimatti. A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In Reiner Hähnle, editor, *Proceedings of the 18th International Conference on Automated Deduction (Copenhagen, Denmark)*, Lecture Notes in Artificial Intelligence. Springer, 2002.
3. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, 2002. Copenhagen, Denmark.

4. Franc Brglez, Xiao Yu Li, and Matthias F. Stallman. The role of a skeptic agent in testing and benchmarking of SAT algorithms. In *Fifth Int. Symp. on the Theory and Applications of Satisfiability Testing*, Cincinnati, Ohio, USA, May 2002.
5. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.
6. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
7. Leonardo de Moura, Harald Ruess, and Maria Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *18th International Conference on Automated Deduction*, 2002.
8. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James Saxe. Theorem Proving using Lazy Proof Explication. In *15th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
9. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
10. E. Goldberg and Y. Novikov. Verification of Proofs of Unsatisfiability for CNF formulas. In *Proceedings of Design, Automation and Test in Europe (Munich, Germany)*, 2003.
11. Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.
12. J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
13. M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.
14. A. Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002. In preparation: check http://verify.stanford.edu/˜ stump/ for a draft.
15. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.
16. Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
17. L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proceedings of Design, Automation and Test in Europe (Munich, Germany)*, 2003.

Abstraction Based Theorem Proving: An example from the theory of Reals

Ashish Tiwari*

SRI International,
333 Ravenswood Ave,
Menlo Park, CA, U.S.A
Tel:+1.650.859.4774
Fax:+1.650.859.2844
tiwari@csl.sri.com

Abstract. This paper presents a sound, but incomplete, logical decision procedure for the quantifier-free theory of reals. It is obtained using a combination of superposition and ordered chaining calculi. The emphasis is on termination and efficiency, rather than completeness. The procedure is used at the core of an hybrid abstraction tool for creating finite state sound abstractions of hybrid systems.

1 Introduction

Soundness, completeness, and termination are three crucial properties of inference systems. If an inference system is sound and complete with respect to a logical theory and terminating as well, then it can be used to obtain a decision procedure for that theory. However, many logical theories, including the full first-order theory, are undecidable. For an undecidable theory T, any inference system will either not be sound and complete with respect to T, or it will be non-terminating.

In light of the undecidability, the emphasis in the theorem proving community has been on designing sound and complete, but nonterminating, inference systems. In this paper, we follow a different paradigm. Our interest is in sound and terminating inference systems, that may not really be complete for the given theory of interest.

One can argue that an inference system that is not sound and complete with respect to a theory T of interest, is not really an inference system *for* T. Indeed, given an inference system I which is sound with respect to T and terminating, we can define a theory T' with respect to which it is also complete. Here T' is defined to be the set of (the negation of) all sentences which are (shown contradictory) proved by I. Of course, for "weak" inference systems I, T' might not even be deductively closed. But, due to soundness, $T' \subseteq T$ and we call any such T' an *abstraction* of the theory T.

Given a fixed theory T of interest, there are several abstractions T' of T for which sound, complete, and terminating inference systems can be designed. Any set of sound

* Research supported in part by DARPA under the MoBIES and SEC programs administered by AFRL under contracts F33615-00-C-1700 and F33615-00-C-3043, respectively, and NASA Langley Research Center contract NAS1-00108 to Rannoch Corporation.

(with respect to T) and terminating inference system defines, as we saw above, an abstraction T' of T. This is also true of abstractions of state transition systems, as is popular in formal verification. The important question, therefore, in both areas, is really the identification of *pragmatically useful* abstractions.

In this paper, the theory T of interest is the quantifier-free theory of reals. We present an inference system which is sound with respect to T and also terminating. However, the abstract theory T' with respect to which it is also complete, is hard to characterize in any other way.

1.1 Motivation

We briefly motivate why identifying such pragmatically useful decidable abstractions T' of logical theories T are of interest. Many applications use reasoning systems in a *failure-tolerant* mode, that is, failure to identify an unsatisfiable formula (as being unsatisfiable) is not catastrophic to the application. For example, in compiler optimization, the ability to perform nontrivial reasoning only results in better optimized code. Failure of the reasoning system does not interfere with the correctness of the compiler itself. As a second example, consider the process of constructing (predicate) abstractions of state transition systems. An incomplete prover, again, only results in a coarser abstractions. Any abstraction constructed by such a prover, would still be a "sound" abstraction. In both these applications, note further that the theorem prover is embedded inside the application, and hence termination is clearly more desirable.

A second motivation comes from the observation that very often the full theory of interest is either undecidable, or has a decision procedure with a very large computational complexity. In either case, it is important to identify pragmatically useful abstractions T', or in other words, identify sound, but incomplete, and terminating inference systems.

2 The theory of Reals

We assume the reader is familiar with the definition of a field K over a signature consisting of binary symbols $+$ and $\cdot$, unary symbols $-$ and $^{-}$, and constants 0 and 1. A real closed field is formally defined as follows.

Definition 1. *An ordered field K is a field together with a subset P of K of positive elements such that (i) $0 \not\in P$, (ii) if $a \in K$, then either $a \in P$, $a = 0$, or $-a \in P$, and (iii) P is closed under the addition $+$ and multiplication $\cdot$ operators.*

A real closed field K is an ordered field such that (i) every positive element of K has a square root in K and (ii) every polynomial f(x) in K[x] of odd degree has a root in K.

The theory of real closed fields, RCF, is defined over the signature $\langle \mathbb{Q}, +, -, \cdot, >, \geq, \approx \rangle$, where $\mathbb{Q}$ is the set of rational numbers, $+, \cdot$ are binary function symbols, $-$ is a unary symbol, and $>, \approx, \geq$ are binary predicates. The field of real numbers $\mathbb{R}$ is a model of RCF, but it is not the only model. However, it can be shown that the full first-order theories $Th(\mathbb{R})$ and $Th(RCF)$ are identical.

The first-order theory of real closed fields was shown to be decidable by Tarski [13] by showing that the theory admitted quantifier elimination. This theory has a known upper-bound of $n^{2^{O(n)}}$, thanks to Collin's method based on cylindrical algebraic decomposition [5]. Over the last 25 years, the CAD algorithm has been improved and made more efficient [9, 11, 12].

Although an excellent implementation of the cylindrical algebraic decomposition based decision procedure is available [10], the double exponential complexity for the decision procedure greatly hampers its utility for applications alluded to above. Moreover, in our application to automatically construct abstractions for hybrid systems, the proof obligations are mostly from the *quantifier-free* theory of the reals and they are often just a *conjunction of atomic formulas*.

We describe below the procedure that has worked best in our application domain. It is a *logical* decision procedure obtained by combining the superposition calculus for constructing Gröbner bases of polynomial equations [3], with an ordered chaining calculus for handling inequalities. Ordered chaining calculi for abstract transitive relations have been developed previously [2]. This paper uses them in the specific context of the greater-than $>$ and greater-than-equal-to $\geq$ relation over the reals.

3 Polynomials

Terms over the signature of reals can be represented as *polynomials*, written as $\Sigma_{1\leq i\leq n} c_i t_i$, where c_i's are elements of $\mathbb{Q}$ and t_i's are products over variables, called *power-products*. The computation of a polynomial form is best described via a convergent rewrite system over the following sorted signature.

$$
\begin{array}{rrl}
0, 1, \ldots, c \in \mathbb{Q} : & & \to \mathit{Coef} \\
- : & \mathit{Coef} & \to \mathit{Coef} \\
+, \times : & \mathit{Coef} \times \mathit{Coef} & \to \mathit{Coef} \\
I, x_1, \ldots, x_n : & & \to \mathit{PProd} \\
\cdot : & \mathit{PProd} \times \mathit{PProd} & \to \mathit{PProd} \\
M : & \mathit{Coef} \times \mathit{PProd} & \to \mathit{Poly} \\
\mathbf{0} : & & \to \mathit{Poly} \\
\oplus, \otimes : & \mathit{Poly} \times \mathit{Poly} & \to \mathit{Poly}
\end{array}
$$

Terms of sort Coef, PProd, and Poly are respectively called coefficients, power-products, and polynomial expressions. We use u, v for variables of sort Coef, μ, ν for PProd variables, and α, β, γ for Poly variables. Every term over the signature of the theory of reals can be represented as a term of sort Poly. Terms of sort Poly can be normalized via the following rewrite system, which is ground convergent modulo the associativity and commutativity axioms (AC), to give the corresponding polynomial form for the

term [3].

$$\mu \cdot I \to \mu \quad (1)$$
$$M(0,\nu) \to \mathbf{0} \quad (2)$$
$$\mathbf{0} \oplus \alpha \to \alpha \quad (3)$$
$$\mathbf{0} \otimes \alpha \to \mathbf{0} \quad (4)$$
$$M(u,\nu) \oplus M(v,\nu) \to M(u+v,\nu) \quad (5)$$
$$M(u,\mu) \otimes M(v,\nu) \to M(u \times v, \mu \cdot \nu) \quad (6)$$
$$\alpha \otimes (\beta \oplus \gamma) \to (\alpha \otimes \beta) \oplus (\alpha \otimes \gamma) \quad (7)$$

The equational theory induced by the above rewrite system (modulo AC) is identical to the equational theory of polynomials [3]. Note that the variables $x_1, \ldots, x_n$ are treated as special constants for purposes of computing normal forms for $Poly$ terms using the above rewrite system. "Ground" terms (of sort *Poly*) in normal form correspond to (and will be called) *polynomials*. They can be written as sums of monomials $M(c_1,t_1) \oplus M(c_2,t_2) \oplus \cdots M(c_n,t_n)$, which will be notationally simplified to $\Sigma_j c_j t_j$. The above sorted signature was introduced purely for the purpose of presenting the convergent rewrite system for normalizing terms in the signature of reals. In the subsequent sections, we will use the popular and simplified notation for writing polynomials. In particular, the symbol **0** will be represented by 0, $\oplus$ will be written as $+$, and M and $\otimes$ will be replaced by juxtaposition. We will use (meta-variables) c, d to denote elements of $\mathbb{Q}$, s, t to denote ground $PProd$ terms, and p, q, r to denote ground $Poly$ terms.

4 A decision procedure for an abstract theory of reals

Our algorithm maintains ground $PProd$ terms p in polynomial normal-form and atomic formulas in the form $p \approx 0$, $p > 0$, or $p \geq 0$. A conjunction of such atomic formulas is described as a tuple (E, R, S), where E, R and S contain atomic formulas of the first, second, and third kinds respectively. Since interest is in the satisfiability of the conjunction of these atomic formulas, the variables can be treated as *constants*, as in [3], and an ordering $\succ$ defined on the power-products, that can be suitably extended to polynomials. More specifically, if $\succ$ is any precedence on the indeterminates $x_1, \ldots, x_n$, then we have a choice between using "lexicographic ordering" or "total degree lexicographic ordering" on power-products [3, 4]. For example, if we choose the precedence $x_1 \succ x_2 \succ x_3$ over the 3 variables, then $x_1 x_2^2 \succ x_1^2$ in the total-degree lexicographic ordering (since degree of $x_1 x_2^2$ is 3, which is greater than the degree of x_1^2), while x_1^2 is greater in the lexicographic ordering since degree of x_1 is greater in x_1^2.

Our inference rules work on the highest power-product in a polynomial, and hence we often write a polynomial p as $c_0 t_0 + \Sigma_{1 \leq i \leq n} c_i t_i$ to distinguish the highest power-products t_0 from the rest. For example, in the polynomial $2x_1^2 x_2 + x_2^2$, $c_0 = 2$ and $t_0 = x_1^2 x_2$, if we use the total degree lexicographic ordering.

4.1 Gröbner Basis for Polynomial Equations

Polynomial equations are handled by using the completion-like method for constructing Gröbner bases [3]. Since the coefficient domain is restricted to rationals, the computation is much simplified. For purposes of efficiency, though, the full blown Gröbner

basis computation is (optionally) replaced by a computation that only performs *simplifications* and avoids all *deduction* steps.

Let $q = d_0 t_0 + \Sigma_i d_i t_i$ be a polynomial with leading power-product t_0. A polynomial equation $q = 0$ can be used as a rewrite rule to simplify other polynomials. If $p = p' + c_k s_k$ such that s_k can be written in the form $t_0 t_0'$, then we define $p \rightarrow_{q=0} p' + p''$, where $p'' = c_k s_k + (-c_k t_0'/d_0)(d_0 t_0 + \Sigma_i d_i t_i)$. Note that p will be simplified into a polynomial form using the convergent rewrite system from Section 3 before it is added in the consequent below.

$$\frac{(E \cup \{p \approx 0,\ q \approx 0\}, R, S)}{(E \cup \{p' \approx 0,\ q \approx 0\}, R, S)} \qquad \texttt{if } p \rightarrow_{q \approx 0} p'$$

$$\frac{(E\{q \approx 0\}, R \cup \{p > 0\}, S)}{(E \cup \{q \approx 0\}, R \cup \{p' > 0\}, S)} \qquad \texttt{if } p \rightarrow_{q \approx 0} p'$$

$$\frac{(E\{q \approx 0\}, R, S \cup \{p \geq 0\})}{(E \cup \{q \approx 0\}, R, S \cup \{p' \geq 0\})} \qquad \texttt{if } p \rightarrow_{q \approx 0} p'$$

It is clear that repeated applications of the above inference rules is bound to terminate: the cardinality of the E, R, S sets does not increase and terms in the sets E, R, S only get smaller in the well-founded ordering $\succ$. When restricted to the equational subcase, these inference rules capture the *collapse* rule, but do not compute nontrivial critical pairs (that is, do not capture the *superposition* rule below), and hence, do not result in a convergent set (Gröbner basis) for the polynomial equations.

$$\frac{(E' = E \cup \{c_0 s_0 + p \approx 0,\ d_0 t_0 + q \approx 0\}, R, S)}{(E' \cup \{c_0 t_0' q - d_0 s_0' p \approx 0\}, R, S)} \qquad \texttt{if } t_0' t_0 = s_0' s_0 = lcm(s_0, t_0) \neq s_0 t_0$$

The full Gröbner basis computation for the equations E is available as an option, but in its default setting, this is turned off. Note that a full Gröbner basis computation is "complete" only for algebraic closed fields, and hence doing the superposition inferences is still not complete for real closed fields. For completeness, among other things, we would need additional rules for polynomial factorization.

Example 1. Let $E = \{x_1^2 x_2 \approx 0,\ x_1 x_2 \approx 1\}$, $R = S = \emptyset$. Now, $x_1^2 x_2$ rewrites to $1x_1$, which normalizes to x_1, by the equation $x_1 x_2 \approx 1$. Hence, using the first inference rule, we can deduce the new equation $x_1 \approx 0$. Now, $x_1 x_2$ can be simplified to $0x_2$, which normalizes to 0, by $x_1 \approx 0$. Thus, we infer $1 = 0$ and add it to the set E. The inconsistency detection rules in Section 4.3 detect this as an inconsistency.

4.2 Ordered Chaining for Polynomial Inequalities

We use an ordered chaining calculus, inspired by the work of Bachmair and Ganzinger [2], to deal with the inequalities.

Definition 2. *Let $p = c_0 s_0 + p'$ be a polynomial with leading power-product s_0 and leading coefficient c_0. Let P be a set of polynomials. Let $q_1, q_2, \ldots, q_k \in P$ be such that*

the product of leading power-products of q_i's is equal to s_0, and the product of leading coefficients of q_i's is d_0. If $c_0 d_0 < 0$, then we say that the polynomial $p + |\frac{c_0}{d_0}| q_1 q_2 \cdots q_k$ is a k-chaining of p and P.

If the polynomial p and each polynomial in P is known to be non-negative, then the k-chaining polynomial of p and P will also be non-negative. If p is strictly positive, then the k-chaining polynomial will also be strictly positive. This reasoning is used for dealing with inequalities and formally it is stated as the following inference rules.

$$\frac{(E, R \cup \{p > 0\}, S)}{(E, R \cup \{p > 0, q > 0\}, S)} \quad \texttt{if } q \text{ is a } k\text{-chaining of } p \text{ and } \mathit{Pols}(R \cup S \cup E \cup E^-)$$

$$\frac{(E, R, S \cup \{p \geq 0\})}{(E, R, S \cup \{p \geq 0, q \geq 0\})} \quad \texttt{if } q \text{ is a } k\text{-chaining of } p \text{ and } \mathit{Pols}(R \cup S \cup E \cup E^-)$$

$$\frac{(E, R, S \cup \{p \geq 0\})}{(E, R \cup \{q > 0\}, S \cup \{p \geq 0\})} \quad \texttt{if } q \text{ is a } k\text{-chaining of } p \text{ and } \mathit{Pols}(R)$$

Here $\mathit{Pols}(ERS) = \{p : p \rhd 0 \in ERS, \rhd \in \{\approx, >, \geq\}\}$ and $E^- = \{-p \approx 0 : p \approx 0 \in E\}$. In affect, each equation $p \approx 0$ is being treated here as the conjunction of two inequalities $p \geq 0$ and $-p \geq 0$.

Note that k is a parameter in the inference rules above. The instances where $k = 1$ are applied preferably. Again, for purposes of efficiency, the implementation only considers k-chaining for $k \leq 3$.

Example 2. Let $R = \{-x_1 x_2 + 5 > 0,\ x_1 - 2 > 0,\ x_2 - 4 > 0\}$ and $E = S = \emptyset$. The polynomial $-x_1 x_2 + 5 + (x_1 - 2)(x_2 - 4)$, which normalizes (by the convergent rewrite system of Section 3) to $-4x_1 - 2x_2 + 3$, is a 2-chaining of $-x_1 x_2 + 5$ and $\mathit{Pols}(R)$. Thus, we deduce the inequality $-4x_1 - 2x_2 + 3 > 0$. A 1-chaining of this new inequality and $\mathit{Pols}(R)$ gives $-4x_1 - 2x_2 + 3 + 4(x_1 - 2) > 0$, which simplifies to $-2x_2 - 5 > 0$. A yet another 1-chaining results in $-2x_2 - 5 + 2(x_2 - 4) > 0$, which normalizes to $-13 > 0$, which is detected as inconsistent by Section 4.3 inference rules.

An important point to note here is that $k = 1$ is sufficient for *linear* formulas, which is an efficiently decidable fragment of the quantifier-free theory of the reals. The specialization of the first of these three inference rules for the linear case can be written as:

$$\frac{(E, R' = R \cup \{c_0 x + p > 0,\ -d_0 x + q > 0\}, S)}{(E, R' \cup \{d_0 p + c_0 q > 0\}, S)} \quad \texttt{if } c_0 > 0, d_0 > 0, \{x\} \succ^m \{p, q\}$$

Note that the ordering restriction constrains x to be the highest precedence variable. In this linear case, the decision procedure obtained using the above inference rules mimics the well-known Fourier-Motzkin decision procedure for linear rational arithmetic [6]. For nonlinear formulas too $k = 1$ is tried in preference to higher values for k. In this way, even on nonlinear formulas, our decision procedure first checks whether it can

detect possible inconsistencies by "ignoring the nonlinearity", or in other words, abstracting the nonlinear parts by new "linear" terms (implicitly).

The chaining inference rules increase the cardinality of $R \cup S$. Note, however, that the deduced polynomial is smaller than the original polynomial $c_0 s_0 + p$ (in the ordering $\succ$). But still, the set of inference rules described above do *not* terminate. This was a surprising result, but it is a consequence of the fact that the coefficient domain, the set of rational numbers, is not finite. The procedure emulates certain iterative root finding algorithms from numerical analysis.

Example 3. Let $R = \{-x^2 + 4x - 4 > 0, x > 0\}$, $E = \emptyset$, and $S = \emptyset$. A 2-chaining of $-x^2 + 4x - 4$ and $Pols(R)$ is the polynomial $4x - 4$, and hence we deduce $4x - 4 > 0$, or $x - 1 > 0$. Now, a 2-chaining polynomial of $-x^2 + 4x - 4$ and $Pols(R \cup \{x - 1 > 0\})$ is the polynomial $2x - 3$. Thus, starting with $x > 0$, we have now obtained $x > 1$ and $x > 3/2$. The next iteration of inference gives $-x^2 + 4x - 4 + (x - 3/2)(x - 3/2) > 0$ which simplifies to $x > 7/4$.

Continuing this way, from $-x^2 + 4x - 4 > 0$ and $x > c$, we can deduce $x > c'$ where $c' = (4 - c^2)/(4 - 2c)$. The value of c' is always less than 2. Hence, we can deduce formulas of the form $x > c$, with c getting successively closer to 2, but never really becoming 2. Thus, the inference rules do not terminate and fail to detect the inconsistency of the original inequalities.

The inference system is clearly sound. We force termination by explicitly bounding the number inferences in a single run of the saturation procedure. We also use strong heuristics and deletion of redundant rules in the procedure. In most examples, the strong heuristics and redundancy elimination rules also ensure termination of the saturation procedure. But, in some rare cases, forcing an explicit termination is required.

4.3 Inconsistency and Redundancy

The decision procedure terminates when it detects an inconsistency. The following inference rules detect trivial inconsistencies.

$$\frac{(E \cup \{c_0 \approx 0\}, R, S)}{\bot} \quad \texttt{if } c_0 \neq 0$$

$$\frac{(E, R \cup \{c_0 > 0\}, S)}{\bot} \quad \texttt{if } c_0 \leq 0$$

$$\frac{(E, R, S \cup \{c_0 \geq 0\})}{\bot} \quad \texttt{if } c_0 < 0$$

Removal of redundant rules is crucial for efficiency of the decision procedure. Equations of the form $0 \approx 0$, and strict inequalities of the form $c > 0$, where c is indeed greater than 0, and inequalities of the form $c \geq 0$, where c is nonnegative are redundant and can be deleted. However, we can a use stronger definition of redundancy and delete more rules.

The removal of redundant equations is implicit in the formulation of the inference rules above. The simplification rules introduce new equations and delete the old redundant equation simultaneously. Hence, we only need to define the notion of redundancy on inequalities.

Definition 3. *Trivial inequalities of the form $c > 0$ (respectively $d \geq 0$) are redundant if the constant c (d) is positive (nonnegative). A rule $q > 0$ (respectively $r \geq 0$) is redundant in the state $(E = \{p_i \approx 0 : i = 1, 2, \ldots\}, R = \{q_j > 0 : j = 1, 2, \ldots\}, S = \{r_k \geq 0 : k = 1, 2, \ldots\})$ if $q \rightarrow_E^* q'$ and $q' = \Sigma_j c_j q_j + \Sigma_k d_k r_k + e$ for some nonnegative constants c_j's and d_k's, and positive constant e.*

The following two inference rules remove redundant inequalities.

$$\frac{(E, R \cup \{p > 0\}, S)}{(E, R, S)} \quad \texttt{if } p > 0 \text{ is redundant in } (E, R, S)$$

$$\frac{(E, R, S \cup \{p \geq 0\})}{(E, R, S)} \quad \texttt{if } p \geq 0 \text{ is redundant in } (E, R, S)$$

Note that strong notions of redundancy can possibly cause incompleteness in deductive systems. But since we make no claims about completeness, the "correctness" of the definition of redundancy is really vacuous. Nevertheless, we briefly motivate Definition 3 here. As described above, an equation $c_0 s_0 + p \approx 0$ in E can be used as rewrite rule $c_0 s_0 \rightarrow -p$. The inequalities in $R \cup S$ can also be used as rewrite rules in the same way, but restricted somehow. Unlike the semantic equality relation $\approx$ over reals, the semantic relations $>$ and $\geq$ are not *congruences* and hence these rewrite rules cannot be used inside arbitrary contexts. In particular, if $c_0 s_0 > \Sigma_i c_i s_i$, (equivalent representation for $c_0 s_0 + \Sigma_i - c_i s_i > 0$) then it is not the case that $c_0 s_0 q > \Sigma_i c_i s_i q$, for arbitrary polynomial q. But this is true for some restricted q, for example, when q is a positive rational. Thus, if we use $R \cup S$ inequalities as rewrite rules, but restricted to within the context of a positive rational, and use the equations in E in the usual way, then we can construct *rewrite derivations*, or a *proof* of inequalities $q > 0$ (or $r \geq 0$). Now, an inequality $q > 0$, or $r \geq 0$, is defined as redundant in Definition 3 if there is such a rewrite proof for it using $E \cup R \cup S$ and the trivially redundant inequalities. In the spirit of proof simplification and proof orderings [1], we can view the process of saturation as simplifying proofs until they are all in some normal form–valley proofs for equations and rewrite proofs for inequalities. Thus, any inequality which already has a simpler normal form proof using existing other inequalities and equations can be deleted.

Definition 3 is not easy to apply in its general form. Therefore, just like the k-chaining inference rule, we only search for restricted kinds of redundant inequalities. More specifically, we first fully simplify q by the equations in E and then check if the resulting polynomial, say q', can be written in one of the following forms: e, $c_j q_j + e$, $d_k r_k + e$, or $c_j q_j + d_k r_k + e$ for some j, k, and positive (or, nonnegative, depending on the case) e.

Example 4. The inequality $2x_1x_2 - 2x_1 - 5 > 0$ is redundant in the state $(\emptyset, R = \{x_1x_2 - 5 > 0,\ -x_1 + 2 > 0\}, \emptyset)$ because $2x_1x_2 - 2x_1 - 5 = 2(x_1x_2 - 5) + 2(-x_1 + 2) + 1$ and $1 > 0$.

4.4 Heuristics

We use some additional heuristics to limit the search space (the number of new rules in the sets R and S). The most crucial heuristic currently in use limits the number of inequalities with the same leading power-product and identical coefficient sign that can be present simultaneously in the sets R and S. In the default setting, this limit is set to 2.

$$\frac{(E, R \cup \{c_0 s_0 + p > 0,\ c_0' s_0 + p' > 0,\ c_0'' s_0 + p > 0\}, S)}{(E, R \cup \{c_0 s_0 + p > 0,\ c_0' s_0 + p' > 0\}, S)} \quad \texttt{if } sign(c_0) = sign(c_0') = sign(c_0'')$$

$$\frac{(E, R, S \cup \{c_0 s_0 + p \geq 0,\ c_0' s_0 + p' \geq 0,\ c_0'' s_0 + p \geq 0\})}{(E, R, S \cup \{c_0 s_0 + p \geq 0,\ c_0' s_0 + p' \geq 0\})} \quad \texttt{if } sign(c_0) = sign(c_0') = sign(c_0'')$$

If one of the rules is redundant, then it is deleted. But if none is redundant, then the choice of rule to be removed (in the above inference rule) is arbitrary.

The decision procedure also uses a special inference rule based on symbolic computation of upper and lower bounds of polynomial expressions. If numerical lower and upper bounds of certain variables are known, then symbolic lower and upper bounds of polynomial expressions can be computed, which would have a lower degree than the original polynomial.

$$\frac{(E, R \cup \{p + c x_i \mu > 0\}, S)}{(E, R \cup \{p + c x_i \mu > 0, p + c d_i \mu > 0\}, S)} \quad \texttt{if } c > 0, (R \cup S) \models 0 \leq x_i \leq d_i$$

The check $(R \cup S) \models 0 \leq x_i \leq d_i$ is done by simply going through the set $R \cup S$ and maintaining a numerical lower and upper bound for x_i. The rule can be suitably modified if the coefficient c is negative, or when the inequality is in the set S.

The inference system is clearly sound. Every rule or equation in the sets E, R, and S is a consequence (in the theory of reals) of the formulas in the initial set.

Proposition 1. *Let (E, R, S) be an initial state consisting of equations E and inequalities R and S. If $\perp$ can be reached from the initial state using the given inference rules, then the set of atomic formulas $E \cup R \cup S$ (interpreted as a conjunction) is unsatisfiable in the theory of reals.*

As mentioned above, termination is explicitly enforced. The inference systems is clearly not complete, as shown by Example 3.

5 Features of the Decision Procedure

Some of the unique and important features of the decision procedure described here are that it is logical, incremental, and produces witnesses. Having a logical decision procedure is important for purposes of integration with other decision procedures and other theorem proving components.

Saturation based procedures are incremental. Equations and inequalities are asserted one at a time in a context specified by the triple (E, R, S). The result of the assertion is

a new triple (E', R', S') (or $\perp$ and a witness, see below) that is obtained by saturating the existing set (E, R, S) and the new assertion under the above inference rules. Only inferences with the newly asserted fact need to be computed.

5.1 Witness

The decision procedure also keeps track of witnesses or proofs for derived formulas. Each atomic formula is represented as a structure containing a polynomial, an operator ($\approx$, $>$, or $\geq$), and a *witness*. A witness is a list of pointers to other atomic formulae in the set $E \cup R \cup S$. Each application of a (deductive) inference rule creates a new atomic formula and stores a witness with it (a pointer to the atomic formulas in the antecedent of the rule).

The witness facility is primarily used by the feasibility checker of the hybrid abstractor, which is described in detail below.

6 Application Domain

The set of inference rules described in Section 4 are used to obtain a terminating procedure for deciding satisfiability of conjunctions of atomic formulas over the theory of reals. Presently, the decision procedure is used as a black box, embedded inside an automated abstraction tool for hybrid systems. The proof obligations generated by the hybrid abstraction tool range from fairly simple obligations to quite complex formulas depending on the particular hybrid system specification. They could be nonlinear in general.

The hybrid abstractor uses the decision procedure in two different ways. First, proof obligations are generated in the process of abstracting the (continuous and discrete) dynamics of the (hybrid) system. When restricted to discrete transition systems, the proof obligations are similar to those generated by a predicate abstraction tool [8]. The proof obligations arising from abstracting the continuous dynamics are similar too.

The hybrid abstractor also uses the decision procedure to identify which abstract states are actually feasible. This step, which can be obliterated while constructing discrete transition system abstractions [8], becomes crucial for hybrid systems. The witness generator is used here for efficiency. When an abstract state is identified as infeasible (by the decision procedure), the decision procedure also returns a witness for the inconsistency and thus the abstraction tool can infer infeasibility of a *set* of abstract states. The check for infeasibility can be done at the time of constructing the abstraction, or during model-checking [15]. For examples of hybrid systems that are handled by the hybrid abstractor and decision procedure, the reader is referred to [7, 14].

Compared to the QEPCAD tool, which we first used for experiments, the new sound, but incomplete, procedure is faster and more efficient. We present here some results of experiments comparing the QEPCAD tool with our decision procedure. All test examples consisted of conjunction of atomic formulas, with all variables assumed to be existentially quantified. Table 1 gives the example formulas and Table 2 gives the running time of the QEPCAD procedure and the decision procedure (DP) described

in this paper. DP correctly decides each of the formulas in this test bench and is considerably faster. QEPCAD (Version 13) fails on two. All experiments were done on a personal desktop i686 machine running Linux.

	Atomic formulas
Test1	$\{x_c - x_d > 0, x_c > 0, x_d > 0, x_a > 0, x_b > 0, x_a > 0, x_a x_b + x_a x_c - 10 < 0\}$
Test2	$\{2a_p - 3 = 0, 2b_p - 1 > 0, 4a - 1 > 0, 1 - 2e > 0, v7 > 0, v_{ba} > 0, v_{bap} > 0, 3v_{ba} - 8v_7 - 8v_{bap} > 0, ap \geq 0, bp \geq 0, a \geq 0, e \geq 0\}$
Test3	$\{r_a + 5r_b^2 - 20r_b = 0, 25r_b^3 - 100r_b^2 + 20r_b - 1 = 0, r_a > 0, r_b > 0, 4 - v > 0, v + 4 > 0, 1 - a > 0, h - 1 > 0, a + 1 > 0, 10 - g > 0, g - 4a - 4v > 0, 1 - 2h > 0, g - r_a v - r_b a = 0\}$
Test4	$\{l^3 + 3l^2 + 4l + 1 = 0, r_a = l^2 + 3l^2, r_b = l, 0 < 2l + 1, 4l + 1 < 0, v_f > 0, v > 0, 0 < a + 2, v_f - v > 0, g + 10 > 0, g - r_a v_f + r_a v - v_f + r_b a > 0\}$
Test5	$Test4 \cup \{g + 10 < 0\}$

Table 1. Atomic formulas for the test examples.

The experimental results reported in Table 2 comparing the two decision procedures should be interpreted in the right context. First, the test bench consists of formulas of a very restricted kind. The QEPCAD decision procedure is much more general and can handle arbitrary first-order formulas, whereas DP cannot. Furthermore, it is quite easy to create very small examples to expose the incompleteness of DP, while QEPCAD works just fine on them. Second, the QEPCAD decision procedure was run in its interactive mode, that is, the input was typed in by hand, in the input format of QEPCAD. The DP procedure is written in Lisp and the input was provided in Lisp. The decision procedure code was compiled and loaded into Lisp.

But despite the unfairness, it is amply clear that QEPCAD is not well suited for applications where termination and soundness are essential, and "logical" aspects are crucial. Test 5 combines a trivial inconsistency $gap + 10 > 0,\ gap + 10 < 0$ with a large formula. But, QEPCAD fails on this "logically simple" example.

	#variables	degree	status	QEPCAD	DP
Test1	4	2	satisfiable	30	1
Test2	7	3	inconsistent	20	1
Test3	6	3	inconsistent	4240	2
Test4	7	3	inconsistent	FAIL	5
Test5	7	3	inconsistent	FAIL	5

Table 2. Total running time averages (in milliseconds) for Tests1–5.

7 Conclusion

Motivated by applications that need embedded symbolic reasoning component (for *enhancing* their quality), we argue for the need for abstraction based theorem proving. The correct abstraction (degree of incompleteness) is application dependent and it is a question of pragmatics as it offers a time versus accuracy tradeoff. In this article, we investigated a sound, incomplete, and terminating procedure for checking satisfiability of formulas over the theory of real closed fields. This has proved very useful for the application domain of hybrid abstraction.

As part of future work, it will be interesting to study additional inference rules that can be added to the above set without compromising efficiency. Characterizing the theory for which such a procedure is complete is also intriguing. One also wonders if the current set of inference rules can be extended to give a *complete* and logical decision procedure for the quantifier-free theory of reals.

The long term goal of this research is to obtain an efficient and practical implementations of many different *logical* algorithms for deciding (certain classes of) formulas over the theory of real closed fields. We envision a collection of procedures that successively decide larger and larger fragments of the theory and that interact with each other based on abstraction and refinement.

Acknowledgements. The author wishes to thank Hassen Saïdi for help in concretizing some ideas presented in this paper.

References

1. L. Bachmair. *Canonical Equational Proofs*. Birkhäuser, Boston, 1991.
2. L. Bachmair and H. Ganzinger. Rewrite techniques for transitive relations. In *9th Annual IEEE Symposium on Logic in Computer Science*, pages 384–393, 1994.
3. L. Bachmair and A. Tiwari. D-bases for polynomial ideals over commutative noetherian rings. In H. Comon, editor, *Rewriting Techniques and Applications, RTA 1997*, volume 1103 of *Lecture Notes in Computer Science*, pages 113–127, Sitges, Spain, July 1997. Springer-Verlag.
4. T. Becker and V. Weispfenning. *Gröbner bases: a computational approach to commutative algebra*. Springer-Verlag, Berlin, 1993.
5. G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Proc. Second GI Conf. Automata theory and formal languages*, pages 134–183, 1975. Vol. 33 of *Lecture Notes in Comp. Sci., Springer, Berlin*.
6. G. B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *J. of combinatorial theory (A)*, 14:288–297, 1973.
7. R. Ghosh, A. Tiwari, and C. Tomlin. Automated symbolic reachability analysis with application to delta-notch signaling automata. In O. Maler and A. Pnuelu, editors, *Hybrid Systems: Computation and Control HSCC*, volume 2623 of *LNCS*, pages 233–248. Springer, April 2003.
8. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. of 9th Conference on Computer-Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
9. H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *Proc. ISAAC 90*, pages 261–264, 1990.

10. H. Hong. Quantifier elimination in elementary algebra and geometry by partial cylindrical algebraic decomposition version 13. In *The world wide web*, 1995. `http://www.gwdg.de/∼cais/systeme/saclib`, `www.eecis.udel.edu/∼saclib/`.
11. D. Lazard. *An improved projection for cylindrical algebraic decomposition*. Technical Report, Informatique, Universite Paris IV, F-75252 Paris Cedex 05, France, 1990.
12. S. McCallum. An improved projection operator for cylindrical algebraic decomposition of three dimensional space. *J. Symbolic Computation*, 5:141–161, 1988.
13. A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948. Second edition.
14. A. Tiwari. Approximate reachability for linear systems. In O. Maler and A. Pnuelu, editors, *Hybrid Systems: Computation and Control HSCC*, volume 2623 of *LNCS*, pages 514–525. Springer, April 2003.
15. A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In C. J. Tomlin and M. R. Greenstreet, editors, *Hybrid Systems: Computation and Control, HSCC 2002*, volume 2289 of *LNCS*, pages 465–478, 2002.

Simplifying OBDDs in Decidable Theories
—Extended Abstract—

Alessandro Armando

MRG-Lab
DIST, University of Genova
armando@dist.unige.it

1 Introduction

In the recent years there has been a number of attempts to extend OBDD-technology beyond the realm of propositional logic. The motivation for this endeavour stems from the need for greater expressivity posed by new, important verification tasks. For instance [6] shows that the correctness of pipelined microprocessors control can be expressed in the quantifier-free fragment of the theory of equality with uninterpreted function symbols. This result stimulated the design of efficient decision procedures for this fragment [2,4,5,10]. More recently, the focus has been shifted to the design of decision procedures for other decidable theories. Particular attention has been devoted to *difference logic*, that is the quantifier-free fragment of Presburger arithmetics where atomic formulae can only have the form $x - y \leq n$ (x and y are real-valued variables and n is a numeric constant). Difference Decision Diagrams (DDDs) [8] are similar to Ordered Binary Decision Diagrams (OBDDs) with the main difference being that difference constraints are used in place of boolean variables. SAT-based decision procedures for difference logic are presented in [1, 11]. An approach to building proof procedures by combining OBDDs and superposition theorem proving is introduced in [3].

In this paper I present a generic approach to building decidable theories into OBDDs. I start in Section 2 by presenting a simplified algorithm for the construction of OBDDs. In Section 3, I introduce a first simple procedure, $\textsf{offline-obdd}_T$, based on the idea of simplifying OBDDs off-line using a satisfiability procedure for a (decidable) background theory T. I then present the main procedure, $\textsf{online-obdd}_T$, which is based on the idea of simplifying OBDDs on-the-fly. The procedure builds OBDDs in a bottom-up fashion (thereby retaining much of the efficiency of standard OBDD technology) and uses the available satisfiability procedure to simplify the OBDD currently being built. The advantages of the approach are manyfold:

1. It combines the efficiency and conciseness of standard OBDD technology with the added expressivity provided by the available satisfiability procedure.
2. It is *generic*, i.e. independent from the theory decided by the satisfiability procedure; this means that new satisfiability procedures (possibly for different decidable theories) can be easily incorporated into the proposed procedure as soon as they become available.
3. The generated OBDDs are *semi-canonical representations*, i.e. there is exactly one OBDD for tautology (the constant `T`) and exactly one OBDD for contradiction (the constant `F`). It is worth pointing out that full canonicity is lost in this general setting.

Finally, in Section 4, I present some experimental results that clearly show that $\mathsf{online\text{-}obdd}_T$ performs significantly better than $\mathsf{offline\text{-}obdd}_T$ on non-trivial instances and that it scales up smoothly to even very big instances.

2 Building OBDDs

Let $\mathtt{T}$ and $\mathtt{F}$ be the propositional constants for truth and falsity respectively and let L be a set of propositional variables equipped with a total ordering $\prec$. A propositional letter is either a propositional constant or a propositional variable. A *literal* is either a propositional letter or the negation of a propositional letter. We consider a propositional language where formulae are built out of propositional letters using the usual logical connectives (i.e. $\neg$, $\vee$, $\wedge$, $\supset$, $\leftrightarrow$, $\ldots$) and the (ternary) logical connective $(_ \Rightarrow _; _)$ called "if-then-else"; thus if α, β, and γ are propositional formulae, then also $(\alpha \Rightarrow \beta; \gamma)$ is. The semantics of the "if-then-else" is given by the logical equivalence $(\alpha \Rightarrow \beta; \gamma) \leftrightarrow ((\alpha \wedge \beta) \vee (\neg\alpha \wedge \gamma))$.

Following [9], OBDDs are modeled as propositional formulae inductively defined in the following way. The propositional constants $\mathtt{T}$ and $\mathtt{F}$ are OBDDs; if t and f are OBDDs and c is a propositional variable $\prec$-smaller than the propositional variables occurring in t and f, then $(c \Rightarrow t; f)$ is an OBDD. An OBDD is *reduced* iff there are no OBDDs of the form $(c \Rightarrow o; o)$ in it. Any OBDD can be always put in reduced form by exhaustively applying the rewrite rule

$$(V \Rightarrow O; O) \mapsto O$$

For instance, if $A, B, C, \ldots$ are propositional variables and $\prec$ is the standard lexicographic ordering, then $(A \Rightarrow (C \Rightarrow \mathtt{F}; \mathtt{F}); (B \Rightarrow \mathtt{T}; (C \Rightarrow \mathtt{T}; \mathtt{F})))$ is an OBDD but it is not reduced, whereas $(A \Rightarrow \mathtt{F}; (B \Rightarrow \mathtt{T}; (C \Rightarrow \mathtt{T}; \mathtt{F})))$ is a reduced OBDD. Furthermore, neither $(A \Rightarrow (C \Rightarrow \mathtt{F}; \mathtt{T}); (C \Rightarrow \mathtt{T}; (B \Rightarrow \mathtt{T}; \mathtt{F})))$ nor $(\mathtt{T} \Rightarrow (C \Rightarrow \mathtt{F}; \mathtt{T}); \mathtt{T})$ are OBDDs. From here on by the term OBDD we mean a reduced OBDD.

We say that ω *is an OBDD for* ϕ iff ω and ϕ are logically equivalent (i.e. they have the same truth-value w.r.t. all truth-value assignments), in symbols $\models (\omega \leftrightarrow \phi)$, where $\models$ is the classical entailment relation. In the above definition we can say "the OBDD" in place of "an OBDD" because OBDDs enjoy the property of being canonical representations of the boolean functions they represent.

Figure 2 provides an abstract account of an algorithm for constructing OBDDs based on the rational reconstruction given in [9]. Given a propositional formula w, $\mathsf{obdd}(w)$ computes and returns the OBDD for w. For instance

$$\mathsf{obdd}(((a \wedge b) \leftrightarrow (a \leftrightarrow b))) = (a \Rightarrow \mathtt{T}; (b \Rightarrow \mathtt{T}; \mathtt{F})) \tag{1}$$

Notice that reduction is implemented by the first clause of mkif.

3 Building decidable theories into OBDDs

Let T be a decidable (first-order) theory of language $\mathcal{L}_T$, called *background theory*. We say that ω *is an OBDD for the formula* $\phi \in \mathcal{L}_T$ *in* T if and only if $T \models (\omega \leftrightarrow \phi)$.

$$
\begin{aligned}
\mathsf{obdd}(\mathtt{T}) &= \mathtt{T} \\
\mathsf{obdd}(\mathtt{F}) &= \mathtt{F} \\
\mathsf{obdd}(c) &= \mathsf{mkif}(c, \mathtt{T}, \mathtt{F}) \quad \text{if } c \text{ is a pc} \\
\mathsf{obdd}(t_1 \; op \; t_2) &= \mathsf{merge}(op, \mathsf{obdd}(t_1), \mathsf{obdd}(t_2)) \\
\\
\mathsf{merge}(\wedge, \mathtt{F}, y) &= \mathtt{F} \\
\mathsf{merge}(\wedge, x, \mathtt{F}) &= \mathtt{F} \\
\mathsf{merge}(\wedge, \mathtt{T}, y) &= y \\
\mathsf{merge}(\wedge, y, \mathtt{T}) &= y \\
\mathsf{merge}(\supset, \mathtt{F}, y) &= \mathtt{T} \\
\mathsf{merge}(\supset, \mathtt{T}, y) &= y \\
\mathsf{merge}(\supset, x, \mathtt{T}) &= \mathtt{T} \\
\mathsf{merge}(\supset, x, \mathtt{F}) &= \mathsf{negate}(x) \\
&\vdots \\
\mathsf{merge}(op, (c \Rightarrow t_1 ; f_1), (c \Rightarrow t_2 ; f_2)) &= \mathsf{mkif}(c, \mathsf{merge}(op, t_1, t_2), \mathsf{merge}(op, f_1, f_2)) \\
\mathsf{merge}(op, (c_1 \Rightarrow t_1 ; f_1), y) &= \mathsf{mkif}(c_1, \mathsf{merge}(op, t_1, y), \mathsf{merge}(op, f_1, y)) \\
& \text{ if } y = (c_2 \Rightarrow t_2 ; f_2) \text{ and } c_1 \prec c_2 \\
\mathsf{merge}(op, x, (c_2 \Rightarrow t_2 ; f_2)) &= \mathsf{mkif}(c_2, \mathsf{merge}(op, x, t_2), \mathsf{merge}(op, x, f_2)) \\
& \text{ if } x = (c_1 \Rightarrow t_1 ; f_1) \text{ and } c_2 \prec c_1 \\
\\
\mathsf{mkif}(\mathtt{T}, t, _) &= t \\
\mathsf{mkif}(\mathtt{F}, _, f) &= f \\
\mathsf{mkif}(c, o, o) &= o \\
\mathsf{mkif}(c, t, f) &= (c \Rightarrow t ; f) \quad \text{if } t \neq f \\
\\
\mathsf{negate}(\mathtt{T}) &= \mathtt{F} \\
\mathsf{negate}(\mathtt{F}) &= \mathtt{T} \\
\mathsf{negate}((c \Rightarrow t ; f)) &= (c \Rightarrow \mathsf{negate}(t) ; \mathsf{negate}(f))
\end{aligned}
$$

Note: Non-determinism is resolved by selecting the first applicable equation.

Fig. 1. Basic OBBD algorithm

$$\begin{aligned}
\mathsf{simplify}_T(C, \mathtt{T}) &= \mathtt{T} \\
\mathsf{simplify}_T(C, _) &= \mathtt{T} \quad \text{if } T \cup C \text{ is unsatisfiable} \\
\mathsf{simplify}_T(C, \mathtt{F}) &= \mathtt{F} \\
\mathsf{simplify}_T(C, (c \Rightarrow t; f)) &= \mathsf{cxt\text{-}mkif}(c, \mathsf{simplify}_T(C \cup \{c\}, t), \mathsf{simplify}_T(C \cup \{\neg c\}, f)) \\
\\
\mathsf{cxt\text{-}mkif}(_, \mathtt{F}, f) &= f \\
\mathsf{cxt\text{-}mkif}(_, t, \mathtt{F}) &= t \\
\mathsf{cxt\text{-}mkif}(C, t, f) &= \mathsf{mkif}(C, t, f)
\end{aligned}$$

Note: Non-determinism is resolved by selecting the first applicable equation.

Fig. 2. A procedure for off-line simplification of OBDDs

Notice that the OBDD for ϕ is an OBDD for ϕ in any theory. For instance, let T be Presburger arithmetics over the reals and ϕ be the formula

$$(2 * x \leq y \supset (y \leq 0 \supset x \leq 0)) \tag{2}$$

and let $\prec$ be such that $(2 * x \leq y) \prec (y \leq 0) \prec (x \leq 0)$, then an OBDD for (2) in T is

$$(x \leq 0 \Rightarrow \mathtt{T}; (y \leq 0 \Rightarrow (2 * x \leq y \Rightarrow \mathtt{F}; \mathtt{T}); \mathtt{T})) \tag{3}$$

However, also $\mathtt{T}$ is an OBDD for (2) in T. As the above example clearly shows, we have lost the canonicity property of OBDDs. On the other hand in the above example it is possible to simplify (3) to $\mathtt{T}$ by using the properties of the background theory.

There are several approaches to building a decidable theory T into OBDDs. Given a formula $\phi \in \mathcal{L}_T$, the simplest approach amounts to building the OBDD ω of ϕ and then simplifying ω w.r.t. T yielding a simplified OBDD ω'. We call this approach *off-line simplification*. An alternative, more sophisticated approach amounts to simplifying on-the-fly the OBDD during its construction, thereby directly yielding the simplified OBDD. We call this approach *on-line simplification*.

A simple algorithm that can be used to carry out an off-line simplification of OBDDs w.r.t. a given decidable theory T is given in Figure 2. C is a set of literals and plays the role of the context of simplification. Notice that C is incrementally extended in the obvious way as the OBDD is transversed. This simplification procedure is essentially the one presented in [8]. It is easy to see that the procedure is both sound and terminating. Moreover it can be easily shown that the simplified OBDDs returned by the procedure enjoy the property that each path from the root to the leaf is T-satisfiable and therefore $\mathtt{F}$ is the only contradictory OBDD and $\mathtt{T}$ is the only tautological one. Therefore the OBDDs returned by the procedure are semi-canonical representations.

An algorithm for computing OBDDs w.r.t. a given theory T can then be defined as follows:

$$\mathsf{offline\text{-}obdd}_T(w) = \mathsf{simplify}_T(\emptyset, \mathsf{obdd}(w)) \tag{4}$$

where obdd is as defined in Section 2. For instance, by applying $\mathsf{offline\text{-}obdd}_T$, (2) gets simplified to $\mathtt{T}$.

$$
\begin{aligned}
\text{online-obdd}_T(\mathtt{T}) &= \mathtt{T} \\
\text{online-obdd}_T(\mathtt{F}) &= \mathtt{F} \\
\text{online-obdd}_T(c) &= \text{simplify}_T(\text{mkif}(c, \mathtt{T}, \mathtt{F})) \quad \text{if } c \text{ is a pc} \\
\text{online-obdd}_T(t_1 \; op \; t_2) &= \text{cmerge}(op, \emptyset, \text{online-obdd}_T(t_1), \text{online-obdd}_T(t_2)) \\
\\
\text{cmerge}(\wedge, C, x, y) &= \mathtt{F} \quad \text{if } x = \mathtt{F} \text{ or } y = \mathtt{F} \\
\text{cmerge}(\wedge, C, \mathtt{T}, y) &= \text{simplify}_T(C, y) \\
\text{cmerge}(\wedge, C, x, \mathtt{T}) &= \text{simplify}_T(C, x) \\
\text{cmerge}(\supset, C, x, y) &= \mathtt{T} \quad \text{if } x = \mathtt{F} \text{ or } y = \mathtt{T} \\
\text{cmerge}(\supset, C, \mathtt{T}, y) &= \text{simplify}_T(C, y) \\
\text{cmerge}(\supset, C, x, \mathtt{F}) &= \text{simplify}_T(\text{negate}(x)) \\
&\vdots \\
\text{cmerge}(op, C, (c \Rightarrow t_1 \,;\, f_1), (c \Rightarrow t_2 \,;\, f_2)) &= \text{cmerge}(op, C, t_1, t_2) \quad \text{if } C \models_T c \\
\text{cmerge}(op, C, (c \Rightarrow t_1 \,;\, f_1), (c \Rightarrow t_2 \,;\, f_2)) &= \text{cmerge}(op, C, f_1, f_2) \quad \text{if } C \models_T \neg c \\
\text{cmerge}(op, C, (c \Rightarrow t_1 \,;\, f_1), (c \Rightarrow t_2 \,;\, f_2)) &= \text{mkif}(c, \\
&\qquad \text{cmerge}(op, C \cup \{c\}, t_1, t_2), \\
&\qquad \text{cmerge}(op, C \cup \{\neg c\}, f_1, f_2)) \\
\text{cmerge}(op, C, (c_1 \Rightarrow t_1 \,;\, f_1), y) &= \text{cmerge}(op, C, t_1, y) \\
&\quad \text{if } y = (c_2 \Rightarrow _ ; _), c_1 \prec c_2, \text{ and } C \models_T c_1 \\
\text{cmerge}(op, C, (c_1 \Rightarrow t_1 \,;\, f_1), y) &= \text{cmerge}(op, C, f_1, y) \\
&\quad \text{if } y = (c_2 \Rightarrow _ ; _), c_1 \prec c_2 \text{ and } C \models_T \neg c_1 \\
\text{cmerge}(op, C, (c_1 \Rightarrow t_1 \,;\, f_1), y) &= \text{mkif}(c_1, \\
&\qquad \text{cmerge}(op, C \cup \{c_1\}, t_1, y), \\
&\qquad \text{cmerge}(op, C \cup \{\neg c_1\}, f_1, y)) \\
&\vdots
\end{aligned}
$$

Note: Non-determinism is resolved by selecting the first applicable equation.

Fig. 3. A procedure for OBDD construction with on-line simplification.

A procedure for on-line simplification is obtained by incorporating simplification into the basic procedure obdd. The resulting procedure, online-obdd$_T$, is given in Figure 3. As above, C is a set of literals which plays the role of the context of simplification. Similarly to offline-obdd$_T$, online-obdd$_T$ is sound, terminating, and the returned OBDDs are semi-canonical representations.

The advantage of off-line over on-line simplification is that the former does not require any modification to the algorithm used to build OBDDs and hence any state-of-the-art OBDD package can be taken off-the-shelf and used for this purpose. The drawback of the off-line approach is that the intermediate OBDD can be very large and hence both its generation and its simplification can be very time and memory expensive. On-line simplification, by simplifying the OBDD eagerly during construction, does not suffer from this problem and this leads to significant savings as shown in Section 4.

4 Experimental Results

I have implemented a prototype version of the procedures described in the previous sections, namely offline-obdd$_T$ and online-obdd$_T$.[1] Currently only Presburger arithmetics over the reals is supported as a background theory. An incremental satisfiability procedure based on the Fourier-Motzkin elimination method [7] provides the functionalities required by the simplification procedures.

In order to compare the performance of offline-obdd$_T$ and online-obdd$_T$ I have used the set of benchmark problems (introduced in [8]) defined by the following parametric formula:

$$(0 \leq x \leq N \wedge 0 \leq y \leq N) \leftrightarrow \left(\begin{array}{c} \bigvee_{i=1,\ldots,N; j=1,\ldots,N} x \geq i-1 \wedge y \leq j \wedge y - x \geq j - i \\ \vee \\ \bigvee_{i=1,\ldots,N; j=1,\ldots,N} x \leq i \wedge y \geq j-1 \wedge y - x \leq j - i \end{array} \right) \tag{5}$$

for any integer $N > 0$. Notice that (5) is a valid formula for any positive integer N since (by resorting to the associated geometric interpretation) the formula in the right-hand-side represents a partition of the square represented by the left-hand-side formula.

The plots of the time spent by offline-obdd$_T$ and online-obdd$_T$ procedures are depicted in Figure 4.[2] The x-axis represents the values of N whereas the y-axis represents time (in milliseconds) spent by the procedures to complete. The size of the intermediate OBDD generated by offline-obdd$_T$ is plotted in Figure 5. The results clearly indicate that online-obdd$_T$ scales up smoothly to very big instances whereas offline-obdd$_T$ has difficulties in dealing with problems of moderate complexity. Not surprisingly the problem with the offline approach is the size of the intermediate OBDD which grows exponentially as N increases.

5 Conclusions

I this paper have explored the use of satisfiability procedures for decidable theories to simplify OBDDs. I have proposed a procedure that simplifies OBDDs on-the-fly

[1] The procedures are implemented in Prolog. Code is available on request from the author.
[2] Experiments have been carried out using a PC with a 2.4 GHz Processor and 1GB of RAM.

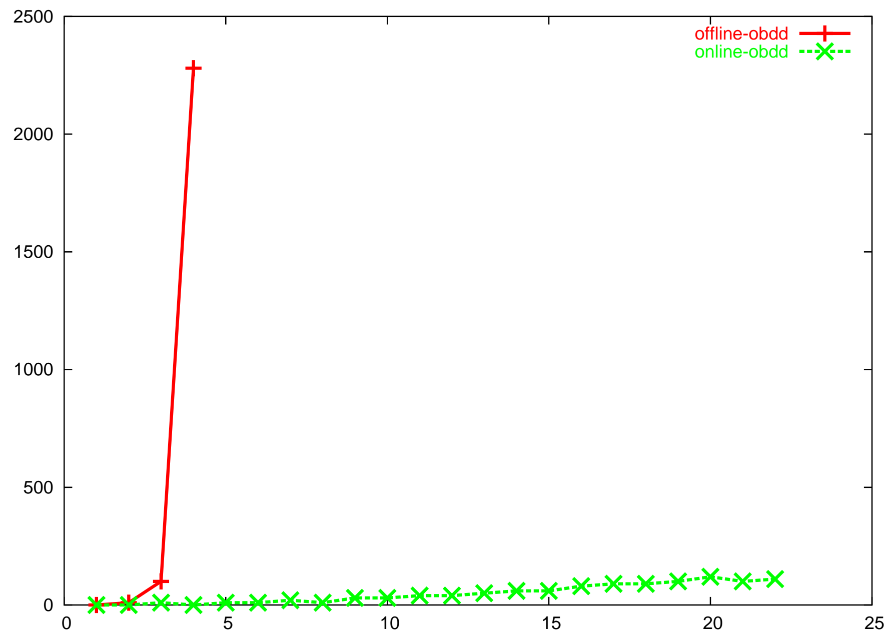


Fig. 4. Offline vs online simplification.

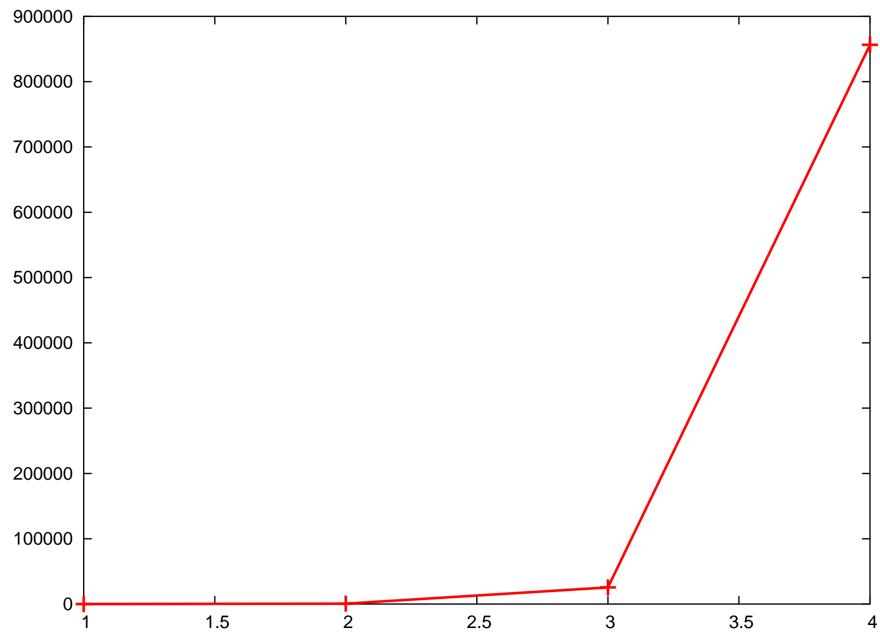


Fig. 5. Size of the intermediate OBDD generated by offline-obdd$_T$.

as opposed to existing approaches (e.g. [8]) that, by simplifying OBDDs a posteriori, generate and simplify (possibly very large) intermediate OBDDs.

References

1. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Lecture Notes in Computer Science*, volume 1809, pages 97–108, 1999.
2. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. 11th International Computer Aided Verification Conference*, pages 470–482, 1999.
3. D. Dharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In *Proc. of 1st International Conference on Software Engineering and Formal Methods (SEFM03)*. IEEE Computer Society Press, 2003. To appear.
4. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In *Proc. 10th International Computer Aided Verification Conference*, pages 244–255, 1998.
5. Jan Friso Groote and Jaco van de Pol. Equational binary decision diagrams. *Lecture Notes in Computer Science*, 1955:161–178, 2000.
6. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.
7. J.-L. Lassez and M.J. Maher. On Fourier's Algorithm's for Linear Arithmetic Constraints. *J. of Automated Reasoning*, 9:373–379, 1992.
8. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, September 1999.
9. J Strother Moore. Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, 12(1):33–45, 1994.
10. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proc. 11th International Computer Aided Verification Conference*, pages 455–469, 1999.
11. Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding separation formulas with SAT. *Lecture Notes in Computer Science*, 2404:209–222, 2002.

Verifying Industrial Hybrid Systems with MathSAT*

Gilles Audemard[1], Marco Bozzano[2], Alessandro Cimatti[2] and Roberto Sebastiani[2,3]

[1] Centre de Recherche en Informatique de Lens
IUT de Lens, Rue de l'université, SP16, F 62307 Lens Cedex
audemard@iut-lens.univ-artois.fr
[2] ITC-IRST, Via Sommarive 18, 38050 Povo, Trento, Italy
{bozzano,cimatti}@itc.it
[3] DIT, Università di Trento, Via Sommarive 14, 38050 Povo, Trento, Italy
rseba@dit.unitn.it

Abstract. Industrial systems of practical relevance can be often characterized in terms of discrete control variables and real-valued physical variables, and can therefore be modeled as hybrid automata. Unfortunately, continuity of the physical behaviour over time, or triangular constraints, must often be assumed, which yield an undecidable class of hybrid automata.

In this paper, we propose a technique for bounded reachability of linear hybrid automata, based on the reduction of a bounded reachability problem to a MATH-SAT problem, i.e. satisfiability of a boolean combination of propositional variables and mathematical constraints. The MathSAT solver can be used to check the existence (or absence) of paths of bounded length.

The approach is very similar in spirit to SAT-based bounded model checking; furthermore, the ability to reason directly about real variables gives computational leverage over discretization-based methods. Despite the undecidability of the general problem, the proposed method is able to provide valuable information on large designs of practical relevance.

1 Introduction

Many systems and plants of industrial relevance (e.g., engines, turbines) are defined in terms of discrete control variables and physical real-valued variables (e.g., speed, pressure), and can be naturally modeled as hybrid automata: depending on a discrete state (e.g., "nominal", "increasing"), different equations describe the behaviour of the physical variable (e.g., speed). Frequently, the dynamics of physical variables is continuous: i.e., transitions from a discrete state to another should not necessarily yield a

* This work has been sponsored by the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. It has also been partly supported by ESACS, an European sponsored project, contract no. G4RD-CT-2000-00361. The fourth author is also sponsored by a MIUR COFIN02 project, code 2002097822_003.

discontinuity in the physical dimension. For instance, in the transition from "increasing" to "decreasing", the velocity should not change its value (but only its derivative). Furthermore, the evolution can depend on the comparison between the values of physical variables. Unfortunately, either imposing continuity or allowing for comparisons between variables (also known as triangular constraints) result in a class of hybrid automata where even reachability is undecidable [HKPV98]. Yet, it is very important to be able to develop tools that allow to formally validate such designs, that often implement critical functionalities (e.g., control systems for avionics).

In this paper, we address the problem of verifying hybrid automata with continuous variables and triangular constraints. We propose a formal verification method for bounded reachability. The approach is based on the encoding of a bounded reachability problem into a MATHSAT problem, i.e. the problem of checking the satisfiability of a boolean combination of propositional variables and mathematical constraints over real variables. The approach is made practical by the use of the efficient MATHSAT solver [ABC+02a], that extends and integrates state-of-the-art techniques for propositional satisfiability (SAT) with a set of mathematical reasoners. The approach presented in this paper is largely similar to bounded model checking [BCCZ99], and enhances the method presented at [ACKS02], limited to timed systems, to dealing with real variables with arbitrary linear dynamics.

The proposed technique is clearly incomplete, and currently limited to the case of linear dynamics. Despite these facts, however, it allows us to represent and to analyze interesting systems from real-world applications [BVÅ+03,BCC+03], providing useful information, especially oriented to debugging and goal-directed simulation. An experimental analysis shows that our techniques is competitive with state of the art verification tools such as HYTECH, and with methods based on the discretization of real variables.

Outline of the paper The rest of the paper is structured as follows. In Section 2 we illustrate a motivating example for our approach; in Section 3 we give a short and informal introduction to our model of hybrid systems; in Section 4 we give a brief overview of SAT-based bounded model checking and we discuss in more detail our encoding of hybrid systems into MATHSAT; in Section 5 we discuss some experiments, and finally in Section 6 and 7 we discuss related work and draw some conclusions.

2 A Motivating Example: the Secondary Power System

Throughout the paper, we use a running example to motivate and illustrate the main concepts we present. Specifically, we discuss the modeling and analysis of a real-world safety-critical system, namely the Secondary Power System (SPS). It is an industrial case study which has been and is being investigated within ESACS (Enhanced Safety Analysis for Complex Systems), a European-Union-sponsored project in the avionics sector, whose goal is to define a methodology to improve the safety analysis practice for complex systems development [BVÅ+03, BCC+03].

The SPS drives the hydraulic and electrical utilities of an aircraft. It is an example of safety-critical system with embedded hardware and software components. The hardware subsystems comprise (electro)-mechanical components (e.g., control valves,

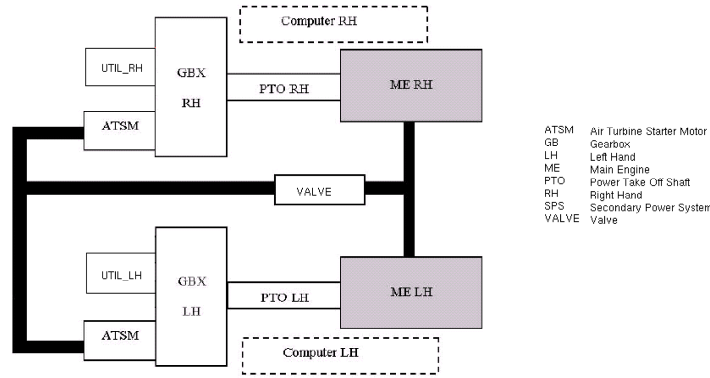


Fig. 1. SPS schematic view

relays, shafts, gearboxes, freewheels) and electronic transducers (e.g., speed and pressure sensors), whereas the software component is given by embedded controllers (SPS computers).

The SPS drives the utilities of both the left and right hand side of the aircraft. To ensure the basic safety requirement, i.e. *no single failures shall cause the total loss of the SPS utilities*, the architecture of the system includes two basic redundancies: there are two independent and perfectly symmetric lines, whose purpose is to drive the left and the right hand side utilities, respectively; for each side, the mechanical drive of the relevant utilities (normal mode) is redounded by a pneumatic drive (*cross-bleed mode*) in case of failure of one of the components in the mechanical line.

Figure 1 shows a simplified schematic view of the SPS. The SPS normal operation consists in transmitting the mechanical power from the engines to the relevant hydraulic and electrical generators. Specifically, the mechanical power of the main engine (ME) is transmitted via the Power Take Off Shaft (PTO) to a gearbox (GBX) which feeds the utilities. A component may fail due to abnormal operational conditions or ruptures. As an example, *flameout* and *grippage* are two possible failure modes of the main engine. To ensure safety of in-flight operation, in case of an engine failure the SPS computers automatically initiate a *cross-bleed* procedure consisting in driving the hydraulic and electrical generators by means of an air turbine motor (ATSM), using bled air from a valve (VALVE), which is in turn fed by the mechanical power coming from the opposite engine. Correct functioning of the cross-bleed procedure is an example of one safety requirement of the SPS. Some experimental results about this will be presented in Section 5.

3 Modeling Hybrid Automata

In this section we briefly present and exemplify our model of hybrid systems. The model is inspired by the *linear* and *rectangular hybrid automata* models presented

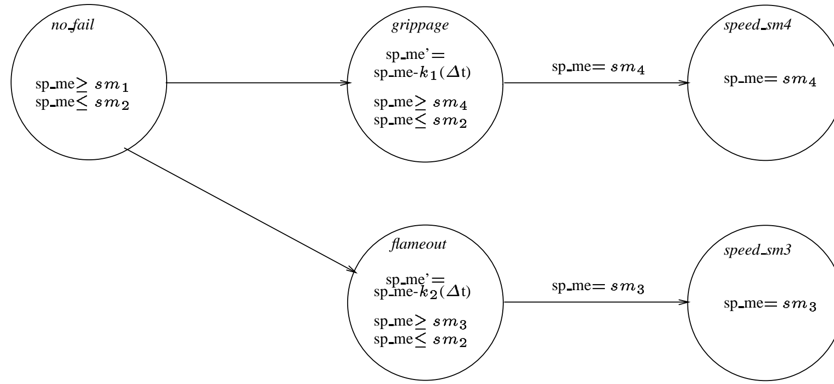


Fig. 2. SPS: main engine automaton (ME)

in [Hen96, HHWT97]. Informally, a hybrid system can be seen as the parallel composition of a collection of hybrid automata, which can communicate either by explicit synchronization on some *channel*, or implicitly by means of *shared variables*. Each automaton models both *discrete* events (e.g., failure of a component) and continuous activities of analog variables (e.g., time, component speed). At any given instant of time, the state of a hybrid automaton is defined by a control location (discrete state) and the values of all the analog variables (continuous state). The state can change either because of an *instantaneous* discrete transition, which changes the control location and may also affect the values of the analog variables (e.g., re-initialization is possible) or because of a *time elapse* (continuous) transition, which changes only the values of the analog variables according to some specified law. Hybrid systems can be seen as an extension of the timed systems model of [ACKS02], in which the only analog variables are clocks. In the following, by *elementary linear expression* we mean an equality and/or (non-strict) inequality over *linear terms* (i.e., linear combinations of real-valued variables with rational coefficients).

Figure 2 and 3 depict two examples of hybrid automata, modeling the *main engine* (ME) and the valve (VALVE) components of the SPS. A hybrid automaton consists of the following components:

Locations A finite set of locations, encoding the discrete states of the hybrid automaton. The automaton in Figure 2 has five locations, drawn as circles, which model the discrete state of the ME of the SPS. Location no_fail models the default behaviour of the engine; locations $grippage$ and $flameout$ model two different faulty states; locations $speed_sm4$ and $speed_sm3$ model states in which the speed of the engine has the constant value sm_4 and sm_3.

Analog Variables A finite vector of real-valued data variables $(w_1, \ldots, w_n)$. The sp_me variable in Figure 2 encodes the speed of the ME. *Clock* variables of [ACKS02] may be seen as a particular case of real-valued variables. Primed variables (e.g., sp_me') are used to denote the value of real-valued variables after execution of a transition.

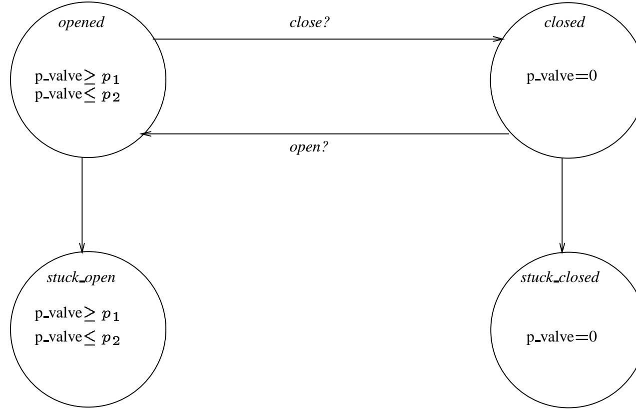


Fig. 3. SPS: valve automaton (VALVE)

Initial and Invariant Conditions Every location of a hybrid automaton may be declared as initial (meaning that it is a legal initial state of the system). Every location may be equipped with *invariants* on the real-valued variables, expressed by means of a set of elementary linear expressions $\{\psi_1, ..., \psi_h\}$ over the variables $w_1, \ldots, w_n$. Location no_fail is the initial location of the ME automaton (Figure 2), and is equipped with an invariant enforcing the sp_me variable to stay between the constant values sm_1 and sm_2. The invariant in location $speed_sm3$ forces sp_me to assume constantly the value sm_3.

Channels A finite set of channels is used for discrete communication between automata. A channel c may be used as an *input* (notation c?) or an *output* (notation c!) channel for synchronizing different automata. For instance, the pressure valve automaton of Figure 3 uses two different input channels called $open$? and $close$?. The intended semantics is that the pressure valve automaton awaits for incoming commands (requesting either opening of closing of the valve) coming from the relevant SPS computer controller.

Transitions A finite set of discrete transitions encodes the *discrete* evolution of the automaton. Each transition (also called *switch*) has a *source* and *target* location, and may be equipped with a set $\{\gamma_1, ..., \gamma_k\}$ of *guards* (pre-conditions) and a set $\{\theta_1, ..., \theta_m\}$ of *jump conditions* (post-conditions) on the real-valued variables. A guard is an elementary linear expression over $w_1, \ldots, w_n$; a jump condition is an elementary linear expression over $w_1, \ldots, w_n, w'_1, \ldots, w'_n$. In Figure 2, the transition from $flameout$ to $speed_sm3$ has a guard $sp_me = sm_3$ and no jump condition. By convention, the absence of jump conditions on a transition forces real-valued variables to preserve their value (e.g., in the previous example $sp_me' = sp_me$ implicitly holds). Transitions may be equipped with one or more optional labels denoting the channels on which the automaton must synchronize. For instance, two transitions in Figure 3 are labeled with the input channels $open$? and $close$?

Variable Dynamics Variable dynamics describe how the real-valued variables change in presence of a time elapse transition, and are expressed, for each location, as a set $\{\Psi_1, \ldots, \Psi_k\}$ of elementary linear expressions over $w_1, \ldots, w_n, w'_1, \ldots, w'_n$. As an example, in Figure 2 the *sp_me* variable in location *grippage* varies according to the law $sp_me' = sp_me - k_1(\Delta t)$ (where k_1 is a constant), that is, the speed decreases linearly (proportionally to the time delay) with first derivative equal to k_1. The expression Δt, encoding the time delay, will be explained in Section 4.2.

The hybrid automata presented here do not fall into the *rectangular* automata class described in [Hen96], since re-initialization of variables is not enforced, and triangular constraints are possible. As a consequence, even the problem of reachability for this class of automata is undecidable [HKPV98].

4 Bounded Model Checking for Hybrid Systems

In this section we give a very short overview of SAT based model checking, and we discuss the encoding of our model of hybrid systems, informally described in Section 3, into MATHSAT.

4.1 SAT Based Bounded Model Checking

Bounded Model Checking (BMC) is a recent approach to symbolic model checking [BCCZ99]. Given a Kripke structure M, and an LTL formula f, the idea is to check whether f is true in M by looking for a counterexample (i.e., a witness to the violation of f) that can be presented within a bound of k steps. Given k, the problem is reduced to the satisfiability of a propositional formula $[[M, \neg f]]_k$. For instance, for a property of the form $f := \mathbf{G}p(\underline{s})$, where $p(\underline{s})$ is a boolean formula in the boolean variables $\underline{s}$, then

$$[[M, \neg f]]_k = \mathcal{I}(\underline{s}^{(0)}) \wedge \bigwedge_{i=0}^{k} \mathcal{C}(\underline{s}^{(i)}) \wedge \bigwedge_{i=0}^{k-1} \mathcal{R}(\underline{s}^{(i)}, \underline{s}^{(i+1)}) \wedge \bigvee_{i=0}^{k} \neg p(\underline{s}^{(i)}),$$

where $\mathcal{I}(\underline{s}^{(0)})$ is a representation of the initial conditions, $\mathcal{C}(\underline{s}^{(i)})$ is a representation of the invariant conditions at step i, and $\mathcal{R}(\underline{s}^{(i)}, \underline{s}^{(i+1)})$ is a representation of the transition relation from step i to step $i+1$. If $[[M, \neg f]]_k$ is satisfiable, the propositional model provides a counterexaple of k steps to f. If $[[M, \neg f]]_k$ is unsatisfiable, then nothing can be said about the existence of counterexamples to $M \models f$ with higher bound. The typical technique is to generate and solve $[[M, \neg f]]_k$ for increasing values of k, until either a counter-example is found, or a given time-out is reached.

BMC is being increasingly accepted as practical technique, effective in particular in the process of falsification, i.e. bug finding. The technique relies on the use of efficient SAT solvers (e.g., based on DPLL procedures [DLL62]) for checking the propositional satisfiability of $[[M, \neg f]]_k$. As shown in [CFG+01], BMC avoids the blow-up in memory that can occur with model checking based on Binary Decision Diagrams, and is therefore able to tackle much larger circuits.

4.2 The encoding

Our approach to the verification of hybrid automata is a generalization of BMC for timed systems, as proposed in [ACKS02]. The approach reduces a BMC problem for timed systems to the problem of deciding the satisfiability of math-formulas, i.e. boolean combinations of boolean variables and linear (in)equalities over real variables, representing absolute time and clocks. The resulting math-formulas are tackled with MATH-SAT [ABC+02b, ABC+02a], a solver combining an efficient DPLL procedure with mathematical constraint solvers of increasing deductive power.

In the encoding for timed automata, boolean variables are used to encode the discrete part of the system, while linear constraints on real variables encode the timed part. In particular, each location l is represented by a bitwise encoding $\underline{l}$, so that $\underline{l_i}$ holds if and only if the system is in the location l_i; each synchronization event (channel, shared variable) is represented by a corresponding boolean variables; each switch is represented by a single boolean variable (say, T) which holds if and only if the system executes the corresponding switch; a boolean variable T_δ, representing a continuous transition, holds if and only if time elapses by some $\delta > 0$; finally, for each process P_i, we introduce a boolean variable T^i_{null}, that holds if and only if process P_i does nothing. In order to deal with time, we introduce a real valued variable t representing absolute time, and, for each clock x, a real valued variable ox representing the difference with respect to absolute time. All mathematical constraints required in the encoding are in the form $v_1 - v_2 \bowtie c$, $\bowtie \in \{\leq, \geq, =, >, <\}$ v_1 and v_2 being real variables representing either absolute time or clock values. The reader can refer to [ACKS02] for details.

We tackle the case of hybrid automata by considering that it is an extension of the case of timed automata. The encoding for the timed case is extended by introducing a set of real variables ω_i's, representing physical entities. To simplify the notation, in the following we write: "Δt" for the difference $t' - t$ between absolute time in the next and in the current state; "$\Delta\omega$" for "$\omega' - \omega$", so that, e.g., we write "$c \cdot \Delta t$..." for "$c \cdot t' - c \cdot t$..."; "$\Delta\omega = 0$" for "$\omega' = \omega$", "$\Delta\omega \leq$ ..." for "$\omega' \leq \omega +$ ...". We also write "$(w \in [t_1, t_2])$" for "$(w \geq t_1) \wedge (w \leq t_2)$", where t_1 and t_2 are linear terms. If ψ is a formula, ψ' denotes the formula obtained by substituting in ψ each propositional variable p_j with p'_j, and each real variable v_i with v'_i.

Initial conditions $\mathcal{I}(\underline{s}^{(0)})$. At step 0, in an initial location l, ω can either:

- be set to a given initial value c_0. If so, we represent this fact by the axiom:

$$\underline{l}^{(0)} \rightarrow (\omega^{(0)} = c_0); \tag{1}$$

- be set nondeterministically to an initial value within a closed interval $[a_0, b_0]$, $a_0, b_0 \in [-\infty, \infty]$. [4] If so, we represent this fact by the axiom:

$$\underline{l}^{(0)} \rightarrow (\omega^{(0)} \in [a_0, b_0]). \tag{2}$$

[4] "$a_0 = -\infty$" and "$b_0 = -\infty$" mean that there is no lower bound and no upper bound for ω respectively.

Invariant conditions $\mathcal{C}(\underline{s})$. For each location l equipped with the set $\{\psi_1, ..., \psi_h\}$ of invariants on real valued variables, we include the axiom

$$\underline{l} \rightarrow \bigwedge_j \psi_j. \tag{3}$$

Transition relation $\mathcal{R}(\underline{s}, \underline{s}')$. For each switch T equipped with a set $\{\gamma_1, ..., \gamma_k\}$ of guards and with a set $\{\theta_1, ..., \theta_m\}$ of jump conditions on the real valued variables ω_i's and t, we include the axioms

$$T \rightarrow \bigwedge_j \gamma_j, \tag{4}$$

$$T \rightarrow \bigwedge_j \theta'_j \tag{5}$$

respectively. For each physical variable ω that is not interested by a jump condition of switch T, and must therefore keep its value, we add the axiom:

$$T \rightarrow (\Delta\omega = 0). \tag{6}$$

When process i does nothing, in correspondence of T^i_{null}, each physical variable ω maintains its value:

$$T_{null} \rightarrow (\Delta\omega = 0). \tag{7}$$

When time elapses in a location l, physical variables ω_i evolve according to the set of variable dynamics $\{\Psi_1, \ldots, \Psi_k\}$ associated with l. For each location, we add the axiom

$$(\underline{l} \wedge T_\delta) \rightarrow \bigwedge_i \Psi_i \tag{8}$$

Different forms of variable dynamics are possible:

– ω maintains its value under a dynamic of the form:

$$(\underline{l} \wedge T_\delta) \rightarrow (\Delta\omega = 0); \tag{9}$$

– ω may evolve deterministically according to a linear function:

$$(\underline{l} \wedge T_\delta) \rightarrow (\Delta\omega = c \cdot \Delta t) \tag{10}$$

c being a constant.

– ω may evolve nondeterministically within two linear functions:

$$\omega' \in [b_1\omega + c_1 \cdot \Delta t - a_1, b_2\omega + c_2 \cdot \Delta t + a_2], \tag{11}$$

$$a_1, a_2 \geq 0,\ b_1, b_2 \in \{0, 1\},\ c_1, c_2 \in (-\infty, \infty). \tag{12}$$

If $a_1 = a_2 = 0$, then (11) encodes a triangular constraint. If $b_1 = b_2 = 0$ and $c_1 = c_2 = 0$, then (11) encodes a rectangular constraint.

– in the general case, the evolution of the variables can be nondeterministic within the space described by the linear inequalities $\{\Psi_1, \ldots, \Psi_k\}$, as in equation 8.

The encoding of properties basically follows the encoding in [ACKS02]. Our approach is bounded complete, in the following sense: if there exists a trace of length k, then the encoding of length k is satisfiable, and can be found by running MATHSAT on it. The undecidability of the class of hybrid automata we are dealing with tells us that it is in general impossible to decide if a counterexample might be found with bigger k, or if the problem is unsolvable.

5 Experimental Evaluation

We evaluated the potential of the approach by tackling an example of hybrid systems of industrial relevance, i.e. the model of the SPS. The bounded reachability method described in Section 4 can be used both for model debugging (i.e., bug hunting) and for simulation of hybrid systems. In the following we provide some hints about the use of our methodology by showing some experimental results. For illustration purposes, we will discuss a simplified *one-sided* model of the SPS case study, including one instance of the ME, GBX, VALVE, ATSM, PTO and SPS computer components of Figure 1. Under this abstraction, the analogous components of the opposite side of the system are assumed to be correctly working. An example of property to be checked is given by (the negation of) the following formula:

```
(!GBX.loc_broken & !GBX.loc_grippage & !VALVE.loc_stuck_closed &
!ATSM.loc_broken & !PTO.loc_fused)  U  GBX.sp_gbx <= sg1          (P1)
```

This is a typical safety property expressed via the LTL *until* operator. The intended semantics is whether there exists a path such that no failures of the GBX, VALVE, ATSM and PTO components happen along the path, and finally the speed of the *gearbox* (GBX component) drops below the constant value sg_1. The negation of the above property can be seen as a *safety* property to be verified by the system (i.e., *in presence of failures due only to the main engine, the gearbox speed cannot drop below* sg_1). The rationale behind this property is that the *cross-bleed* procedure initiated by the SPS computer (see Section 2) is able to recover from an engine failure by using power coming from the opposite engine (which is assumed to be working in this one-sided model).

The property may or may not hold depending on the value chosen for the constant sg_1. In particular, if the value chosen for sg_1 exceeds a given threshold, the property is falsified by MathSAT (this means that the cross-bleed procedure is not able to prevent the gearbox speed to drop below that particular value). In this case, MathSAT generates an output trace showing an execution of the system which leads to the violation. The trace includes information on the discrete transitions and the time elapse transitions taken by the automata, the exact time delays and time points at which the transitions take place, and the synchronization channels between different automata. If the value of the constant sg_1 is chosen below a suitable threshold, property (P1) holds, and therefore MathSAT correctly does not find any counterexample. Regarding the choice of the constant sg_1, see the discussion in Section 7.

Time T_0 :
Locations : no_fail, gbx_pto_driven, $atsm_idle$, sps_ok, $closed$
Analog Variables : $sp_me = sm_2$, $sp_gbx = sg_2$, $sp_atsm = 0$
Discrete Trans : $me_grippage$
Synchronizations : none
Time T_1 :
Locations : $grippage$, gbx_pto_driven, $atsm_idle$, sps_ok, $closed$
Analog Variables : $sp_me = sm_5$, $sp_gbx = sg_3$, $sp_atsm = 0$
Discrete Trans : $atsm_inc_a$, sps_inc_a, $valve_open$
Synchronizations : SPS and ATSM on inc_a, SPS and VALVE on $open$
Time T_2 :
Locations : $grippage$, gbx_pto_driven, $atsm_inc_a$, sps_inc_a, $open$
Analog Variables : $sp_me = sm_6$, $sp_gbx = sg_4$, $sp_atsm = sa_2$
Discrete Trans : $atsm_inc_a_inc_b$, $sps_inc_a_inc_b$
Synchronizations : SPS and ATSM on inc_b
Time T_3 :
Locations : $grippage$, gbx_pto_driven, $atsm_inc_b$, sps_inc_b, $open$
Analog Variables : $sp_me = sm_7$, $sp_gbx = sg_1$, $sp_atsm = sa_3$

Fig. 4. An example of MathSAT trace

The trace generated by MathSAT is schematically shown in Figure 4. For each time instant, the trace shows the current locations of the ME, GBX, ATSM, and VALVE automata, the current values of the sp_me, sp_gbx, sp_atsm analog variables, the discrete transitions which take place at that time instant, and the synchronizations channels. For a better understanding of the trace, in Figure 5 we show a simplified version of the SPS computer automaton (only the part relevant to the simulation is shown). Notice that this automaton shows as example of triangular constraint, i.e. $sp_gbx - sp_atsm \leq c_1$ $[\geq c_1]$, and of communication with shared variables (variables sp_gbx and sp_atsm model, respectively, the speed of the GBX and ATSM components).

The simulation begins at time T0, when an engine grippage takes place. Both the engine and the gearbox speeds begin to decrease. At time T1, the SPS computer detects a gearbox low speed condition, and therefore issues the opening of the valve (the VALVE and SPS computer automata synchronize on the *open* channel); as a result, the ATSM begins to increase its speed (SPS and ATSM synchronize on the inc_a channel). At time T2, the SPS computer issues a change in the ATSM dynamics (SPS and ATSM synchronize on inc_b). The simulation stops at time T3, when the gearbox speed reaches the value sg_1.

The same approach can be used for guided simulation. To give an example, we consider the following formula:

```
(!ME.loc_eng_flameout & !GBX.loc_broken & !GBX.loc_grippage &
 !VALVE.loc_stuck_closed & !ATSM.loc_broken & !PTO.loc_fused)
    U  GBX.sp_atsm >= sa1                                        (P2)
```

It is a variation of the previous reachability property, here we require that at the end of the path the speed of the ATSM component is greater than the constant value sa_1.

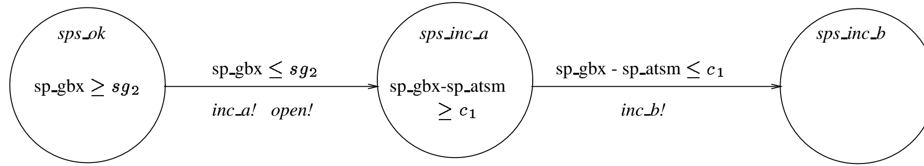


Fig. 5. SPS computer automaton (fragment)

	Property P1				Property P2			
k	Time	Σ Time	Mem.	Result	Time	Σ Time	Mem.	Result
2	0.06	0.06	5.6	UNSAT	0.10	0.10	5.5	UNSAT
3	0.20	0.26	6.2	UNSAT	0.16	0.26	6.0	UNSAT
4	0.53	0.79	7.1	UNSAT	0.30	0.56	6.8	UNSAT
5	1.81	2.60	7.7	UNSAT	0.49	1.05	7.5	UNSAT
6	6.49	9.09	8.4	UNSAT	0.84	1.89	8.2	UNSAT
7	4.53	13.62	8.9	SAT	1.53	3.42	8.8	UNSAT
8					2.88	6.30	9.4	UNSAT
9					4.94	11.24	10.0	UNSAT
10					8.69	19.93	10.7	UNSAT
11					8.88	28.81	11.3	SAT

Table 1. Experimental results (Time in seconds, Memory in MB)

Furthermore, by explicitly ruling out an engine *flameout*, we limit the possible failure modes of the main engine to *grippage*. As explained in Section 2, in presence of an engine failure, the ATSM component is responsible of carrying out the *cross-bleed* procedure, which consists in driving the gearbox with the pneumatic power coming from the valve. Correct functioning of the cross-bleed procedure requires the ATSM (which is initially idle) to start and bring up the gearbox speed. Using MathSAT, we are able to reconstruct a trace corresponding to the above property, which illustrates how the cross-bleed procedure is carried out. It is possible to tune the above simulation and perform further ones by adding further constraints on the trace to look for.

The performance of our method on the examples described above are reported in Table 1. For each problem length, we show computation time, total computation time up to that problem instance, and memory usage. Computation times include both parse and search time. The results have been obtained on a Pentium III machine 1.0 GHz, with 256 Mb memory, running Linux Redhat 7.1. The minimal length trace generated by MathSat for P1 has length 7, whereas the one generated for P2 has length 11.

We also attempted a comparison with HYTECH [HHWT97], a state-of-the-art tool for the analysis of hybrid systems. Differently from our approach, HYTECH is based on the calculation of the reachable state space, and is therefore not limited to the bounded case. In principle, HYTECH may not terminate when tackling an undecidable class of automata (as in the case of the SPS).

We encoded the models of the SPS, as closely as possible, into HYTECH. Overflow errors in the underlying polyhedral libraries made it impossible for HYTECH to com-

pute the space of reachable configurations beyond the 5th iteration. We also attempted to use the `-o1` and `-o2` options, that are sometimes able to limit such problems, but in our case obtained no effect. From the point of view of performance, the time required by HyTech to reach the 5th iteration was 32 seconds, when run without options; the use of `-o1` and `-o2` required 50 and 86 seconds, respectively. The analysis is very preliminary, but seems to suggest that there is a clear potential in our techniques.

6 Related Work

The work presented in this paper builds upon our previous work on *timed systems* [ACKS02]. In [ACKS02], we showed how to reduce the problem of bounded model checking for timed systems to the satisfiability of a math-formula, which can then be checked by a SAT-solver. We also presented the MathSat solver [ABC+02b,ABC+02a], an efficient SAT-solver which is based on the integration of SAT techniques [BCCZ99] with some specialized decision procedures for linear mathematical constraints. A work related to ours, but still limited to timed systems, is [Sor02]. In the present work, as explained in Section 4.2, we have extended the encoding in order to deal with hybrid systems.

Our model for hybrid systems is closely related with the linear and rectangular hybrid automata models presented in [Hen96, HM00], the main difference being in the definition of the dynamics of the real-valued variables. In [Hen96], the dynamics (called *flow conditions*) of the real-valued variables are defined by means of linear constraints over the first derivatives of such variables, whereas in our model dynamics can be characterized by means of linear functions of the time delay, which directly constrain the behaviour of the variables. This approach is analogous to restricting the flows of the real-valued variables to stay inside a rectangular region, as in the rectangular automata model of [Hen96]. In fact, as noted in [HKPV98], under the hypothesis of working with *convex* linear constraints, requiring the flow to be inside a rectangular region amounts to requiring the existence of a smooth function inside the corresponding piecewise-linear envelope.

The model of hybrid I/O automata presented in [LSV03] is general enough to accommodate our model of hybrid automata. Discrete and continuous communication are achieved by means of, respectively, shared actions and shared variables. However, discrete events are not allowed to change the value of shared variables, as in our case.

As an alternative approach to the verification of hybrid systems, we cite [SRKC00], where the CheckMate tool is presented. CheckMate performs verification of hybrid systems using finite-state approximations called *quotient transition systems*. Although this approach is not restricted to linear hybrid automata, the verification analysis may be inconclusive, in which case a refinement of the current approximation may be attempted. An analysis of the current trends in model checking of hybrid systems can be found in [SSKE01].

This line of research has been carried on inside the ESACS [BVÅ+03] project (see `http://www.esacs.org`), an European-Union-sponsored project whose main goals are to define a methodology and a shared environment to improve the safety analysis practice for complex systems development. The Secondary Power System

[BCC+03] is one of the case-studies investigated in ESACS. One of the main motivations for our research is the realization that the use of traditional finite-state model checking, based on the discretization of real variables, has a very hard time in dealing with the complexity of hybrid systems [BCC+03]. In fact, the results may depend on the step of discretization, and the state explosion problem makes such an approach infeasible in practice.

7 Conclusions and Future Work

In this paper, we have addressed the problem of verification of industrial systems that are naturally modeled as linear hybrid automata. The approach is an enhancement of the bounded model checking approach for timed systems proposed in [ACKS02] to the case of linear hybrid automata. Efficiency is gained by the use of the MathSAT solver. The main limitations are given by the undecidability of the analyzed class, and the constraints on the linearity of real variables dynamics. Despite this, however, the approach allows us to model and analyze systems of practical relevance, that HyTech is currently unable to deal with.

In the future, we will provide a more thorough experimental evaluation, by enlarging both the set of tools we compare with (some of them are cited in Section 6), and the set of case studies to analyze. Regarding the SPS example, we plan to experiment with more complex models, at different levels of granularity and abstraction (e.g., a two-sided model of the system). We will investigate how to optimize the MathSAT solver on these specific problems (e.g., by constraining the splitting variables in the style of [GMS98, Str00]), and will experiment with different encodings. As a first step towards bridging the gap between *bounded* model checking and *unbounded* verification, inductive reasoning techniques to prove invariant properties will be investigated. An important point we plan to address in the near future is concerned with *parametric* analysis, which is currently supported in HyTech. To exemplify, parametric analysis would allow us to replace the constant value sg_1 in property (P1) (see Section 5) with a parameter α in order to find out constraints on the parameter for which the property does or does not hold. Finally, in the future we plan to extend the framework to properties expressed in full LTL.

References

[ABC+02a] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In Andrei Voronkov, editor, *CADE-18: Conference on Automated Deduction*, volume 2392 of *LNAI*, pages 195–210, Copenhagen, Denmark, 2002. Springer.

[ABC+02b] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements. In Jacques Calmet, Bernard Benhamou, Olga Caprotti, Laurent Henocque, and Volker Sorge, editors, *CALCULEMUS-2002: Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, volume 2385 of *LNAI*, pages 231–245, Marseille, France, 2002. Springer.

[ACKS02] Gilles Audemard, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. Bounded Model Checking for Timed Systems. In Doron A. Peled and Moshe Y. Vardi, editors, *FORTE 2002: Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of *LNCS*, Houston, Texas, November 2002. Springer.

[BCC+03] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving Safety Assessment of Complex Systems: An industrial case study. In *Proc. Formal Methods Europe (FME 2003)*, 2003. To appear.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *Proc. 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[BVÅ+03] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Lüdtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, and G. Zacco. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *European Safety and Reliability Conference (ESREL'03)*, pages 237–245. Balkema Publisher, 2003.

[CFG+01] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. CAV'2001*, LNCS. Springer, 2001.

[DLL62] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[GMS98] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. In *Proc. AAAI'98*, pages 948–953, 1998.

[Hen96] T.A. Henzinger. The Theory of Hybrid Automata. In *Proceedings 11th Annual International Symposium on Logic in Computer Science (LICS'96)*, pages 278–292, New Brunswick, New Jersey, 1996. IEEE Computer Society Press.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[HKPV98] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's Decidable About Hybrid Automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[HM00] T.A. Henzinger and R. Majumdar. Symbolic Model Checking for Rectangular Hybrid Systems. In S. Graf and M. I. Schwartzbach, editors, *Proceedings 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 142–156, Berlin, Germany, 2000. Springer-Verlag.

[LSV03] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O Automata. *Information and Computation*, 2003. To appear.

[Sor02] M. Sorea. Bounded Model Checking for Timed Automata. In *Proceedings 3rd Workshop on Models for Time-Critical Systems (MTCS'02)*, Brno, Czech Republic, 2002.

[SRKC00] B.I. Silva, K. Richeson, B.H. Krogh, and A. Chutinan. Modeling and verification of hybrid dynamical system using CheckMate. In *Proc. ADPM 2000, Automation of mixed processes: Hybrid Dynamic Systems*. Shaker Verlag, 2000.

[SSKE01] B.I. Silva, O. Stursberg, B.H. Krogh, and S. Engell. An assessment of the Current Status of Algorithmic Approaches to the Verification of Hybrid Systems. In *Proc. 40th Conference on Decision and Control*, 2001.

[Str00] Ofer Strichman. Tuning SAT checkers for bounded model checking. In *Proc. CAV00*, volume 1855 of *LNCS*, pages 480–494, Berlin, 2000. Springer.

Collection of EUFM Benchmark Suites from Formal Verification of Microprocessors

Miroslav N. Velev
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract. The paper presents a collection of 12 benchmark suites, containing a total of 948 formulas in the logic of Equality with Uninterpreted Functions and Memories (EUFM). These benchmarks are from formal verification of pipelined, superscalar, and VLIW processors, defined in the high-level hardware description language AbsHDL. The formulas are expressed in an extension of the SVC format. They were generated with the term-level symbolic simulator TLSim, and were checked for validity with the Equality Validity Checker (EVC)—a decision procedure that exploits the property of Positive Equality and other optimizations when translating EUFM formulas to equivalent SAT formulas, allowing the use of efficient SAT solvers to prove the validity of the original formulas. The suites are available from [76].

1 Introduction

Historically, every time the design process was shifted to a higher level of abstraction, productivity increased. The logic of Equality with Uninterpreted Functions and Memories (EUFM) [12]—see Sect. 2 for a complete description—has constructs that allow us to abstract the functional units, memories, and word-level values, while completely modeling the control path of a processor. Word-level values are abstracted with terms whose only property is that of equality with other terms. EUFM was proposed by Burch and Dill [12], who implemented the first decision procedure for that logic—a prototype of the Stanford Validity Checker (SVC) [61]. Jones et al. [32], and Levitt and Olukotun [35][36] extended SVC with heuristics that sped up the verification, but as observed by Barrett et al. [5] these heuristics are not flexible and do not scale for complex benchmarks.

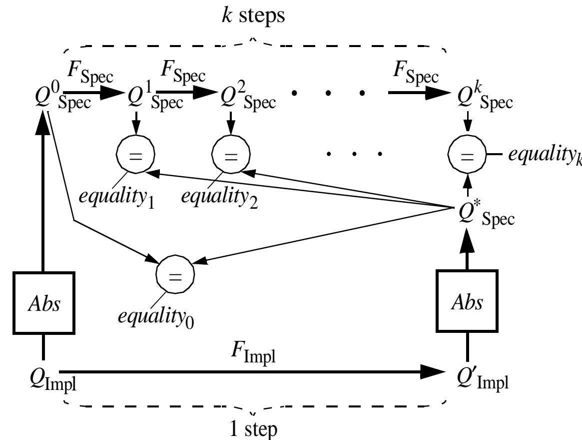
In our work on formal verification of pipelined processors, we imposed some simple restrictions on the style for defining high-level implementations [63][64]. The result was a significant reduction in the number of terms that appear in both positive and negated equality comparisons—and are so called *g-terms* (for general terms)—and an increase in the number of terms that appear only in positive polarity—and are so called *p-terms* (for positive terms). The property of Positive Equality [63][64] allowed us to treat syntactically different p-terms as not equal when evaluating the validity of EUFM formulas, thus achieving significant simplifications, and orders of magnitude speedup. The reader is referred to [9] for a correctness proof.

In the current paper, the formulas are expressed in an extension of the SVC format [61] for EUFM formulas, such that the only addition is a construct for defining constraints. The formulas were generated with the term-level symbolic simulator TLSim [69] that takes a pipelined implementation and a non-pipelined specification—both defined in the high-level hardware description language AbsHDL [69][72]—as well as

a simulation-command file indicating how to simulate symbolically the two processors and how to form the EUFM formula for correctness of the implementation. The benchmark suites in this paper were created when developing and improving the decision procedure EVC [69] that exploits Positive Equality and other optimizations to translate an EUFM formula to an equivalent Boolean formula that has to be a tautology for the EUFM formula to be valid. The tool flow consisting of TLSim, EVC, and an efficient SAT solver was used at Motorola [34] to formally verify a model of the M•CORE processor, and detected bugs. The tool flow was also used in an advanced computer architecture course [72], where undergraduate and graduate students designed and formally verified pipelined DLX processors [19], including variants with exceptions and branch prediction, as well as dual-issue superscalar implementations; formulas from the student solutions, both buggy and correct, form 3 of the benchmark suites in this paper.

2 Background

The formal verification is done by correspondence checking—comparison of a pipelined implementation against a non-pipelined specification, using flushing [12] to automatically compute an *abstraction function* that maps an implementation state to an equivalent specification state. The safety property (see Figure 1) is expressed as a formula in the logic of EUFM, and checks that one step of the implementation corresponds to between 0 and k steps of the specification, where k is the issue width of the implementation. F_{Impl} is the transition function of the implementation, and F_{Spec} is the transition function of the specification. We will refer to the sequence of first applying the abstraction function and then exercising the specification as the *specification side* of the commutative diagram in Figure 1, and to the sequence of first exercising the implementation for one step and then applying the abstraction function as the *implementation side* of the commutative diagram.



Safety property:

$equality_0 \lor equality_1 \lor \ldots \lor equality_k = true$

Figure 1. The safety correctness property for an implementation processor with issue width k: one step of the implementation should correspond to between 0 and k steps of the specification, when the implementation starts from an arbitrary initial state Q_{Impl} that is possibly restricted by invariant constraints.

The safety property is a proof by induction, since the initial implementation state, Q_{Impl}, is completely arbitrary. If the implementation is correct for all transitions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimized by using unreachable states as don't-care conditions, we might have to impose a set of *invariant constraints* for the initial implementation state in order to exclude unreachable states. Then, we need to prove that those constraints will be satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. The reader is referred to [1][2] for a discussion of correctness criteria.

To illustrate the safety property in Figure 1, let the implementation and specification have three architectural state elements—program counter (PC), register file, and data memory. Let PC^i_{Spec}, $RegFile^i_{\text{Spec}}$, and $DMem^i_{\text{Spec}}$ be the state of the PC, register file, and data memory, respectively, in specification state Q^i_{Spec} (i = 0, ..., k) along the specification side of the diagram. Let PC^*_{Spec}, $RegFile^*_{\text{Spec}}$, and $DMem^*_{\text{Spec}}$ be the state of the PC, register file, and data memory, respectively, in specification state Q^*_{Spec}, reached after the implementation side of the diagram. Then, each disjunct $equality_i$ (i = 0, ..., k) is defined as:

$$equality_i \leftarrow pc_i \wedge rf_i \wedge dm_i,$$

where

$$pc_i \leftarrow (PC^i_{\text{Spec}} = PC^*_{\text{Spec}}),$$
$$rf_i \leftarrow (RegFile^i_{\text{Spec}} = RegFile^*_{\text{Spec}}),$$
$$dm_i \leftarrow (DMem^i_{\text{Spec}} = DMem^*_{\text{Spec}}).$$

That is, $equality_i$ is the conjunction of the pair-wise equality comparisons for all architectural state elements, thus ensuring that the architectural state elements are updated in synchrony by the same number of instructions. In processors with more architectural state elements, an equality comparison is conjuncted similarly for each additional state element. Hence, for this implementation processor, the safety property

$$equality_0 \vee equality_1 \vee \ldots \vee equality_{n \times k} = true, \qquad (1)$$

is equivalently represented as:

$$pc_0 \wedge rf_0 \wedge dm_0 \vee pc_1 \wedge rf_1 \wedge dm_1 \vee \ldots \vee pc_k \wedge rf_k \wedge dm_k = true. \qquad (1')$$

The validity checking of (1′) can be decomposed [66] by proving that:

$$pc_0 \vee pc_1 \vee \ldots \vee pc_k = true$$
$$pc_i \wedge pc_j = false, \quad i \neq j, \quad i, j = 0,\ldots,k$$
$$pc_i \Rightarrow rf_i, \quad i = 0,\ldots,k$$
$$pc_i \Rightarrow dm_i, \quad i = 0,\ldots,k$$

We will call the above *PC-based decomposition* of (1′). This decomposition was used to speedup BDD-based evaluation of the correctness formulas for VLIW processors [66], and helped when formally verifying out-of-order processors in on-going work.

To prove *liveness*—that the processor will complete at least one new instruction after a finite number of steps, *n*—we can prove that:

$$equality_1 \vee equality_2 \vee \ldots \vee equality_{n \times k} = true, \quad (2)$$

omitting $equality_0$. Special abstractions and an indirect method for proving liveness, resulting in orders of magnitude speedup, are presented in [75].

The syntax of EUFM [12] includes *terms* and *formulas*—see Fig. 2. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied to a list of argument terms, a term variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, $term_1$, $term_2$) will evaluate to $term_1$ when *formula* = *true*, and to $term_2$ when *formula* = *false*. The syntax for terms can be extended to model memories by means of the functions *read* and *write* [12][67]. Formulas are used to model the control path of a microprocessor, as well as to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied to a list of argument terms, a propositional variable, an *ITE* operator selecting between two argument formulas based on a controlling formula, or an equation (equality comparison) of two terms. Formulas can be negated and combined with Boolean connectives. We will refer to both terms and formulas as *expressions*.

term ::= ***ITE***(*formula*, *term*, *term*)
| *uninterpreted-function*(*term*, . . . , *term*)
| ***read***(*term*, *term*)
| ***write***(*term*, *term*, *term*)
formula ::= **true** | **false** | (*term* = *term*)
| (*formula* ∧ *formula*) | (*formula* ∨ *formula*) | ¬*formula*
| *uninterpreted-predicate*(*term*, . . . , *term*)

Figure 2. Syntax of the logic of EUFM.

UFs and UPs are used to abstract the implementation details of functional units by replacing them with “black boxes” that satisfy no particular properties other than that of *functional consistency*. Namely, that the same combinations of values to the inputs of the UF (or UP) produce the same output value. Then, it no longer matters whether the original functional unit is an adder or a multiplier, etc., as long as the same UF (or UP) is used to replace it in both the implementation and the specification. Thus, we will prove a more general problem—that the processor is correct for any functionally consistent implementation of its functional units. However, this more general problem is easier to prove.

Two possible ways to impose the property of functional consistency of UFs and UPs are Ackermann constraints [3] and nested *ITE*s [64]. The Ackermann scheme replaces each UF (UP) application in the EUFM formula *F* with a new term variable (Boolean variable) and then adds external consistency constraints. For example, the UF applica-

tion $f(a_1, b_1)$ will be replaced by a new term variable c_1, another application of the same UF, $f(a_2, b_2)$, will be replaced by a new term variable c_2. Then, the resulting EUFM formula F' will be extended as $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$. In the nested *ITE*s scheme, the first application of the UF above will still be replaced by a new term variable c_1. However, the second one will be replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new term variable. A third one, $f(a_3, b_3)$, will be replaced by $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new term variable, and so on. UPs are eliminated similarly.

To compare the sequence of *write* operations that form the final states of memories along the implementation and specification sides of the commutative diagram, the decision procedure EVC [69] automatically introduces a new term variable to serve as arbitrary read address for each memory. That address is used to perform a *read* from the memory's final states that are compared for equality in the EUFM correctness formula, then replacing each equation between memory states with an equation between the corresponding *read*s from the new address. For example, if *cmp_addr* is the new term variable introduced for the register file, then each equation $(RegFile^{i}_{\mathrm{Spec}} = RegFile^{*}_{\mathrm{Spec}})$, will be replaced with $(read(RegFile^{i}_{\mathrm{Spec}}, cmp_addr) = read(RegFile^{*}_{\mathrm{Spec}}, cmp_addr))$, thus proving that an arbitrary address in that memory is modified in the same way by both sides of the diagram. EVC replaces a *read* from a sequence of *write*s with a sequence of nested *ITE*s, according to the forwarding property of the memory semantics, such that each *ITE* is controlled by an equation between the new term variable and the destination address for the eliminated *write*. These equations appear in dual polarity—in positive polarity when selecting the then-expression of the *ITE*, but in negative polarity when selecting the else-expression—and so are g-equations that need to be encoded with Boolean variables.

We will call *complete equality* the usual equality, where two (syntactically) different term variables a and b can be either equal or not equal to each other, and will use = to denote it. Reasoning about complete equality requires a case split, in order to account for both cases, and so the need to encode it with Boolean variables when translating an EUFM formula to an equivalent Boolean formula. We will call *syntactic equality* the subset of complete equality where a term variable is equal only to itself, and will use $=_{\mathrm{SYN}}$ to denote it. We will call *delta equality* the difference between complete equality and syntactic equality, and will use $=_{\Delta}$ to denote it. That is, if t_1 and t_2 are two terms consisting of *ITE* operators, term variables, and formulas controlling the *ITE* operators, then $(t_1 =_{\Delta} t_2)$ is defined as $(t_1 = t_2) \wedge \neg(t_1 =_{\mathrm{SYN}} t_2)$, or equivalently, complete equality $(t_1 = t_2)$ is defined as $(t_1 =_{\mathrm{SYN}} t_2) \vee (t_1 =_{\Delta} t_2)$. We will call *hybrid equality* the extension of syntactic equality with a proper subset of the delta equality between two terms, and will denote it with $=_{\mathrm{HYB}}$.

The property of Positive Equality is due to the observation that the formulas for safety (1), and liveness (2) consist of top-level p-equations that are combined with the monotonically positive connectives of conjunction and disjunction, but are not negated. Then, if a formula for safety/liveness is valid (true) when the complete equality in the top-level p-equations is replaced with syntactic equality, the formula will be valid with the original complete equality in the top-level p-equations, since then the formula can only get bigger because of the omitted delta equality that will be included

through monotonically positive connectives. However, using only syntactic equality for the top-level p-equations results in a significant reduction of the solution space. Similarly, we exploit syntactic functional consistency when eliminating UFs and UPs in that the property of functional consistency is enforced only for the cases of syntactic equality between the corresponding arguments of different applications of the same UF/UP. Syntactic functional consistency is a conservative approximation, since functional consistency is enforced only for a subset of the conditions for complete functional consistency (based on complete equality). If F is a formula obtained after eliminating all UFs/UPs by accounting for syntactic functional consistency only, and F is valid, then so will be the formula obtained from F by accounting for complete functional consistency, e.g., by extending F with Ackermann constraints for complete functional consistency.

A *low-level g-equation* is one where both arguments are term variables. A *top-level g-equation* is one where the arguments can be either term variables or nested *ITE* expressions selecting term variables. Both the e_{ij} [16] and the small-domain [49] methods for encoding g-equations with Boolean variables eliminate top-level g-equations by pushing them to the argument term variables, and then encode the resulting low-level g-equations. In our previous work [71], we found the e_{ij} encoding to outperform the small-domain encoding when formally verifying microprocessors. An alternative method [73] eliminates top-level g-equations by abstracting them with a special interpreted predicate that satisfies the properties of transitivity, reflexivity, syntactic symmetry, and syntactic functional consistency. This method resulted in an order of magnitude speedup, compared to the e_{ij} and small-domain methods, when formally verifying wide-issue superscalar benchmarks, presented in suite 8; not enforcing transitivity for benchmarks that do not require it, or enforcing partial transitivity based on heuristics improved the scaling. In EVC, transitivity is enforced based on a greedy heuristic [10], such that the equality comparison graph (between low-level or top-level g-equations, as used in the encoding method) is triangulated with extra equations added greedily, and transitivity is enforced for each resulting cycle of length 3. All transitivity constraints are added to the CNF formula, i.e., the translation to SAT is *eager* as is also the case in [11][49][57]. Other researchers [4][5][44] use *lazy* translation to SAT—incrementally adding constraints to prevent recurrence of detected false counterexamples—but as observed by Seshia et al. [57] this approach significantly degrades the performance when deciding formulas in an extended set of EUFM.

3 File Format

The EUFM formulas are saved in an extension of the SVC format [61], such that the only addition is a construct for defining constraints. The file format is illustrated next with Fig. 4 that contains the output of TLSim after symbolically simulating the processor in Fig. 3 for 1 cycle. Expressions are defined with command `set`, and their names begin with "`$`"; a suffix such as "`___0_0`" indicates the simulation step when TLSim generated the expression. Command `check_valid` specifies an expression that has to be checked for validity, and command `constraint` an expression that if *true* should imply the validity of expressions checked for validity. A file in the extended SVC for-

mat can have multiple constraints, and multiple expressions checked for validity.

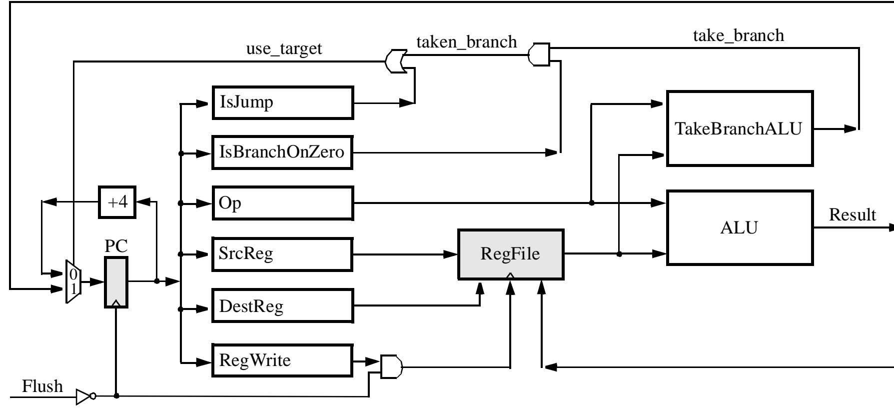


Figure 3. Example processor. The Register File is read-before-write.

```
set $Flush_bar___0_0 (not Flush)
constraint Flush
set $SrcReg___1_1 (SrcReg PC)
set $Data___5_0 (read RegFile $SrcReg___1_1)
set $take_branch___5_1 (TakeBranchALU $Op___1_1 $Data___5_0)
set $IsBranchOnZero___1_1 (IsBranchOnZero PC)
set $taken_branch___5_1 (and $take_branch___5_1 $IsBranchOnZero___1_1)
set $IsJump___1_1 (IsJump PC)
set $use_target___5_1 (or $taken_branch___5_1 $IsJump___1_1)
set $Op___1_1 (Op PC)
set $Result___5_1 (ALU $Op___1_1 $Data___5_0)
set $sequentialPC___1_1 (PCAdder PC)
set $nextPC___5_1
   (ite $use_target___5_1 $Result___5_1 $sequentialPC___1_1)
set $PC_0 (ite $Flush_bar___0_0 $nextPC___5_1 PC)
set $RegWrite___1_1 (RegWrite PC)
set $writeRegFile___7_0 (and $RegWrite___1_1 $Flush_bar___0_0)
set $DestReg___1_1 (DestReg PC)
set $RegFile_0 (ite $writeRegFile___7_0
   (write RegFile $DestReg___1_1 $Result___5_1) RegFile)
set $comparemem_RegFile_0 (= RegFile $RegFile_0)
check_valid $comparemem_RegFile_0
quit
```

Figure 4. Output of TLSim for 1 cycle of term-level symbolic simulation of the example processor. This formula is in the extended SVC format; construct `constraint` is the only addition.

In Fig. 4, `Flush` is a Boolean variable, while `PC` and `RegFile` are term variables for the initial state of the PC and the Register File, respectively. `SrcReg`, `DestReg`, and `Op` are uninterpreted functions mapping the PC to the corresponding field of an instruction. Similarly, `IsBranchOnZero`, `IsJump`, and `RegWrite` are uninterpreted predicates mapping the PC to a control bit for the fetched instruction. `ALU` is an uninterpreted function abstracting the ALU, and `TakeBranchALU` is an uninterpreted predicate abstracting the logic that decides whether to take a conditional branch.

4 Description of the EUFM Benchmark Suites

Suite 1: Variants of Single- and Dual-Issue DLX Processors

Included are the formulas from: a single-issue, 5-stage pipelined DLX [19] processor, `1dlx_c`; a dual-issue superscalar DLX with one complete (i.e., capable of executing all instruction types) and one arithmetic pipeline, `2dlx_ca`; a dual-issue superscalar DLX with two complete pipelines, `2dlx_cc`; and intermediate dual-issue superscalar designs that were created when developing `2dlx_ca` and `2dlx_cc`. These benchmarks were used for the experiments in [64].

Number of formulas in this suite: 8 (all valid)

Suite 2: Single- and Dual-Issue DLX Models with Exceptions, Multicycle ALUs, and Branch Prediction

The 3 main processors used in Suite 1 and described above were extended with branch prediction; multicycle ALUs, multicycle Instruction Memory, and multicycle Data Memory; exceptions that could be raised from the ALUs, the Instruction Memory, and the Data Memory; and a return-from-exception instruction. Variants with both multicycle functional units and exceptions, as well as with all the features were also created. These benchmarks are from the experiments in [65].

Number of formulas in this suite: 18 (all valid)

Suite 3: 100 Buggy Variants of the Most Complex Dual-Issue DLX

This suite contains formulas from 100 buggy versions of `2dlx_cc_mc_ex_bp` from Suite 2. The bugs were variants of actual errors made when creating the correct designs in that suite. These formulas were used for the experiments in [68][71].

Number of formulas in this suite: 100 (all invalid)

Suite 4: VLIW Processors with Speculative Execution

The most complex benchmarks are variants of the VLIW architecture shown in Fig. 5: `9vliw` is the base processor; `9vliw_bp` is an extension with branch prediction; `9vliw_bp_mc` is a further extension with multicycle ALUs, instruction memory and data memory; and `9vliw_bp_mc_ex` is a version with both branch prediction, multicycle functional units, and exceptions. These models have a fetch engine that supplies the execution engine with a packet of 9 instructions with no read-after-write dependencies between them. Each of these instructions is already matched with one of 9 execution pipelines of 4 stages: 4 integer pipelines, two of which can perform both integer and floating-point memory accesses; 2 floating-point pipelines; and 3 branch-address computation pipelines. Data values are stored in 4 register files: integer, floating-point,

predicate, and branch-address. Additionally, the architectural state includes a PC, a Data Memory, and two state elements from Intel's 64-bit architecture (IA-64) [28][58]—a Current Frame Marker (CFM) that is used for register remapping, and an Advanced Load Address Table (ALAT) that is used to implement advanced loads. Every instruction is predicated with a qualifying predicate-register identifier that, if *true*, enables the instruction to modify architectural state. There can be up to 42 instructions in flight. The reader is referred to [66][68] for detailed description of the VLIW benchmarks.

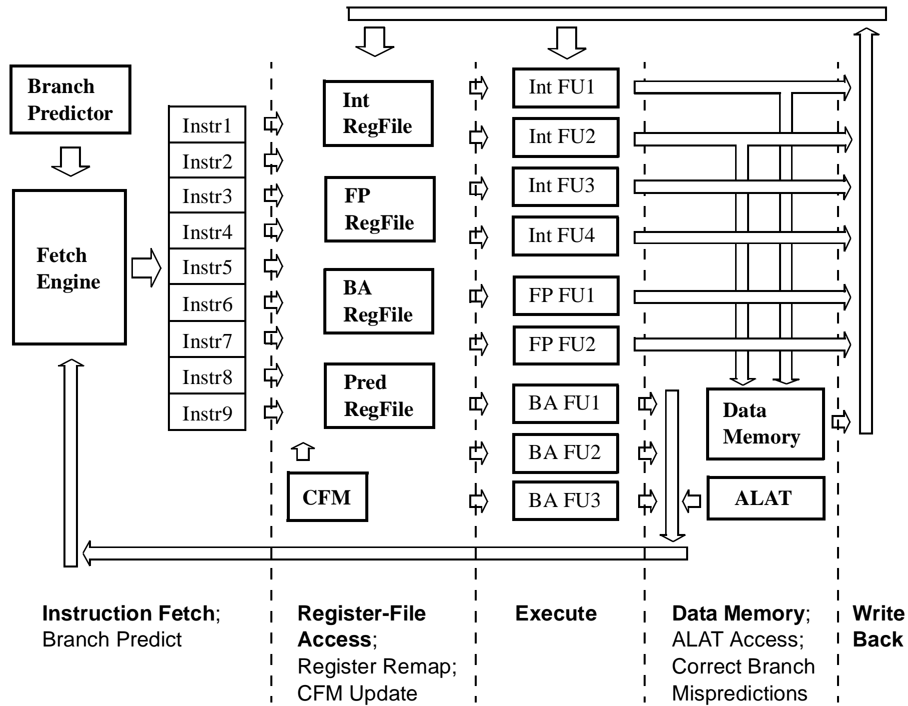


Figure 5. Block diagram of the VLIW architecture.

Number of formulas in this suite: 14 (10 valid, and 4 invalid)

Suite 5: 100 Buggy VLIW Processors

This suite contains formulas from 100 buggy versions of `9vliw_bp_mc` from Suite 4. The bugs were variants of actual errors made when creating the correct designs in that suite. These formulas were used for the experiments in [68][71].

Number of formulas in this suite: 100 (all invalid)

Suite 6: Wide-Issue Out-of-Order Processors with Abstracted Reorder Buffer

The out-of-order processor that was formally verified is shown in Fig. 6. The design can execute only register-register instructions. Up to k of them are fetched in program order by the Fetch Engine on every clock cycle, where k is the issue width of the design. The newly fetched instructions are placed at the end of the Reorder Buffer (ROB), a FIFO structure that maintains the program order of executed instructions. The ROB is abstracted as described in [67]. The actual number of fetched instructions is determined by the Scheduler, based on the available ROB entries and structural resources (decoding units, buses, etc.), the state of the processor, and the implemented scheduling algorithm. The scheduler was also abstracted [67] with a generator of arbitrary Boolean values. The Scheduler communicates that number by signals $fetch_i$, for $1 \leq i \leq k$. The Fetch Engine also computes the next value of the Program Counter (PC), based on the values of signals $fetch_i$. This model has no register renaming, and the operands are kept in the ROB entries, each associated with a computation slice that non-deterministically computes the result for that ROB entry as soon as both data operands become available. Included are benchmarks with up to 1,500 ROB entries, and issue/retire widths of up to 128 instructions per cycle.

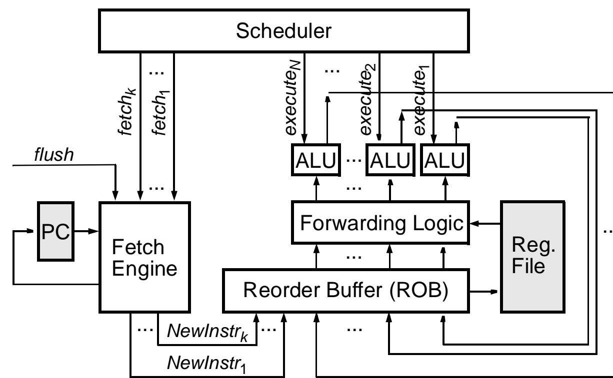


Figure 6. Block diagram of the out-of-order processor with abstracted Reorder Buffer.

Number of formulas in this suite: 76 (all valid)

Suite 7: Liveness of Superscalar and VLIW Processors

The main benchmarks from Suite 2, and the VLIW benchmark from suite 4, as well as their extensions with exceptions and branch prediction, were checked for liveness after being simulated for up to 16 cycles according to formula (2) in Sect. 2. An indirect method to prove liveness is presented in [75].

Number of formulas in this suite: 15 (all valid)

Suite 8: Wide-Issue Superscalar Processors

These benchmarks are from formal verification of wide-issue superscalar processors that were used in [73]. Included are formulas from two groups of correct designs:

Superscalar processors with in-order issue. Labeled `2pipe`, `3pipe`, ... , `12pipe`, where the number designates the issue width, these processor can execute register-register-ALU and load instructions. The benchmarks have the five stages of the DLX processor [19]: fetch, decode (and issue), execute, memory access, and write back, such that each stage can have as many instructions as the issue width of the processor. Since there is no forwarding from the memory-access stage to the execute stage for data values just loaded from the Data Memory, instructions that are in the decode stage and have a data dependency on a load in the execute stage are stalled in the decode stage for one cycle. Instructions are issued (allowed to advance from the decode to the execute stage) in program order, as long as an instruction does not have a data dependency on a load in the execute stage or on an older instruction in the decode stage, and all older instructions in the decode stage are issued in the same cycle. The issue logic in the decode stage is implemented as a shift register, compressing all unissued instructions to the beginning of that register, and filling emptied slots with new instructions.

Superscalar processors with out-of-order issue. Labeled `2pipe_ooo`, `3pipe_ooo`, ... , `7pipe_ooo`, where again the number designates the issue width. These benchmarks are variants of the superscalar processors with in-order issue, except that the instructions in the decode stage are allowed to advance to the execute stage out of program order, as long as they do not have data dependencies on loads in the execute stage or on older instructions in the decode stage, and will not create write-after-read or write-after-write hazards with respect to older instructions that are in the decode stage and are not issued in the same clock cycle. The issue logic is again implemented as a shift register, compressing unissued instructions in the decode stage to the beginning of that register, and filling the emptied slots with newly fetched instructions. However, the out-of-order issue of instructions significantly increases the complexity of the shift register, which has to be able to compress all possible combinations of unissued instructions.

6 buggy designs were inadvertently made when creating the correct models, and the resulting EUFM formulas are also included.

Number of formulas in this suite: 23 (17 valid, and 6 invalid)

Suite 9: Student Buggy Designs 1

These formulas are from student implementations of single-issue 5-stage pipelined DLX processors, designed as a project in an advanced computer architecture course [72]. The models were created by 52 students, working in 24 groups, with 9 of them having a single student. The different coding styles of the many designers make these benchmarks ideal for developing and tuning rewriting rules. The processors were implemented in a sequence of 6 steps, with both the correct and buggy implementations of each group for each step included. Furthermore, the abstraction function was

computed with regular flushing, as opposed to Burch's controlled flushing [13], thus resulting in more complex formulas than the one from a single-issue pipelined DLX in Suite 1. For a full description of this project, as well as of the projects that resulted in the next two benchmark suites, the reader is referred to [72], and for a detailed discussion and classification of the bugs to [74]. The same tool flow as used in these projects—TLSim, and EVC, but combined with BDDs—was applied at Motorola to formally verify a model of the M•CORE processor [34], detecting two bugs in the forwarding logic and one in the issue logic; multiple variants of such bugs appear among those in this and the next two suites.

Number of formulas in this suite: 294 (148 valid, and 146 invalid)

Suite 10: Student Buggy Designs 2

These models were created in a sequel to the above project, where the same students extended their implementations of single-issue pipelined DLX processors with ALU exceptions, a return-from-exception instruction, and branch prediction.

Number of formulas in this suite: 153 (61 valid, and 92 invalid)

Suite 11: Student Buggy Designs 3

Benchmarks from another project in the same course—the students created dual-issue superscalar DLX implementations, where the first pipeline was capable of executing ALU, load, and store instructions, while the second pipeline could execute ALU and branch instructions. Since controlled flushing [13] was not used again, these formulas are significantly more complex than those from the dual-issue models in Suite 1, taking up to an order of magnitude longer to check for validity.

Number of formulas in this suite: 67 (25 valid, and 42 invalid)

Suite 12: Single-Issue DLX Models with Instruction Queues

These models are extensions of `1dlx_c` from Suite 1, and of `1dlx_c_mc_ex_bp` (a processor with multicycle functional units, exceptions, and branch prediction) from Suite 2 with instruction queues that are implemented as shift registers, based on the description of the instruction queue in the PowerPC 750 [27]. The instruction queues have between 1 and 40 entries, for a total of 40 variants of each benchmark.

Number of formulas in this suite: 80 (all valid)

5 Summary of Results

The experiments were performed on a Dell OptiPlex GX260 with a 3.06-GHz Intel Pentium 4 processor, having a 512-KB on-chip level-2 cache, 2 GB of physical memory, and running Red Hat Linux 8.0. The tool flow, consisting of TLSim and EVC, was combined with the SAT-checker Siege [52], which was up to 3 times faster than BerkMin [17], and significantly outperformed Chaff [43][79], Jerusat [46], Limmat [7], and Satzoo [15]—top performers in the 2002 and 2003 SAT competitions [59].

Each benchmark in Suite 1 was formally verified in less than 0.7 seconds, and each benchmark in Suite 2 in less than 1.2 seconds. For the formulas in Suite 3, the maximum time was 2.1 seconds, and the average 0.5 seconds. For the VLIW benchmarks in Suite 4, the maximum time was 46 seconds, and for the 100 buggy VLIW variants in Suite 5, the maximum was 18.3, while the average was 5.2 seconds.

With the available 2 GB of memory, the experiments for the out-of-order processors with abstracted Reorder Buffer (ROB)—Suite 6—scaled for up to 800 ROB entries and issue/retire widths of 128 instructions per cycle, taking 672 seconds. On a 336-MHz Sun4 with 4 GB of memory [70], processing the formula from a model with 1,500 ROB entries and issue/retire widths of 128 instructions per cycle took 3.5 hours. Using an indirect method [75] to solve the formulas in Suite 7 took 33 seconds for the most complex VLIW benchmark. Applying an automatic abstraction of top-level general equations [73] resulted in an order of magnitude speedup when solving the formulas for safety of the wide-issue superscalar processors in Suite 8—the formula from the processor with issue width of 12 instructions was solved in less than 3,200 seconds, compared to over 12 hours without that abstraction. Each of the formulas from student buggy designs in Suites 9, 10, and 11 require, respectively, lass than 1 second, less than 3 seconds, and less than 20 seconds. The most complex benchmark in Suite 12 was formally verified in 3,660 seconds.

Not exploiting Positive Equality when processing the formula from the most complex benchmark in Suite 1—a dual-issue DLX processor with 2 complete pipelines—resulted in CPU time of over 24 hours. In contrast, with Positive Equality, that formula was proved valid in less than 0.7 seconds, i.e., 5 orders of magnitude faster. Hence, even with the tremendous advances in SAT solvers, the property of Positive Equality continues to be the main reason for the efficiency of EVC when checking the validity of formulas from formal verification of pipelined, superscalar, and VLIW processors.

6 Related Work

Before the use of Positive Equality and other optimizations to translate EUFM formulas to SAT, the most widely used method for formal verification of pipelined processors was theorem proving. However, the formal verification of a 5-stage pipelined DLX or ARM—comparable to the model for formula `1dlx_c` from EUFM Suite 1—required extensive manual work by experts, and often long run times [8][14][24][25][26][31][33][45][62][77]. Even 3-stage pipelines, executing only ALU instructions, took significant manual intervention to formally verify with theorem proving [38][39][55][56], or with assume-guarantee reasoning [20][21][50]. Symbolic Trajectory Evaluation (STE) also required extensive manual work to prove the correctness of just a

register-immediate `OR` instruction in a bit-level 5-stage ARM processor [47]. Other researchers had to limit the data values to 4 bits, the register file to 1 register, and the ISA to 16 instructions, to symbolically verify a bit-level pipelined processor [6]. Various symbolic tools ran for a long time when formally verifying a pipelined DLX [23][53][54], or ran out of memory [29]. Custom-tailored, manually defined rewriting rules were used to formally verify a 5-stage DLX [35][36], and similar 4-stage processors [18][37][40][41], but would require modifications to work on designs described in a different coding style, and significant extensions to scale for dual-issue superscalar processors. Other researchers proved only few properties of a pipelined DLX [30][51], or did not present completeness argument [42]—that the properties proved will ensure correctness under all possible scenarios.

Burch and Dill [12] proposed flushing as a way to automatically compute an abstraction function—mapping the state of a pipelined processor to the state of its non-pipelined specification—and were first to formally verify a pipelined DLX. However, they had to manually provide a case-splitting expression for the conditions when the processor will fetch and complete a new instruction. Burch [13] applied the same method to a dual-issue superscalar DLX, but had to manually define 28 case-splitting expressions, and to decompose the safety correctness criterion. That decomposition was subtle enough to warrant publication of its correctness proof as a separate paper [78]. Hosabettu et al. [24][25] used theorem-proving to formally verify a single-issue pipelined DLX and a dual-issue superscalar DLX, but reported one month of manual work for each. DLX models with exceptions and multicycle functional units of fixed latency were verified by Mishra and Dutt [42], who wrote scripts to generate symbolic sequences that test for various hazards, and used SVC [61] to verify the resulting formulas. However, those authors did not present a completeness argument.

Recent decision procedures [11][57]—extending EUFM with counter arithmetic, lambda expressions, and inequalities—exploit most of the optimizations in EVC, including the use of Positive Equality.

7 Conclusions

The paper presented a large collection of EUFM benchmarks from formal verification of microprocessors. The formulas can be used to develop and tune new decision procedures for the logic of EUFM. The key to efficient validity checking of these benchmarks with the decision procedure EVC was the property of Positive Equality, and translation to equivalent Boolean formulas, allowing the use of efficient SAT solvers. Additionally, abstraction of top-level equations, decomposition of the correctness proofs, and rewriting rules helped to check the validity of formulas from complex superscalar processors. The benchmark suites are available from [76].

References

[1] M.D. Aagaard, N.A. Day, and M. Lou, "Relating Multi-Step and Single-Step Microprocessor Correctness Statements," *Formal Methods in Computer-Aided Design (FMCAD '02)*, M.D. Aagaard, and J.W. O'Leary, *eds.*, LNCS 2517, Springer-Verlag, November 2002, pp. 123–141.

[2] M.D. Aagaard, B. Cook, N.A. Day, and R.B. Jones, "A Framework for Superscalar Microprocessor Correctness Statements," Software Tools for Technology Transfer (STTT), Vol. 4, No. 3 (May 2003).

[3] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.
[4] G. Audemard, P. Bertoli, A. Cimatti, A. Korniowicz, and R. Sebastiani, "A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions," *11th International Conference on Automated Deduction (CADE '02)*, LNCS 2392, Springer-Verlag, July 2002, pp. 195–210.
[5] C. Barrett, D. Dill, and A. Stump, "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT," *Computer-Aided Verification (CAV '02)*, LNCS 2404, Springer-Verlag, July 2002, pp. 187–201.
[6] V. Bhagwati, and S. Devadas, "Automatic Verification of Pipelined Microprocessors," *31st Design Automation Conference (DAC '94)*, June 1994, pp. 603–608.
[7] A. Biere, Limmat Satisfiability Solver, http://www.inf.ethz.ch/personal/biere/projects/limmat/.
[8] E. Börger, and S. Mazzanti, "A Practical Method for Rigorously Controllable Hardware Design," *10th International Conference of Z Users (ZUM '97)*, J. Bowen, M. Hinchey, and D. Till, *eds.*, LNCS 1212, Springer-Verlag, April 1997, pp. 151–187.
[9] R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic," ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (January 2001), pp. 93–134.
[10] R.E. Bryant, and M.N. Velev, "Boolean Satisfiability with Transitivity Constraints," ACM Transactions on Computational Logic (TOCL), Volume 3, Number 4 (October 2002), pp. 604–627.
[11] R.E. Bryant, S.K. Lahiri, and S.A. Seshia, "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," *Computer-Aided Verification (CAV '02)*, LNCS 2404, Springer-Verlag, July 2002, pp. 78–92.
[12] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification* (*CAV '94)*, D.L. Dill, *ed.*, LNCS 818, Springer-Verlag, June 1994, pp. 68–80.
[13] J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996.
[14] D. Cyrluk, "Inverting the Abstraction Mapping: A Methodology for Hardware Verification," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996, pp. 172–186.
[15] N. Eén, and N. Sörensson, "Temporal Induction by Incremental SAT Solving," *submitted for publication*, 2003. http://www.cs.chalmers.se/~een/Satzoo/.
[16] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244–255.
[17] E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver," *Design, Automation, and Test in Europe (DATE '02)*, March 2002, pp. 142–149.
[18] N.A. Harman, "Verifying a Simple Pipelined Microprocessor Using Maude," *15th International Workshop on Recent Trends in Algebraic Development Techniques (WADT '01),* M. Cerioli, and G. Reggio, *eds.*, LNCS 2267, Springer-Verlag, April 2001, pp. 128–151.
[19] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann Publishers, San Francisco, CA, 2002.
[20] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "You Assume, We Guarantee: Methodology and Case Studies," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 440–451.
[21] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning," *International Conference on Computer-Aided Design*, November 2000.
[22] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer, "Temporal-Safety Proofs for Systems Code," *Computer-Aided Verification (CAV '02)*, LNCS 2404, Springer-Verlag, July 2002, pp. 526–538.
[23] H. Hinrichsen, H. Eveking, and G. Ritter, "Formal Synthesis for Pipeline Design," *2nd International Conference on Discrete Mathematics and Theoretical Computer Science (DMTCS '99) and the 5th Australasian Theory Symposium (CATS '99)*, C. S. Calude, and M. J. Dinneen, *eds.*, Australian Computer Science Communications, Vol. 21, No. 3, Springer-Verlag, 1999.
[24] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Decomposing the Proof of Correctness of Pipelined Microprocessors," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 122–134.
[25] R. Hosabettu, "Systematic Verification of Pipelined Microprocessors," Ph.D. Thesis, Department of Computer Science, University of Utah, August 2000.
[26] J.K. Huggins, and D. Van Campenhout, "Specification and Verification of Pipelining in the ARM2 RISC Microprocessor," ACM Transactions on Design Automation of Electronic Systems, Vol. 3, No. 4 (October 1998), pp. 563–580.
[27] IBM Corporation, *PowerPC 740™/PowerPC 750™: RISC Microprocessor User's Manual*, 1999.
[28] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999. Available from: `http://developer.intel.com/design/ia-64/architecture.htm`
[29] A.J. Isles, R. Hojati, and R.K. Brayton, "Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 256–267.
[30] L. Ivanov, "Modeling and Verification of a Pipelined CPU," *Midwest Symposium on Circuits and Systems (MWSCAS '02)*, August 2002.

[31] C. Jacobi, and D. Kröning, "Proving the Correctness of a Complete Microprocessor," *30. Jahrestagung der Gesellshaft für Informatik*, Springer-Verlag, 2000.
[32] R.B. Jones, D.L. Dill, and J.R. Burch, "Efficient Validity Checking for Processor Verification," *International Conference on Computer-Aided Design (ICCAD '95)*, November 1995, pp. 2–6.
[33] D. Kröning, and W.J. Paul, "Automated Pipeline Design," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 810–815.
[34] S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE™ Microprocessor Core," *6th Annual IEEE International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001, pp. 109–114.
[35] J. Levitt, and K. Olukotun, "Verifying Correct Pipeline Implementation for Microprocessors," *International Conference on Computer-Aided Design (ICCAD '97)*, November 1997, pp. 162–169.
[36] J.R. Levitt. "Formal Verification Techniques for Digital Systems," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, December 1998.
[37] M.N. Lis, "Superscalar Processors via Automatic Microarchitecture Transformations," M.S. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., June 2000.
[38] P. Manolios, "Correctness of Pipelined Machines," *Formal Methods in Computer-Aided Design (FMCAD '00)*, W.A. Hunt, Jr., and S.D. Johnson, *eds.*, LNCS 1954, Springer-Verlag, November 2000, pp. 161–178.
[39] P. Manolios, "Mechanical Verification of Reactive Systems," Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, August 2001.
[40] J. Matthews, and J. Launchbury, "Elementary Microarchitecture Algebra," *Computer-Aided Verification (CAV '99)*, N. Halbwachs and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, June 1999.
[41] J.R. Matthews, "Algebraic Specification and Verification of Processor Microarchitectures," Ph.D. Thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, October 2000.
[42] P. Mishra, and N. Dutt, "Modeling and Verification of Pipelined Embedded Processors in the Presence of Hazards and Exceptions," *IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES '02)*, August 2002.
[43] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 530–535.
[44] L. de Moura, H. Rueß, and M. Sorea, "Lazy Theorem Proving for Bounded Model Checking over Infinite Domains," *11th International Conference on Automated Deduction (CADE '02)*, LNCS 2392, Springer-Verlag, July 2002, pp. 438–455.
[45] S.M. Müller, and W.J. Paul, *Computer Architecture: Complexity and Correctness*, Springer-Verlag, 2000.
[46] A. Nadel, Jerusat 1.2 SAT Solver, http://www.geocities.com/alikn78/.
[47] V.A. Patankar, A. Jain, and R.E. Bryant, "Formal Verification of an ARM Processor," *12th International Conference on VLSI Design*, January 1999, pp. 282–287.
[48] A. Pnueli, M. Siegel, and O. Strichman, "The Code Validation Tool (CVT): Automatic Verification of a Compilation Process," Software Tools for Technology Transfer (STTT), No. 2, 1998, pp. 192–201.
[49] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel, "The Small Model Property: How Small Can It Be?", Journal of Information and Computation, Vol. 178, No. 1 (October 2002), pp. 279–293.
[50] S. Qadeer, "Algorithms and Methodology for Scalable Model Checking**,"** Ph.D. Thesis, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, Fall 1999.
[51] S. Ramesh, and P. Bhaduri, "Validation of Pipelined Processor Designs Using Esterel Tools: A Case Study," *Computer-Aided Verification (CAV '99)*, N. Halbwachs, and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, July 1999, pp. 84–95.
[52] L. Ryan, Siege SAT Solver v.3, http://www.cs.sfu.ca/~loryan/personal/.
[53] G. Ritter, H. Eveking, and H. Hinrichsen, "Formal Verification of Designs with Complex Control by Symbolic Simulation," *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999.
[54] G. Ritter, "Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation," Ph.D. Thesis, Department of Electrical and Computer Engineering, Darmstadt University of Technology, March 2001.
[55] J. Sawada, "Formal Verification of an Advanced Pipelined Machine," Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, December 1999.
[56] J. Sawada, "Verification of a Simple Pipelined Machine Model," in *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J.S. Moore, *eds.*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2000, pp. 137–150.
[57] S.A. Seshia, S.K. Lahiri, and R.E. Bryant, "A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions," *40th Design Automation Conference (DAC '03)*, June 2003, pp. 425–430.
[58] H. Sharangpani, and K. Arora, "Itanium Processor Microarchitecture," IEEE Micro, Vol. 20, No. 5 (September–October 2000), pp. 24–43.
[59] L. Simon, D. Le Berre, and E.A. Hirsch, "SAT Solver Competition Report," 2002, 2003. Available from: http://www.satlive.org.
[60] S.K. Srinivasan, and M.N. Velev, "Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Imprecise Data-Memory Exceptions," *Formal*

Methods and Models for Codesign (MEMOCODE '03), June 2003.
[61] Stanford Validity Checker (SVC), http://sprout.Stanford.EDU/SVC.
[62] S. Tahar and R. Kumar, "A Practical Methodology for the Formal Verification of RISC Processors," Formal Methods in Systems Design, Vol. 13, No. 2 (September 1998), pp. 159–225.
[63] M.N. Velev, and R.E. Bryant, "Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors," *36th Design Automation Conference (DAC '99)*, June 1999, pp. 397–401.
[64] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre, and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 37–53.
[65] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *37th Design Automation Conference (DAC '00)*, June 2000, pp. 112–117.
[66] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution," *Computer-Aided Verification (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 296–311.
[67] M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, T. Margaria and W. Yi, *eds.*, LNCS 2031, Springer-Verlag, April 2001, pp. 252-267.
[68] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 226–231.
[69] M.N. Velev, and R.E. Bryant, "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations," *Computer-Aided Verification (CAV '01)*, G. Berry, H. Comon, and A. Finkel, *eds.*, LNCS 2102, Springer-Verlag, July 2001, pp. 235–240.
[70] M.N. Velev, "Using Rewriting Rules and Positive Equality to Formally Verify Wide-Issue Out-Of-Order Microprocessors with a Reorder Buffer," *Design, Automation and Test in Europe (DATE '02)*, March 2002, pp. 28–35.
[71] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2 (February 2003), pp. 73–106.
[72] M.N. Velev, "Integrating Formal Verification into an Advanced Computer Architecture Course," *ASEE 2003 Annual Conference & Exposition*, June 2003.
[73] M.N. Velev, "Automatic Abstraction of Equations in a Logic of Equality," *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, LNAI, Springer-Verlag, September 2003.
[74] M.N. Velev, "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs," *International Test Conference (ITC '03)*, October 2003.
[75] M.N. Velev, "Using Positive Equality to Prove Liveness for Pipelined Microprocessors," submitted for publication, 2003.
[76] M.N. Velev, EUFM Benchmark Suites. http://www.ece.cmu.edu/~mvelev
[77] P.J. Windley, "Verifying Pipelined Microprocessors," *Conference on Hardware Description Languages (CHDL '95)*, August 1995.
[78] P.J. Windley, and J.R. Burch, "Mechanically Checking a Lemma Used in an Automatic Verification Tool," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas, and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996, pp. 362–376.
[79] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," *International Conference on Computer-Aided Design (ICCAD '01)*, November 2001, pp. 279–285.

The SMT-LIB Format: An Initial Proposal

Silvio Ranise[1] and Cesare Tinelli[2]

[1] LORIA & INRIA-Lorraine
Nancy, France
ranise@loria.fr
[2] The University of Iowa
Iowa City, IA, USA
tinelli@cs.uiowa.edu

1 Introduction

This paper is a first proposal for a common format for the Satisfiability Modulo Theories Library, or SMT-LIB for short. The main goal of the SMT-LIB initiative [2], coordinated by these authors and supported by a growing number of researchers world-wide, is to produce a on-line library of benchmarks for *satisfiability modulo theories*. By benchmark we mean a logical formula to be checked for satisfiability modulo (combinations of) background theories of interest. Examples of background theories typically used in computer science are real and integer arithmetic and the theories of various data structures such as lists, arrays, bit vectors and so on.

A lot of work has been done in the last few years by several research groups on building systems for satisfiability modulo theories. We believe that having a library of benchmarks will greatly facilitate the evaluation and the comparison of these systems, and advance the state of the art in the field, in the same way as, for instance, the TPTP library [3] has done for theorem proving, or the SATLIB library [1] has done for propositional satisfiability.

2 The Satisfiability Modulo Theories Library

We envision a library consisting of two main sections: one containing the specification of several background theories, and another containing benchmark sets, grouped under a number of indexes such as their corresponding background theory, the class of formulas they belong to, the type of problem they originate from and so on.

For the library to be viable and useful it should adopt a common standard for expressing the benchmarks, and for defining the background theories in a rigorous way—so that there is no doubt on which theories are intended. In this respect, some natural questions arise.

1. Is it sufficient to use a first-order language for the benchmarks?
   (After all, most of the research in decision procedure has been done in a first-order setting.)
2. If so, should the library be limited to ground (i.e. quantifier-free) satisfiability problems? Or should it also consider problems with quantifiers?

3. Should the library adopt a sorted or an unsorted input language?
4. How should the background theories and their various combinations be defined and specified?
5. Which concrete syntax should we use for the benchmarks?

In the rest of the paper we discuss some possible answers to these questions and put forward an initial proposal. The proposal described here has been prepared by unifying a number of ideas contributed by the members of the SMT-LIB interest group.

3 Basic Assumptions and Proposals

We start by fixing the terminology and making some basic assumptions.

We assume that input problems, i.e. logical formulas, are to be checked for satisfiability, not validity.[3] In particular, given a theory T and a formula φ, we are interested in whether φ is *satisfiable in* T, or is *satisfiable modulo* T, that is, whether there is a model of φ that satisfies (the existential closure of) φ.

Informally speaking, let us call a *satisfiability procedure* any procedure for satisfiability modulo some given theory. With satisfiability procedures one can distinguish among

1. the procedure's *underlying logic* (first-order, many-sorted, modal, intuitionistic, higher order, etc.),
2. the procedure's *background theory*, the theory against which satisfiability is checked (typically a set of closed formulas in the underlying logic's syntax or a set of models in the logic's semantics), and
3. the procedure's *input language*, that is, the class of formulas the procedure accepts as input (ground, CNF, first-order, temporal, etc.).

For instance, the underlying logic of a typical solver for linear arithmetic is first-order logic with equality, the background theory is the theory of real numbers, and the input language is the class of conjunctions of linear equations and inequations.

We believe that to ease organization and classification it would be helpful for SMT-LIB to follow this distinction as well. This entails that

> *SMT-LIB should (i) adopt at least one underlying logic, (ii) define a number of background theories, and (iii) specify a general syntax for the various benchmarks, providing a way to indicate to which class of formulas a benchmark belongs.*

We briefly discuss these three points in the following subsections, together with our proposal for each.

[3] The difference matters only for those classes of problems that are not closed under logical negation.

3.1 The Logic

There is unanimous agreement within the SMT-LIB interest group that SMT-LIB should concentrate on a single underlying logic for its benchmarks and that this logic should be first-order logic with equality ($FOL^=$) or some variations of it. There is less agreement on whether it should be classical unsorted logic or a multi-sorted version.

An argument against supporting a multi-sorted version of $FOL^=$ is that it is a more complicated framework than classical first-order logic, and that almost all theoretical results in satisfiability modulo theories (e.g. on combining satisfiability procedures) are given in the context of unsorted $FOL^=$ only.

The issue of sorts, however, is very important in this field, because a large number of background theories and input problems are more naturally formulated in a sorted framework. As a matter of fact, some of the existing solvers actually have a sorted input language. Therefore, we feel that SMT-LIB should provide some kind of support for sorts.

In an attempt to balance the pros and cons of a multi-sorted framework, we propose here a compromise solution that seeks to combine the simplicity and familiarity of unsorted logic with the convenience of a sorted language.

We propose for SMT-LIB to adopt unsorted $FOL^=$ as the sole underlying logic, but to allow benchmarks and theories to be specified in a many-sorted language whose semantics is provided by a translation into $FOL^=$.

The proposed language and its translation into $FOL^=$ are discussed in the next sections.

3.2 The Background Theories

One of the goals of the SMT-LIB initiative is to clearly define a catalog of background theories, starting with a small number of popular ones, and adding new ones as solvers for them are developed. Theories will be specified in SMT-LIB independently of any benchmarks or solvers. Each set of benchmarks then will contain a reference to its own background theory.

We distinguish between *basic* (or *component*) theories and *combined* theories. By combined theory we mean a theory that is defined as some kind of combination of basic theories. Example of basic theories include the theory of real numbers, the theory of arrays, the theory of lists and so on.

We propose that basic theories be specified informally albeit as rigorously as possible. We do not prescribe any specific way to specify a theory.

The issue of specifying combined theories modularly in terms of their component theories is a bit tricky, in particular in the presence of sorts. We leave its discussion to a later version of this paper. Although this is not ideal, for the time being we can specify a combined theory in SMT as if it was a basic one, that is, disregarding the fact that it is the combination of other theories.

3.3 The Input Language

We propose to adopt a general first-order (sorted) language in which to write all SMT-LIB benchmarks.

We realize, however, that many benchmarks are typically expressed in a some fragment of the language of first-order logic. The particular fragment of $FOL^{=}$ considered does matter because one can often write a solver specialized on that fragment that is a lot more efficient than a solver meant for a larger fragment.[4]

An extreme case of this situation occurs when satisfiability modulo a given theory T is decidable for a certain fragment (quantifier-free, say) but undecidable for a larger one (full first-order, say), as for instance happens with the theory of arrays specified in Section 7.2. But it is also true when the decidability of the satisfiability problem is preserved across various fragments. For instance, if T is the theory of real numbers, the satisfiability in T of full-first order formulas is decidable. However, one can build increasingly faster solvers by restricting the language respectively to quantifier-free formula, linear equations and inequations, difference equations, inequations between variables, and so on. As a consequence,

it is useful for a benchmark to specify which specific fragment of the general first-order language it belongs to. The attribute-based format we propose for writing benchmark sets provides an attribute to specify just that.

4 The SMT-LIB Logic and Language

We argued earlier for adopting classical (unsorted) first-order logic with equality as the sole underlying logic for SMT-LIB.

For specifying benchmarks and possibly theory axioms, however, we propose to adopt a many-sorted language with subsorts. As a typed language, the proposed language is intentionally limited in expressive power. In essence,

the language allows one only to declare sorts (types) only by means of sort symbols, to define a preordering on the sorts, to specify the interface of function and predicate symbols in terms of the declared sorts, and to specify the sort of quantified variables.

In type theory terms, the language has no type constructors, no type quantifiers, no provisions for parametric polymorphism, and so on. The only form of polymorphism it allows is subsort polymorphism, akin to subtype polymorphism in object-oriented languages. Also, explicit (ad-hoc) overloading of function or predicate symbols—by which a symbol could be explicitly given more than one interface—is not allowed. The idea is to provide, at least at the beginning of this project, just enough expressive power to represent typical benchmarks without getting bogged down in the complexity (and higher-orderness) of type theory.

[4] By efficiency here we do not necessary refer to worst-case time complexity, but to efficiency "in practice".

An abstract syntax for expressing formulas in the proposed language is provided in the following. In order for us to provide already actual examples of benchmarks at this stage, the abstract syntax is actually not as abstract as it could be. We wrote it so that an actual, concrete syntax is immediately derivable from it.

The proposed syntax is attribute-based and Lisp-like. In designing it, we followed the recommendation of several members of the SMT-LIB interest group that the benchmarks be easily parsable. Preferring ease of parsing over human readability is reasonable in this context because we expect not only that benchmarks will be typically read by solvers but also that, by and large, they will be produced in the first place by automated tools like verification condition generators or translators from other formats.

The syntax of formulas in the SMT-LIB language extends the standard abstract syntax of $FOL^{=}$ with the following additional constructs:
- *a construct for declaring the sort of quantified variables,*
- *an* if-then-else-*like logical connective,*
- *a* let *construct for terms,*
- *a* let *construct for formulas, and*
- *a* distinct *construct for declaring a number of values as pairwise distinct.*

Except for the first extension, dictated by our goal of supporting sorts, the other extensions are provided for greater convenience. We discuss each of them in turn.

The *if-then-else* construct This construct is very common in benchmarks coming for instance from hardware verification. Although it can be defined in terms of more basic constructs such as conjunctions and implications, it provides important structural information that a solver can use to speed up its computation. Since such information is lost if one imposes a previous translation into the more basic constructs, it seems important to support an *if-then-else* construct natively in the SMT-LIB language.

We point out that conceptually there are two kinds of *if-then-else*-like constructs: a logical connective of the form

$$\textbf{if } \varphi \textbf{ then } \varphi_1 \textbf{ else } \varphi_2$$

where $\varphi, \varphi_1, \varphi_2$ are formulas, and function symbol of the form

$$\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2$$

where t, t_1, t_2 are terms—with t being typically, but not necessarily, a Boolean term. We propose to support only the first type, which has a straightforward general semantics and provides already enough flexibility. The second type is problematic because its semantics is domain dependent as it is based on the type of t.[5] The complications introduced by the second *if-then-else* construct can be easily avoided because its effect can be achieved (with no substantial loss of structural information) by means of the

[5] In the literature, the second *if-then-else* evaluates to the value of t_1, as opposed to the value of t_2 if t evaluates to some value v which, depending on problem domain, is chosen to be $true$, 0, a positive number, an empty list, a non-empty one, and so on.

first *if-then-else* construct and fresh constants. In fact, every atomic formula of the form $\varphi(\textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2)$, that is, containing $\textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2$ as a subterm, can be equivalently rewritten as

$$(\textbf{if}\ t = v\ \textbf{then}\ c = t_1\ \textbf{else}\ c = t_2) \wedge \varphi(c)$$

where c is a fresh constant and v is the value t must evaluate to for $\textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2$ to evaluate to t_1's value.

The *let* construct for terms This is a construct of the form

$$\textbf{let}\ x = t\ \textbf{in}\ \varphi$$

where x is a variable, t is a term and φ is a formula. This construct is convenient for benchmark compactness as it allows one to replace multiple occurrences of the same term by a variable. It is of course also useful for a solver because it saves the solver the effort to recognize the various occurrences of the same term as such.

We propose to include this kind of construct, but with the restriction that t be a ground term, that is, a term with no variables. This restriction is perhaps overly strong. With it, however, the interpretation of $\textbf{let}\ x = t\ \textbf{in}\ \varphi$ is straightforward and unproblematic: it stands for the formula obtained from φ by replacing each unbound occurrence of x in φ by t.[6] With t ground we do not have to worry about the problem of "variable capturing", which occurs when a variable that is originally free in t becomes bound after the substitution of t for x in φ.

The *let* construct for formulas This is a construct of the form

$$\textbf{let}\ p \equiv \varphi\ \textbf{in}\ \psi$$

where p is a propositional variable, φ is a formula with no free variables, and ψ is any formula. The rationale for this construct and the restriction on φ to be a closed formula is entirely similar to that for the previous *let* construct.

To simplify parsing, the proposed language will distinguish syntactically between the two *let* constructs.

The *distinct* construct This construct is also added for conciseness. It has the form

$$distinct(t_1, \ldots, t_n)$$

for a variable $n \geq 2$, where $t_1, \ldots, t_n$ are terms, and it stands for all pairwise disequations between the t_i's.

[6] By unbound we mean not in the scope of another *let* or of a quantifier in φ.

4.1 The language of formulas

The proposed language of formulas is the *well-sorted* subset of the language generated by the non-terminal symbol *let_formula* in the grammar below. Derivation rules defining the well-sorted formulas and terms of the language follow later.

The grammar is given as a set of BNF-style production rules. In these rules, we use boldface text to denote terminal symbols, the (_)* operator for denoting zero or more repetitions of the operand, the (_)+ operator for one or more repetitions, and the [_] operator for zero or one repetitions.

let_formula ::= **(** :**let** *var term let_formula* **)**
  | **(** :**flet** *pred_sym formula let_formula* **)**
  | *formula*

formula ::= **(** *quant_symb* **(** *var sort_symb* **)** *formula* **)**
  | **(** *connective formula** **)**
  | *atomic_formula*

quant_symb ::= :**exists** | :**forall**
var ::= **?***identifier*
sort_symb ::= *identifier*
connective ::= :**not** | :**impl** | :**iff** | :**ite** | :**and** | :**or** | :**xor**

atomic_formula ::= :**true** | :**false** | *pred_symb* | **(** *pred_symb term*+ **)**
  | **(** = *term term*+ **)** | **(** :**distinct** *term term*+ **)**

term ::= *var* | *numeral* | *fun_symb* | **(** *fun_symb term*+ **)**

pred_symb ::= *identifier* | *operator*
fun_symb ::= *identifier* | *operator*

numeral ::= a sequence of digits
identifier ::= a sequence of letters, digits and underscores (_),
  starting with a letter
operator ::= a "math operator" symbol such as +, ∗, <, <=, &, etc.

Here follow some salient features of the grammar that we think deserve discussion.

The *let* declarations generated by *let_formula* are in a sense global: although they can be nested, they must all come before the "main formula". In other words, *let* expressions are not allowed to appear below quantifiers or logical connectives. Consistently with this restriction, in expressions of the form (:**flet** $p\ \varphi\ \psi$) the formula φ cannot itself contain a *let* expression. These two restrictions as well are dictated by simplicity concerns. However, they are only tentative at the moment, and might be relaxed in the future if needed.

In this grammar we do not distinguish between constant and function symbols (they are all defined as *fun_symb*), and between propositional variables and predicate symbols (they are all defined as *pred_symb*). These distinctions are really a matter of arity, which is taken care of later by the well-sortedness rules. A similar observation applies to the logical connectives (the members of *connective*) and the number of arguments they are allowed take.

Finally, we do not enforce here any disjointness constraints between the sets of sort, function, and predicate symbols. Such constraints will be enforced later at the level of specific theories and benchmark sets.

In the rest of the paper, we will often treat a non-terminal symbols of the grammar above as the set of expressions it generates.

4.2 Well-sorted Formulas

The SMT-LIB language of formula is the largest set of well-sorted formulas contained in the language generated by the previous grammar. Well-sorted formulas are defined below by means of a set of sorting rules, similar in format and spirit to the kind of typing rules found in the programming languages literature.

We first provide a set of subsorting rules that define the subsort relation as a preorder on sort symbols. Then we provide a set of rules defining well-sorted terms, and another set defining well-sorted formulas.

To specify these rules we need to presuppose the existence of a sorted signature Σ, defined formally below. Strictly speaking then, the SMT-LIB language is a family of languages parametrized by Σ. As we will see later, for each benchmark ψ and theory T, the specific signature is going to be jointly defined by the specification of of T and that of the benchmark set containing ψ.

Definition 1. An *order-sorted signature* Σ is a tuple consisting of:

– a non-empty subset Σ^{S} of *sort_symb*, a subset Σ^{F} of *fun_symb*, a subset Σ^{P} of *pred_symb*, all pairwise disjoint,
– a binary relation $\Sigma^{<:}$ over Σ^{S},
– a mapping of the elements of *var* to elements of Σ^{S},
– a mapping of the elements of Σ^{F} to a non-empty sequence of elements of Σ^{S}, and
– a mapping of the elements of Σ^{P} to a possibly empty sequence of elements of Σ^{S}.

Note that the absence of overloading of function or predicate symbols in the SMT-LIB language is enforced by the last two mappings in the definition of signature.

To simplify the notation in this section we use the following meta-variables, possibly with subscripts: S, U and T ranging over *sort_symb*, x ranging over *var*, a and f ranging over *fun_symb*, p ranging over *pred_symb*, t ranging over *term*, k ranging over *connective*, φ ranging over *formula*, ψ ranging over *let_formula*, and n ranging over the non-negative integers.

We also use a distinguished sort symbol, denoted by F and assumed not to be in *sort_symb*, as the sort of well-sorted formulas.

In the following we will fix a signature Σ. Then, we will write $x : S \in \Sigma$ to mean that Σ maps the variable x to the sort S. We will write $f : S_1 \cdots S_{n+1} \in \Sigma$ to mean that $f \in \Sigma^{\mathrm{F}}$ and Σ maps f to the sort sequence $S_1 \cdots S_{n+1}$ (and similarly for predicate symbols). We will write $\Sigma, x : S$ to denote the signature that coincides with Σ except (possibly) that it maps x to S. Similarly, we will write $\Sigma, p : \epsilon$ to denote the signature that coincides with Σ except (possibly) that it contains the predicate symbol p and maps p to the empty string of sorts (in other words, declares p as a propositional variable).

Subsorting rules

$$\frac{S \in \Sigma^{\mathrm{S}}}{\Sigma \vdash S <: S} \qquad \frac{(S,T) \in \Sigma^{<:}}{\Sigma \vdash S <: T} \qquad \frac{\Sigma \vdash S <: U \qquad \Sigma \vdash U <: T}{\Sigma \vdash S <: T}$$

We say that a sort S is a *subsort* of a sort T (according to Σ) if $\Sigma \vdash S <: T$ is derivable by the rules above.

Note that the subsorting relation $<:$ induced by Σ coincides with the reflexive-transitive closure of $\Sigma^{<:}$.

Well-sortedness rules for terms

$$\frac{x : S \in \Sigma}{\Sigma \vdash x : S} \qquad \frac{f : S_1 \cdots S_{n+1} \in \Sigma}{\Sigma \vdash f : S_1 \cdots S_{n+1}} \qquad \frac{\Sigma \vdash t : S \quad \Sigma \vdash S <: T}{\Sigma \vdash t : T}$$

$$\frac{\Sigma \vdash f : S_1 \cdots S_{n+1} \quad \Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_n : S_n}{\Sigma \vdash (f\ t_1\ \cdots\ t_n) : S_{n+1}}$$

We call an element t of term *well-sorted* (with respect to Σ) if $\Sigma \vdash t : S$ is derivable by the sort rules above for some sort S. In that case, we also say that *t is of sort* S.

The rules are pretty standard and self-explanatory. The only interesting rule is perhaps the third one which basically says that any term of sort S is also of sort T for any supersort T of S. Subsort polymorphism is allowed in the language be virtue of this rule.

Well-sortedness rules for formulas

$$\frac{}{\Sigma \vdash \textbf{:true} : \mathsf{F}} \qquad \frac{}{\Sigma \vdash \textbf{:false} : \mathsf{F}} \qquad \frac{p : S_1 \cdots S_n \in \Sigma}{\Sigma \vdash p : S_1 \cdots S_n\ \mathsf{F}}$$

$$\frac{\Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_{n+2} : S_{n+2}}{\Sigma \vdash (=\ t_1\ \cdots\ t_{n+2}) : \mathsf{F}} \qquad \frac{\Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_{n+2} : S_{n+2}}{\Sigma \vdash (\textbf{:distinct}\ t_1\ \cdots\ t_{n+2}) : \mathsf{F}}$$

$$\frac{\Sigma \vdash p : S_1 \cdots S_n\ \mathsf{F} \quad \Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_n : S_n}{\Sigma \vdash (p\ t_1\ \cdots\ t_n) : \mathsf{F}}$$

$$\frac{\Sigma \vdash \varphi : \mathsf{F}}{\Sigma \vdash (\textbf{:not}\ \varphi) : \mathsf{F}} \qquad \frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \Sigma \vdash \varphi_2 : \mathsf{F}}{\Sigma \vdash (\textbf{:impl}\ \varphi_1\ \varphi_2) : \mathsf{F}} \qquad \frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \Sigma \vdash \varphi_2 : \mathsf{F}}{\Sigma \vdash (\textbf{:iff}\ \varphi_1\ \varphi_2) : \mathsf{F}}$$

$$\frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \Sigma \vdash \varphi_2 : \mathsf{F} \quad \Sigma \vdash \varphi_3 : \mathsf{F}}{\Sigma \vdash (\textbf{:ite}\ \varphi_1\ \varphi_2\ \varphi_3) : \mathsf{F}} \qquad \frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \cdots \quad \Sigma \vdash \varphi_n : \mathsf{F}}{\Sigma \vdash (k\ \varphi_1\ \cdots\ \varphi_n) : \mathsf{F}}$$

$$\frac{\Sigma,\ x : S \vdash \varphi : \mathsf{F}}{\Sigma \vdash (\textbf{:exists}\ (x\ S)\ \varphi) : \mathsf{F}} \qquad \frac{\Sigma,\ x : S \vdash \varphi : \mathsf{F}}{\Sigma \vdash (\textbf{:forall}\ (x\ S)\ \varphi) : \mathsf{F}}$$

$$\frac{\Sigma \vdash t : S \quad \Sigma, x : S \vdash \psi : \mathsf{F}}{\Sigma \vdash (\textbf{:let}\ x\ t\ \psi) : \mathsf{F}} \text{ if } t \text{ is ground}$$

$$\frac{\Sigma \vdash \varphi : \mathsf{F} \quad \Sigma, p : \epsilon \vdash \psi : \mathsf{F}}{\Sigma \vdash (\text{:}\mathbf{flet}\ p\ \varphi\ \psi) : \mathsf{F}} \text{ if } \varphi \text{ is closed and } p \notin \Sigma^{\mathrm{P}}$$

By *ground* and *closed* in the last two rules we mean containing no occurrences of elements of *var* and no free occurrences of elements of *var*, respectively, according to the standard meaning of free occurrence in $FOL^{=}$.

To reduce the level of nesting of formulas the *and*, *or*, and exclusive *or* connectives are defined, thanks to their associativity, as varyadic connectives taking zero or more arguments. The semantics of the connectives applied to zero arguments is the expected one: (:**and**) is equivalent to :**true**, while (:**or**) and (:**xor**) are both equivalent to :**false**.

We say that an element ψ of *let_formula* is *well-sorted* (with respect to Σ) if $\Sigma \vdash \psi : \mathsf{F}$ is derivable by the rules above.

Definition 2. The SMT-LIB language for formulas is the set of all closed well-formed formulas generated by *let_formula*.

Note that we consider closed formulas only. This is mostly a technicality, motivated by considerations of convenience. In fact, with a closed formula ψ of a signature Σ the particular mapping of variables to sorts defined by Σ is irrelevant. The reason is that the formula itself contains its own sort declaration for its variables, either explicitly, for the variables bound by a quantifier, or implicitly, for the variables bound by a *let*. Using only closed formulas then simplifies the task of specifying their signature, as it becomes unnecessary to specify how the signature maps the elements of *var* to the signature's sorts.

There is no loss of generality in this approach because, since we are interested in formula satisfiability, every formula $\varphi(\boldsymbol{x})$ with free variables $\boldsymbol{x}$ of sort $\boldsymbol{S}$ can be rewritten as $\exists \boldsymbol{x} : \boldsymbol{S}.\, \varphi(\boldsymbol{x})$. An alternative way to avoid free variables in benchmarks is proposed in Section 5.2

5 The SMT-LIB Format

In this section we propose a format for specifying theories and benchmark sets. As with formulas, this too is an attribute-value-based format. The main difference with formulas is that some of the attributes do not have a formally specified value—they just contain free text.

Ideally, a formal specification of these free-text attributes would be preferable to free text in order to avoid ambiguities and misinterpretation.

> *The choice of using free text for these attributes is motivated by practicality reasons: (i) these attributes are meant to be read by human readers, not programs and (ii) the amount of effort needed to devise a formal language for these attributes first and to specify their values in this language later does not seem justified by the current goals of SMT-LIB.*

As mentioned in the introduction, we propose that background theories and benchmark sets be specified separately in SMT-LIB. We described the proposed format for each in the following using the same kind of grammar used in Section 4.1 and also some of the non-terminal symbols defined there.

5.1 Specifying theories

In this version of the paper we consider the specification of basic theories only. We leave the problem of specifying combined theories to a later version.

> *A theory specification defines both an order-sorted signature for a theory and the theory itself.*

Formally, a theory specification is an element of the language generated by the non-terminal *theory* in the grammar below.

```
theory ::= ( :name string
             :sorts ( sort_symb+ )
             [:subsorts ( subsort_decl+ )]
             :funs ( fun_symb_decl+ )
             [:preds ( pred_symb_decl+ )]
             :definition string
             [:extensions string]
           )

subsort_decl   ::= ( sort_symb sort_symb )
fun_symb_decl  ::= ( fun_symb sort_symb+ )
pred_symb_decl ::= ( pred_symb sort_symb* )
```

Of all the elements of *theory* we consider only those that satisfy the following constraints:

1. sort symbols do not occur as function or predicate symbol as well, and similarly for function and predicate symbols—the sets of sort symbols names, function symbols, and predicate symbols are pairwise disjoint;
2. the sort symbols occurring in the :**subsorts**, :**funs** or :**preds** attribute are among the symbols listed in the :**sorts** attribute—all sort symbols used must be declared;
3. a function symbol does not occur more than once in :**funs**—no overloading of function symbols;
4. a predicate symbol does not occur more than once in :**preds**—no overloading of predicate symbols;
5. every sort symbol that occurs only on the left in the pairs of the :**subsorts** attribute occurs as the last symbol in a function symbol declaration in :**funs**—no empty sorts.

The signature of a theory is defined by the :**sorts**, :**subsort**, :**funs** and :**preds** attributes in the obvious way.

The :**subsort** and :**preds** attributes are optional because a theory might have a flat sort structure or lack predicate symbols. The :**sorts** attribute, however, is not optional. The way the language is designed, the signature must have at least one sort otherwise is is not possible to specify the interfaces of function and predicate symbols. This is no real limitation because for instance unsorted theories can be always seen as one-sorted.

The :**funs** attribute is also not optional. This is a consequence of constraint 5 above which forces the existence of at least one function symbol for at least one sort. The

purpose of this constraint is to guarantee that for each sort S with no subsorts there is at least one well-sorted term of sort S. This is a technicality, well-known in the many-sorted logic literature. Although it is not strictly necessary, it somewhat simplifies the semantics and the proof theory of many-sorted logics without real loss of generality. In our case it is almost invariably satisfied. When it is not, it is enough to add in the **:funs** attribute the declaration of a new constant symbol of the appropriate sort.

The **:definition** attribute is meant to contain a natural language definition of the theory. While this definition is expected to be as rigorous as possible, it does not have to be a formal one. Some theories (like the theory of real numbers) are well known, and so just a reference to their common name might be enough. For theories that have a small set of axioms (or axiom schemas), it might be convenient to list the actual axioms. For some other theories, a mix a formal notation and informal explanation might be more appropriate.

The optional **:extension** attribute is meant to document any notational conventions used in the listed benchmarks. This is useful because often the syntax of a theory is extended for convenience with syntactic sugar.[7]

5.2 Specifying Benchmarks

We propose to group benchmarks in benchmark sets.

> *The specification of a benchmark set contains, in addition to the benchmarks themselves, a reference to their background theory, a description of the language fragment to which the benchmarks belong, and an optional specification of additional function and predicate symbols.*

More formally, a benchmark set specification is an element of the language generated by the non-terminal *benchmark_set* in the grammar below.

benchmark_set ::= (**:name** *string*
  :theory *string*
  :language *string*
  [**:extra_funs** (*fun_symb_decl*$^+$)]
  [**:extra_preds** (*pred_symb_decl*$^+$)]

  :benchmarks (*benchmark*$^+$)
)

benchmark ::= (**:formula** *let_formula* **:status** *status*)
status ::= **:sat** | **:unsat** | **:unknown**

Of all the elements of *benchmark_set* we consider only those that satisfy the following constraints:

[7] Think for instance of Presburger arithmetic, where a numeral n abbreviates the n-fold application of the successor function to zero, and the predicate $t_1 \leq t_2$ abbreviates the formula $t_1 < t_2 \vee t_1 = t_2$.

1. the value of the :**theory** attribute coincides with the value of the :**name** attribute of some theory specification *tspec* in SMT-LIB;
2. the sort symbols occurring in (the value of) the :**extra_funs** or the :**extra_preds** attribute are among the symbols listed in the :**sorts** attribute of *tspec*;
3. a function symbol does not occur more than once in :**extra_funs**,
4. a predicate symbol does not occur more than once in :**extra_preds**;
5. a function symbol occurs neither in :**extra_preds** nor in *tspec*;
6. a predicate symbol occurs neither in :**extra_funs** nor in *tspec*;
7. all function (resp. predicate) symbols occurring in the :**benchmarks** attribute are declared either in *tspec* or in the :**extra_funs** (resp. :**extra_preds**) attribute.

The :**name** attribute provides a name for the set, for reference purposes.

The :**theory** attribute simply contains the name of a background theory specified in SMT-LIB.

The :**language** attribute specifies the specific subset of *let_formula* to which the listed benchmarks belong. The attribute is text valued because it has mostly documentation purposes. It is meant to help the benchmark user decide whether his solver can process the benchmark set. A natural language description of the sublanguage seems therefore adequate for this purpose.

The :**extra_funs** attribute complements the :**funs** attribute of the corresponding theory specification by declaring additional function symbols with their interface. The :**extra_preds** attribute has a similar purpose, but for predicate symbols. In contrast to the symbols possibly defined in the :**extensions** attribute of a theory specification, which are interpreted in terms of the symbols in the theory, the symbols in :**extra_funs** and :**extra_funs** are uninterpreted in associated theory.

Uninterpreted function or predicate symbols are found often in applications of satisfiability modulo theories, typically as a consequence of Skolemization or abstraction transformations applied to more complex formulas. Hence SMT-solvers typically accept formulas containing uninterpreted symbols in addition to the symbols of their background theory. The :**extra_funs** and :**extra_preds** attributes serve to declare any uninterpreted symbols occurring in benchmarks listed in the :**benchmarks** attribute. The value of :**extra_funs** and :**extra_preds** attributes is specified formally because, in effect, it dynamically expand the signature of the associated background theory, hence it is convenient for it to be directly readable by satisfiability procedures for that theory.

The :**extra_funs** attribute is also useful for specifying benchmarks with free variables (such as quantifier-free ones). As mentioned in Section 4.1, our language does not allow benchmarks with free variables. As we saw, one way to circumvent this restriction is to close such formulas existentially. Another one is to replace the free variables by fresh constant symbols of the proper sort. In the second case, these constant symbols are declared in the :**extra_funs** attribute.

The :**benchmarks** attribute lists the actual benchmarks—formally defined as members of *let_formula*—specifying for each of them whether the benchmark is known to be (un)satisfiable in the background theory. Knowing about the satisfiability of a benchmark is useful for debugging new solvers.

At the moment we require that all function or predicates symbols in a benchmark set be declared either in the corresponding theory specification or in the **:extra_funs** and **:extra_preds** attribute. Depending on the feedback we will get from the SMT-LIB interest group, this restriction might be relaxed in a later version of this paper. Specifically, one might think of following the convention that any undeclared function or predicate symbol occurring in a benchmark is automatically considered as uninterpreted. Technically, this is feasible if we add to the proposed sort system a predefined top sort, i.e., a sort that is by definition a supersort of any other sort. In that case, an interface for a symbol (i.e. its input and output sorts) can be alway inferred automatically from the formula. We point out, however, that in essence this requires a solver to be able to do type inference, as opposed to just type checking, a considerable more complex task in case of signatures with subsorts.

6 Semantics

The semantics of the SMT-LIB language is provided by a translation of let_formulas *and their signature into formulas of $FOL^{=}$.*

We define a translation operator $\mathbf{Tr}(_)$ below, following a well-known relativization process for translating many-sorted logics into classical logic. Formally, the semantics of SMT-LIB benchmarks is defined as follows.

Definition 3. Let Σ be an order-sorted signature generated by a set Sub of *subsort_decl*'s, a set F of *fun_symb_decl*'s, and a set P of *fun_symb_decl*'s. Let T be a theory (of signature Σ) axiomatized by a set Ax of closed formulas well-sorted with respect to Σ. Then let ψ be a closed formula well-sorted with respect to Σ. We say that ψ is satisfiable in T iff the set

$$\{\mathbf{Tr}(sd) \mid sd \in Sub\} \cup \{\mathbf{Tr}(fd) \mid fd \in F\} \cup \{\mathbf{Tr}(ax) \mid ax \in Ax\} \cup \{\mathbf{Tr}(\psi)\}$$

of $FOL^{=}$ sentences is satisfiable in the classical sense.

The particular sets Sub, F, P, Ax in the definition above are meant to be elicited from the specification of a benchmark set containing the formula ψ and from the specification of the benchmark set's background theory.

A fine point to note about our translation semantics is that it effectively requires all background theories of SMT-LIB to be defined axiomatically (so that one can identify the set Ax needed in Definition 3). Defining the background theories by a set of axioms, as opposed to a set of models, say, looks like the only option if we want to avoid specifying a native algebraic semantics—one with order-sorted models—for our language, instead of the translation semantics below. Further discussion on this point is needed.

The translation operator $\mathbf{Tr}(_)$ is defined in the following on sort declarations, function symbol declarations, and formulas. To describe the translation into $FOL^{=}$, we use a conventional syntax for $FOL^{=}$ sentences and abuse the notation a bit by sometimes ignoring the fact that the terms of the SMT–language have a LISP-like syntax instead of the usual mathematical notation.

Translation of sort declarations and function symbols declarations

$$\mathbf{Tr}(\textit{sort_decl}) = \mathbf{Tr}((S_1\ S_2)) = \forall x\ (S_1(x) \Rightarrow S_2(x))$$

$$\begin{aligned}\mathbf{Tr}(\textit{fun_symb_decl}) &= \mathbf{Tr}((f\ S_1 \cdots S_{n+1})) \\ &= \forall x_1, \ldots, x_n\ (S_1(x_1) \wedge \cdots \wedge S_n(x_n) \Rightarrow S_{n+1}(f(x_1, \ldots, x_n)))\end{aligned}$$

Translation of formulas For formulas, the translation operator is defined inductively on the structure of *let_formula*.

Recall that variable occurrences in a formula can be bound by a quantifier or by a :**let** binder, while propositional variable occurrences in a formula can be bound by an :**flet** binder. In the following, we denote by $\psi\{x \mapsto t\}$ the formula obtained from the formula ψ by simultaneously replacing every unbound occurrence of the variable x in ψ by the term t. Similarly, we denote by $\psi\{p \mapsto \varphi\}$ the formula obtained from ψ by simultaneously replacing every unbound occurrence of the propositional variable p in ψ by the formula φ.

$$\begin{aligned}
\mathbf{Tr}((\text{:}\mathbf{let}\ a\ t\ \psi)) &= \mathbf{Tr}(\psi\{x \mapsto t\}) \\
\mathbf{Tr}((\text{:}\mathbf{flet}\ p\ \varphi\ \psi)) &= \mathbf{Tr}(\psi\{p \mapsto \varphi\}) \\
\mathbf{Tr}((\text{:}\mathbf{not}\ \varphi)) &= \neg\mathbf{Tr}(\varphi) \\
\mathbf{Tr}((\text{:}\mathbf{and}\ \varphi_1\ \cdots \varphi_n)) &= \mathbf{Tr}(\varphi_1) \wedge \cdots \wedge \mathbf{Tr}(\varphi_n) \\
\mathbf{Tr}((\text{:}\mathbf{or}\ \varphi_1\ \cdots \varphi_n)) &= \mathbf{Tr}(\varphi_1) \vee \cdots \vee \mathbf{Tr}(\varphi_n) \\
\mathbf{Tr}((\text{:}\mathbf{xor}\ \varphi_1\ \cdots \varphi_n)) &= \mathbf{Tr}(\varphi_1) \oplus \cdots \oplus \mathbf{Tr}(\varphi_n) \\
\mathbf{Tr}((\text{:}\mathbf{ite}\ \varphi_1\ \varphi_2\ \varphi_3)) &= (\mathbf{Tr}(\varphi_1) \Rightarrow \mathbf{Tr}(\varphi_2)) \wedge (\neg\mathbf{Tr}(\varphi_1) \Rightarrow \mathbf{Tr}(\varphi_3)) \\
\mathbf{Tr}((\text{:}\mathbf{exists}\ (x\ S)\ \varphi)) &= \exists x\ (S(x) \wedge \mathbf{Tr}(\varphi)) \\
\mathbf{Tr}((\text{:}\mathbf{forall}\ (x\ S)\ \varphi)) &= \forall x\ (S(x) \Rightarrow \mathbf{Tr}(\varphi)) \\
\mathbf{Tr}(\text{:}\mathbf{false}) &= \bot \\
\mathbf{Tr}(\text{:}\mathbf{true}) &= \neg\bot \\
\mathbf{Tr}(p) &= p \\
\mathbf{Tr}((p\ t_1\ \cdots\ t_{n+1})) &= p(t_1, \ldots, t_{n+1}) \\
\mathbf{Tr}((=\ t_1\ \cdots\ t_{n+2})) &= t_1 = t_2 \wedge \cdots \wedge t_{n+1} = t_{n+2} \\
\mathbf{Tr}((\text{:}\mathbf{distinct}\ t_1\ \cdots\ t_{n+2})) &= \textstyle\bigwedge_{1 \le i < j \le n+2} \neg(t_i = t_j)
\end{aligned}$$

7 Examples

In this section we present some examples of theory and benchmark set specifications written in the proposed format.

Most examples are about the theory of real numbers because this theory is particularly apt at illustrating the various aspects of the format, and in particular the point of distinguishing between logic, theories, and input language in SMT-LIB.

7.1 The theory of real numbers

There really is one theory T_R that captures the real numbers, the one that logicians usually call the theory of ordered real-closed fields. The literature, however, is full with theories and "logics" that one way or another have to do with the real numbers. For

the purposes of SMT-LIB, most of these various theories and logics are best described in terms of the restriction that they impose on the class of formulas considered for satisfiability in T_R.

For instance, the satisfiability of formulas in so called "linear arithmetic" can be described as the satisfiability in T_R of formulas built as conjunctions of equations and inequations ($\leq$) between *linear* terms, that is, terms reducible to linear polynomials.

Similarly, the satisfiability of formulas in so the called "separation logic" or "difference logic"can be described as the satisfiability in T_R of conjunctions of formulas of the form $m * (x - y) < m$ where x and y are variables and m, n are two numerals.

To avoid an undue proliferation of theories in STM-LIB we would specify only the full first-order theory of the real numbers, and consider its restrictions only at the level of benchmark sets. Similar considerations of course would apply to other theories such as, for instance, the full-first order theory of the natural numbers and the restriction of it known as Presburger arithmetic.

We present two versions of the theory of the real numbers to highlight the different possibilities offered by the order-sorted framework.

Real numbers I

```
(:name "REALS-I"
 :sorts (Real)
 :funs ((0 Real) (1 Real)
        (~ Real Real)
        (+ Real Real Real) (* Real Real Real))
 :preds ((< Real Real))
 :definition "The standard, one-sorted theory of real numbers"
 :extensions "A numeral n > 1 abbreviates the sum (+ 1 ... (+ 1 1))
              of n ones;
              (- t_1 t_2) abbreviates (+ t_1 (~ t_2));
              (<= t_1 t_2) abbreviates (:or (< t_1 t_2) (= t_1 t_2))
              and similarly for >= and >"
)
```

A benchmark set for this theory might look like the following.

```
(:name "RB1"
 :theory "REALS-I"
 :language "Conjunctions of linear ground equations and inequations"
 :extra_funs ((x Real) (y Real) (f Real Real))
 :benchmarks (
  (:formula (:and (= (+ (* 5 x) y) 0)
                  (= (- x y) 4))
   :status :sat)
  (:formula (:and (= (+ x (+ y (f x))) 0)
                  (<= (- x y) 4)
                  (< (* 4 x) (f x)))
   :status :sat))
)
```

Note how this example uses the uninterpreted constants `x` and `y` as the unknowns of the first benchmark, and how the second benchmark also contains occurrences of the uninterpreted function symbol `f`.

The second example considers first-order benchmarks with no extra symbols.

```
(:name "RB2"
 :theory "REALS-I"
```

```
 :language "all first-order formulas"
 :benchmarks (
  (:formula (:not (:forall (?x Real) (:forall (?y Real)
                   (impl (< ?x ?y)
                         (:exists (?z Real) (:and (< ?x ?z) (< ?z ?y)))))))
   :status :unsat))
)
```

The third example considers difference logic-like benchmarks.

```
(:name "RB3"
 :theory "REALS-I"
 :language "Boolean combinations of atoms of the form
            (op (* m (- x y)) n)
            where op is =, <, <, >, or >=,
            m, n are numerals and x,y are variables."
 :extra_funs ((x Real) (y Real) (z Real))
 :benchmarks (
  (:formula (:or (:and (<= (- x y) 0)
                       (= (- x z) 7))
                 (< (* 3 (- y z)) 1))
   :status :unknown))
)
```

Real numbers II

Here is a second take on the theory of real numbers, motivated by the fact that in some applications one is often interested in restricting the range of problem variables to integer values only. For those applications an order-sorted version of the theory seems more convenient.

```
(:name "REALS-II"
 :sorts (Nat Int Real)
 :subsorts ((Nat Int) (Int Real))
 :funs ((0 Nat) (1 Nat)
        (~ Real Real)
        (+ Real Real Real) (* Real Real Real))
 :preds ((< Real Real))
 :definition "The standard theory of real numbers over the sort Real,
              together with the Nat and Int subsorts defined by the
              following axioms, which specify that the sort Nat is
              generated by 0,1 and +, while the sort Int is generated
              by 0,1, + and ~(unary minus).
              (:forall (?x Real)
               (:iff (:exists (?n Nat) (= ?x ?n))
                     (:or (= ?x 0)
                          (exists (?m Nat) (= ?x (+ ?m 1))))))
              (:forall (?x Real)
               (:iff (:exists (?z Int) (= ?x ?z))
                     (:exists (?n Nat) (:or (= ?x ?n) (= ?x (~ ?n))))))"
 :extensions "A numeral n > 1 abbreviates the sum (+ 1 ... (+ 1 1))
              of n ones.
              (- t_1 t_2) abbreviates (+ t_1 (~ t_2))
              (<= t_1 t_2) abbreviates (:or (< t_1 t_2) (= t_1 t_2))"
)
```

The following benchmark set should make clear that subsorts so add to the power of the language, as changing sort constraints on a formula's variables changes its satisfiability in the theory.

```
(:name "RB4"
 :theory "REALS-II"
 :language "Conjunctions of linear equations"
 :extra_funs ((x Real) (y Real) (n Int) (m Int))
 :benchmarks (
  (:formula (:and (= (+ x y) 1)
                  (= (- (* 3 x) (* 3 y)) 1))
   :status :sat)
  (:formula (:and (= (+ m n) 1)
                  (= (- (* 3 n) (* 3 n)) 1))
   :status :unsat))
)
```

The benchmarks in this set represent the same system of equations. The only difference is that the system's unknowns are required to be real valued in the first case and integer valued in the second case. Since the system admits only a non-integer solution, the first benchmark is satisfiable in the theory while the second is not.

7.2 The theory of arrays

We conclude by providing a specification for another popular theory in the SMT literature, the theory of functional arrays (without the extensionality axiom). The specification of this theory is unproblematic because the theory is defined by just two axioms.

```
(:name "ARRAYS-I"
 :sorts (Index Element Array)
 :funs ((select Array Index Element) (store Array Index Element Array))
 :definition "The theory of arrays defined by the following axioms:
              (:forall (?a Array) (:forall (?i Index)
              (:forall (?e Element)
                (= (select (store ?a ?i ?e) ?i) ?e))))
              (:forall (?a Array)
              (:forall (?i Index) (:forall (?j Index)
              (:forall (?e Element) (:or (= ?i ?j)
                                         (= (select (store ?a ?i ?e) ?j)
                                            (select (?a ?j))))))))"
)
```

8 Acknowledgments

We would like to thank the following members of the SMT-LIB interest group, in alphabetical order, for their initial suggestions and comments on the SMT-LIB format: Clark Barrett, Greg Nelson and the members of Sparta group at HP SRC, Harald Ruess, and Aaron Stump.

References

1. Holger Hoos and Thomas Stützle. SATLIB–The Satisfiability Library. Web site at: `http://www.satlib.org/`.
2. Silvio Ranise and Cesare Tinelli. SMT-LIB–The Satisfiability Modulo Theories Library. Web site at: `http://combination.cs.uiowa.edu/smtlib/`.
3. Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library for Automated Theorem Proving. Web site at: `http://www.cs.miami.edu/ tptp/`.

Author Index

Armando, A., 61
Audemard, G., 76

Barrett, C., 39
Berezin, S., 39
Bozzano, M., 76

Cantone, D., 14
Cimatti, A., 76

Deivanayagam, A., 24

Formisano, A., 14

Kathol, S., 24

Lingelbach, D., 24

Omodeo, E.G., 14

Ranise, S., 111

Schobel, D., 24
Schwarz, J.T., 14
Sebastiani, R., 76
Stickel, M., 1
Stump, A., 24

Tinelli, C., 111
Tiwari, A., 52

Velev, M., 76