Proceedings of the 9th International Workshop on Satisfiability Modulo Theories (SMT) 2011

Shuvendu Lahiri, Ed.
Sanjit A. Seshia, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-80
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-80.html

July 4, 2011

Acknowledgement

The support of UC Berkeley and industrial sponsors Microsoft Research, NEC Labs, and Intel is gratefully acknowledged.

Proceedings of SMT'11:
9th International Workshop on Satisfiability Modulo Theories

Co-located with the 23rd International Conference on Computer-Aided Verification (CAV), 2011

Editors: Shuvendu Lahiri (Microsoft Research) and Sanjit A. Seshia (UC Berkeley)

July 4, 2011

This technical report serves as an informal proceedings for the 9th International Workshop on Satisfiability Modulo Theories (SMT'11), held as a satellite workshop of the 2011 Interational Conference on Computer-Aided Verification (CAV), held in Snowbird, Utah, USA.

The workshop spans two days. The program comprises two morning keynote talks, and presentations of peer-reviewed full articles, extended abstracts, and presentation-only papers that have appeared elsewhere. These proceedings include a brief statement of the goals of the workshop, the workshop program, abstracts for the invited talks and presentation-only papers, and full-length papers.

Motivation and Goals of the Workshop

Determining the satisfiability of first-order formulas modulo background theories, known as the Satisfiability Modulo Theories (SMT) problem, has proved to be an enabling technology for verification, synthesis, test generation, compiler optimization, scheduling, and other areas. The success of SMT techniques depends on the development of both domain-specific decision procedures for each background theory (e.g., linear arithmetic, the theory of arrays, or the theory of bit-vectors) and combination methods that allow one to obtain more versatile SMT tools, usually leveraging Boolean satisfiability (SAT) solvers. These ingredients together make SMT techniques well-suited for use in larger automated reasoning and verification efforts.

The aim of the SMT 2011 workshop is to bring together researchers working on SMT and users of SMT techniques. Relevant topics include, but are not limited to:

- New decision procedures and new theories of interest;
- Combinations of decision procedures;
- Novel implementation techniques;
- Benchmarks and evaluation methodologies;
- Applications and case studies, and
- Theoretical results.

Workshop Program

Day 1: July 14

Time	Details	Page No.
8:45 – 9:00	Welcome and Introduction	
9:00 – 10:00	INVITED TALK *Viktor Kuncak.* Software Construction using Executable Constraints.	5
10:00 – 10:30	Break I	
	INTERPOLANTS	
10:30 – 11:00	*Maria Paola Bonacina and Moa Johansson.* Towards interpolation in an SMT solver with integrated superposition.	9
11:00 – 11:30	*Amit Goel, Sava Krstic, Rupak Majumdar and Sai Deep Tetali.* Quantified Interpolation for SMT.	19
11:30 – 12:00	*Andrew Reynolds, Cesare Tinelli and Liana Hadarean.* Certified Interpolant Generation for EUF.	30
12:00 – 2:00	Lunch	
	DECISION PROCEDURES I	
2:00 – 2:30	*David Deharbe, Pascal Fontaine, Stephan Merz and Bruno Woltzenlogel Paleo.* Exploiting Symmetry in SMT Problems (presentation only).	6
2:30 – 3:00	*Ruzica Piskac and Thomas Wies.* Decision Procedures for Automating Termination Proofs (presentation only).	6
3:00 – 3:30	*Thomas Wies, Marco Muniz and Viktor Kuncak.* An Efficient Decision Procedure for Imperative Tree Data Structures (presentation only).	6
3:30 – 4:00	Break II	
	DECISION PROCEDURES II	
4:00 – 4:30	*Dejan Jovanovic and Leonardo De Moura.* Cutting to the Chase: Solving Linear Integer Arithmetic (presentation only).	7
4:30 – 5:00	*Duckki Oe and Aaron Stump.* Extended Abstract: Combining a Logical Framework with an RUP Checker for SMT Proofs.	40
5:00 – 6:00	SMT Business Meeting	

Day 2: July 15

Time	Details	Page No.
9:00 – 10:00	INVITED TALK *Bud Mishra.* When Biology Meets (Symbolic) Computing: Algebra, Biology, Computability and Diophantus.	5
10:00 – 10:30	Break I	
	MODEL CHECKING AND SYNTHESIS	
10:30 – 11:00	*Hyondeuk Kim, Fabio Somenzi and Hoonsang Jin.* Selective SMT Encoding for Hardware Model Checking.	49
11:00 – 11:30	*Timothy King and Clark Barrett.* Exploring and Categorizing Error Spaces using BMC and SMT.	59
11:30 – 12:00	*Steve Haynal and Heidi Haynal.* Generating and Searching Families of FFT Algorithms (presentation only).	7
12:00 – 2:00	Lunch	
	SMT FOR SOFTWARE	
2:00 – 2:30	*Arie Gurfinkel, Sagar Chaki and Samir Sapra.* Efficient Predicate Abstraction of Program Summaries (presentation only).	7
2:30 – 3:00	*Mahmoud Said, Chao Wang, Zijiang Yang and Karem Sakallah.* Generating Data Race Witnesses by an SMT-based Analysis (presentation only).	8
3:00 – 3:30	*Stephan Falke, Florian Merz and Carsten Sinz.* A Theory of C-Style Memory Allocation.	71
3:30 – 4:00	Break II	
4:00 - 6:00	SMTLIB INITIATIVE AND SMTCOMP	

Program Committee

- Clark Barrett (NYU)
- Maria Paola Bonacina (Universit degli Studi di Verona)
- Alessandro Cimatti (FBK Trento)
- Scott Cotton (Atrenta)
- Bruno Dutertre (SRI International)
- Martin Frnzle (Carl von Ossietzky Universitt Oldenburg)
- Amit Goel (Intel)
- Franjo Ivancic (NEC Labs)
- Vijay Ganesh (MIT)
- Daniel Kroening (Oxford)
- Andreas Kuehlmann (Coverity)
- Shuvendu K. Lahiri (Microsoft Research) (Co-chair)
- Leonardo de Moura (Microsoft Research)
- Sanjit Seshia (UC Berkeley) (Co-chair)
- Ofer Strichman (Technion)
- Cesare Tinelli (Iowa)

Abstracts of Invited Talks

Software Construction using Executable Constraints.
Viktor Kuncak (EPFL)

Constructing software that satisfies the desired properties can greatly benefit from solvers based on satisfiability modulo theories (SMT) paradigm. We propose a research program in which software construction and SMT technology become even more interconnected than today. Instead of checking properties of low-level legacy programs, new generations of SMT solvers could focus on constructing programs and values within compilers and run-time systems of declarative programming languages designed with constraint solving in mind. This agenda has implications on both theories and interfaces of future SMT solvers.

The theories of interest come in part from functional programming languages and from data structure libraries. They include algebraic data types (term algebras), arrays, maps, sets, and multisets, extended with recursively defined functions. New interfaces for SMT solvers are driven by their extended role: in addition to yes/no satisfiability answers, we need synthesis of values and programs. To answer declarative queries at run-time, we need efficient generation and fair enumeration of models, as well as their mapping to programming language values. To support the compilation of implicit specifications, we need SMT solvers that can solve parameterized problems and perform constraint simplification. Ultimately, SMT solvers can become virtual machines for constraint programming, working with data structures directly manipulated by programs, and performing selective specialization of frequently invoked constraint solving paths.

This is joint work with: Barbara Jobstmann, Ali Sinan Koeksal, Ruzica Piskac, and Philippe Suter. More information at http://lara.epfl.ch/w/impro and http://richmodels.org .

When Biology Meets (Symbolic) Computing: Algebra, Biology, Computability and Diophantus.
Bud Mishra (NYU)

In this talk, I will introduce a new approach to compositional and hierarchical modeling of biological systems and its relations to certain problems in algebra and algorithmics: namely, decision procedures for systems of linear Diophantine equations and inequalities, whose coefficients range over algebraic numbers and intervals. The questions, addressed here, are central to the success of the emerging field of systems biology and relate to questions in decidability theory, algorithmic algebra, hybrid automata models, etc.

Abstracts of Presentation-Only Papers

Presentation-only papers are those that have appeared (or will appear) at other conference or journal venues, but which have been judged by the program committee to be of interest to the SMT community.

Exploiting Symmetry in SMT Problems.
David Deharbe, Pascal Fontaine, Stephan Merz and Bruno Woltzenlogel Paleo.

Methods exploiting problem symmetries have been very successful in several areas including constraint programming and SAT solving. We here recast a technique to enhance the performance of SMT-solvers by detecting symmetries in the input formulas and use them to prune the search space of the SMT algorithm. This technique is based on the concept of (syntactic) invariance by permutation of constants. An algorithm for solving SMT by taking advantage of such symmetries is presented. The idea here is very simple: given a formula G invariant by all permutations of some uninterpreted constants $c_0, \ldots, c_n$, for any model M of G, if term t does not contain these constants and M satisfies $t = c_i$ for some i, then there should be a model in which t equals c_0. While checking for unsatisfiability, it is thus sufficient to look for models assigning t and c_0 to the same value. We present a heuristic to guess symmetries, and we show that checking if a formula is invariant by all permutations of some uninterpreted constants can be done in linear time. The overhead of this technique is thus very small, and we show that the technique can account for an exponential decrease of running times on some series of crafted benchmarks based on the pigeonhole problem. The implementation of this algorithm in the SMT-solver veriT is used to illustrate the practical benefits of this approach. It results in a significant improvement of veriT's performances on the SMT-LIB benchmarks that places it ahead of the winners of the last editions of the SMT-COMP contest in the QF_UF category. Used as a preprocessing technique, similar improvements are observed for all the five other tested SMT solvers.

Decision Procedures for Automating Termination Proofs.
Ruzica Piskac and Thomas Wies.

Automated termination provers often use the following schema to prove that a program terminates: construct a relational abstraction of the program's transition relation and show that the relational abstraction is well-founded. The focus of current tools has been on developing sophisticated techniques for constructing the abstractions while relying on known decidable logics (such as linear arithmetic) to express them. We believe we can significantly increase the class of programs that are amenable to automated termination proofs by identifying more expressive decidable logics for reasoning about well-founded relations. We therefore present a new decision procedure for reasoning about multiset orderings, which are among the most powerful orderings used to prove termination. We show that, using our decision procedure, one can automatically prove termination of natural abstractions of programs.

An Efficient Decision Procedure for Imperative Tree Data Structures.
Thomas Wies, Marco Muniz and Viktor Kuncak.

We present a new decidable logic called TREX for expressing constraints about imperative tree data

structures. In particular, TREX supports a transitive closure operator that can express reachability constraints, which often appear in data structure invariants. We show that our logic is closed under weakest precondition computation, which enables its use for automated program verification. We further show that satisfiability of formulas in TREX is decidable in NP. The low complexity makes it an attractive alternative to more expensive logics such as monadic second-order logic (MSOL) over trees, which have been traditionally used for reasoning about tree data structures.

Cutting to the Chase: Solving Linear Integer Arithmetic.
Dejan Jovanovic and Leonardo De Moura.

We describe a new algorithm for solving linear integer programming problems. The algorithm performs a DPLL style search for a feasible assignment, while using a novel cut procedure to guide the search away from the conflicting states. This paper was accepted at CADE'11.

Generating and Searching Families of FFT Algorithms.
Steven Haynal and Heidi Haynal.

A fundamental question of longstanding theoretical interest is to prove the lowest exact count of real additions and multiplications required to compute a power-of-two discrete Fourier transform (DFT). For 35 years the split-radix algorithm held the record by requiring just $4n \log n - 6n + 8$ arithmetic operations on real numbers for a size-n DFT, and was widely believed to be the best possible. Recent work by Van Buskirk et al. demonstrated improvements to the split-radix operation count by using multiplier coefficients or "twiddle factors" that are not nth roots of unity for a size-n DFT.

This paper presents a Boolean Satisfiability-based proof of the lowest operation count for certain classes of DFT algorithms. First, we present a novel way to choose new yet valid twiddle factors for the nodes in flowgraphs generated by common power-of-two fast Fourier transform algorithms, FFTs. With this new technique, we can generate a large family of FFTs realizable by a fixed flowgraph. This solution space of FFTs is cast as a Boolean Satisfiability problem, and a modern Satisfiability Modulo Theory solver is applied to search for FFTs requiring the fewest arithmetic operations. Surprisingly, we find that there are FFTs requiring fewer operations than the split-radix even when all twiddle factors are nth roots of unity.

Efficient Predicate Abstraction of Program Summaries.
Arie Gurfinkel, Sagar Chaki and Samir Sapra.

Predicate abstraction is an effective technique for scaling Software Model Checking to real programs. Traditionally, predicate ab- straction abstracts each basic block of a program P to construct a small finite abstract model a Boolean program BP , whose state-transition relation is over some chosen (finite) set of predicates. This is called Small- Block Encoding (SBE). A recent advancement is Large-Block Encoding (LBE) where abstraction is applied to a summarized program so that the abstract transitions of BP correspond to loop-free fragments of P. In this paper, we expand on the original notion of LBE to promote flexibility. We explore and describe efficient ways to perform CEGAR bot- tleneck operations: generating and solving predicate abstraction queries (PAQs). We make the following contributions. First, we define a gen-

eral notion of program summarization based on loop cutsets. Second, we give a linear time algorithm to construct PAQs for a loop-free fragment of a program. Third, we compare two approaches to solving PAQs: a classical AllSAT-based one, and a new one based on Linear Decision Diagrams (LDDs). The approaches are evaluated on a large benchmark from open- source software. Our results show that the new LDD-based approach significantly outperforms (and complements) the AllSAT one.

Generating Data Race Witnesses by an SMT-based Analysis.
Mahmoud Said, Chao Wang, Zijiang Yang and Karem Sakallah.

Data race is one of the most dangerous errors in multi-threaded programming, and despite intensive studies, it remains a notorious cause of failures in concurrent systems. Detecting data races, statically or dynamically, is already a hard problem, and yet it is even harder for a programmer to decide whether or how a reported data race can appear in the actual program execution. In this paper we propose an algorithm for generating debugging aid information called witnesses, which are concrete thread schedules that can deterministically trigger the data races. More specifically, given a concrete execution trace of the program, which may be non-erroneous but have triggered a warning in Eraser-style data race detectors, we use a symbolic analysis based on SMT solvers to search for such a witness among alternative interleavings of events of that trace. Our symbolic analysis precisely encodes the sequential consistency semantics using a scalable predictive model to ensure that the reported witness is always feasible.

Towards interpolation in an SMT-solver with integrated superposition*

Maria Paola Bonacina **Moa Johansson**
Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, I-39134 Verona, Italy
mariapaola.bonacina@univr.it *moakristin.johansson@univr.it*

Abstract

Interpolation is a technique for extracting intermediate formulæ from a proof. It has applications in formal verification, where interpolation may enable a program analyser to discover information about intermediate program locations and states. We study interpolation in the theorem proving method DPLL($\Gamma+\mathcal{T}$), which integrates tightly a superposition based prover Γ in a DPLL($\mathcal{T}$) based SMT-solver to unite their respective strengths. We show how a first interpolation system for DPLL($\Gamma+\mathcal{T}$) can be obtained from interpolation systems for DPLL, equality sharing and Γ. We describe ongoing work on an interpolation system for Γ, by presenting and proving complete an interpolation system for the ground case, followed by a discussion of ongoing work on an extension to the general case. Thanks to the modular design of DPLL($\Gamma+\mathcal{T}$), its interpolation system can be extended easily beyond the ground case once a general interpolation system for Γ becomes available.

1 Introduction

Interpolation is a theorem proving technique which has recently found several applications in verification. Informally, interpolants are formulæ 'in-between' other fomulæ in a proof: for a proof of $A \vdash B$ with interpolant I, $A \vdash I$ and $I \vdash B$, with I only containing symbols shared between A and B. Interpolation was first proposed for *abstraction refinement* in software model checking, initially for propositional logic and propositional satisfiability [16], and then for quantifier free fragments of first-order theories and their combinations [21, 17, 11, 5, 8, 3]. In the Counter-Example Guided Abstraction Refinement paradigm, interpolants from the proof of unsatisfiability of the formula produced from a spurious counter-example may capture intermediate states in an error trace, and can be used to refine the abstraction by re-introducing predicates from the interpolants to exclude states leading to spurious errors.

Interpolation has also found applications for *invariant generation* in the context of inference systems for first-order logic with equality [18, 13, 9]. Here, one assumes that a k-step unwinding of a loop does not satisfy the post-condition. The formulæ expressing this produces a contradiction if the loop *does* satisfy the post-condition. An interpolant, containing only the symbols occurring in the loop body, can be extracted and used to guide the construction of a loop invariant [18].

A third application of interpolation is to supplement *annotation generation* by a weakest pre-condition calculus [19]. In this context, interpolation allows a static analyser to avoid inserting irrelevant program variables in annotations, such as procedure summaries.

*Research supported in part by grant no. 2007-9E5KM8 of the Ministero dell'Istruzione Università e Ricerca, Italy, and by COST Action IC0901 of the European Union.

The aim of this work is to develop an interpolation system for DPLL($\Gamma + \mathcal{T}$) [2], a new theorem proving method which integrates a first-order inference system Γ, based on resolution and superposition, into the DPLL($\mathcal{T}$) framework for satisfiability modulo theories. The motivation for DPLL($\Gamma + \mathcal{T}$) is to unite the strengths of resolution based provers, such as automated treatment of quantifiers, with those of SMT-solvers, such as built-in theories and scalability on large ground problems. All these features are crucial for applications to verification. For instance, formulæ with quantifiers are necessary to state invariants and to axiomatise theories without decision procedures. Heuristic techniques for instantiating variables in SMT-solvers can be used, but they can be fragile and require a lot of user effort to get right [15]. Thus, DPLL(Gamma+T) has properties attractive to the application areas, exemplified above, where also interpolation has uses. Hence an interpolating version of DPLL($\Gamma + \mathcal{T}$) would be of interest for the formal verification community. The work described in this paper is still in progress; we describe how a first interpolation system for DPLL(Γ+$\mathcal{T}$), thanks to its modular design, is built from interpolation systems for DPLL, equality sharing and Γ.

We will use the propositional interpolation system for DPLL independently discovered by Huang, Krajíček and Pudlàk [10, 20, 14], later reformulated and proved correct in the context of satisfiability modulo theories by Yorsh and Musuvathi [21]. We call this algorithm HKPYM from the initials of the authors. Yorsh and Musuvathi also gave an interpolation system for equality sharing, which we refer to as EQSH [21]. EQSH requires that the satisfiability procedures for the built in theories can produce proofs and interpolants. Then HKPYM and EQSH can be integrated to yield an interpolation system for DPLL($\mathcal{T}$) [21, 5, 8]. What remains for an interpolation system for DPLL($\Gamma + \mathcal{T}$) is an interpolation system for Γ. We present a novel complete interpolation system for Γ in the ground case and give a modular interpolation system for DPLL($\Gamma + \mathcal{T}$). We consider our interpolation system for superposition to be clearer and more general than previous work, because its working is specified for each generative inference, which was not done before. We conclude with a discussion of related work and an overview of ongoing work aiming at extending the ground interpolation system for Γ to proofs involving substitutions, under suitable restrictions. The interpolation system for DPLL(Γ+$\mathcal{T}$) is currently restricted to the ground case, but easily extendable to the non-ground case once such an interpolation system for Γ is available, which is the ultimate goal of this project.

2 Preliminaries

We assume the basic definitions commonly used in theorem proving. Equality in the inference systems will be denoted by $\simeq$ and the symbol $\bowtie$ stands for either $\simeq$ or $\not\simeq$.

Let A and B be two formulæ. We denote by Σ_A, and Σ_B, the set of constant, function and predicate symbols that occur in A, and B, respectively, and we use $\setminus$ for set difference. A non-variable symbol is *A-coloured*, if it is in $\Sigma_A \setminus \Sigma_B$, *B-coloured*, if it is in $\Sigma_B \setminus \Sigma_A$, and *transparent*, if it is in $\Sigma_T = \Sigma_A \cap \Sigma_B$. This extends to terms, literals and clauses:

Definition 2.1 *A term, literal or clause is* transparent *if all its symbols are transparent,* A-coloured *if it contains at least one A-coloured symbol, and the rest are transparent (similarly for* B-coloured*). Otherwise it is AB-mixed.*

A clause is *colourable* if it contains no AB-mixed literals. We use, ambiguously, $\mathcal{L}_A$ for the language of terms, literals or formulæ made of symbols in Σ_A; $\mathcal{L}_B$ and $\mathcal{L}_T$ are defined similarly for Σ_B and Σ_T, respectively. We let $\mathcal{L}_X$ stand for either $\mathcal{L}_A$, $\mathcal{L}_B$ or $\mathcal{L}_T$.

A theory is presented by a set $\mathcal{T}$ of sentences, meaning that the theory is the set of all logical consequences of $\mathcal{T}$. It is customary to call $\mathcal{T}$ itself a theory. Let $\Sigma_\mathcal{T}$ be the signature of $\mathcal{T}$, and $\mathcal{L}_\mathcal{T}$ the language of terms, literals or formulæ built from $\Sigma_\mathcal{T}$. Then, let $\mathcal{L}_T$ be the language of terms, literals or formulæ built from $\Sigma_\mathcal{T} \cup \Sigma_T$: in other words, whenever a theory is involved, theory symbols are considered transparent:

Definition 2.2 (Theory Interpolant) *A formula I is a* theory interpolant *of formulæ A and B such that $A \vdash_{\mathcal{T}} B$, if (i) $A \vdash_{\mathcal{T}} I$, (ii) $I \vdash_{\mathcal{T}} B$ and (iii) I is in $\mathcal{L}_T$. A formula I is a* reverse theory interpolant *of formulæ A and B such that $A, B \vdash_{\mathcal{T}} \bot$, if (i) $A \vdash_{\mathcal{T}} I$, (ii) $B, I \vdash_{\mathcal{T}} \bot$ and (iii) I is in $\mathcal{L}_T$.*

Reverse interpolants are more widely used in the context of theorem proving, since practical theorem provers work refutationally. In keeping with most of the literature, in the following we shall write "interpolant" for "reverse interpolant", unless the distinction is relevant. Furthermore, when it is clear from the context, we may omit the "theory" prefix and just write "interpolant". Similarly, we may use $\vdash$ instead of $\vdash_{\mathcal{T}}$.

Definition 2.3 (Projection) *Let C be a disjunction (conjunction) of literals. The* projection *of C on language $\mathcal{L}_X$, denoted $C|_X$, is the disjunction (conjunction) obtained from C by removing any literal whose atom is not in $\mathcal{L}_X$. By convention, if C is a disjunction and $C|_X$ is empty, then $C|_X = \bot$; if C is a conjunction and $C|_X$ is empty, then $C|_X = \top$.*

Many approaches to interpolation work by annotating each clause C in a refutation of A and B with auxiliary formulæ, called *partial interpolants*:

Definition 2.4 (Partial interpolant) *A* partial interpolant $PI(C)$ *of a clause C occurring in a refutation of $A \cup B$ is an interpolant of $A \wedge \neg(C|_A)$ and $B \wedge \neg(C|_B)$.*

By Definition 2.2 applied to Definition 2.4, a partial interpolant needs to satisfy the following requirements:

Proposition 2.1 *A partial interpolant for a clause C have to satisfy:*

1. $A \wedge \neg(C|_A) \vdash PI(C)$ *or, equivalently,* $A \vdash C|_A \vee PI(C)$
2. $B \wedge \neg(C|_B) \wedge PI(C) \vdash \bot$ *or, equivalently,* $B \wedge PI(C) \vdash C|_B$*, and*
3. $PI(C)$ *is transparent.*

We now give a brief overview of the DPLL($\mathcal{T}$) and DPLL($\Gamma + \mathcal{T}$) theorem proving methods for satisfiability modulo theories. DPLL($\mathcal{T}$) combines propositional reasoning by DPLL with decision procedures for specific theories. DPLL($\Gamma + \mathcal{T}$) is a further extension which also features an interface to a first-order prover with resolution and superposition. We refer to [2] for a description of DPLL($\Gamma + \mathcal{T}$), which includes DPLL($\mathcal{T}$). DPLL($\Gamma + \mathcal{T}$) works with *hypothetical clauses*, where the hypotheses are the connection between Γ-inferences and the partial model M maintained by DPLL($\mathcal{T}$). Hypothetical clauses have the form $H \triangleright C$, where C is a clause and the hypothesis H is a set of ground literals. The literals in H come from M and are the literals that were used as premises to infer C by a Γ-inference. DPLL($\Gamma + \mathcal{T}$) employs model-based theory combination [6], which is a version of equality sharing where only equalities between ground terms are propagated. DPLL($\Gamma + \mathcal{T}$) can be described as a transition system with two kinds of states: $M \parallel F$ (candidate model and set of clauses) and $M \parallel F \parallel C$ (candidate model, set of clauses and conflict clause). Let $S = \mathcal{R} \uplus P$ stand for the set of input clauses, where $\mathcal{R}$ is a set of non-ground clauses, without occurrences of $\mathcal{T}$-symbols, while P is a set of ground clauses that typically do contain $\mathcal{T}$-symbols. A transition system derivation for DPLL($\Gamma + \mathcal{T}$) is defined as follows:

Definition 2.5 (Transition system derivation) *Let $\mathcal{U}$ stand for DPLL($\Gamma + \mathcal{T}$), and S be the input set $\mathcal{R} \uplus P$. A* transition system derivation*, or* $\mathcal{U}$-derivation*, is a sequence of state transitions:*
$\Delta_0 \Longrightarrow_{\mathcal{U}} \Delta_1 \Longrightarrow_{\mathcal{U}} \ldots \Delta_i \Longrightarrow_{\mathcal{U}} \Delta_{i+1} \Longrightarrow_{\mathcal{U}} \ldots$*, where* $\forall i \geq 0$*,* Δ_i *is of the form* $M_i \parallel F_i$ *or* $M_i \parallel F_i \parallel C_i$*, each transition is determined by a transition rule in* $\mathcal{U}$ *and* $\Delta_0 = \parallel F_0$*, where* $F_0 = \{\emptyset \triangleright C \mid C \in S\}$.

A transition system derivation is characterised by the sets $F^* = \bigcup_{i \geq 0} F_i$ of all generated clauses and $C^* = \{C_i | i > 0\}$ of all conflict clauses. A DPLL($\Gamma + \mathcal{T}$) refutation is a refutation by propositional resolution plus $\mathcal{T}$-conflict clauses, which are derived when one of the theory solvers discovers an inconsistency with the current model, and inferences performed by Γ. We denote the proof tree produced by the $\mathcal{T}$-solver for a $\mathcal{T}$-conflict clause C, by $\Pi_{\mathcal{T}}(C)$.

Definition 2.6 (DPLL($\Gamma+\mathcal{T}$)-proof tree) *Given a DPLL($\Gamma+\mathcal{T}$)-derivation,*

$$\Delta_0 \underset{\mathcal{U}}{\Longrightarrow} \Delta_1 \underset{\mathcal{U}}{\Longrightarrow} \dots \Delta_i \underset{\mathcal{U}}{\Longrightarrow} \Delta_{i+1} \underset{\mathcal{U}}{\Longrightarrow} \dots,$$

for all $C \in C^*$ and $H \triangleright C \in F^*$, the DPLL($\Gamma+\mathcal{T}$)-proof tree $\Pi_{\mathcal{U}}(C)$ *of C is defined as follows:*

- *If $C \in F_0$, $\Pi_{\mathcal{U}}(C)$ consists of a node labelled by C;*
- *If C is generated by resolving conflict clause C_1 with justification C_2, $\Pi_{\mathcal{U}}(C)$ consists of a node labelled by C with sub-trees $\Pi_{\mathcal{U}}(C_1)$ and $\Pi_{\mathcal{U}}(C_2)$;*
- *If C is a $\mathcal{T}$-conflict clause, $\Pi_{\mathcal{U}}(C) = \Pi_{\mathcal{T}}(C)$;*
- *If $H \triangleright C$ is inferred by a Γ-based transition from hypothetical clauses $\{H_1 \triangleright C_1, \dots, H_m \triangleright C_m\}$ and literals $\{l_{m+1}, \dots, l_k\}$, $\Pi_{\mathcal{U}}(H \triangleright C)$ consists of a node labelled by $H \triangleright C$ with m sub-trees $\Pi_{\mathcal{U}}(H_1 \triangleright C_1), \dots, \Pi_{\mathcal{U}}(H_m \triangleright C_m)$.*

If the derivation halts reporting unsatisfiable, $\Pi_{\mathcal{U}}(\Box)$ is a DPLL($\Gamma+\mathcal{T}$)-refutation.

Hypotheses need to be discharged when the hypothetical clause $H \triangleright \Box$ is generated. The system then switches to conflict resolution mode, with $\neg H$ as the conflict clause. A refutation is reached only when $\neg H$ is reduced to $\Box$. Thus, a DPLL($\Gamma+\mathcal{T}$)-refutation is obtained by attaching a non-ground proof tree with $H \triangleright \Box$, or $\neg H$, as root, to a ground proof tree with $\neg H$ among its leaves and $\Box$ as root.

An *interpolation system* is a mechanism to annotate each clause C in a refutation of A and B with a partial interpolant. To define an interpolation system, one needs to define the partial interpolants that it associates to the clauses in a proof. Since each clause in a proof is generated by some inference rule, the definition of an interpolation system needs to cover all possibilities. The fundamental property of an interpolation system is *completeness*:

Definition 2.7 (Complete interpolation system) *An interpolation system is* complete *for inference system Γ, or transition system $\mathcal{U}$, if for all sets of clauses A and B, such that $A \cup B$ is unsatisfiable, and for all refutations of $A \cup B$ by Γ, or $\mathcal{U}$, respectively, it generates an interpolant of (A, B).*

The key property of partial interpolants is that $PI(\Box)$ is an interpolant of A and B. Thus, in order to prove that an interpolation system is complete, it is sufficient to show that it annotates the clauses in any refutation with clauses that are indeed partial interpolants.

3 An Interpolation System for DPLL($\Gamma + \mathcal{T}$)

A complete interpolation system for DPLL($\Gamma + \mathcal{T}$) must be able to compute partial interpolants for each clause in the proof tree in Def. 2.6, that is: propositional resolvents, $\mathcal{T}$-conflict clauses, and clauses derived by Γ. The latter are covered by the new interpolation system given in this section. Propositional resolvents are dealt with by a propositional interpolation system such as HKPYM [10, 20, 14, 21]. Since $\mathcal{T}$ is a combination, $\mathcal{T}$-conflict clauses are handled by EQSH [21], which requires that the component theories are *equality interpolating*:

Definition 3.1 (Equality Interpolating Theory) *A theory $\mathcal{T}$ is* equality interpolating *if for all $\mathcal{T}$-formulæ A and B, whenever $A \wedge B \models_{\mathcal{T}} t_a \simeq t_b$, where t_a is an A-coloured ground term and t_b is a B-coloured ground term, then $A \wedge B \models_{\mathcal{T}} t_a \simeq t \wedge t_b \simeq t$ for some transparent ground term t.*

Several theories used in practice are indeed equality interpolating, for example quantifier-free theories of uninterpreted functions and linear arithmetic [21]. Without this requirement, the notion of transparency is not stable: if $t_a \simeq t_b$ without any transparent t such that $t_a \simeq t$ and $t_b \simeq t$, the congruence class of t_a and t_b includes no transparent term, which means a coloured term should "become" transparent, to serve as a representative for terms of both colours. This is clearly undesirable as transparent terms are those used to build interpolants. A similar issue arises for Γ, which also reasons about equalities. If

an AB-mixed equality $t_a \simeq t_b$ is derived, it can be used to simplify clauses in A, introducing B-coloured symbols, which now should be considered transparent as they have become shared between A and B. Proofs without AB-mixed equalities were termed *colourable* in [8]:

Definition 3.2 (Colourable proof) *A proof is* colourable *if it contains no AB-mixed literals.*

We proceed to connect the notion of equality-interpolating theory with the following requirement, that appeared in [18], under the name *AB-oriented ordering*, and then in [12]:

Definition 3.3 (Separating ordering) *An ordering $\succ$ is* separating *if $t \succ s$ whenever s is transparent and t is not, for all ground terms, or literals, s and t.*

If the theory is equality-interpolating, whenever $t_a \simeq t_b$ holds, $t_a \simeq t$ and $t_b \simeq t$ also hold, and a separating ordering ensures that, if $t_a \simeq t$ and $t_b \simeq t$ are derived, t replaces t_a and t_b, or becomes the representative of the congruence class of t_a and t_b.

Lemma 3.1 *If the ordering is separating, all ground Γ-proof-trees are colourable.*

The proof is by induction on the structure of the proof tree (see [1]). To get a superposition based theorem prover to produce ground colourable proofs, it is thus sufficient to adopt a separating ordering. Separating orderings exist, and were implemented, for instance, in Vampire [9]. From now on, we assume that *the built-in theories $\mathcal{T}_i$, $1 \leq i \leq n$, are equality-interpolating*, and that *the ordering $\succ$ for Γ-inferences is separating*, so that all ground proofs are colourable. Under these assumptions, we present a complete interpolation system for Γ in the ground case, where the inferences rules, with premises and consequences labelled for later reference, are as follows (see [2] for full details):

Resolution:

$$\frac{p_1 \colon (C \vee l) \quad p_2 \colon (D \vee \neg l)}{c \colon (C \vee D)} \qquad \forall m \in C \colon l \succ m \quad \forall m \in D \colon \neg l \succ m$$

Paramodulation:

$$\frac{p_1 \colon (C \vee s \simeq r) \quad p_2 \colon (D \vee l[s])}{c \colon (C \vee l[r] \vee D)} \qquad (i) \quad (ii) \quad (iii)$$

Superposition:

$$\frac{p_1 \colon (C \vee s \simeq r) \quad p_2 \colon (D \vee l[s] \bowtie t)}{c \colon (C \vee l[r] \bowtie t \vee D)} \qquad (i) \quad (ii) \quad (iv) \quad (v)$$

where (i) $s \succ r$, (ii) $\forall m \in C \colon (s \simeq r) \succ m$, (iii) $\forall m \in D \colon l[s] \succ m$, (iv) $l[s] \succ t$, (v) $\forall m \in D \colon (l[s] \bowtie t) \succ m$; and Simplification inferences are instances of Paramodulation/Superposition, where c replaces p_2, C is empty, and (i) is the only side condition.

Definition 3.4 (GΓI interpolation system) *Let $c \colon C$ be a clause that appears in a ground Γ-refutation of $A \cup B$:*

- *If $c \colon C \in A$, then $PI(c) = \bot$, if $c \colon C \in B$, then $PI(c) = \top$.*
- *If $c \colon C$ is generated from premises p_1 and p_2 by a Γ-inference, $PI(c)$ is defined as follows:*
  - *Resolution: $c \colon C \vee D$ generated from $p_1 \colon C \vee l$ and $p_2 \colon D \vee \neg l$*
    - *l is A-coloured: $PI(c) = PI(p_1) \vee PI(p_2)$*
    - *l is B-coloured: $PI(c) = PI(p_1) \wedge PI(p_2)$*
    - *l is transparent: $PI(c) = (l \vee PI(p_1)) \wedge (\neg l \vee PI(p_2))$*
  - *Paramodulation/Superposition/Simplification: $c \colon C \vee l[r] \vee D$ generated from $p_1 \colon C \vee s \simeq r$ and $p_2 \colon D \vee l[s]$*
    - *$s \simeq r$ is A-coloured: $PI(c) = PI(p_1) \vee PI(p_2)$*
    - *$s \simeq r$ is B-coloured: $PI(c) = PI(p_1) \wedge PI(p_2)$*

$*$ $s \simeq r$, $l[s]$ *are transparent:* $PI(c) = (s \simeq r \vee PI(p_1)) \wedge (l[s] \vee PI(p_2))$
$*$ $s \simeq r$ *is transparent,* $l[s]$ *is not:* $PI(c) = (s \simeq r \vee PI(p_1)) \wedge (s \not\simeq r \vee PI(p_2))$.

Superposition is treated like Paramodulation, with $l[s]$ replaced by $l[s] \bowtie t$, and the case for Simplification is subsumed by those for Paramodulation and Superposition. As we assume a separating ordering, transparent terms are smaller than coloured ones and we do not need to consider the case where $s \simeq r$ is coloured and $l[s]$ is transparent for paramodulation inferences. In such a case, s must be transparent, as it also occurs in the transparent literal $l[s]$, and r must be coloured. The separating ordering would thus re-orient such an equation to $r \simeq s$, and only inferences rewriting a coloured term are allowed.

Theorem 1 *If the ordering is separating, GΓI is a complete interpolation system for all ground Γ-refutations.*

Proof: By induction on the structure of the proof. We need to prove that for all clauses $c\colon C$ in the refutation, the partial interpolants satisfy the requirements in Proposition 2.1.
Base cases: $c : C$ *is an input clause*. Trivial.
Inductive cases:
Inductive hypothesis: for $k \in \{1, 2\}$ it holds that:

1. $A \wedge \neg(p_k|_A) \vdash PI(p_k)$ or, equivalently, $A \vdash p_k|_A \vee PI(p_k)$
2. $B \wedge \neg(p_k|_B) \wedge PI(p_k) \vdash \bot$ or, equivalently, $B \wedge PI(p_k) \vdash p_k|_B$
3. $PI(p_k)$ is transparent.

Resolution: $c\colon C \vee D$ generated from $p_1\colon C \vee l$ and $p_2\colon D \vee \neg l$

- l is A-coloured: $l|_A = l$, $(\neg l)|_A = \neg l$, $l|_B = \bot = (\neg l)|_B$
  1. $A \vdash (C \vee D)|_A \vee PI(p_1) \vee PI(p_2)$. From inductive hypothesis (1) we have $A \vdash (C \vee l)|_A \vee PI(p_1)$ and $A \vdash (D \vee \neg l)|_A \vee PI(p_2)$. A resolution step gives $A \vdash (C \vee D)|_A \vee PI(p_1) \vee PI(p_2)$ as desired.
  2. $B \wedge (PI(p_1) \vee PI(p_2)) \vdash (C \vee D)|_B$. From inductive hypothesis (2) we have $B \wedge PI(p_1) \vdash C|_B$ and $B \wedge PI(p_2) \vdash D|_B$ from which the inductive conclusion follows.
  3. Transparency of the partial interpolant follows from the inductive hypothesis.
- l is B-coloured: Symmetric to the previous case.
- l is transparent: $l|_A = l = l|_B$, $(\neg l)|_A = \neg l = (\neg l)|_B$
  1. $A \wedge \neg(C \vee D)|_A \vdash (l \vee PI(p_1)) \wedge (\neg l \vee PI(p_2))$ or, equivalently, $A \wedge \neg C|_A \wedge \neg D|_A \vdash (l \vee PI(p_1)) \wedge (\neg l \vee PI(p_2))$. From inductive hypothesis (1) we have $A \wedge \neg C|_A \vdash l \vee PI(p_1)$ and $A \wedge \neg D|_A \vdash \neg l \vee PI(p_2)$, which together give the desired result.
  2. $B \wedge (l \vee PI(p_1)) \wedge (\neg l \vee PI(p_2)) \vdash (C \vee D)|_B$. By case analysis on l in $PI(c)$: if l is true, l holds, l subsumes $l \vee PI(p_1)$ and simplifies $\neg l \vee PI(p_2)$ to $PI(p_2)$; if l is false, $\neg l$ holds, $\neg l$ subsumes $\neg l \vee PI(p_2)$ and simplifies $l \vee PI(p_1)$ to $PI(p_1)$; so that we need to establish:
     (a) $B \wedge l \wedge PI(p_2) \vdash (C \vee D)|_B$. From inductive hypothesis (2) we have $B \wedge PI(p_2) \vdash D|_B \vee \neg l$ whence $B \wedge l \wedge PI(p_2) \vdash D|_B$.
     (b) $B \wedge \neg l \wedge PI(p_1) \vdash (C \vee D)|_B$. From inductive hypothesis (2) we have $B \wedge PI(p_1) \vdash C|_B \vee l$ whence $B \wedge \neg l \wedge PI(p_1) \vdash C|_B$.
  3. Transparency of the partial interpolant follows from the inductive hypothesis and the assumption that l is transparent.

Paramodulation/Superposition/Simplification: $c\colon C \vee l[r] \vee D$ generated from $p_1\colon C \vee s \simeq r$ and $p_2\colon D \vee l[s]$

- $s \simeq r$ is A-coloured: either s and r are both A-coloured, or, since $s \succ r$, s is A-coloured and r is transparent; since there are no AB-mixed literals, either $l[s]$ and $l[r]$ are both A-coloured, or $l[s]$ is A-coloured and $l[r]$ is transparent; thus, we have: $(s \simeq r)|_A = (s \simeq r)$, $l[s]|_A = l[s]$, $l[r]|_A = l[r]$, $(s \simeq r)|_B = \bot$, $l[s]|_B = \bot$

1. $A \vdash (C \vee l[r] \vee D)|_A \vee PI(p_1) \vee PI(p_2)$. Inductive hypothesis (1) gives $A \vdash C|_A \vee s \simeq r \vee PI(p_1)$ and $A \vdash D|_A \vee l[s] \vee PI(p_2)$; Thus, the inductive conclusion follows by a paramodulation step.
2. $B \wedge (PI(p_1) \vee PI(p_2)) \vdash (C \vee l[r] \vee D)|_B$. From inductive hypothesis (2) we have $B \wedge PI(p_1) \vdash C|_B$ and $B \wedge PI(p_2) \vdash D|_B$, which proves the inductive conclusion.
3. The partial interpolant is transparent by inductive hypothesis.

- $s \simeq r$ is B-coloured: Symmetric to the previous case.
- $s \simeq r$ and $l[s]$ are transparent: $l[r]$ is also transparent, and all three literals are unaffected by projections.
  1. $A \wedge \neg(C|_A) \wedge \neg l[r] \wedge \neg(D|_A) \vdash (s \simeq r \vee PI(p_1)) \wedge (l[s] \vee PI(p_2))$. From inductive hypothesis (1) we have $A \wedge \neg C|_A \vdash s \simeq r \vee PI(p_1)$ and $A \wedge \neg D|_A \vdash l[s] \vee PI(p_2)$, which together give the desired result.
  2. $B \wedge (s \simeq r \vee PI(p_1)) \wedge (l[s] \vee PI(p_2))) \vdash (C \vee l[r] \vee D)|_B$. We do a case analysis on $s \simeq r$ and $l[s]$:
     (a) If $s \simeq r$ and $l[s]$ are both true, then $l[r]$ is true.
     (b) If $s \simeq r$ is true and $l[s]$ is false, then $PI(p_2)$ must be true and $D \vee l[s]$ is equivalent to D, so that induction hypothesis (2) gives $B \wedge PI(p_2) \vdash D|_B$.
     (c) If $s \simeq r$ is false and $l[s]$ is true, then $PI(p_1)$ must be true and $C \vee s \simeq r$ is equivalent to C, so that induction hypothesis (2) gives $B \wedge PI(p_1) \vdash C|_B$.
     (d) If $s \simeq r$ and $l[s]$ are both false, then $PI(p_1)$ and $PI(p_2)$ must be true and induction hypothesis (2) gives $B \wedge PI(p_1) \vdash C|_B$ and $B \wedge PI(p_2) \vdash D|_B$.
  3. Transparency of the partial interpolant follows from the inductive hypothesis and the assumption that $s \simeq r$ and $l[s]$ are transparent.
- $s \simeq r$ is transparent, $l[s]$ is not:
  1. $A \vdash (C \vee l[r] \vee D)|_A \vee (s \simeq r \vee PI(p_1)) \wedge (s \not\simeq r \vee PI(p_2))$. This is equivalent to: $A \wedge ((s \not\simeq r \wedge \neg PI(p_1)) \vee (s \simeq r \wedge \neg PI(p_2))) \vdash (C \vee l[r] \vee D)|_A$. We perform a case analysis on $s \simeq r$:
     (a) If $s \simeq r$ is false, $s \simeq r \wedge \neg PI(p_2)$ is false, and it suffices to establish $A \wedge s \not\simeq r \wedge \neg PI(p_1) \vdash (C \vee l[r] \vee D)|_A$. By induction hypothesis (1) we have $A \wedge s \not\simeq r \wedge \neg PI(p_1) \vdash C|_A$, which suffices.
     (b) If $s \simeq r$ is true, $s \not\simeq r \wedge \neg PI(p_1)$ is false, and it suffices to establish $A \wedge s \simeq r \wedge \neg PI(p_2) \vdash (C \vee l[r] \vee D)|_A$ or, equivalently, $A \wedge s \simeq r \wedge \neg PI(p_2) \vdash (C \vee l[s] \vee D)|_A$ since $s \simeq r$ holds. By induction hypothesis (1) we have $A \wedge \neg PI(p_2) \vdash (l[s] \vee D)|_A$, and we are done.
  2. $B \wedge (s \simeq r \vee PI(p_1)) \wedge (s \not\simeq r \vee PI(p_2))) \vdash (C \vee l[r] \vee D)|_B$. By case analysis on $s \simeq r$:
     (a) If $s \simeq r$ is true, $s \simeq r \vee PI(p_1)$ is subsumed, $s \not\simeq r$ is false and $PI(p_2)$ must be true. Thus, it suffices to establish $B \wedge s \simeq r \wedge PI(p_2) \vdash (C \vee l[r] \vee D)|_B$, which is equivalent to $B \wedge s \simeq r \wedge PI(p_2) \vdash (C \vee l[s] \vee D)|_B$, since $s \simeq r$ holds. By induction hypothesis (2) we have $B \wedge PI(p_2) \vdash l[s]|_B \vee D|_B$, which closes this case.
     (b) If $s \not\simeq r$ is true, $s \not\simeq r \vee PI(p_2)$ is subsumed, $s \simeq r$ is false and $PI(p_1)$ must be true. Thus, we need to establish $B \wedge s \not\simeq r \wedge PI(p_1) \vdash (C \vee l[r] \vee D)|_B$. By induction hypothesis (2) we have $B \wedge s \not\simeq r \wedge PI(p_1) \vdash C|_B$, which suffices.
  3. Transparency follows from the transparency of $s \simeq r$ and the inductive hypothesis.

□

Having obtained a complete interpolation system for Γ, we can now define an interpolation system for DPLL($\Gamma+\mathcal{T}$):

Definition 3.5 (I^* interpolation system) *Let $c\colon C$ be a clause that appears in a DPLL($\Gamma+\mathcal{T}$)-refutation of $A \cup B$:*

- *If* $c\colon C \in A$*, then* $PI(c) = \bot$*, if* $c\colon C \in B$*, then* $PI(c) = \top$.
- *If* $c\colon C$ *is a* $\mathcal{T}$*-conflict clause,* $PI(c)$ *is the* $\mathcal{T}$*-interpolant of* $((\neg C)|_A, (\neg C)|_B)$ *produced by EQSH from the refutation* $\neg C \vdash_{\mathcal{T}} \bot$*;*
- *If* $c\colon C \vee D$ *is a propositional resolvent of* $p_1\colon C \vee l$ *and* $p_2\colon D \vee \neg l$ *then:*
  - *If* l *is A-coloured, then* $PI(c) = PI(p_1) \vee PI(p_2)$*,*
  - *If* l *is B-coloured, then* $PI(c) = PI(p_1) \wedge PI(p_2)$ *and*
  - *If* l *is transparent, then* $PI(c) = (l \vee PI(p_1)) \wedge (\neg l \vee PI(p_2))$.
- *If* $c\colon C$ *is a hypothetical clause* $H \triangleright C$ *inferred by a generative* Γ*-based transition from premises* $\{H_1 \triangleright C_1, \ldots, H_m \triangleright C_m\}$ *and* $\{l_{m+1}, \ldots, l_k\}$*, then* $PI(c)$ *is the partial interpolant produced by the interpolation system* $G\Gamma I$ *for the* Γ*-inference inferring* C *from premises* $C_1, \ldots, C_m, l_{m+1}, \ldots, l_k$.

The partial interpolant for a hypothetical clause $H \triangleright C$ is given by the partial interpolant for the corresponding regular clause C, because the Γ-inference embedded in a Γ-based transition ignores hypotheses, and, when $H \triangleright \square$ is generated, the hypotheses in H are discharged by propositional resolution steps, whose partial interpolant is computed as in HKPYM. In summary, the modular construction of DPLL($\Gamma + \mathcal{T}$) allows us to define its interpolation system from the interpolation systems of its components. Furthermore, this allows us to simply replace $G\Gamma I$ by a general interpolation system for Γ once available. The requirement that all theories in $\mathcal{T}$ are equality-interpolating guarantees that the $\mathcal{T}$-conflict clauses do not introduce in the proof AB-mixed literals, and the completeness of I^* thus follows from the completeness of its component interpolation systems.

4 Related Work

Interpolation for *coloured* superposition proofs was first considered by McMillan [18], and further studied, with some criticism that restricted it to ground proofs, by Kovàcs and Voronkov [13]. *Coloured* is a stronger requirement than *colourable*: each inference may involve at most one colour, so that not only AB-mixed literals, but also AB-mixed clauses are forbidden. The main similarity between our work and these is the adoption of a separating ordering, where transparent literals are smaller than coloured ones. However, our approach differs is several ways: Firstly, in the ground case, we relax the requirement of coloured proofs, and only require the notion of colourable proofs from [8]. We showed that when a separating ordering is used, every ground Γ-refutation is colourable. Secondly, the target inference system in [13] is LASCA (Linear Arithmetic Superposition CAlculus), which is superposition with linear arithmetic built in. We do not consider arithmetic within Γ, because in DPLL($\Gamma + \mathcal{T}$) arithmetic is handled by the DPLL($\mathcal{T}$) part, and therefore by an interpolating decision procedure for arithmetic (e.g. [17]). Thirdly, our notion of partial interpolant is different from [13], which focused on proving existence of partial interpolants[1] only for transparent ground clauses in coloured proof-trees. No explicit interpolation system is given in either [13] or [18]. We define explicitly the partial interpolants for every generative rule in Γ. Thus, we consider the interpolation system I^* to be more concrete and representing a more direct generalisation of propositional interpolation systems to ground first-order logic.

Christ and Hoenicke considers interpolation in the presence of quantifiers in the context of DPLL($\mathcal{T}$) [4]. They assume instantiations are found by heuristic methods, such as triggering, rather than by unification as in superposition. The interpolation method is based on McMillan's ground interpolation system for resolution [17], extended to introduce quantifiers in interpolants, when instantiations introduce coloured terms. This approach thus goes beyond colourable proofs for resolution. Equality reasoning is assumed to be handled by an interpolating decision procedure.

Our interpolation system for ground superposition also covers proofs in EUF (Equality with Uninterpreted Functions). McMillan's interpolation system for EUF in [17] consists of inference rules for reflexivity, symmetry, congruence, transitivity and contradiction, instrumented to compute formulas

[1] Referred to as *C-interpolants* in [13].

akin to partial interpolants for each inference step. There are several versions of each rule, depending on side conditions relating to the colour of the inferences leading up to the conclusion. The interpolation system by Fuchs et al. [7], on the other hand, works on colourable congruence graphs, generated by the congruence closure algorithm. While we rely on the separating ordering to ensure that no AB-mixed literals are present in the proof, Fuchs et al. use the fact that EUF is equality interpolating and perform some modifications on the congruence graph, introducing transparent constants to separate A-coloured and B-coloured terms as needed, essentially implementing the requirement that the theory be equality-interpolating. While our interpolation system will include all transparent literals derived from A, the specialised congruence closure method can summarise chains originating only from A, and only consider adding the last transparent term in such a chain. This means that it tend to produce shorter interpolants, which for some applications may be desirable. At this stage of research, we have focused on completeness of the interpolation system, with analysis of the properties of interpolants for various applications left as further work. Last, the algorithm in [7] is restricted to EUF only, while our aim is a much more general interpolation system.

5 Current and Future Work

We reported on ongoing work on interpolation for the theorem proving method DPLL(Γ+$\mathcal{T}$). We showed how an interpolation system for DPLL($\Gamma+\mathcal{T}$) can be constructed modularly from interpolation systems for DPLL, equality sharing and for Γ, a first-order resolution and superposition based prover. We presented and proved correct a novel interpolation system for Γ in the ground case for colourable proofs. Current work in progress aims at extending the interpolation system to general proofs with substitution under suitable restrictions. The interpolation system for general Γ-proofs can then simply be plugged into the interpolation system for DPLL($\Gamma+\mathcal{T}$), to extend it beyond ground proofs. In order to generalise the ground interpolation system for Γ to some class of proofs in full first order logic, we need to extend our approach to handle variables and substitutions. In the ground case, we can ensure colours are stable by imposing a separating ordering, which prevents equations between A-coloured and B-coloured terms from being generated. In the general case, a separating ordering is no longer sufficient to ensure that proofs are colourable, as we may unify two literals of different colour, which may have the side effect of generating AB-mixed literals by substitution. One way of avoiding AB-mixed literals is to impose the restriction that the proof is coloured. For coloured proofs, we have that substitution and projection commute, allowing a straightforward extension from the ground interpolation system of the cases where both pivots, or literals paramodulated from and into, have the same colour. However, non-ground coloured proofs also have to deal with the cases where one of the premises is transparent and the other coloured. Thus, excluding AB-mixed literals is not enough to ensure colours are stable, as substitutions may also paint transparent literals as a side effect. Our current work is concerned with extending the interpolation system to these inferences, in which the partial interpolants may contain quantifiers. One of the approaches we are studying is *procrastination*, suggested by McMillan [18], which involves the addition of a special inference rule that delays superposition steps and record restrictions on variable instantiations. We are also considering *instance purification* [4], where coloured literals occurring in partial interpolants are replaced by quantified variables.

References

[1] M.P. Bonacina and M. Johansson. On theorem proving with interpolation for program checking. Technical report, Università degli Studi di Verona, April 2011.

[2] M.P. Bonacina, C.A. Lynch, and L. de Moura. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning*, In press:1–29. Published online 22 December 2010 (doi: 10.1007/s10817-010-9213-y).

[3] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *Proceedings of the International Joint Conference on Automated Reasoning*, volume 6173 of *LNAI*, pages 384–399. Springer, 2010.

[4] J. Christ and J. Hoenicke. Instantiation-based interpolation for quantified formulae. Satisfiability Modulo Theories Workshop, 2010.

[5] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo a theory. In *Proceedings of the Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 397–412. Springer, 2008.

[6] L. de Moura and N. Bjørner. Model-based theory combination. In *Proceedings of the Workshop on Satisfiability Modulo Theories 2007*, volume 198(2) of *ENTCS*, pages 37–49. Elsevier, 2008.

[7] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground interpolation for the theory of equality. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427. Springer, 2009.

[8] A. Goel, S. Krstić, and C. Tinelli. Ground interpolation for combined theories. In *Proceedings of the Conference on Automated Deduction*, volume 5663 of *LNAI*, pages 183–198. Springer, 2009.

[9] K. Hoder, L. Kovàcs, and A. Voronkov. Interpolation and symbol elimination in Vampire. In *Proceedings of the International Joint Conference on Automated Reasoning*, volume 6173 of *LNAI*, pages 188–195, 2010.

[10] G. Huang. Constructing Craig interpolation formulas. In *Proceedings of the First Annual International Conference on Computing and Combinatorics*, pages 181–190. Springer, 1995.

[11] D. Kapur, R. Majumdar, and C.G. Zarba. Interpolation for data structures. In Premkumar Devambu, editor, *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 2006.

[12] L. Kovàcs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[13] L. Kovàcs and A. Voronkov. Interpolation and symbol elimination. In *Proceedings of the Conference on Automated Deduction*, volume 5663 of *LNAI*, pages 199–213. Springer, 2009.

[14] J. Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, 62(2):457–486, 1997.

[15] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper–V hypervisor with VCC. In *Proceedings of the Second World Congress on Formal Methods*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.

[16] K.L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the Conference on Computer Aided Verification*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[17] K.L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[18] K.L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of the Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[19] K.L. McMillan. Lazy annotation for program testing and verification. In *Proceedings of the Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

[20] P. Pudlàk. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[21] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *Proceedings of the Conference on Automated Deduction*, volume 3632 of *LNAI*, pages 353–368, 2005. Early version in MSR-TR-2004-108, October 2004.

Quantified Interpolation for SMT

Amit Goel[1] Sava Krstić[1] Rupak Majumdar[2,3] Sai Deep Tetali[3]
[1]Strategic CAD Labs, Intel [2]MPI-SWS [3]UC Los Angeles

Abstract

Interpolation algorithms form the basis of several successful verification systems. We describe a generic interpolation algorithm for (quantified) first-order logic with background theories, pointing out implementation choices in (1) partitioning literals, (2) introducing quantifiers, and (3) partitioning theory lemmas. The choices lead to different quantified interpolants and to different known algorithms in the literature. We show how to incorporate ground-interpolating decision procedures directly in the interpolation algorithm, enabling direct integration with SMT solvers. We provide some initial evaluation of interpolation-based parameterized verification of protocols and conclude that more research is needed in tuning the quality of interpolants.

1 Introduction

Given two formulas A and B whose conjunction is unsatisfiable, an *interpolant* for the pair (A, B) is a formula I in the common language of A and B such that A implies I and B implies $\neg I$ [3] (see [8] for a textbook account). Interpolants for propositional or first-order formulas form the basis for several successful software verification systems [9, 16, 17]. While interpolants exist for full first-order logic [3] together with recursively enumerable background theories [11], their use in verification tools, until recently, has been restricted to either propositional logic, or to quantifier-free formulas in special theory combinations (EUF and rational linear arithmetic or difference logic). On the other hand, many verification problems for software require the use of quantification (e.g., to say every object in a collection has some property), or the use of theories for which quantifier-free interpolants do not exist, even if the two formulas are quantifier-free (e.g., the theory of arrays).

This limitation of current interpolation-based software verification tools is understood, and there have been several recent attempts to compute and apply *quantified* interpolants for first-order logical formulas with background theories in interpolation-based model checking. One way to generate quantified interpolants in the presence of background theories is to extend a first-order theorem prover (such as Spass [21]) with an explicitly coded axiomatization of the background theory, to generate a proof of unsatisfiability with pure first-order reasoning, and to construct an interpolant out of this proof, using standard techniques [3, 8, 10]. This approach, described and implemented by McMillan [18], has been used to verify small array and list processing programs. However, it fails to exploit the power of modern SMT solvers to generate ground interpolants for many theories.

In this paper, we give a description of quantified interpolation algorithms on top of SMT solvers, parameterizing over *all* different choices that arise in the algorithms: the construction of propositional interpolants [4, 10, 12, 17, 20], the position and introduction of quantifiers [1, 10], and the integration with ground decision procedures [2]. As special cases, we get known interpolation algorithms [1, 10]. While many of these ideas have been published before, our contribution is the way we organize the ideas into a single coherent account.

2 Preliminary Definitions

Syntax of First-Order Logic. A *signature* $\Sigma = (S, F, P)$ consists of a set S of *sorts*, a set F of *function symbols*, and a set P of *predicate symbols*, where the arities of the symbols in F and P are constructed using the sorts in S (we consider the arity of a function or a predicate to be a built-in part of the function or predicate symbol). A *constant* is a function of arity zero. For a signature Σ, we write Σ^S (respectively, Σ^F, Σ^P) for S (respectively F, P). For signatures Σ_1 and Σ_2, we write $\Sigma_1 \subseteq \Sigma_2$ if $\Sigma_1^S \subseteq \Sigma_2^S$, $\Sigma_1^F \subseteq \Sigma_2^F$, and $\Sigma_1^P \subseteq \Sigma_2^P$. The union and intersection of signatures is defined as the pointwise union and intersection of their component sets. For each sort σ, we fix a set $\mathcal{X}_\sigma$ of *free variable symbols of sort* σ which are disjoint from the function and predicate symbols $\Sigma^F \cup \Sigma^P$. We also fix a set $\mathcal{X}_{\mathsf{bool}}$ of free propositional symbols.

In what follows, we use meta-variables a, b, c to represent constants, f, g, h to represent (non-constant) function symbols, and p, q to represent predicate symbols.

For a signature Σ, the set of Σ-terms is the smallest set such that (1) each free variable symbol $u \in \mathcal{X}_\sigma$ is a Σ-term of sort σ for all $\sigma \in \Sigma^S$, (2) each constant symbol $u \in \Sigma^F$ of sort σ is a Σ-term of sort σ, and (3) $f(t_1, \ldots, t_n)$ is a Σ-term of sort σ, given $f \in \Sigma^F$ is a function symbol of arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ and t_i is a Σ-term of sort σ_i for $i = 1, \ldots, n$.

The set of Σ-atoms is the smallest set such that (1) each propositional symbol $u \in \mathcal{X}_{\mathsf{bool}}$ is a Σ-atom, (2) $p(t_1, \ldots, t_n)$ is a Σ-atom given that $p \in \Sigma^P$ is a predicate symbol of arity $\sigma_1 \times \ldots \times \sigma_n$ and t_i is a Σ-term of sort σ_i for $i = 1, \ldots, n$.

The set of quantifier-free Σ-formulas is the smallest set such that (1) each Σ-atom is a Σ-formula, (2) if φ, ψ are Σ-formulas, so are $\neg\varphi$, $\varphi \wedge \psi$. The set of Σ-formulas is the smallest set such that (1) every quantifier-free Σ-formula is a Σ-formula, and (2) if φ is a Σ-formula and $x \in \mathcal{X}_\sigma$ a free variable, then $\forall x : \mathcal{X}_\sigma.\varphi$ and $\exists x : \mathcal{X}_\sigma.\varphi$ are Σ-formulas. We shall use the usual derived formulas $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$. A *sentence* is a formula with no free variables. We will use the notation $\phi[t \mapsto s]$ for the formula obtained from ϕ by replacing every occurrence of a subterm t in it simultaneously with a term s. We omit the prefix Σ- when it is clear from the context. We write $vars(\varphi)$ for the free variable symbols in φ.

Semantics. For a signature $\Sigma = (S, F, P)$ and a set X of free variable symbols over sorts in S, a Σ-structure $\mathcal{A}$ over X is a map which interprets (a) each sort $\sigma \in S$ as a non-empty domain A_σ, (b) each free variable constant symbol $u \in X_\sigma$ as an element $u^{\mathcal{A}} \in A_\sigma$, (c) each free propositional symbol $u \in \mathcal{X}_{\mathsf{bool}}$ as a truth value in $\{true, false\}$, (d) each function symbol $f \in F$ of arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ as a function $f^{\mathcal{A}} : A_{\sigma_1} \times \ldots \times A_{\sigma_n} \to A_\sigma$, and (e) each predicate symbol $p \in P$ of arity $\sigma_1 \times \ldots \times \sigma_n$ as a relation $p^{\mathcal{A}} \subseteq A_{\sigma_1} \times \ldots \times A_{\sigma_n}$. For a Σ-formula φ with free variables $X_0 \subseteq X$, we denote by $\varphi^{\mathcal{A}}$, the evaluation of φ under $\mathcal{A}$ (defined in the usual way). For a formula φ, we write $\mathcal{A} \models \varphi$ if $\varphi^{\mathcal{A}} = true$. A formula φ is *satisfiable* if $\mathcal{A} \models \varphi$ for some structure $\mathcal{A}$ over $vars(\varphi)$. For formulas φ, φ', we write $\varphi \models \varphi'$ if for every structure $\mathcal{A}$ such that $\mathcal{A} \models \varphi$ we have $\mathcal{A} \models \varphi'$.

A *literal* is either an atomic formula p or the negation $\neg p$ of an atomic formula. A *clause* is a prenex formula of the form $\forall x_1 \ldots \forall x_m C$ where C is a disjunction of distinct literals. We assume for convenience that the sets of bound variables are disjoint for any two clauses. A *ground clause* is a clause which has no variables (and no quantifiers). The *empty clause* $\Box$ is just the constant $false$. For simplicity, we omit the universal quantifiers and assume that each variable is implicitly universally quantified. Also, we do not explicitly deal with existential quantifiers in the formulas, assuming that we have introduced Skolem functions for each existentially quantified variable.

Interpolants. Let Σ_A and Σ_B be two signatures, and let A and B be two sets of clauses over Σ_A and Σ_B respectively. Suppose that $A \cup B$ is unsatisfiable. By Craig's theorem [3], there is a sentence I over the signature $\Sigma_A \cap \Sigma_B$ such that $A \models I$ and $B \models \neg I$. We call such an I an *interpolant* for (A, B).

Resolution Proof System. Let A be a set of ground clauses. A *resolution proof tree* for A is a rooted, node-labeled, binary tree where each node n is labeled with a clause $\theta(n)$ so that every leaf is labeled by a clause from A, and every internal node is an instance of the *resolution proof rule*:

$$\frac{L \vee \phi \qquad \bar{L} \vee \phi'}{\phi \vee \phi'} \text{ (RESOLUTION)}$$

That is, for every internal node n with children n_1 and n_2, we have that $\theta(n_1) \equiv (L \vee \phi)$, $\theta(n_2) \equiv (\bar{L} \vee \phi')$, and $\theta(n) \equiv \phi \vee \phi'$, for some literal L and clauses ϕ and ϕ'. A *refutation* of A is a resolution proof tree where the root is labeled with $\Box$.

Suppose now that A is a set of clauses, not necessarily ground. To prove that A is unsatisfiable, we can use the *ground resolution* technique: generate a finite number of ground instances of each clause in A (by substituting ground terms for variables) and construct a refutation tree with these ground clauses at the leaves. The technique is sound and complete: A is unsatisfiable if and only if there exists such a ground refutation. See, e.g., [8].

3 Quantified Interpolants for First-Order Logic

We first give an algorithm to derive an interpolant from a resolution proof of unsatisfiability for first-order logic without equality. In §4, we generalize the technique to compute interpolants for first-order logic in the presence of background theories.

Let A and B be sets of clauses such that $A \cup B$ is unsatisfiable and suppose Ω is a refutation tree for $A^* \cup B^*$, where A^* and B^* are finite sets of ground instances of A and B respectively. As above, for every node n, the clause that labels it is denoted $\theta(n)$.

Example 1 *Let $\Sigma_A = \{f, g, \sim\}$, $\Sigma_B = \{a, b, f, \sim\}$, and let A and B consist of the following clauses, where x, y are (implicitly universally quantified) variables.*[1]

$$\begin{array}{ll} A: & x \not\sim y \Rightarrow g(f(x)) \not\sim g(f(y)) \qquad x \sim y \Rightarrow g(x) \sim g(y) \\ B: & a \not\sim b \qquad f(a) \sim f(b) \end{array}$$

Figure 1 shows a refutation of $A \cup B$ where the clauses superscripted with the symbol † are obtained by instantiating the clauses of A via substitutions $[x \mapsto a, y \mapsto b]$ and $[x \mapsto f(a), y \mapsto f(b)]$ respectively. These two clauses constitute the set A^*, while the set B^* is just B.

Working inductively from leaves of Ω towards the root, we will associate with every node n its *partial interpolant* $I(n)$, which is a first-order formula, not necessarily ground and not necessarily a clause. The partial interpolant for the root of Ω will be the desired interpolant for the pair A, B.

We say that a literal is A^*-*local* if it or its negation occurs in an A^*-clause, but neither occurs in any of the B^*-clauses. B^*-*local* literals are defined analogously. The remaining literals are *shared*; each of them occurs, negated or not, in an A^*-clause *and* in a B^*-clause.

The construction of partial interpolants begins with assigning one of the marks a, b, or ab to every literal in every leaf clause [5]. The rule is: mark each occurrence of any shared literal arbitrarily, and mark each occurrence of a non-shared literal either a or b, depending on whether the literal is A^*-local or B^*-local.[2]

[1] Treat $\sim$ just as a binary relation. We are currently in the logic without equality.

[2] Pudlak's [20] and McMillan's [17] constructions are the special cases that correspond to marking all shared literals with ab or b respectively.

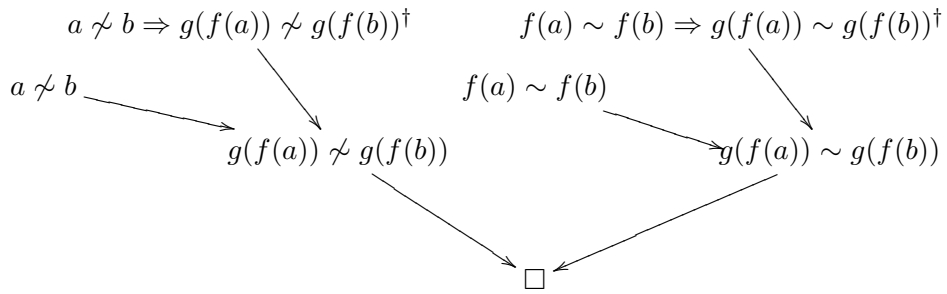


Figure 1: Refutation proof for Example 1

The marking propagates to internal nodes as follows. If n_1 and n_2 are the children nodes of n, then every literal of $\theta(n)$ that occurs in only one of the children clauses $\theta(n_1)$, $\theta(n_2)$ inherits the marking from that child clause. For literals that occur in both children clauses with the same marking, that marking is preserved in $\theta(n)$. In the remaining case (the literal occurs in the children clauses with different markings), the literal is marked ab in $\theta(n)$.

Removing all literals marked b from the clause $\theta(n)$ produces a subclause that will be denoted $\alpha(n)$; the additional removal of literals marked ab produces $\alpha^\sharp(n)$. The subclauses $\beta(n)$ and $\beta^\sharp(n)$ are defined analogously. Note that

$$\alpha(n) \vee \beta^\sharp(n) \;\equiv\; \theta(n) \;\equiv\; \alpha^\sharp(n) \vee \beta(n) \tag{1}$$

Partial interpolants $I(n)$ will be crafted with the aim to prove that

$$A \models I(n) \vee \alpha^\sharp(n) \;\text{ and }\; B \models \neg I(n) \vee \beta^\sharp(n) \tag{2}$$

hold for every node n of Ω. Since both $\alpha^\sharp(\mathit{root})$ and $\beta^\sharp(\mathit{root})$ are empty clauses, it will follow from (2) that the formula $I(\mathit{root})$ is an interpolant for A and B provided it is written over the common signature $\Sigma_A \cap \Sigma_B$.

The inductive definition of partial interpolants begins with

$$I(n) = \beta(n), \text{ if } n \text{ is an } A\text{-leaf}; \quad \text{and} \quad I(n) = \neg\alpha(n), \text{ if } n \text{ is a } B\text{-leaf} \tag{3}$$

which satisfies (2) in view of (1).

To define $I(n)$ for an internal node n with children n_1 and n_2, we first obtain a propositional combination $J(n)$ of $I(n_1)$ and $I(n_2)$. Assuming L is the literal that is resolved at the node n, and that L and $\bar{L}$ occur in $\theta(n_1)$ and $\theta(n_2)$ respectively, we look at the mark of L in $\theta(n_1)$ and the mark of $\bar{L}$ in $\theta(n_2)$, and define

$$J(n) = \begin{cases} I(n_1) \vee I(n_2) & \text{if both marks are a} \\ I(n_1) \wedge I(n_2) & \text{if both marks are b} \\ (L \vee I(n_1)) \wedge (\bar{L} \vee I(n_2)) & \text{otherwise} \end{cases} \tag{4}$$

Note that the last case applies when the two marks are distinct or both are equal to ab.

Lemma 1 (D'Silva [4]) *If both $I(n_1)$ and $I(n_2)$ satisfy condition* (2)*, then so does $J(n)$.* ∎

The partial interpolant $I(n)$ will be obtained by adding some quantifiers to the formula $J(n)$, in a non-unique manner. There is a degree of freedom here, expressed by the parameter F in the formula $I(n) \equiv \mathsf{Q}^A_F(J(n))$ of Lemma 4 below. The parameter F ranges over a set of sequences of subterms of $J(n)$ called *flags*.

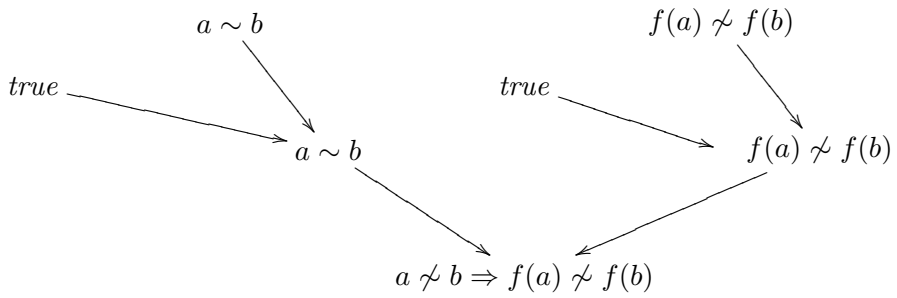


Figure 2: Partial interpolants for Example 1

Before precise definitions it behooves us to look at some examples.

Observe that if we set $I(n) = J(n)$ for every node, we will get the formula $I(root)$ satisfying $A \models I(root)$ and $B \models \neg I(root)$. It need not be an interpolant because non-common symbols may occur in it. We give three examples, and in each of them we show how to fix $I(root)$ into an interpolant by replacing non-common subterms with quantified variables.

Example 2 *The signatures are $\Sigma_A = \{a, R\}$ and $\Sigma_B = \{b, R\}$. Each of the clause sets A and B consists of a single clause: $R(a, v)$ and $\neg R(u, b)$ respectively. (u and v are variables.) The instances $R(a, b)$ and $\neg R(a, b)$ of these clauses are the labels of the leaves of a one-step refutation. The marking algorithm allows us to arbitrarily choose the marking of the only occurring literal $R(a, b)$. Each of the nine (3×3) choices leads by the formula* (4) *to $J(root) = R(a, b)$. Replacing the non-common terms a and b with fresh variables gives us $R(x, y)$. Of all the possible ways to close this formula, two are interpolants: $\exists x\, \forall y\, R(x, y)$ and $\forall y\, \exists x\, R(x, y)$.*

Example 3 *The signatures are $\Sigma_A = \{g, f, \sim\}$ and $\Sigma_B = \{h, f, \sim\}$, and we again have singleton clause sets $A = \{f(u, g(u)) \sim f(g(u), u)\}$ and $B = \{f(h(v), w) \not\sim f(w, h(v))\}$. The substitutions $[u \mapsto h(p)]$ and $[v \mapsto p, w \mapsto g(h(p))]$ applied to these clauses produce the ground clause $f(h(p), g(h(p))) \sim f(g(h(p)), h(p))$ and its negation. As in Example 2, $J(root)$ is exactly this clause. With fresh variables x for $h(p)$ and y for $g(h(p))$, it becomes $f(x, y) \sim f(y, x)$. It is not difficult to check that the clause can be closed only in one way to become an interpolant: $\forall x\, \exists y\, f(x, y) \sim f(y, x)$.*

Example 4 *In Example 1, the literals $a \sim b$ and $f(a) \sim f(b)$ occur (possibly negated) in both A^* and B^*. We can mark their occurrences arbitrarily, but for definiteness, let us use McMillan's marking and mark them* b. *The remaining two literals occur only in A^*-clauses and so must be marked* a. *With $I(n) = J(n)$ for all nodes, the formula* (4) *produces partial interpolants shown in Figure 2. Abstraction of non-common terms a and b from $J(root)$ with fresh variables and subsequent quantification leads to the interpolant $\forall x\, \forall y\, (x \not\sim y \Rightarrow f(x) \not\sim f(y))$. The choice of quantifiers is again unique.*

Huang [10] shows how to replace the non-common symbols in $I(root)$ with fresh variables and prefix the resulting formula with quantifiers that bound the fresh variables, so that at the end we have the desired interpolant for A and B. Christ and Hoenicke [1] observe that quantification does not need to wait for the end of the process, when $J(root)$ is obtained. Instead, there is an opportunity to introduce some quantifiers when generating the partial interpolant $I(n)$ from $J(n)$ at each node. The amount of quantifier introductions is arbitrary to a certain extent and we will precisely specify it. Huang's algorithm comes out as a special case when all quantification is postponed until the root is reached. At the other extreme, choosing to introduce as many quantifiers as possible at each node results in the algorithm of [1].

Example 5 *With the clause sets $A = \{q \vee p_1(u), \neg q \vee p_2(u)\}$ and $B = \{\neg(p_1(u)), \neg(p_2(u))\}$ over the respective signatures $\Sigma_A = \{p_1, p_2, q\}$ and $\Sigma_B = \{p_1, p_2, a\}$, the two extreme quantification strategies produce distinct intepolants:*

$$I_{\text{Huang}} = \forall x\,(p_1(x) \vee p_2(x)) \qquad I_{\text{Ch-Ho}} = (\forall x\, p_1(x)) \ \vee \ (\forall x\, p_2(x))$$

Example 6 *This example demonstrates that there may be even more quantification choices than the two extremes I_{Huang} and $I_{\text{Ch-Ho}}$. Start with any pair A, B, where, as in Example 5, I_{Huang} and $I_{\text{Ch-Ho}}$ are distinct. Let s be a fresh propositional variable, let A^* be the set that contains the clauses $s \vee c$ and $\neg s \vee c$ for every clause $c \in A$, and let B^* be derived from B in the same manner. Let Ω be a refutation tree for the pair A, B. Take two copies Ω_1 and Ω_2 of Ω. For every leaf n of Ω labelled with the clause c (say), label the corresponding node n_1 of Ω_1 with $s \vee c$ and the corresponding node of Ω_2 with $\neg s \vee c$. Add the root node to the union of Ω_1 and Ω_2 to obtain a refutation tree Ω^* for A^*, B^*.*

Suppose two partial interpolation strategies applied to Ω produce interpolants I_1 and I_2. Then we can apply a combination of these strategies to Ω^*; the first on its subtree Ω_1 and the second on Ω_2. The resulting interpolant I^* for A^*, B^* will be $(s \wedge I_1) \vee (\neg s \wedge I_2)$. Thus, among other choices, I^* can be I_{Huang} (when $I_1 = I_2 = I_{\text{Huang}}$), or $I_{\text{Ch-Ho}}$, or $(s \wedge I_{\text{Huang}}) \vee (\neg s \wedge I_{\text{Ch-Ho}})$.

The FOL Interpolation Algorithm. We classify terms over $\Sigma_A \cup \Sigma_B$ into *straight*, *A-bent* and *B-bent* depending on whether their top symbol is common (belongs to $\Sigma_A \cap \Sigma_B$), or is exclusively in Σ_A, or exclusively in Σ_B. [3] For a formula ϕ over $\Sigma_A \cup \Sigma_B$ and a bent term t, define

$$\mathsf{Q}_t^A \phi = \begin{cases} \exists z\, \phi[t \mapsto z] & \text{if } t \text{ is } A\text{-bent} \\ \forall z\, \phi[t \mapsto z] & \text{if } t \text{ is } B\text{-bent} \end{cases}$$

where z is a fresh variable. Define $\mathsf{Q}_t^B \phi$ dually (swapping $\forall$ and $\exists$ above). The following lemma is proved by induction on the derivation of $A \vdash \phi$ and $B \vdash \phi$ in any proof system for first-order logic.

Lemma 2 *Let t be a bent ground term. If $A \models \phi$ then $A \models \mathsf{Q}_t^A \phi$. If $B \models \phi$ then $B \models \mathsf{Q}_t^B \phi$.*

By definition, a *bent factor* of a formula ϕ is a bent ground term that has an occurrence in ϕ at which all its superterms are straight and ground.

A sequence $F = \langle t_1, t_2, \ldots, t_k \rangle$ of bent factors of ϕ will be called a *flag* if it respects the subterm ordering (t_i is a subterm of t_j only if $i < j$); and it is upward-closed in the sense that every bent factor that contains some member of F as a subterm is itself a member of F. A flag $\langle t_1, t_2, \ldots, t_k \rangle$ is *maximal* if it is not a subsequence of another flag. For any flag F of ϕ, we define $\mathsf{Q}_F^A \phi = \mathsf{Q}_{t_1}^A\, \mathsf{Q}_{t_2}^A \cdots \mathsf{Q}_{t_k}^A \phi$ and $\mathsf{Q}_F^B \phi = \mathsf{Q}_{t_1}^B\, \mathsf{Q}_{t_2}^B \cdots \mathsf{Q}_{t_k}^B \phi$.

Lemma 3 *Let F be a flag of ϕ. (1) If $A \models \phi$ then $A \models \mathsf{Q}_F^A \phi$. If $B \models \phi$ then $B \models \mathsf{Q}_F^B \phi$.* ∎ *(2) Then $\models \mathsf{Q}_F^A(\neg\phi) \Leftrightarrow \neg(\mathsf{Q}_F^B\, \phi)$.*

Proof: Part (1) is a simple generalization of Lemma 2. For part (2), obtain $\models \mathsf{Q}_t^A(\neg\phi) \Leftrightarrow \neg(\mathsf{Q}_t^B \phi)$ from definitions. The general case follows by induction. ∎

Lemma 4 *Let n be a node of Ω with children n_1 and n_2. Suppose*

[3] The *top symbol* of a term t is that f for which t is of the form $f(t_1, \ldots, t_n)$.

- $I(n_1)$ *and* $I(n_2)$ *have been defined and both satisfy property* (2)*;*
- $J(n)$ *is as in* (4)*;*
- F *is a flag of* $J(n)$ *that does not contain any subterms of* $\alpha^\sharp(n)$ *or* $\beta^\sharp(n)$.

Then (2) *holds at* n *for* $I(n) \equiv \mathsf{Q}_F^A(J(n))$.

Proof: Since F contains no subterms of $\alpha^\sharp(n)$, it is a flag of $J(n) \vee \alpha^\sharp(n)$ and the formulas $\mathsf{Q}_F^A(J(n) \vee \alpha^\sharp(n))$ and $(\mathsf{Q}_F^A(J(n))) \vee \alpha^\sharp(n)$ are equivalent. The latter is the same as $I(n) \vee \alpha^\sharp(n)$. Since $A \models J(n) \vee \alpha^\sharp(n)$ holds by Lemma 1, we can deduce $A \models I(n) \vee \alpha^\sharp(n)$ with the aid of Lemma 3.

The proof of $B \models \neg I(n) \vee \beta^\sharp(n)$ is similar after we represent $\neg I(n)$ as $\mathsf{Q}_F^B(\neg J(n))$ (Lemma 3). ∎

We summarize the computation of an interpolant for the pair A, B of sets of clauses:

Algorithm 1

1. *Obtain a refutation tree* Ω *for* $A^* \cup B^*$*, where* A^* *is some finite set of ground instances of* A*, and similarly for* B^*.
2. *Starting with a legitimate choice of marking of literal occurrences in* $\theta(n)$ *at the leaves of* Ω*, extend the marking to literal occurrences in the clauses labeling the internal nodes of* Ω.
3. *Define* $I(n)$ *at the leaves as in* (3).
4. *For each internal node* n*, define* $J(n)$ *as in* (4)*, then choose a flag* F *of* $J(n)$ *that does not contain subterms of* $\alpha^\sharp(n)$ *or* $\beta^\sharp(n)$*, then set* $I(n)$ *to be* $\mathsf{Q}_F^A(J(n))$.
5. *At the root, choose a maximal flag* F *and set* $I(root) = \mathsf{Q}_F^A(J(root))$.

Theorem 1 *The formula* $I(root)$ *produced at the end of a run of Algorithm 1 is an interpolant for* A, B.

Proof: The conditions in (2) hold for every node n of Ω. This follows by induction, where the induction step is justified by Lemma 4 and the base case (when n is a leaf) holds as observed after the defining equation (3). The instance of (2) with $n = root$ gives us $A \models I(root)$ and $B \models \neg I(root)$, so it only remains to check that $I(root)$ does not contain symbols that are not in the common signature $\Sigma_A \cap \Sigma_B$. This follows from: (1) the definition of $I(root)$ as $\mathsf{Q}_F^A(J(root))$, where F is a maximal flag of $J(root)$; and (2) the general observation that when F is a maximal flag of ϕ, then $\mathsf{Q}_F^A\phi$ has no bent factors and therefore no occurrences of symbols not in $\Sigma_A \cap \Sigma_B$. ∎

4 Quantified Interpolants for Theories

We proceed to extend this method to allow the construction of interpolants so that it works in the presence of background theories. For recursively enumerable theories, interpolants can be generated by reduction to pure first-order reasoning by enumerating the (finitely many, by compactness) theory lemmas used in a proof of unsatisfiability [11]. We show how interpolants can be computed directly on top of decision procedures.

Interpolation Modulo Theories. A Σ-*theory* is a set of Σ-sentences closed under logical deduction.[4] Given a Σ-theory T, a T-*model* is a Σ-structure that satisfies all sentences in T. A Σ-formula φ over a set V of free variable symbols is T-*valid*, denoted $\models_T \varphi$, if it is satisfied by all T-models over V. A Σ-formula is

[4] A set T of Σ-sentences is closed under logical deduction if whenever $\varphi \in T$ and $\varphi \Rightarrow \psi$ is valid, we have $\psi \in T$.

T-*satisfiable* if it is satisfied by some T-model over V, and is T-*unsatisfiable* if it is not T-satisfiable. The *satisfiability problem* of a Σ-theory T is the problem of deciding, for every Σ-formula φ, whether or not φ is T-satisfiable.

Let T be a Σ-theory. Let A be a Σ_A-formula and B a Σ_B-formula, with $\Sigma \subseteq \Sigma_A \cap \Sigma_B$, such that $A \wedge B$ is T-unsatisfiable. A formula I is a T-*interpolant* of (A, B) if: (1) I is over $\Sigma_A \cap \Sigma_B$; (2) $\models_T A \Rightarrow I$; and (3) $\models_T B \Rightarrow \neg I$.

The rest of §4 explains how satisfiability-modulo-theory solvers (based on decision procedures) can be extended into procedures for T-interpolant generation. We will need the following modulo T generalizations of results of §3. The proofs apply verbatim.

Proposition 1 *Lemma 1 holds if $\models$ in* (2) *is replaced with $\models_T$. Lemmas 2–4 also remain true if $\models$ in them is replaced with $\models_T$.* ∎

(I) Theory Lemmas and Partitioning Them. Modern SMT solvers combine a propositional SAT solver with decision procedures that can check T-satisfiability of sets of literals. When a decision procedure is called on an input set $\mathcal{L}$ of literals and it finds this set unsatisfiable, then it usually finds an unsatisfiable subset $\{L_1, \ldots, L_k\}$ of $\mathcal{L}$, and we know that the clause $\overline{L}_1 \vee \cdots \vee \overline{L}_k$ is a theorem of T. Theorems that arise this way in runs of an SMT solver are called *theory lemmas*.

During a successful refutation run of an SMT solver applied to an input clause set $A \cup B$, the solver will generate finite sets A^* of ground instances of clauses of A, B^* of ground instances of clauses of B, and Λ of theory lemmas, the union of which is *propositionally* unsatisfiable. Moreover, it is possible to extract from the run of the solver a refutation tree Ω with the clauses from A^*, B^*, and Λ at the leaves.

With the refutation tree Ω at hand, we can partition Λ arbitrarily into Λ_A and Λ_B and then use the Algorithm 1 of §3 with the sets $A^\dagger = A^* \cup \Lambda_A$ and $B^\dagger = B^* \cup \Lambda_B$ in place of A^* and B^*. The formula $I(root)$ produced by the algorithm will be a T-interpolant for the pair (A, B). To prove this, we can repeat the proof of Theorem 1, replacing $\models$ in it with $\models_T$, and referring to Proposition 1. The only place where an additional argument is needed is for the base case of the induction: checking that the $\models_T$-version of (2) holds for every leaf node n. In view of (3), this amounts to proving that $A \models_T \theta(n)$ holds if $\theta(n) \in A^\dagger$, and the $B \models_T \theta(n)$ holds if $\theta(n) \in B^\dagger$. For the proof of the first statement, just note that $A \models \theta(n)$ holds when $\theta(n) \in A^*$ and that $\models_T \theta(n)$ holds when $\theta(n) \in \Lambda_A$. The second statement is symmetric.

Example 7 *The choice of the partition of Λ into Λ_A and Λ_B affects the resulting interpolant. Let T be the theory of extensional arrays [14] over the signature $\Sigma = \{read, write\}$, with the axioms*

$$read(write(u, i, x), i) = x \qquad i = j \vee read(write(u, i, x), j) = read(u, j)$$
$$u = v \Leftrightarrow \forall i(read(u, i) = read(v, i))$$

(Axioms are implicitly universally quantified.) Let $\Sigma_A = \Sigma \cup \{r, s, a, e\}$, $\Sigma_B = \Sigma \cup \{r, s, b, c\}$, and let $A = \{r = write(s, a, e)\}$ and $B = \{b \neq c, read(r, b) \neq read(s, b), read(r, c) \neq read(s, c)\}$ be sets of unit (single-literal) ground clauses. Theory lemmas

$$b \neq c \Rightarrow a \neq b \vee a \neq c \qquad a \neq b \wedge r = write(s, a, e) \Rightarrow read(r, b) = read(s, b)$$
$$a \neq c \wedge r = write(s, a, e) \Rightarrow read(r, c) = read(s, c)$$

can be easily combined with the clauses of A and B into a refutation tree. If we treat all theory lemmas as A-clauses (that is, set $\Lambda_B = \emptyset$) then we obtain the interpolant $\forall j \, \forall k \, (j = k \vee read(r, j) = read(s, j) \vee read(r, k) = read(s, k))$; at the other extreme, treating theory lemmas as B-clauses produces the interpolant $\exists i \, \exists x \, (r = write(s, i, x))$. In both cases, we use Huang's marking and quantification procedure.

Instead of dumping all theory lemmas in one of the sets $A^\dagger, B^\dagger$, we can attempt to minimize the number of shared literals—and with it the number of quantifiers in the interpolant—with the following symmetric partitioning algorithm. Let $\mathcal{L}$ be the set of all literals (and their negations) that occur in $A^* \cup B^* \cup \Lambda$. Let $\mathcal{L}_{\mathsf{ab}}$, $\mathcal{L}_{\mathsf{a}}$, $\mathcal{L}_{\mathsf{b}}$ be the sets of shared, A^*-local, and B^*-local literals respectively, as defined in §3. We have a partition $\mathcal{L} = \mathcal{L}_{\mathsf{a}} + \mathcal{L}_{\mathsf{b}} + \mathcal{L}_{\mathsf{ab}} + \mathcal{L}_{\text{new}}$, where $\mathcal{L}_{\text{new}}$ contains literals that occur only in theory lemmas. Let Λ_{a} be the set of theory lemmas that do not use any $\mathcal{L}_{\mathsf{b}}$-literals, and define Λ_{b} analogously. Now add Λ_{a} to A^*, add Λ_{b} to B^*, and repeat the procedure with these new A^* and B^*. Repeat the procedure until it stabilizes, which happens when both Λ_{a} and Λ_{b} are empty. At this point, put each of the remaining theory lemmas in $A^\dagger$ or $B^\dagger$ arbitrarily.

For example, consider the unsatisfiable pair $s = \mathit{write}(r, a, x) \wedge \mathit{read}(s, b) \neq x$ and $t = \mathit{write}(r, a, y) \wedge \mathit{read}(t, b) = y \wedge \mathit{read}(r, b) \neq y$. A proof of unsatisfiability can be generated using theory lemmas $s = \mathit{write}(r, a, x) \Rightarrow \mathit{read}(s, a) = x$; $\mathit{read}(s, a) = x \wedge \mathit{read}(s, b) \neq x \Rightarrow a \neq b$; and $a \neq b \wedge t = \mathit{write}(r, a, y) \Rightarrow \mathit{read}(t, b) = \mathit{read}(r, b)$. The partitioning algorithm above would put the first two lemmas in $A^\dagger$, and the last in $B^\dagger$, and the interpolant would be $a \neq b$. If, instead, we put all the lemmas in $A^\dagger$, then t and y need to be quantified. Similarly, if all the lemmas were put in $B^\dagger$, then s and x need to be quantified.

(II) Ground Interpolation. A theory T is *ground-interpolating* if for every T-lemma split into two clauses $A \vee B$ there exists a ground formula ϕ such that $\models_T A \vee \phi$, $\models_T B \vee \neg\phi$, and ϕ uses only symbols that either belong to the signature of T or are shared by A and B.[5] Efficient ground-interpolating algorithms exist for some common theories (rational/real linear arithmetic; EUF—"equality with uninterpreted functions") [2, 6, 15]. On the other hand, there are theories of practical interest that are not ground-interpolating; in fact, Example 7 shows the theory of arrays is not ground-interpolating.

A ground-interpolating algorithm can replace the generic treatment of theory lemmas in §4(I) with a more streamlined construction of partial interpolants. Suppose that the resolution tree Ω and the clause sets A^*, B^*, and Λ are as in §4(I), and let us make a simplifying assumption that $\mathcal{L}_{\text{new}} = \emptyset$, i.e., that the theory lemmas involve only literals that occur in A^* or B^*.[6] We apply the construction of partial interpolants for the nodes of Ω much as presented in §3. The only substantial addition is the extension of the base-case equations (3) to cover theory lemmas; for every leaf node n, we now define

$$I(n) = \begin{cases} \beta(n) & \text{if } \theta(n) \in A^* \\ \neg\alpha(n) & \text{if } \theta(n) \in B^* \\ I_1 \vee I_2 \vee \gamma(n) & \text{if } \theta(n) \in \Lambda \end{cases} \tag{5}$$

where $\gamma(n)$ denotes the part of $\theta(n)$ marked ab and I_1, I_2 are the theory interpolants for the clause $\theta(n)$ split as $(\alpha^\sharp(n) \vee \gamma(n)) \vee \beta^\sharp(n)$ and $\alpha^\sharp(n) \vee (\gamma(n) \vee \beta^\sharp(n))$ respectively. Thus, in the case $\theta(n) \in \Lambda$, we have

$$A \models_T I_1 \vee \alpha^\sharp(n) \vee \gamma(n) \qquad B \models_T I_1 \vee \beta^\sharp(n)$$
$$A \models_T I_2 \vee \alpha^\sharp(n) \qquad B \models_T I_2 \vee \gamma(n) \vee \beta^\sharp(n)$$

and this implies $A \models_T I(n) \vee \alpha^\sharp(n)$ and $B \models_T \neg I(n) \vee \beta^\sharp(n)$, which is condition (2) with $\models_T$ in place of $\models$. Apart from this, we leave Algorithm 1 unchanged. The formula $I(\mathit{root})$ it produces is a T-interpolant for (A, B). The proof is again a repetition of the proof of Theorem 1 relativized modulo T.

[5] Another way to phrase this definition would be by requiring that a T-interpolant exists for every pair of mutually inconsistent sets of literals. Since this condition implies the existence of a ground interpolant for every T-inconsistent pair of ground *formulas* [2, 7, 15], the name *ground-interpolating* is justified.

[6] This is actually no restriction if the theory lemmas are produced by a $DPLL(T)$ solver [19]. Solvers of this kind work on a set of literals present in the input clauses, and do not extend it [7].

Note that if $\theta(n)$ is a theory lemma in which $\gamma(n)$ is an empty clause, then there is only one splitting of $\theta(n)$, namely $\alpha^\sharp(n) \vee \beta^\sharp(n)$, to which we need to apply ground interpolation, and $I(n)$ is the resulting theory interpolant. This simple situation always arises when one uses "McMillan's marking" (see footnote 5 in §3), as done in [2, 7, 15].

Example 8 *Using ground interpolation algorithms, we can produce simpler interpolants than with the generic procidure in §4(I). The theory is EUF over the signature $\Sigma = \{\circ, c, d, e\}$. Let $\Sigma_A = \Sigma \cup \{a\}$, $\Sigma_B = \Sigma \cup \{b\}$, and consider the clause sets $A = \{a = c, a \circ d = e\}$ and $B = \{b = d, c \circ b \neq e\}$. The theory lemma*

$$\Lambda: \quad a \neq c \ \vee \ a \circ d \neq e \ \vee \ b \neq d \ \vee \ c \circ b = e$$

combines with the four literals of A and B into a (linear) resolution proof. The theory interpolant for the lemma is $I \equiv c \circ d = e$ and it is also the final interpolant for A and B.[7] On the other hand, if we do as suggested in §4(I) and add the theory lemma to A, the algorithm of §3 would produce the interpolant $I_A \equiv \forall y\,(y = d \Rightarrow c \circ y = e)$. And if we add the lemma to B, the same algorithm would produce the interpolant $I_B \equiv \exists x\,(x = c \wedge x \circ d = e)$.

Clearly, I is a better interpolant than I_A or I_B. One might object that I can be reconstructed from either I_A or I_B by a simple simplification procedure, but that happens by accident. To see this, let us modify the example by taking $\Sigma_A = \Sigma \cup \{a_1, a_2\}$ and replacing a with $a_1 \circ a_2$ in the two clauses of A. The theory interpolant remains the same I, but I_A and I_B become $\forall y_1\,\forall y_2\,(y_1 \circ y_2 = d \Rightarrow c \circ (y_1 \circ y_2) = e)$ and $\exists x_1\,\exists x_2\,((x_1 \circ x_2) = c \wedge (x_1 \circ x_2) \circ d = e)$, and neither is now equivalent with I.

(III) Combination of Theories. Modern SMT solvers typically use decision procedures for several theories (say, $T_1, \ldots, T_n$) combined in a "DPLL plus Nelson-Oppen" fashion, abstractly described in, e.g., [13]. As mentioned in §4(I), a successful refutation run of the solver on the input $A \cup B$ produces a refutation tree Ω at the leaves of which are the clauses, each of which is either an instance of an A-clause, an instance of a B-clause, or a theory lemma, that is, a T_i-lemma for some $i \in \{1, \ldots, n\}$. This results in the partition $A^* + B^* + \Lambda_1 + \ldots + \Lambda_n$ of the set of clauses that label the leaves of Ω.

Some of the participating theories may come with a ground-interpolating algorithm, some may not, so the interpolation algorithm to be added to a multiple-theories solver is a combination of the algorithms described in §4(I) and §4(II). It works as follows.

Assume that $T_1, \ldots, T_k$ are ground-interpolating, and $T_{k+1}, \ldots, T_n$ are not. In the first stage of the algorithm, we ignore $\Lambda_1, \ldots, \Lambda_k$ and add every lemma of $\Lambda_{k+1}, \ldots, \Lambda_n$ to either A^* or B^* as described in §4(I). The process consists of a few rounds of unambiguous decisions (based on the analysis of literal sharing between each theory lemma and A^* or B^*), finished with arbitrary decisions for the lemmas that do not exhibit a predisposition for either side.

At this point, all clauses at the leaves of Ω are from the set $A^* + B^* + \Lambda_1 + \ldots + \Lambda_k$, with A^* and B^* now including the original non-ground-interpolating theory lemmas. We proceed to define partial interpolants for the leaves of Ω. As earlier, this begins with marking every literal occurence in every leaf clause $\theta(n)$ with a, b, or ab. There is no guarantee, however, that every literal that occurs in a theory lemma must also occur in a clause of $A^* \cup B^*$. We leave such literal occurrences unmarked and redefine $\gamma(n)$ (§4(II)) to be the clause consisting of literals of $\theta(n)$ that are either maked ab or are unmarked. The partial interpolants at the leaves of Ω are now defined by the formula (5) of §4(II), where the "theory interpolants" I_1 and I_2 are obtained using the ground interpolation procedure for the appropriate theory T_i (the i is determined by $\theta(n) \in \Lambda_i$). The proof of correctness does not require any new ideas.

[7] It is easy to verify that I is an interpolant for (A, B). In fact, the ground-intepolating algorithm of [6] would produce exactly this interpolant.

References

[1] J. Christ and J. Hoenicke. Instantiation-based interpolation for quantified formulae. In *SMT 2010: 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[2] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS 08*, pages 397–412. Springer, 2008.

[3] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic*, 22(3):269–285, 1957.

[4] V. D'Silva. Propositional interpolation and abstract interpretation. In *ESOP*, volume 6012 of *LNCS*, pages 185–204. Springer, 2010.

[5] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *VMCAI 10*, volume 5944 of *LNCS*, pages 129–145. Springer, 2010.

[6] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground interpolation for the theory of equality. In *TACAS 09*, pages 413–427. Springer, 2009.

[7] A. Goel, S. Krstić, and C. Tinelli. Ground interpolation for combined theories. In *CADE 09*, volume 5663 of *LNAI*, pages 183–198. Springer, 2009.

[8] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[9] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[10] G. Huang. Constructing Craig interpolation formulas. In *COCOON 95*, LNCS, pages 181–190. Springer, 1995.

[11] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *FSE 06*, pages 105–116. ACM, 2006.

[12] J. Krajicek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symbolic Logic*, 62:457–486, 1997.

[13] S. Krstić and A. Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *FroCoS 07*, pages 1–27. Springer, 2007.

[14] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[15] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[16] K. L. McMillan. Lazy abstraction with interpolants. In *In Proc. CAV, LNCS 4144*, pages 123–136. Springer, 2006.

[17] K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer, 2003.

[18] K.L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *In LPAR04, LNAI 3452*, pages 36–50. Springer, 2005.

[20] P. Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62:981–998, 1997.

[21] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *CADE 09*, LNCS, pages 140–145. Springer, 2009.

Certified Interpolant Generation for EUF*

Andrew Reynolds
The University of Iowa

Cesare Tinelli
The University of Iowa

Liana Hadarean
New York University

Abstract

Logical interpolants have found a wide array of applications in automated verification, including symbolic model checking and predicate abstraction. It is often critical to these applications that reported interpolants exhibit desired properties, correctness being first and foremost. In this paper, we introduce a method in which interpolants are computed by type inference within the trusted core of a proof checker. Interpolants produced this way from a proof of the joint unsatisfiability of two formulas are certified as correct by construction. We focus our attention to the quantifier-free theory of equality and uninterpreted functions (EUF) and present an interpolant generating proof calculus that can be encoded in the LFSC proof checking framework with limited reliance upon computational side conditions. Our experimental results show that our method generates certified interpolants with small overhead with respect to solving.

1 Introduction

Given a logical theory T, a T-interpolant for a pair of formulas A, B that are jointly unsatisfiable in T is a formula φ over the symbols of T and the free symbols common to A and B such that (i) $A \models_T \varphi$ and (ii) $B, \varphi \models_T \bot$, where $\models_T$ is logical entailment modulo T. For certain theories, interpolants can be generated efficiently from a refutation of $A \wedge B$.

Interpolants have been used successfully in a variety of contexts, including symbolic model checking [9] and predicate abstraction [8]. In many applications, it is critical that formulas computed by interpolant-generation procedures be indeed interpolants, that is, exhibit the defining properties above. For example, in [9] McMillan shows how interpolants can be used to produce a sound and complete method for checking safety properties of finite state systems based on a fixed-point computation that over-approximates the set of reachable states. Using as interpolants formulas that violate property (ii) above makes the method incomplete, as it leads to spurious counterexamples that do not contribute to the overall progress of the main algorithm. Even worse, using interpolants formulas that violate property (i) above may cause the algorithm to reach a fixed point prematurely, thereby reporting a unsound result. While there are known methods to generate interpolants efficiently [4, 3], none of them do so in a verified way. This is an obstacle to the use of interpolation-based model checking in applications of formal methods that require independently checkable proof certificates. We present a way of addressing this deficiency by means of a method for generating *certified interpolants*.

We describe how T-interpolants can be efficiently produced using LFSC, a framework for defining logical calculi that extends the Edinburgh Logical Framework [14] with computational side conditions. In our approach, a proof system for the theory T of interest is encoded declaratively as a set of types and typing rules in a user-defined *signature*, separate from the core of the proof checker. The interpolant itself

*Work partially supported by grants #0914956 and #0914877 from the National Science Foundation.

is computed by *type inference* as a side effect of proof checking. Because the interpolant is computed using the trusted core of the proof checker, it is certified as correct by construction.

Interpolant-generating calculi exist for many theories, including quantifier-free linear integer arithmetic [2] and linear real arithmetic [10]. In this work, we focus on the quantifier-free (fragment of the) theory of equality and uninterpreted functions (EUF). Our approach, however, applies to other quantifier-free theories. Extending previous work [6], we present an intuitive interpolant-generating calculus for EUF that is more flexible than previous approaches, and can be implemented in LFSC in a natural way. We have encoded this calculus in LFSC, and instrumented the SMT-solver Cvc3 [1] to output proofs in this format to verify the viability of our approach.

Related Work. In [10], McMillan gives a calculus for interpolant generation in EUF that is implemented within an interpolating theorem prover, intended for the purposes of interpolation-based model checking and predicate refinement. Fuchs *et al.* give an alternative approach in [6] for interpolant generation in EUF in terms of an algorithmic procedure. An approach for efficient interpolant generation in SMT is given by Cimatti *et al.* in [4] and is implemented within the MathSAT SMT solver.

Contributions. The preliminary work discussed here has a two-fold contribution. Firstly, we develop a general framework for generating interpolants in a certified manner via type checking. Secondly, we provide a novel calculus for interpolant generation in EUF based on the procedure of [6] that can be encoded within this framework. We present an implementation of our approach and discuss comparative performance results in various configurations that provide experimental evidence of practical feasibility. We are working on extending this work to arbitrary quantifier-free formulas, as opposed to only conjunction of literals.

Paper Outline. Section 2 gives an overview of the LFSC framework, in particular, how a proof signature can incorporate proof rules that carry additional information. Section 3 contains a description of our interpolant generating calculus for EUF, and details on how it is encoded. A detailed proof of soundness and (relative) completeness can be found in a longer version of this paper [13]. Finally, experimental results from our implementation are given in Section 4.

1.1 Formal Preliminaries

We work in the context of first-order logic with equality, and use standard notions of signature, term, literal, formula, clause, Horn clause, entailment, satisfiability, and so on. We use the symbol $\approx$ to denote the equality predicate in the logic. We abbreviate $\neg(s \approx t)$ as $s \not\approx t$. We identify finite sets of formulas with the conjunction of their elements. For terms or formulas we use "ground", i.e., variable-free, and "quantifier-free" interchangeably since for our purposes free variables can be always seen as free constants. For convenience, we follow Nieuwenhuis and Oliveras [11] in considering only *Curried signatures*, signatures with no function symbols of positive arity except for a distinguished infix binary symbol $\cdot$. Then, EUF can be defined as the (empty) theory of $\cdot$. This is without loss of generality since a ground formula over any signature can be converted into an equisatisfiable ground formula over the signature above [11]. For example, the formula $g(a) \approx f(a, g(a))$ can be converted into $g \cdot a \approx (f \cdot a) \cdot (g \cdot a)$ with f, g and a all treated as constant symbols.

Throughout the paper we will work with two ground EUF formulas A and B. Let Σ_A and Σ_B be the sets of non-logical symbols (i.e., variables/constants) that occur in A and B, respectively. Terms, literals and formulas over Σ_A are *A-colorable*, and those over Σ_B are *B-colorable*. Such expressions are *colorable* if they are either A- or B-colorable, and are *AB-colorable* if they are both.

$$\frac{t_1 \approx t_2 \quad t_2 \approx t_3}{t_1 \approx t_3}\ \textsf{trans}$$

```
term : type
formula : type
eq : (! t1 term (! t2 term  formula))
proof : (! f formula type)
```

```
trans : (! t1 term (! t2 term (! t3 term
        (! p1 (proof (eq t1 t2)) (! p2 (proof (eq t2 t3))
          (proof (eq t1 t3)))))))
```

$$\frac{t_1 \approx t_2\ [l_1] \quad t_2 \approx t_3\ [l_2]}{t_1 \approx t_3\ [l_1 + t_2 :: l_2]}\ \textsf{trans\_aug}$$

```
...
term_list : type
proof_aug : (! f formula
              (! tl term_list  type))
```

```
trans_aug : (! t1 term (! t2 term (! t3 term
              (! tl1 term_list (! tl2 term_list
                (! p1 (proof_aug (eq t1 t2) tl1)
                (! p2 (proof_aug (eq t2 t3) tl2)
                  (proof_aug (eq t1 t3) (concat tl1 t2 tl2)))))))))
```

Figure 1: Example of two proof systems in LFSC, the second being an augmented version of the first. In LFSC's LISP-like concrete syntax, the symbol ! represents LF's Π binder, for the dependent function space.

2 Interpolant Generation via Type Inference

Edinburgh LF is a framework for defining logics by means of a dependently-typed lambda calculus, the $\lambda\Pi$-calculus [7]. LFSC (Logical Framework with Side Conditions) extends LF by adding support for computational side conditions, i.e., functional programs used to test logical conditions on proof rules [14]. As in LF, a proof system can be defined in LFSC as a list of typing declarations, called a *signature*. Each proof rule is encoded as a constant symbol whose type represents the inference allowed by the rule. An example of a minimal proof system in LFSC is given in the upper half of Figure 1, which shows how the transitivity rule for equality can be encoded in LFSC syntax. In mathematical notation, the LFSC signature in the figure declares eq, for example, as a symbol of type $\Pi\, x{:}\mathsf{term}.\, \Pi\, y{:}\mathsf{term}.\, \mathsf{formula}$, or, written in more conventional form, of type $\mathsf{term} \to \mathsf{term} \to \mathsf{formula}$. The symbol trans, encoding the transitivity proof rule, has type $\Pi\, t_1{:}\mathsf{term}.\, \Pi\, t_2{:}\mathsf{term}.\, \Pi\, t_3{:}\mathsf{term}.\, (\mathsf{proof}\ (\mathsf{eq}\ t_1\ t_2) \to \mathsf{proof}\ (\mathsf{eq}\ t_2\ t_3) \to \mathsf{proof}\ (\mathsf{eq}\ t_1\ t_3))$, indexed by three terms. Intuitively, for any terms t_1, t_2, t_3, trans produces a proof of the equality $t_1 \approx t_3$ from two subproofs p_1 and p_2 of $t_1 \approx t_2$ and $t_2 \approx t_3$, respectively.

The LF metalanguage provides the user with the freedom to choose how to represent logical constructs in a signature. In particular, we may augment proof rules to carry additional information, through the use of suitably modified types. The lower half of Figure 1 shows a modified version of the aforementioned calculus. In this version, the modified proof rule takes as premises *annotated* equalities and produce annotated equalities. The annotation consists of a list of terms occurring in equalities used to prove the overall equality. The corresponding proof judgment in LFSC, encoded as the type proof_aug, is used by the trans_aug rule, which combines a chain of equalities and concatenates their respectively lists with the intermediate term t_2.[1]

One can write proof terms in this calculus where every subterm of type term_list is a (type) variable. Concretely, this is done by using a distinguished *hole* symbol _ , each occurrence of which stands for a different variable. Then, the lists of intermediate terms can be computed by an LFSC type checker by type

[1] Although not shown here, the list concatenation constant concat, of type $\mathsf{term\_list} \to \mathsf{term} \to \mathsf{term\_list} \to \mathsf{term\_list}$, can be given as a logical definition in LFSC, and does not need to be implemented algorithmically.

$$\frac{}{t \approx t}\ \textsf{refl} \qquad \frac{t_2 \approx t_1}{t_1 \approx t_2}\ \textsf{sym} \qquad \frac{s_1 \approx s_2 \quad t_1 \approx t_2}{s_1 \cdot t_1 \approx s_2 \cdot t_2}\ \textsf{cong}$$

$$\frac{t_1 \approx t_2 \quad t_2 \approx t_3}{t_1 \approx t_3}\ \textsf{trans} \qquad \frac{t_1 \approx t_2 \quad \neg(t_1 \approx t_2)}{\bot}\ \textsf{contra}$$

Figure 2: A standard proof calculus for EUF literals. We assume that all terms are in Curried form, and so the only function symbol of non-zero arity is $\_ \cdot \_$, denoting function application.

inference when type checking the proof term. In this example, type checking a proof term P against a type of the form (proof_aug (eq t_1 t_2) _) for some terms t_1 and t_2 will cause the LFSC checker to verify that P has type (proof_aug (eq t_1 t_2) l) for some list of terms l, and if successful, output the computed value of l.

In this example, an account of the steps taken for the proven equality is recorded as specificational data, in the rule's annotation. One can follow the same approach to capture information useful to compute interpolants. In general, one may augment a proof signature to operate on judgments carrying additional terms that satisfy some specific invariant. We will use the common notion of a *partial interpolant* in the next section, which can be encoded as an LF type declaration in a natural way.

Our approach allows for two options for obtaining certified interpolants. First, an LFSC proof term P can be tested against the type (interpolant F) for some formula F. In this option, the interpolant F is explicitly provided as part of the proof, and if proof checking succeeds, then both the proof P and the interpolant F are certified to be correct. Note that the user and the proof checker must agree on the exact form for the interpolant F. Alternatively, P can be tested against the type (interpolant _). If the proof checker verifies that P has type (interpolant F) for some formula F, it will output the computed value of F. In this approach, interpolant generation comes as a side effect of proof checking, via type inference, and the returned interpolant F is correct by construction.

3 Interpolant Generation in EUF with Partial Interpolants

In this section we provide an *interpolating calculus* for EUF whose rules can be used to build refutations in EUF but are also annotated with information for generating interpolants from these refutations. For simplicity, we restrict ourselves to the core case of input sets of formulas containing only literals. A calculus for general quantifier-free formulas is not substantially harder. It can be developed along similar lines, relying on existing methods for extending in a uniform way any interpolation procedure for sets of literals to sets of arbitrary qffs [10]; but this is left to future work.

Our interpolating calculus is an annotated version of the basic calculus for EUF, shown in Figure 2. To compute an interpolant for two jointly unsatisfiable sets of literals A and B, we take a refutation of $A \cup B$ in the basic calculus and *lift* it to a refutation in our interpolating calculus. The lifting is straightforward and consists in essence in (i) annotating the literals of A and B with a suitable *color* (see later) and (ii) replacing each rule application by a corresponding set of rule applications in the interpolating calculus. Every rule in the latter calculus is such that the annotation of the rule's conclusion is derived from the annotations of its premises. Thus, these annotations can be left unspecified in the proof, and derived by type inference in LFSC during proof checking.

Note that since both the proof generation and the proof lifting steps are done before proof checking, neither of them needs to be trusted. This allows us to handle various complications outside the trusted

$$\frac{\{\ t \text{ is } A\text{-colorable}\ \}}{t \approx t\,[\top, \top, A]}\ \mathsf{refl\_A} \qquad \frac{t_1 \approx s_1\,[\varphi_1, \psi_1, A] \qquad t_2 \approx s_2\,[\varphi_2, \psi_2, A]}{t_1 \cdot t_2 \approx s_1 \cdot s_2\,[\varphi_1 \wedge \varphi_2, \psi_1 \wedge \psi_2, A]}\ \mathsf{cong\_A}$$

$$\frac{t_1 \approx t_2\,[A]}{t_1 \approx t_2\,[\top, \top, A]}\ \mathsf{base\_A} \qquad \frac{t_1 \approx t_2\,[\varphi, \psi, A] \quad t_1 \not\approx t_2\,[A]}{\bot\,[\varphi \wedge \neg\psi]}\ \mathsf{contra\_A}$$

$$\frac{t_1 \approx t_2\,[\varphi_1, \psi_1, c] \qquad t_2 \approx t_3\,[\varphi_2, \psi_2, c'] \qquad \{\ t_1, t_3 \text{ are } A\text{-colorable}\ \}}{t_1 \approx t_3\,[\varphi_1 \wedge \varphi_2, \psi_1 \wedge \psi_2, A]}\ \mathsf{trans\_A}$$

$$\frac{\{\ t \text{ is } B\text{-colorable}\ \}}{t \approx t\,[\top, t \approx t, B]}\ \mathsf{refl\_B} \qquad \frac{t_1 \approx s_1\,[\varphi_1, \psi_1, B] \qquad t_2 \approx s_2\,[\varphi_2, \psi_2, B]}{t_1 \cdot t_2 \approx s_1 \cdot s_2\,[\varphi_1 \wedge \varphi_2, t_1 \cdot t_2 \approx s_1 \cdot s_2, B]}\ \mathsf{cong\_B}$$

$$\frac{t_1 \approx t_2\,[B]}{t_1 \approx t_2\,[\top, t_1 \approx t_2, B]}\ \mathsf{base\_B} \qquad \frac{t_1 \approx t_2\,[\varphi, \psi, B] \quad t_1 \not\approx t_2\,[B]}{\bot\,[\varphi]}\ \mathsf{contra\_B}$$

$$\frac{t_1 \approx t_2\,[\varphi_1, \psi_1, c_1] \qquad t_2 \approx t_3\,[\varphi_2, \psi_2, c_2] \qquad \{\ t_1, t_3 \text{ are } B\text{-colorable}\ \}}{t_1 \approx t_3\,[\varphi_1 \wedge \varphi_2 \wedge (\psi_1 \wedge \psi_2) \rightarrow (t_1 \approx t_3), t_1 \approx t_3, B]}\ \mathsf{trans\_B}$$

Figure 3: A-Prover and B-Prover Partial Interpolant Rules. The text in braces denote computational side conditions. Note that the trans rules are actually a summary of multiple rules for cases of colors c and c'. In some of these cases, formulas in the annotation of conclusions may be simplified to respect colorability constraints.

core of LFSC, such as applications of congruence between uncolorable terms. In the end, our interpolating calculus can be encoded in 203 lines of LFSC type declarations. The core rules contain just one kind of computational side conditions, which test for term colorability.

3.1 An Interpolating Calculus for EUF

For the rest of this section, we fix two sets A and B of literals such that $A \cup B$ is unsatisfiable in EUF. Without loss of generality, we assume that A and B are disjoint.

An interpolant-generating calculus for EUF can be thought of as one that explicitly records the communication between two provers, a *A-prover* and a *B-prover*, whose initial assumptions are the sets A and B, respectively [6]. In our calculus, this communication is achieved through *proof judgments*. The rules of the calculus derive or use as premises one of three kinds of judgments, of the following forms:

$$A, B \vdash L\,[c], \qquad A, B \vdash t_1 \approx t_2\,[\varphi, \psi, c], \qquad A, B \vdash \bot\,[\varphi]$$

where L is an equational literal, s, t are terms, φ, ψ are quantifier-free formulas, and c is an element of a binary set of *colors*. For convenience, we name these colors A and B.[2] Although the judgments are parameterized by the literal sets A and B, these sets do not change within a proof. So from now on, we will omit $A, B \vdash$ from proof judgments. Each judgment will consist of a literal annotated with additional information (the information enclosed in square brackets). The calculus starts with the set of judgments

$$\{L\,[A] \mid L \in A\} \cup \{L\,[B] \mid L \in B\},$$

and derives new judgments according to the proof rules defined in Figure 3.

[2]It will be clear from context whether A (B) refers to the input clause set or its associated color.

Many of the rules in Figure 3 are annotated versions of those in the basic calculus of Figure 2. In addition, the rules base_A and base_B are used to provide an equality with the proper annotation. The only side condition our calculus requires is a test for whether a particular term is colorable. Although not shown here, symmetry may be applied to judgments of the form $t_1 \approx t_2\,[c]$ to conclude $t_2 \approx t_1\,[c]$. This was done for simplicity, and does not impact the relative completeness of the calculus. The A and B provers communicate through judgments of the form $t_1 \approx t_2\,[\varphi, \psi, c]$, where $t_1 \approx t_2$ is the overall equality that A and B have cooperated in proving; φ contains interpolation information provided by the A-prover for the benefit of the B-prover; ψ contains interpolation information provided in turn by the B-prover for the A-prover, possibly using information provided by the A-prover. By construction, φ is a conjunction of Horn clauses and ψ a conjunction of literals.

In addition to its limited dependence on computational side conditions, a clear advantage of our calculus for EUF is its flexibility. In particular, the user has the option of applying either the trans_A or trans_B rules for equalities between AB-colorable terms. This choice produces different interpolants, with different size and logical strengths.

To show that our calculus is interpolating, we will use the fact that all derived judgments of the form $s \approx t\,[\varphi, \psi, c]$ are partial interpolants in the sense below.

Definition 1. *A judgment J of the form $t_1 \approx t_2\,[\varphi, \psi, c]$ is a* partial interpolant *if the following hold:* (1) $A \models \varphi$; (2) $B, \varphi \models \psi$; (3) $A, \psi \models t_1 \approx t_2$; (4) t_i *is c-colorable for* $i = 1, 2$; (5) *either*

a. J is $t_1 \approx t_2\,[\varphi, \psi, A]$, and φ and ψ are AB-colorable, or

b. J is $t_1 \approx t_2\,[\varphi, t_1 \approx t_2, B]$ and φ is AB-colorable.

In the definition above, when J is $t_1 \approx t_2\,[\varphi, \psi, A]$, the formula $\varphi \wedge \neg\psi$ is an interpolant for $(A \wedge t_1 \not\approx t_2, B)$. Similarly, when J is $t_1 \approx t_2\,[\varphi, t_1 \approx t_2, B]$, the formula φ is an interpolant for $(A, B \wedge t_1 \not\approx t_2))$.

Our calculus is sound and complete for interpolation in EUF in the following sense.

Theorem 2. *The following hold:*

a. For all derivable judgments $\bot\,[I]$ we have that (i) $A \models I$, (ii) $B, I \models \bot$ and (iii) I is AB-colorable.

b. For every jointly unsatisfiable set of ground EUF literals A and B, there exists a derivation of the judgment $\bot\,[I]$, for some formula I.

The proof of soundness above uses fairly standard arguments. That of completeness relies on the fact that EUF is equality interpolating [15], i.e,, for all colorable terms s, t such that $A \wedge B \models s \approx t$, there exists an AB-colorable term u such that $A, B \models s \approx u \wedge u \approx t$. The term u, and its subterms, which may not occur in $A \cup B$, can be introduced as needed in a proof in our calculus by the refl_A and refl_B rules.

3.2 Encoding Into LFSC

Our calculus for interpolant generation in EUF can be encoded in an LFSC signature with limited reliance on computational side conditions. The encoding of the judgements and the rules is relatively straightforward. An excerpt of the encoding is provided in Figure 4. Looking at it salient features, we encode color as a base type in our signature, with two nullary term constructors, A and B. Interpolant judgments $\bot\,[\varphi]$ are encoded as the type (interpolant φ). Partial interpolants $t_1 \approx t_2\,[\varphi, \psi, c]$ are encoded as the type (p_interpolant t_1 t_2 φ ψ c). In proof terms, we specify whether an input literal L occurs in the set A or B by introducing (local) lambda variables of type (colored L c), where c is the color A or the color B, respectively. Since formulas in the literal sets A and B can be inferred from the types of the free variables in our proof terms, the sets do not need to be explicitly recorded as part of the proof judgment types.

```
formula : type          colored : (! f formula (! c color type))
term : type             interpolant : (! f formula type)
color : type            p_interpolant : (! t1 term (! t2 term
A: color                                  (! f1 formula (! f2 formula
B : color                                  (! c color type)))))
```

Figure 4: An excerpt of the LFSC signature that encodes the interpolation calculus.

3.3 Side Conditions for Testing Colorability

Proving colorability for expressions (that is, proving that the non-logical symbols in an expression occur all in A or all in B) is a common requirement for interpolant generating calculi. To prove such facts a purely declarative calculus would require $\Omega(n)$ proof rule applications for an expression E, where n is the number of symbols in E. Since LFSC's side conditions can be used for this purpose, we decided in this work to verify colorability through them.

Side conditions in LFSC are expressed in a simply typed functional programming language with minimal imperative features, and are intended to be simple enough to be verified by manual inspection. As described in [14], LFSC contains limited support for computational tests run on terms with a mutable state. In particular, each lambda variable introduced in a proof term contains 32 bit fields that are accessible to the writer of the signature. The semantics of LFSC's side condition language provides constructs for toggling (markvar C) and scrutinizing (ifmarked C C_1 C_2) these bit fields, where in both cases C is a code term that evaluates to an LF variable.

We may enforce a scheme for testing colorability using two of these bit fields, one for denoting A-colorable and the other for denoting B-colorable. Whenever a variable of type (colored L c) is introduced in our proof, we use a side condition to traverse L and mark the field specified by color c for all variables. Since LFSC supports term sharing of variables, each mark applies globally for all occurrences of that variable within our proof. To test the c-colorability of a term t, we use another side condition that traverses t and succeeds if and only if the field specified by color c has been marked for all variables occurring in t. In total, the two side condition functions account for 21 lines of functional side condition code.

4 Experimental Results

To evaluate the feasibility of our approach for producing certified interpolants, we used a version of the Cvc3 SMT solver instrumented to produce LFSC proofs from its refutations of EUF formulas. We ran experiments on a set of 25,246 unsatisfiable EUF benchmarks, all of which were a conjunction of equational literals. The benchmarks were extracted from the set of the quantifier-free EUF benchmarks in the SMT-LIB repository as follows.

In its native proof generation mode, Cvc3 produces unsatisfiability proofs with a two-tiered structure, where a propositional resolution style skeleton is filled with theory-specific subproofs of *theory lemmas*, that is, valid (ground) clauses. First, we selected all unsatisfiable EUF benchmarks that Cvc3 could solve in less than 60 seconds. We reran Cvc3 on these benchmarks with native proof generation enabled, and examined all theory lemmas within all proofs. Each theory lemma produced by Cvc3 for EUF is an equational Horn clause. For each theory lemma $e_1 \wedge \cdots \wedge e_n \rightarrow e$ we encountered, we created a new EUF benchmark consisting of the unsatisfiable formula $e_1 \wedge \cdots \wedge e_n \wedge \neg e$. We only considered unique[3] theory lemmas and

[3]Benchmarks were passed through multiple filters for recognizing duplication. It was infeasible to verify that certain theory

		Solving + Pf Gen + Pf Conv (sec)				Proof Size (KB)			Pf Check Time (sec)	
Benchmark	#	cvc	cvcpf	euf	eufi	cvcpf	euf	eufi	euf	eufi
eq_diamond	28	0.07	0.07	0.08	0.08	56.1	37.9	50.9	0.01	0.014
NEQ	2185	4.00	4.75	4.34	5.15	2765.8	2276.3	3873.8	0.27	0.524
PEQ	2252	4.65	6.27	5.81	6.91	4901.2	4256.3	7458.9	0.55	1.044
QG-loops6	2854	5.01	5.64	4.90	5.80	2446.0	1872.9	3052.2	0.21	0.418
QG-qg5	5337	9.05	10.08	10.67	9.60	4189.2	3415.2	5514.4	0.39	0.762
QG-qg6	1789	3.16	3.55	3.43	3.62	1970.1	1493.8	2669.7	0.172	0.368
QG-qg7	7860	16.96	22.77	23.07	25.96	19161.2	18843.2	35527.7	2.544	5.024
SEQ	2941	6.04	7.61	7.11	7.86	5517.8	4315.4	6926.6	0.52	0.948
	25246	48.95	60.74	59.40	64.98	41007.4	36511.1	65074.2	4.666	9.102

Figure 5: Cumulative Results for average of Runs $1 \ldots 5$, grouped by benchmark class. Columns 3 through 6 give Cvc3's (aggregate) runtime for each of the four configurations. Columns 7 through 9 show the proof sizes for each of the three proof-producing configurations. Columns 10 and 11 show LFSC proof checking times for the **euf** and **eufi** configurations.

whose corresponding proof contained at least five deduction steps.

We collected runtimes for the following four configurations of our instrumented version of Cvc3:

cvc: Default, solving benchmarks but with no proof generation.
cvcpf: Solving with proof generation in Cvc3's native format.
euf: Solving with proof generation, translation to LFSC format for EUF.
eufi: Solving with proof generation, translation to LFSC format for interpolant generation in EUF.

In each configuration, Cvc3's decision procedure for EUF was used when solving. We ran each configuration five times and took the average for all runs. For each benchmark used by configuration **eufi**, the sets A and B were determined by randomly placing $\frac{k}{6}$ of the benchmarks into set A and the rest in B on run k for $k = 1 \ldots 5$. This did not affect the difficulty of solving, and we believe provided a sufficient measurement of the effectiveness of our interpolant generation scheme.

We measured total time to solve all benchmarks grouped by benchmark class. These results are shown in Figure 5. In this data set, proof generation came at a 25% overhead with respect to solving. Converting proofs to the LFSC format required very little additional overhead. In fact, LFSC proofs in the **euf** configuration were generally smaller in size (number of bytes) than Cvc3's native proofs, thus slightly reducing proof generation times. LFSC proofs in the interpolanting calculus were nearly twice as large as non-interpolating proofs. The difference in proof size can be attributed to differences in syntax between the two calculi, as well as additional information added to the header of interpolanting proofs for specifying A and B.

As expected from previous work [12], proof checking times using the LFSC checker were very small with respect to solving times. Proof checking times without interpolant generation were about an order of magnitude faster than solving times. Since additional information is inferred within judgments in the **eufi** configuration, proof checking took approximately twice as long with interpolant generation as without. Proof checking times in the **eufi** configuration were a factor of 5 faster than solving times.

We estimate the time to produce uncertified interpolants by measuring solving and proof generation without proof checking (configuration **cvcpf**). Overall, configuration **eufi** shows a 22% overhead with

lemmas were symbolic permutations of others. However, by visual inspection, we believe that such cases were rare.

respect to **cvcpf**, indicating that the generation of certified interpolants is practicably feasible with high-performance SMT solvers.

5 Conclusion and Future Work

We have introduced a method of generating interpolants by type inference within the trusted core of a proof checker for LFSC. Our experiments show that this method is efficient and practical for use with high performance solvers. Overall, interpolant generation in LFSC can be performed 5.38 times faster than solving time on average. Our method is based on a novel calculus for interpolant generation in EUF. By performing certain colorability tests during a proof lifting phase, we are able to simplify the amount of side conditions required during proof checking.

We plan to extend our method to ground EUF formulas with an arbitrary Boolean structure by incorporating an interpolating version of the propositional resolution calculus. Future work includes instrumenting Cvc4, the forthcoming successor of Cvc3, to output interpolating proofs for arbitrary ground EUF formulas as well as in combination with linear real arithmetic. Cvc4 currently supports the theories of EUF and arithmetic, and preliminary work has been planned for a proof generating infrastructure that aims at minimizing performance overhead.

The flexibility of our proof lifting phase allows the user to make various decisions when assigning colors to a congruence graph. In previous work [6], a coloring strategy is used that aims to minimize interpolant size. We plan to explore other strategies with desired properties in mind, including logical strength, which has been explored in recent work for the propositional case [5]. In applications such as interpolation-based model checking [9], logical strength is desirable for an interpolant since a stronger interpolant may produce tighter over-approximations of the reachability relation.

We also plan to explore other applications of this framework related to automated verification, including interpolant generation procedures where additional constraints are considered, such as relative strength with respect to other interpolants. By encoding more restrictive calculi in the LFSC framework, other properties of produced interpolants may be certified by construction.

References

[1] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, Lecture Notes in Computer Science. Springer, 2007.

[2] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *Proceedings of IJCAR'10*, volume 6173 of *LNCS*, pages 384–399. Springer, 2010.

[3] R. Bruttomesso, S. Rollini, N. Sharygina, and A. Tsitovich. Flexible interpolation with local proof transformations. In *Proceedings of ICCAD'10*, pages 770–777. IEEE, 2010.

[4] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *Proceedings of TACASA'08*, Lecture Notes in Computer Science. Springer, 2008.

[5] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proceedings of VMCAI*, pages 129–145, 2010.

[6] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground interpolation for the theory of equality. In S. Kowalewski and A. Philippou, editors, *Proceedings of TACAS'09*, volume 5505 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2009.

[7] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[8] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proceedings of TACAS'06*, pages 459–473, 2006.

[9] K. McMillan. Interpolation and SAT-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *Proceedings of CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[10] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[11] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In J. Giesl, editor, *Proceedings of RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[12] A. Reynolds, L. Hadarean, C. Tinelli, Y. Ge, A. Stump, and C. Barrett. Comparing proof systems for linear real arithmetic with LFSC. In A. Gupta and D. Kroening, editors, *Proceedings of SMT'10*, 2010.

[13] A. Reynolds, C. Tinelli, and L. Hadarean. Certified interpolant generation for EUF. Technical report, Department of Computer Science, The University of Iowa, June 2011.

[14] A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[15] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.

Extended Abstract: Combining a Logical Framework with an RUP Checker for SMT Proofs

Duckki Oe and Aaron Stump
Computer Science
The University of Iowa

June 16, 2011

Abstract

We describe work in progress on a new approach, and associated tools, for checking proofs produced by SMT solvers. The approach extends our previous work on LFSC ("Logical Framework with Side Conditions"), a meta-language in which different proof systems for different SMT solvers can be declaratively specified. In this paper, we show how the LFSC proof checker can delegate the checking of propositional inferences (within a proof of an SMT formula) to a propositional proof checker `clcheck` based on Reverse Unit Propagation (RUP). This approach shows promising improvements in proof size and proof checking time for benchmark proofs produced by the `clsat` QF_IDL solver. We also discuss work in progress to replace `clcheck` with a different RUP checker we are developing called `vercheck`, whose soundness we are in the process of statically verifying.

1 Introduction

The problem of devising a standardized proof format for SMT solvers is an ongoing challenge. A number of solvers are proof-producing; for example, CVC3, veriT, and Z3 all produce proofs, in different formats [1, 2, 5]. In previous work, we advocated for the use of a flexible meta-language for proof systems called LFSC ("Logical Framework with Side Conditions"), from which efficient proof-checkers could be generated by compilation [6, 7]. Our team at The University of Iowa is currently working on a new implementation of LFSC, intended for public release.

For many SMT problems, propositional reasoning is a large if not the dominating component of proofs. Compressing the size of propositional proofs is therefore of significant interest (see, e.g., [3]). In the current paper, we describe an approach, and tools in progress, to compress the size of such proofs, by using an external propositional proof checker called `clcheck`, based on the idea of Reverse Unit Propagation.

2 SMT Proofs in LFSC

In previous work, we have advocated the use of a meta-language called LFSC ("Logical Framework with Side Conditions") for describing proof systems for SMT solvers [6, 7]. We use a meta-language to avoid imposing a single proof system on all solvers. SMT solvers support many different *logics*, and different solving algorithms naturally give rise to different schemes for representing deductions. Pragmatically, it may not be realistic to ask solver implementors to support a specific axiomatization, which may not fit well

```
(declare var type)
(declare lit type)
(declare pos (! x var lit))
(declare neg (! x var lit))

(declare clause type)
(declare cln clause)
(declare clc (! x lit (! c clause clause)))
(declare concat (! c1 clause (! c2 clause clause)))
(declare in_and_remove (! l lit (! c clause clause)))
```

Figure 1: Data Structures in LFSC for Generalized Clauses

```
(declare holds (! c clause type))
(declare R (! c1 clause (! c2 clause
           (! u1 (holds c1)
           (! u2 (holds c2)
           (! n var
            (holds (concat (in_and_remove (pos n) c1)
                     (in_and_remove (neg n) c2)))))))))

(program simplify_clause ((c clause)) clause ...)

(declare satlem (! c1 clause
                (! c2 clause
                (! c3 clause
                (! u1 (holds c1)
                (! r (^ (simplify_clause c1) c2)
                (! u2 (! x (holds c2) (holds c3))
                   (holds c3)))))))))
```

Figure 2: LFSC Rules for Resolution Proofs

with their internal data structures or algorithms. Instead, we are working towards a common meta-language, in which different proof systems may be described. This at least would establish a common meta-language for comparison of proofs and for (meta-language) proof checkers, and could facilitate later adoption of at least a common core proof system for SMT. Other researchers are working towards similar goals, and we anticipate development of a common solution in the coming year [1].

Signatures. In LFSC, proof systems are described by *signatures*. Figures 1 and 2 give part of the signature we use to produce proofs from our `clsat` QF_IDL solver. Most of the 1000-line signature is elided here, including rules for CNF conversion and arithmetic reasoning. The rules shown were developed in our previous work [6], and defer binary resolutions (constructed using the `R` proof rule) until many of them can be processed at once when a lemma is added. Resolutions are deferred by constructing a *generalized clause* (the `clause` type declared in Figure 1) using the `concat` and `in_and_remove` constructors. These constructors represent deferred operations required in order to compute the actual binary resolvent. The side-condition program `simplify_clause` (code omitted from Figure 2) executes those deferred operations in an optimized way, to construct the final resolvent of a series of binary resolutions without

constructing the intermediate resolvents.

Rules can be thought of as richly typed constructors, accepting arguments (via the `!` construct) whose types may mention earlier arguments. For example, the `R` rule has 5 inputs: `c1`, `c2`, `u1`, `u2`, and `n`. The first two are mentioned in the types of the second two. As an optimization, arguments for `c1` and `c2` may be elided in proofs built using these proof rules, since their values can be determined during proof checking from the types of the arguments for `u1` and `u2`.

The rule `satlem` uses the caret (`^`) notation to invoke the `simplify_clause` side-condition program on a clause `c1`, to compute a clause `c2` without deferred operations `concat` and `in_and_remove`. The rule specifies (via the `u2` argument) that the next subproof of a `satlem` inference should prove clause `c3` under the assumption (`x`) that the simplified clause `c2` holds. Using an assumption here allows the proof to refer to the proven (simplified) clause without repeating its proof multiple times.

Efficient Proof-Checking. Our current C++ implementation of LFSC compiles a signature into an efficient C++ proof checker optimized for that signature. Compilation includes compiling side-condition functions like `simplify_clause` to efficient C++ code. The side-condition programming language is a simply typed first-order pure functional programming language, augmented with the limited imperative feature of setting marks on LFSC variables. For details of the optimizations implemented, see our previous work [6]. There, we demonstrated significant performance gains using the deferred resolution method, and significantly better proof-checking times than for two other proof checkers (CVC3+HOL and Fx7+Trew).

3 Compressing SMT Proofs Using RUP Inferences

Our goal now is to take advantage of recent advances in proof-checking for SAT to obtain further improvements in LFSC's runtime performance on SMT proofs. In the format described above, a proof consists of CNF conversion steps and lemmas, which contain theory reasoning steps and propositional reasoning steps. In most SMT implementations, propositional inferences are performed by the internal SAT solver in the form of conflict analysis or other procedures. Reverse Unit Propagation (RUP) has been proposed by van Gelder as an efficient propositional proof format [4]. The idea behind RUP is to check $F \vdash C$ by refuting $(F \cup \neg C)$ using only unit propagation. In this case, there is a proof of the empty clause using only *unit resolution*, which is like standard binary resolution except that one of the two resolved clauses is required to be a unit clause. Unit resolution is not refutation complete in general, but it has been shown to be complete when C is a conflict clause generated according to standard conflict-analysis algorithms [4]. In a proof based on RUP, only the clause C is recorded, and the sequence of such resolutions can be calculated from that clause. Thus, a long resolution proof of a RUP inference can be compressed to the concluded clause. It can happen, however, that writing down the clause itself takes more space in the proof than a short resolution proof would (a point worth exploring further in seeking smaller proofs).

3.1 Delegating Propositional Proofs

In principle, one could implement an RUP checker in the LFSC side condition language. This would require pure functional data structures for unit propagation, which would largely negate the benefits of the RUP proof format, which relies on the efficient unit propagation of modern SAT solvers. So instead, we delegate RUP proof checking to an external RUP checker; see Figure 3 for our work flow. The LFSC rules used to delegate the checking of propositional inferences from LFSC to the external RUP checker are presented in Figure 4. The external RUP checker confirms that certain *check clauses* follow by purely propositional reasoning from certain *assert clauses*. Assert clauses include the propositional clauses derived by CNF

Figure 3: Workflow of New Proof System

```
(declare lemma (! c1 clause
               (! c2 clause
               (! z unit
               (! u1 (holds c1)
               (! r (^ (print_assert c1) z)
               (! u2 (! x (holds c1) (holds c2))
                 (holds c2))))))))
(declare check (! c2 clause
               (! z unit
               (! c1 clause
               (! r (^ (print_check c1) z)
               (! u (! x (holds c1) (holds c2))
                 (holds c2)))))))
```

Figure 4: LFSC Proof Rules for Delegating Checking of Propositional Inferences

conversion from the original input formula; and also the boolean skeletons of all theory lemmas. In between some of these asserts, a proof can request that the proof checker confirm that a check clause follows by RUP from the (propositional) assert clauses, as well as previous check clauses.

The proof rule `lemma` (Figure 4) is used to assert a clause to the external RUP checker. It requires a proof (`u1`) that the asserted clause actually holds. The `check` rule then delegates checking that a clause `c1` follows from earlier clauses, including both asserted and already checked clauses, by purely propositional reasoning. Note that it does not require a proof of the clause `c1` which is being checked, since checking that this clause holds in the current logical context is being delegated to the external RUP checker. Both rules use side-condition functions (`print_assert` and `print_check`) to print out the clauses in question as either assert clauses or check clauses. Additionally, before printing any assert or check clauses, proofs in this signature must print an initial header, giving the number of propositional variables used.

Implementing Delegation. To support delegating propositional proofs, the LFSC compiler was modified to support printing of numbers, string literals, LFSC variables (used directly to encode propositional variables) from side-condition functions. To enable a very straightforward implementation, variables are printed out as their hexadecimal memory addresses. A simple post-processing phase, currently implemented by a short OCAML program, is used to map hexadecimal addresses to numbers starting with 1.

3.2 Propositional Proof Format

This section describes the proof format that the LFSC checker produces to delegate propositional reasoning to a RUP checker. It can be best explained by an example. Figure 5 shows an example propositional proof. We see the initial header, starting with `p`, specifying the maximum number of different variables that can appear in the file (here this is a loose bound). Then come assert clauses, which begin with `a` and are terminated with 0; and check clauses, which begin with `c` and are similarly terminated. The format of clauses is similar to the DIMACS format for CNF SAT problems. The example has four assert clauses and two check

```
p 2
a 1 2 0
a 1 -2 0
c 1 0
a -1 2 0
a -1 -2 0
c 0
```

Figure 5: A Simple Propositional Proof

clauses intermixed. Obviously, the set of those assert clauses is refutable. It is not refutable, however, by unit resolution, because the assert clauses are all binary. Thus, the first check clause is necessary. The first two assert clauses and the negation of the first check clause are refutable by resolving the negated check clause with the first two assert clauses and then resolving their resolvents. Now, the first check clause is verified. The last check clause, which is empty, simply asks if the entire clauses above it are refutable only using unit propagation without any more assumptions, which is true in this example.

3.3 The `clcheck` RUP Proof Checker

We implemented a RUP proof checker, called `clcheck` that supports the proof format explained above. Other proof checkers like `checker3` combined with `rupToRes`, which is used in the SAT competition, could be used with a proper translation. To the best of our knowledge, they do not support intermixed assertions and checks. Thus, assert clauses and check clauses have to be split into separate files. In SMT solvers, theory inferences and propositional inferences are naturally intermixed, and those theory inferences are asserted as clauses to be used in propositional inferences later on. We believe that intermixing assertions and checks in proofs allows concurrent processing of theory lemmas on the LFSC checker and propositional lemmas on the RUP checker, which can lead to more efficient proof checking on a modern multicore system. In our settings, the output of LFSC is directly streamed to `clcheck` using Unix pipes. So, while `clcheck` is checking a RUP inference, LFSC can check the next theory lemma at the same time.

A RUP inference $F \vdash C$ is verified as follows. First, for each literal in C, add a unit clause with the negation of that literal to the clause database. Now, the clause database has $F \cup \neg C$. Second, propagate all unit clauses in the database. If it leads to a conflicting clause, C is proved; otherwise, the inference is invalid. That can be also justified in terms of unit-resolution proof. Because every assignment is caused by a unit clause, which is the antecedent clause, the empty clause can be derived by applying unit resolution on each literal of C and that literal's antecedent clause. Finally, remove those unit clauses added in the first step and cancel all assignments. One can work more cleverly by avoiding redundancy. Instead of canceling all assignments, just cancel assignment only caused by $\neg C$ and, after C is verified, incrementally propagate the new unit clauses in $F \cup C$, which will be the new F for the next check. This approach is implemented in `clcheck`, which is written in C++ and which uses standard efficient data structures (in particular, watch lists for literals) for efficient unit propagation.

4 Preliminary Results

Our SMT solver `clsat` has been modified to generate proofs in the new format in addition to the original format. We have chosen 39 QF_IDL benchmarks that `clsat` solved in 900 seconds in the SMT competition

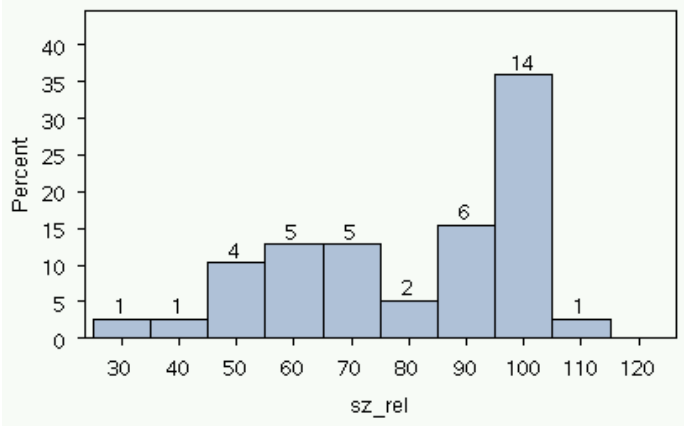


Figure 6: Distribution of Relative Proof Sizes

2009. Because `clsat` does not support the SMT-LIB 2.0 file format, they are in the SMT-LIB 1.2 format. Table 1 (page 10) shows the results. The test machine had Intel Xeon X5650 2.67GHz CPU and 12GB of memory. Times (in seconds) are measured for solving and checking combined so that we can see how the new format improves the whole work flow, not just proof overhead. That measurement includes I/O overhead between the solver and checkers. The proof formats in comparison have different syntactic characteristics that may affect proof sizes. So, we wanted to compare the amount of information as the smallest number of bits needed to store the proof. That means you cannot modify the syntax to achieve a smaller proof. Instead of developing such a proof syntax, we used gzip-compression to approximate the amount of information in a proof. The table shows the gzip-compressed sizes (in bytes) of proofs. The uncompressed proof sizes did not change the conclusion. However, we believe the compressed sizes are more meaningful as data (especially, when we see the relative sizes). Note that one benchmark, `diamonds.18.5.i.a.u` did generate proofs, but failed to check in both formats due to memory overflow (uncompressed proof sizes reach 2GB in size).

Figure 6 shows the distribution of the relative sizes (ratios) of the new proofs. The horizontal axis is the relative size in percent (the size of new proof over the size of old proof times 100). And the vertical axis is the percentage of instances in each range. The number on each bar shows the number of instances in the range. For 14 benchmarks (accounting 35%), the new proof has almost the same size as the old counterpart (in the range of 95%-105%). However, there are a variety of compression ratios and mostly the new proofs are smaller or similar in size. One new proof is as small as 30% of the old counterpart. At the other extreme, there is one case that the new proof is 12% bigger than the old one. Figure 7 shows the correlation between the relative proof sizes and the relative proof checking times. For time comparison, we considered 11 benchmarks that take more than 1 second to solve and check on any system. Because small checking times have relatively big measurement errors, their relative times are not reliable. In the figure, each vertex represents a benchmark where its horizontal coordinate is the relative proof size and its vertical coordinate is the relative checking time. The figure shows a rough linear relationship between relative proof size and relative checking time along the regression line. The R^2 value of the regression is 0.7675. That can be summarized as the more a proof compresses in the new format, the more checking speeds up.

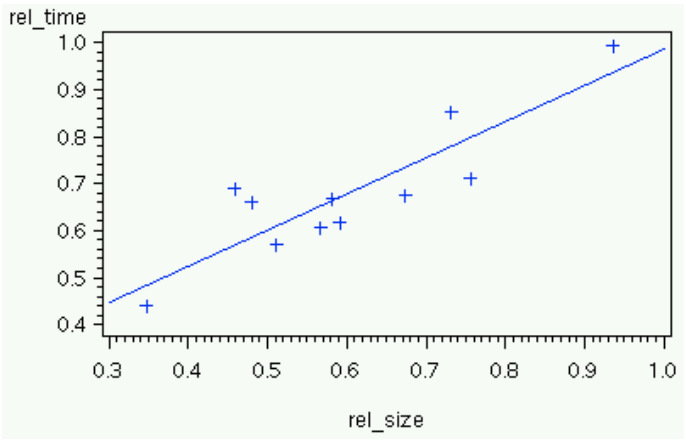


Figure 7: Correlation between Relative Checking Times and Relative Proof Sizes

5 Conclusion and Future Work

We have presented an approach for integrating an RUP checker for propositional proofs with the LFSC proof meta-language, based on delegation to an RUP checker. We have seen promising improvements over pure LFSC proof-checking, in both proof size and proof-checking time.

Improved LFSC implementation. As mentioned in the introduction, our team at The University of Iowa is implementing a new version of the LFSC checker, which we anticipate amplifying the benefits we have observed in our preliminary empirical results. Profiling the current version of LFSC on these benchmarks shows that at least in some cases, running the side-condition code (`simplify_clause` referenced in Figure 2) needed to check propositional resolution proofs is not taking a large part of the time for proof checking. Overhead in other parts of the proof checker outweighs this. Our new implementation is designed to take advantage of optimizations we described in earlier work on fast proof-checking for LF, the Edinburgh Logical Framework on which LFSC is based [8]. These optimizations are missing in the current LFSC checker. We anticipate they will lower the overhead of the rest of the proof-checking algorithm, and thus amplify the benefits of delegating propositional proofs to the RUP checker.

From `clcheck` to `vercheck`. In a separate line of research, the authors are implementing a statically verified modern SAT solver called `versat`. The specification we are establishing is that if the solver reports a set of input clauses unsatisfiable, then there exists a resolution proof of the empty clause from those input clauses. This resolution proof is not constructed at runtime. Rather, we prove that it is guaranteed to exist whenever the solver reports unsatisfiable. The `versat` solver uses standard efficient low-level data structures, based on mutable arrays, and implements standard modern SAT-solving techniques like conflict-driven clause learning, non-chronological backtracking, and watched literals.

Using the unit-propagation code in `versat`, we are implementing a trusted RUP checker called `vercheck`. The specification we are proving for this tool is that if it confirms an RUP proof of the kind described above, then the check clauses really do follow from the earlier check and assert clauses. Using `vercheck` will help mitigate the expansion of the trusted computing base incurred by delegating from LFSC. Our current LFSC C++ checker is around 6kloc C++. The new version currently in progress will be around 4.5kloc OCAML when complete. The `clcheck` solver is just under 1kloc C++. The trusted specification for `vercheck` is just 355 lines of GURU code (GURU is the research programming language we are using for implementation

and static verification of `versat` and `vercheck`). Also, the old signature for QF_IDL proofs from `clsat` is 870 lines of LFSC, while the new one is 795 lines. So using `vercheck`, the new approach based on delegation will only increase the number of lines of trusted code by 280 lines total, which seems a worthwhile price to pay for decreased proof size and improved proof-checking time.

References

[1] T. Bouton, D. Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE)*, pages 151–156, 2009.

[2] L. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In B. Konev, R. Schmidt, and S. Schulz, editors, *7th International Workshop on the Implementation of Logics (IWIL)*, 2008.

[3] P. Fontaine, S. Merz, and B. Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction (CADE)*, 2011. to appear.

[4] Allen Van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.

[5] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electr. Notes Theor. Comput. Sci.*, 144(2):43–51, 2006.

[6] D. Oe, A. Reynolds, and A. Stump. Fast and Flexible Proof Checking for SMT. In B. Dutertre and O. Strichman, editors, *Workshop on Satisfiability Modulo Theories (SMT)*, 2009.

[7] A. Stump and D. Oe. Towards an SMT Proof Format. In C. Barrett and L. de Moura, editors, *International Workshop on Satisfiability Modulo Theories*, 2008.

[8] M. Zeller, A. Stump, and M. Deters. Signature Compilation for the Edinburgh Logical Framework. In C. Schürmann, editor, *Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, 2007.

	solve+check time		compressed proof size	
benchmarks	old	new	old	new
BubbleSort_safe_blmc010	0.48	0.48	224122	220316
BubbleSort_safe_blmc016	0.94	0.94	406296	401250
CELAR7_SUB1	1.39	1.38	49754	46620
ckt_PROP0_tf_15	0.21	0.11	98277	59202
ckt_PROP1_tf_25	0.2	0.16	123252	99474
ckt_PROP2_tf_10	0.02	0.02	13762	13057
ckt_PROP5_tf_25	0.9	0.54	385940	252867
diamonds.18.5.i.a.u	Error	Error	112879127	126579755
DTP_k2_n35_c245_s19	1.32	0.89	363027	186938
DTP_k2_n35_c245_s5	2.51	1.66	709885	340784
DTP_k2_n35_c245_s6	3.36	2.32	924559	423192
FISCHER11-6-ninc	0.76	0.47	352386	241472
FISCHER13-1-ninc	0.03	0.03	25785	25642
FISCHER14-9-ninc	56.78	48.31	9851980	7214785
FISCHER6-2-ninc	0.03	0.03	24634	24146
FISCHER8-1-ninc	0.02	0.02	16079	15929
inf-bakery-invalid-2	0.01	0.01	7212	6690
int_incompleteness1	0	0	565	525
jobshop6-2-3-3-4-4-11	0.01	0.01	3923	3672
lpsat-goal-12	5.91	3.18	2124140	907239
lpsat-goal-15	19.48	11.09	5613533	2867306
lpsat-goal-2	0.05	0.05	43171	41749
lpsat-goal-8	0.92	0.6	486810	271645
plan-18.cvc	7.26	4.41	1570043	890322
plan-22.cvc	0.9	0.57	286853	190437
plan-30.cvc	5.71	2.51	1571073	545799
plan-33.cvc	23.6	14.55	4485851	2650662
plan-35.cvc	68.74	46.02	10937058	6369656
plan-9.cvc	0.06	0.06	37021	33638
PO2-2-PO2	0.01	0.01	10918	10691
PO2-6-PO2	0.04	0.05	40422	38543
PO4-10-PO4	2.64	1.78	1286666	867640
PO4-4-PO4	0.32	0.32	237111	232804
PO4-8-PO4	1.46	1.04	795994	602860
SelectionSort_safe_bgmc005	0.06	0.07	36991	36033
SelectionSort_safe_bgmc009	0.13	0.13	74919	72530
SortingNetwork4_safe_bgmc002	0	0	1833	1812
SortingNetwork8_safe_bgmc006	0.05	0.05	25317	25167
SortingNetwork8_safe_blmc006	0.18	0.17	75659	75482

* Measurement units: times in seconds, sizes in bytes

Table 1: Results of Old and New Proof Systems

Selective SMT Encoding for Hardware Model Checking*

Hyondeuk Kim[1], Fabio Somenzi[2], and HoonSang Jin[1]

[1] Cadence Design Systems
[2] University of Colorado at Boulder
{hkim}@cadence.com, {Fabio}@Colorado.EDU
{hsjin}@cadence.com

Abstract. In this paper, we study the translation of a hardware model language into a verification condition to be checked by SMT solvers. In today's hardware designs, bit-level and word-level operations are often tightly intermingled. On some designs, a bit-level model checker may perform better than a word-level model checker or vice versa. Depending on the characteristics of the design, we selectively choose an encoding method (either bit-level or word-level) to improve the efficiency of hardware model checking. We present a model analysis method for the encoding selection and evaluate the method on a set of hardware verification problems.

1 Introduction

In recent years, model checkers have increasingly used propositional SAT solvers as decision procedures. In spite of their effectiveness, SAT solvers only consider the model at the bit level. Encoding word-level operations into bits often increases the size of the formula and loses information such as high-level data structures. Recently, word-level model checking [2] has received growing attention. In particular, Satisfiability Modulo Theories (SMT) solvers have been effectively applied to software verification with predicate abstraction [14] and bounded model checking [10]. Only to a lesser extent, they have been applied to hardware verification. The most natural SMT encodings for hardware description are bit-vector (BV) and linear integer arithmetic (LIA) encodings. LIA encoding for RTL constructs is presented in [6], where control variables are encoded as Boolean variables and data path variables as integer variables.

Our work is motivated by the results shown in Fig. 1. We have encoded each pair of Verilog design and property into SMT for bounded model checking (BMC). In particular, we used BV and LIA encodings for each design. The details of these encoding methods will be discussed in Sect. 4. The Verilog designs we used are from VIS Verilog benchmarks [21], Opencores [16] and Altera design examples [19]. We compared BV solvers (Boolector-1.4 [3], Z3-2.8 [24], Beaver [1] with Precosat-456r2 [18]) and LIA solvers (MathSAT-4.3 [15], Yices-1.0.28, Z3-2.8) for the encodings. These solvers are the ones that performed best on our BMC problems. In the experiment, the timeout was set to 1000 seconds. Figure 1 shows the comparison of average CPU times of the solvers for the two encodings. The points above the diagonal are wins for the BV solvers, and the ones below are wins for the LIA solvers. As the scatterplot shows, few models work well with both encodings. We introduce a model analysis method that considers each bit-vector operation in the design and selects the encoding based on the analysis. In addition, we present several enhancements to SMT encoding for hardware designs. Our experiments show that our approach selects the right encoding for the hardware design and improves the efficiency of bounded model checking and equivalence checking.

The rest of this paper is organized as follows. Section 2 reviews logics for hardware modeling. Section 3 describes the translation to BV logic. Section 4 describes the translation to LIA logic. Section 5 presents a model analysis method and Sect. 6 presents experiments. After a survey of related work in Sect. 7, conclusions are offered in Sect. 8.

* This work was supported by SRC contract 2009-TJ-1859.

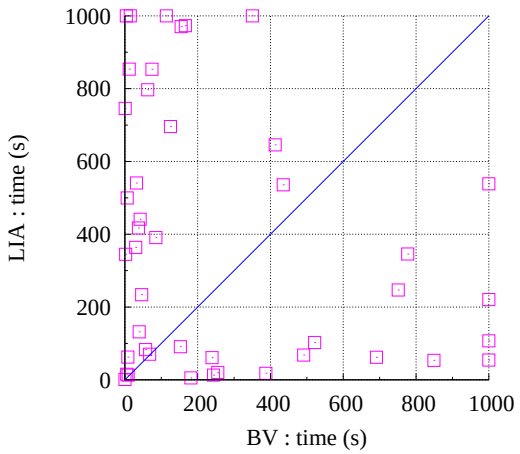


Fig. 1. BV vs. LIA

2 Logics for Hardware Verification

In this section, we recall the definitions of the logics BV and LIA which we use to encode hardware.

2.1 BV Logic

Let $V_B(n)$, $n \in \mathbb{Z}^+$, be the set of BV variables whose domains are n-bit vectors. We assume that $i \neq j \rightarrow V_B(i) \cap V_B(j) = \emptyset$. Let $T_B(n)$ be the set of BV terms whose values are n-bit vectors. The formulae in BV logic are inductively defined as follows.

- If $c \in \mathbb{N}$ and $c \leq 2^n - 1$, then $c[n] \in T_B(n)$.
- If $x \in V_B(n)$, then $x[n] \in T_B(n)$.
- If $x \in V_B(n)$ and $0 \leq j \leq i < n$, then $x[i : j] \in T_B(i - j + 1)$, and if $t[n] \in T_B(n)$, then $\sim t[n] \in T_B(n)$. ($\sim$ is the bit-wise negation operator.)
- If $t_1[n], t_2[n] \in T_B(n)$, and $\diamond$ is an arithmetic or bit-wise operator in $\{+, -, \cdot, /, \%, \&, |\}$, then $t_1[n] \diamond t_2[n] \in T_B(n)$.
- If $t_1[i] \in T_B(i)$ and $t_2[j] \in T_B(j)$, then $concat(t_1[i], t_2[j]) \in T_B(i + j)$.
- A propositional variable $a \in V_P$ is a formula.
- If $t_1[n], t_2[n] \in T_B(n)$, and $\diamond$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1[n] \diamond t_2[n]$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and *ite*(f_1, f_2, f_3) are formulae, and if $t_1[n], t_2[n] \in T_B(n)$ and f is a formula, then *tite* $(f, t_1[n], t_2[n]) \in T_B(n)$.

Further formulae can be defined as abbreviations. For instance, $x[n] \ll k$, a left shift of $x[n]$ by a constant k, is defined as $concat(\ x[n - k - 1 : 0], 0[k])$. An *atomic formula* is one of the form $t_1[n] \diamond t_2[n]$, where $\diamond$ is a relational operator. The semantics are defined in the usual way; in particular, arithmetic is modular, $x[i : j]$ is the subfield of $x[n]$ comprising the bits from i to j included, $concat(t_1[i], t_2[j])$ concatenates $t_1[i]$ and $t_2[j]$, and *ite*(f_1, f_2, f_3) is equivalent to $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$. In addition, the *term if-then-else* (*tite*) operator is defined by the equivalence, for all formulae f and g and for all terms $t_1[n]$ and $t_2[n]$, of $f($*tite*$(g, t_1[n], t_2[n]))$ and *ite*$(g, f(t_1[n]), f(t_2[n]))$. For $A, B, C, D, E \in V_B(2)$, (1) is a BV formula.

$$(C[2] = A[2] \ \& \ B[2]) \wedge (D[2] = C[2] + E[2]) \ . \tag{1}$$

2.2 LIA Logic

Let V_Z be a set of integer-valued variables. The formulae in LIA logic are inductively defined as follows.

- An integer number $c \in \mathbb{Z}$ is a (constant) LIA term, and a variable $x \in V_Z$ is an LIA term.
- A variable $x \in V_Z$ is an LIA term, and the product $c \cdot x$ of an integer number $c \in \mathbb{Z}$ and a variable $x \in V_Z$ is an LIA term.
- If t_1 and t_2 are LIA terms, so are $t_1 + t_2$ and $t_1 - t_2$.
- A propositional variable $a \in V_P$ is a formula.
- If t_1 and t_2 are LIA terms, and $\diamond$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1 \diamond t_2$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and $\mathit{ite}(f_1, f_2, f_3)$ are formulae.
- If t_1 and t_2 are LIA terms, and f is a formula, then $\mathit{tite}(f, t_1, t_2)$ is an LIA term.

For $A, B, C, D, E \in V_Z$, (2) is an LIA formula:

$$(C = A - B) \wedge (D = C + E) \ . \tag{2}$$

3 From Hardware Description to BV

In this section, we outline the conversion from hardware description to BV formula. Hardware is assumed to be described in a subset of the Verilog hardware description language (HDL) [20] suitable for the modeling of synchronous hardware. The subset supports the mixture of blocking and non-blocking assignments in the procedural blocks, and allows non-deterministic interleaving of procedural blocks. We impose restrictions to the description to ensure that the evaluation of each procedural block is not affected by the interleaving of the assignments in different procedural blocks. The restrictions are compatible with common design guidelines used in the industry (e.g., blocking assignments for combinational logic and non-blocking assignments for memory elements) and allow us to produce concise verification conditions. Although the subset includes essential features of Verilog, it does not support delays, strengths, and other features that are not needed for RTL verification of synchronous designs.

We represent a hardware description as a Concurrent Control Flow Graph (CCFG) [13] in Static Single Assignment (SSA) form [7]. With the CCFG, we generate a set of constraints in BV logic for blocking and non-blocking assignments in each procedural block. If there are multiple assignments to the same target in different procedural blocks, we generate an additional conflict arbitration constraint.

In Fig. 2, the two procedural blocks at the top are converted into the SSA form at the bottom. In each procedural block, we generate the BV formula for each target variable. Suppose $u, v \in V_P$ and $w, x, y, z \in V_B(4)$. For the target z, we generate the BV formula

$$\mathit{ite}(v_0, \bar{z}_1[4] = 1[4] \wedge \bar{z}_2[4] = \bar{z}_1[4], \bar{z}_2[4] = z_1[4]) \tag{3}$$

in the first procedural block, and

$$w_1[4] = y_0[4] \wedge \mathit{ite}(u_0, w_2[4] = x_0[4] \wedge w_3[4] = w_2[4], w_3[4] = w_1[4]) \wedge \bar{z}_3[4] = w_3[4] \tag{4}$$

in the second procedural block. Then, we introduce z' for z and generate a conflict arbitration constraint $z'[4] = \bar{z}_2[4] \vee z'[4] = \bar{z}_3[4]$. This formula conjoined with (3) and (4) is the transition relation for the description, where z_1 and z' are the current and next state variables for z.

```
initial #0 #0 z = 0;                    always @(posedge clk) begin
                                            w = y;
always @(posedge clk)                       if (u) w = x;
    if (v) z ⇐ 1;                           z ⇐ w;
                                        end

initial #0 #0 z1 = 0;                   always @(posedge clk) begin
                                            w1 = y0;
always @(posedge clk) begin                 if (u0) w2 = x0;
    if (v0) z̄1 = 1;                         w3 = φ(w2, w1);
    z̄2 = φ(z̄1, z1);                         z̄3 = w3;
end                                     end
```

Fig. 2. Conversion from HDL to SSA form

4 From BV to LIA

In Sect. 3, we showed how a hardware description is converted into a BV formula. In this section, we discuss the translation from BV encoding to LIA encoding. SMT encoding for hardware design (RTL Verilog) was first presented in [6] where both BV and LIA encodings for combinational circuits were introduced. We present selective value enumeration and efficient term-ITE introduction for BV arithmetic terms as enhancements to the basic LIA encoding methods presented in [6].

The basic encoding methods often introduces the product $k \cdot X$ where k is a constant and X is a variable. The coefficient k may be large, and large coefficients often degrade the performance of LIA solvers because they often require many pivots in the simplex-based ILP (Integer Linear Programming) algorithm [9]. We tackle the problem with selective enumeration. If the range of X is small enough to express it with few term-ITEs, term-ITEs replace the multiplication. For instance, if $0 \le X \le 1$ in $Z = 2^j \cdot X + Y$, then the new encoding with a term-ITE is $Z = \mathit{tite}(X = 1, 2^j + Y, Y)$.

For arithmetic terms, two types of encoding are introduced in [6]: One with a fresh constant and the other with a term-ITE. The LIA encodings for an equality $z[n] = \sum_{i=1}^{m} x_i[n]$ with a general arithmetic term can be

$$(Z = (\textstyle\sum_{i=1}^{m} X_i) - 2^n \cdot \alpha) \wedge (0 \le \alpha \le m-1) \ , \tag{5}$$

and

$$\begin{aligned} t_m = \textstyle\sum_{i=1}^{m} X_i \wedge t_{m-1} = \mathit{tite}(t_m \ge (m-1) \cdot 2^n, t_m - (m-1) \cdot 2^n, t_m) \wedge \\ t_{m-2} = \mathit{tite}(t_{m-1} \ge (m-2) \cdot 2^n, t_{m-1} - (m-2) \cdot 2^n, t_{m-1}) \wedge \cdots \wedge \\ t_2 = \mathit{tite}(t_3 \ge 2^{n+1}, t_3 - 2^{n+1}, t_3) \wedge Z = \mathit{tite}(t_2 \ge 2^n, t_2 - 2^n, t_2) \ . \end{aligned} \tag{6}$$

We prefer (6), which introduces term-ITEs, to (5), because (5) often introduces a large coefficient for the fresh variable α. For multiplication, we use the encoding

$$\begin{aligned} t_{N_t-1} = \mathit{tite}(k \cdot X \ge 2^{N_t-1} \cdot 2^n, k \cdot X - 2^{N_t-1} \cdot 2^n, k \cdot X) \wedge \\ t_{N_t-2} = \mathit{tite}(t_{N_t-1} \ge 2^{N_t-2} \cdot 2^n, t_{N_t-1} - 2^{N_t-2} \cdot 2^n, t_{N_t-1}) \wedge \cdots \wedge \\ t_1 = \mathit{tite}(t_2 \ge 2^{n+1}, t_2 - 2^{n+1}, t_2) \wedge Z = \mathit{tite}(t_1 \ge 2^n, t_1 - 2^n, t_1) \ . \end{aligned} \tag{7}$$

The conditions of the term-ITEs in (7) enumerate the different overflow cases. If a condition is true, the value of $k \cdot X$ overflows; hence, the true branch of the term-ITE subtracts a power of 2 from the value of $k \cdot X$ to

satisfy the condition $0 \leq k \cdot X < 2^n$. The number of term-ITEs N_t required for encoding a multiplication $k[n] \cdot x[n]$ in LIA is given by $N_t = \lceil log_2(k) \rceil$.

For a BV equality $z[n] = k_1[n] \cdot x[n] + k_2[n] \cdot y[n]$, the number of term-ITEs can be computed by computing N_t for each multiplication and the addition, or by computing N_t for the whole term. Using the first method $N_t = \lceil log_2(k_1) \rceil + \lceil log_2(k_2) \rceil + 1$, and $N_t = \lceil log_2(k_1 + k_2) \rceil$ with the second method. Since

$$\lceil log_2(k_1 + k_2) \rceil \leq \lceil log_2(k_1) \rceil + \lceil \log_2(k_2) \rceil + 1 \ ,$$

we use the second method. The number of term-ITEs in (6) can be reduced from $m - 1$ to $\lceil log_2(m) \rceil$.

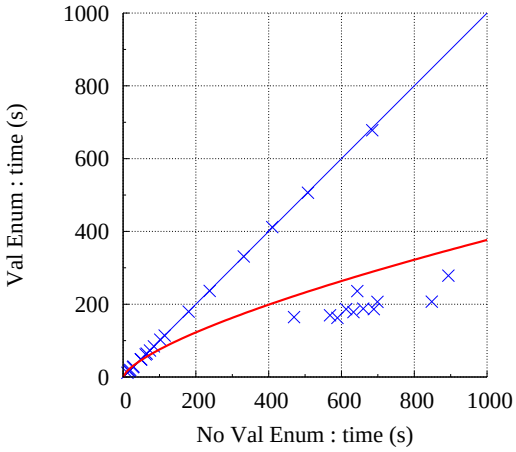


Fig. 3. VAL ENUM vs. No Val Enum

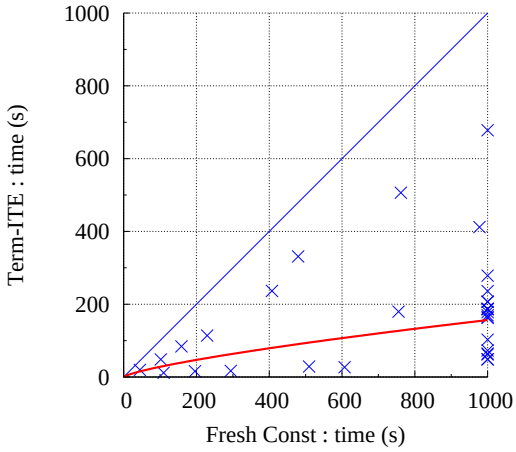


Fig. 4. TERM-ITE vs. Fresh Variable

The results of Yices (LIA) on the hardware verification problems in Fig. 1 with and without the enhanced encodings are shown in Fig. 3 and Fig. 4. In the experiments, the timeout was set to 1000 seconds. Figure 3 compares the encodings with and without value enumeration. Figure 4 compares the encodings with and without term-ITE introduction. Points below the diagonal represent wins for the enhanced encoding. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 3 shows that the encoding with the value enumeration outperforms the one without. Figure 4 shows that the encoding with the term-ITE introduction often outperforms the one without significantly.

5 Model Analysis

Figure 1 shows that choosing the proper encoding is important. Given a hardware design, we analyze the model to choose between BV and LIA (plus, possibly, bit blasting). If the model contains many bit-wise and bit-select operators, or it uses only a narrow data path, then the BV encoding is more likely to be suitable. On the other hand, if the model contains a large number of arithmetic and relational operators with a wide data path, the LIA encoding may be preferable. In practice, we often encounter designs with a mixture of bit-wise, bit-select, and arithmetic operators. On those problems, it is hard to apply LIA solvers even though they contain a large number of arithmetic operators with wide data paths. On the other hand, there is still a chance to apply LIA solver if certain conditions are met. We discuss these conditions in the following.

5.1 Analysis of Bit-Select Operations

The bit-select operators in hardware designs often produce LIA encodings that are hard for SMT solvers. Each bit-select operator generates three fresh variables possibly with large coefficients. If there are multiple bit-select operations applied to one bit-vector, there is no benefit in encoding them in LIA. In [6], the author showed degradation of performance in an LIA solver as the number of slices of a bit-vector grows. When a slice includes either the MSB (most significant bit) or the LSB (least significant bit) of a bit-vector, only two fresh variables are needed. However, the LIA encoding may not be efficient depending on the location of the slice. According to our experiments, if the bit-vector is decomposed only into two and the slicing bit is close to the MSB, then LIA encoding can be still effective.

5.2 Analysis of Bit-Wise Operation

Bit-wise operators make LIA encoding much harder compared to the encodings for other BV operators. There is not much choice but to bit-blast the bit-vectors in the bit-wise operations. On the other hand, some designs contain a large number of arithmetic operations with wide data paths and a small number of bit-wise operations. In those designs, the combinational (BV ∪ LIA) encoding can be used to encode the bit-wise operations with BV encoding, and still maintain the arithmetic operations with LIA encoding. Unfortunately, SMT solvers for BV ∪ LIA encoding do not perform well compared to other solvers (BV or LIA) according to our experiments. Instead of using the BV ∪ LIA encoding, we apply bit blasting for the bit-wise operations and use LIA encoding for the arithmetic operations. The experimental result in Fig. 5 compares LIA encodings with and without bit blasting for the Palu and the retherRTF designs [21] and shows that LIA encoding with bit blasting gives much better performance compared to pure LIA encoding.

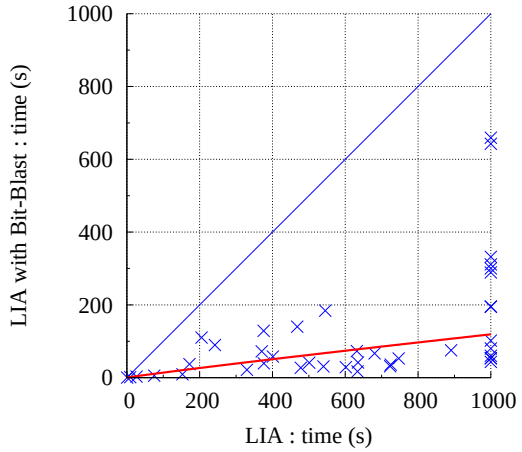


Fig. 5. LIA WITH BIT-BLAST vs. LIA

5.3 Scoring System

The model analysis method decides the encoding method based on a scoring system. Let $Score_B$ be the score for BV encoding and $Score_L$ be the score for LIA encoding. Let w_{ar}, w_{re}, w_{bw}, and w_{bs} be the weights

for the arithmetic, relational, bit-wise, and bit-select operators, with $w_{bw} > w_{bs} > w_{ar} > w_{re}$. We give a larger value to w_{bw} and w_{bs} because the numbers of bit-wise and bit-select operators have a stronger impact on the effectiveness of the LIA encoding than the numbers of arithmetic and relational operators have on the effectiveness of the BV encoding. The score is computed for each relational expression e in the transition system based on (8) and (9), in which $n(bw)$ is the number of bit-wise operators, $n(bs)$ is the number of bit-select operators, $n(ar)$ is the number of arithmetic operators, $n(re)$ is the number of relational operators, and $nbits(e)$ is the number of bits in e.

$$Score_B = \sum_{i=1}^{n(bw)} nbits(e_i) \times w_{bw} + \sum_{i=1}^{n(bs)} nbits(e_i) \times w_{bs}, \quad (8)$$

$$Score_L = \sum_{i=1}^{n(ar)} nbits(e_i) \times w_{ar} + \sum_{i=1}^{n(re)} nbits(e_i) \times w_{re} \ . \quad (9)$$

A bit-select operator that decomposes the data path into only two and whose slicing bit is close to the MSB is considered a weak bit-select and is not counted in $n(bs)$. Non-linear operations are linearized as in [4] and counted in $n(ar)$. Each score represents the amount of bit-vector operations that are suitable for encoding in either BV or LIA.

Given the scores $Score_B$ and $Score_L$ and their thresholds th_B and th_L, we compare the score with its threshold and decide the encoding method. If $Score_L > th_L$ and $Score_B < th_B$, then we select LIA encoding, otherwise we select BV encoding. When encoding in LIA, the bit-vectors in the bit-wise operations are bit-blasted, and the bit-vectors only in the relational operators are also bit-blasted. The selective bit blasting in LIA encoding often improves the efficiency of SMT solvers.

6 Experimental Results

We have implemented a translator called *Vl2smt* that uses Icarus Verilog [11] as front end, accepts a Verilog design as input, and generates an SMT formula for the verification condition of the design. The translator chooses the encoding method for a given design between BV and LIA with bit blasting as discussed in Sect. 5.3. We used the set of designs of Fig. 1 as training set for the predictor. For each design, three BMC problems with different bounds are used for the encoding prediction. All results are for the solvers listed as in Sect. 1 with a timeout of 1000 seconds. Figure 6 shows the comparison of average CPU times of BV solvers (Boolector-1.4, Z3-2.8, Beaver with Precosat-456r2) and LIA solvers (MathSAT-4.3, Yices-1.0.28, Z3-2.8) with the designs classified according to the predicted encoding method. The symbol $\circ$ is used for designs with BV encoding prediction, and the symbol $\times$ is used for the design with LIA encoding prediction. The scatterplot shows that most designs for which BV encoding was predicted to work better actually end up above the diagonal, while most designs for which LIA encoding was predicted to work better actually end up under the diagonal. This result shows that *Vl2smt* predicts the right encoding for most of the problems in the training set.

A set of hardware model checking problems from VIS Verilog benchmarks [21], Opencores [16] and Altera design examples [19] disjoint from the training set was used for evaluation of *Vl2smt*. The result of the evaluation in Fig. 7 shows that *Vl2smt* predicts the right encoding method for each of these model checking problem.

Table 1 shows the average number of bits, the numbers of addition, multiplication, relational (re1: with a constant, re2: with a variable), bit-wise, and bit-select and ignored bit-select operations, the scores, and the encoding predictions for the models in the training (T-Model) and evaluation (E-Model) sets. For (8) and (9), we use the weights $w_{bw} = bits(e)$, $w_{bs} = bits(e)$, $w_{add} = 1$, w_{mul}, $w_{re1} = 0.1$, $w_{re2} = 0.4$, and the thresholds $th_B = 500$, $th_L = Score_L/13$ that we got from T-Model. While computing $Score_B$, the slicing bit of a bit-select operator (ibs) that is close to either LSB or MSB ($bits(e) < 5$) is ignored.

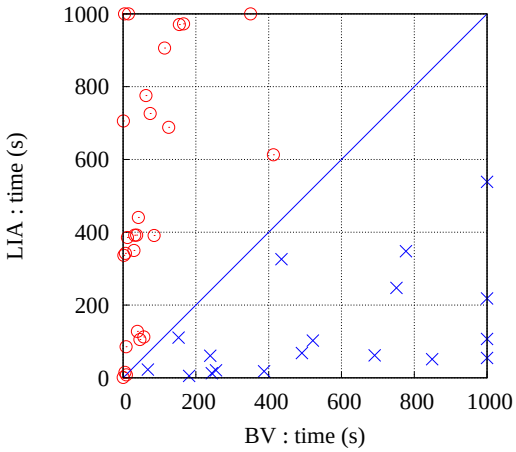


Fig. 6. BV vs. LIA for T-Model set

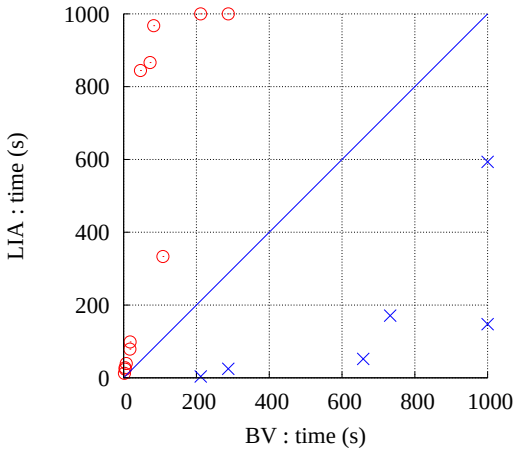


Fig. 7. BV vs. LIA for E-Model set

T-Model	bit	add	mult	re1	re2	bw	bs	ibs	$Score_L$	$Score_B$	**Enc**
Am2910	11	1	0	21	11	0	0	0	75.6	0	BV
Bakery	5	0	0	371	23	0	0	0	243.8	0	BV
Blackjack	5	4	0	75	14	0	0	0	83.9	0	BV
Cube	2	0	0	0	52	0	0	0	89.6	0	BV
FPMult	10	26	14	0	54	20	11	13	1218.8	10229	BV
Palu	15	9	3	9	19	9	0	4	692.4	9216	BV
RetherRTF	5	0	0	0	8	0	0	0	16	0	BV
Swap	3	0	0	28	11	0	0	0	22.4	0	BV
Miim	3	10	9	0	18	4	3	0	281.2	1540	BV
Timeout	51	1	0	0	20	0	0	0	504	0	LIA
cf_fir	8	47	43	0	69	0	0	12	2347.2	0	LIA
FIFOs	55	0	0	75	30	0	0	0	1046.6	0	LIA
FIR	17	20	20	0	4	0	0	9	1715.2	0	LIA
DSP_Adder	22	34	64	0	32	0	0	0	3290.8	0	LIA
MinMax	56	2	0	0	21	0	0	0	540.4	0	LIA
E-Model	bit	add	mult	re1	re2	bw	bs	ibs	$Score_L$	$Score_B$	**Enc**
cf_cordic	2	9	7	0	11	0	0	1	150	0	BV
Daio	2	9	7	0	11	0	0	1	150	0	BV
Dekker	2	0	0	48	4	0	0	0	17.6	0	BV
Unidec	4	0	0	0	55	0	14	14	352	3584	BV
soc_ram	46	0	0	186	10	0	0	0	1050.4	0	LIA
AltMult	8	30	28	0	48	0	0	0	1247.2	0	LIA

Table 1. Encoding predictions for the models in T-Model and E-Model sets

7 Related Work

As we discussed in Sect. 4, the basic LIA encoding for combinational circuits was presented in [6]. In contrast to our selective approach for hardware verification, they adopted the layered approach inside the solver that deals with EUF, the incomplete BV, and the complete LIA encodings. In [2], the author presented a word-level reduction method for industrial netlist verification. He focused on simplifying the netlist as much as possible by applying word-level reductions to equality and disequality comparators. Then, the

simplified netlist was bit-blasted, and solved with either SAT or BDDs. In [12], the authors applied BV solvers to equivalence checking of a system-level model and an RTL design. In [22], the authors presented a normalization technique to simplify the word-level description of an arithmetic circuit for SAT-based BMC. In [17], the authors presented a simplification method for RTL-SAT instances with the combination of interval-arithmetic and Boolean reasoning. Earlier references of word-level hardware verification include [5], [8], and [25]. Finally, the authors of [23] presented an algorithm selection approach that selects one among the SAT solvers that performed best on a representative set of problem instances.

8 Conclusions

The choice of the right encoding style has great effect on the efficiency of model checkers at the word level. In this paper, we have presented a selective SMT encoding for hardware model checking. The approach is based on a model analysis method that selects the encoding by considering several characteristics of the model. In particular, the effects of bit-vector and bit-select operations have been studied. Experiments show that our approach selects the right encoding for most of the designs. This greatly improves the efficiency of hardware model checking. Enhanced encoding techniques have also been introduced and their effectiveness demonstrated experimentally.

References

[1] URL: http://www.eecs.berkeley.edu/ jha/beaver.html.

[2] P. Bjesse. Word level bitwidth reduction for unbounded hardware model checking. *Form. Methods Syst. Des.*, 35(1):56–72, 2009.

[3] URL: http://fmv.jku.at/boolector.

[4] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding RTL constructs for MathSAT: a preliminary report. *Electr. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006.

[5] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 741, 2002.

[6] R. Bruttomesso. *RTL Verification: From SAT to SMT(BV)*. PhD thesis, University of Trento, 2008. Available at "URL: http://www.inf.unisi.ch/postdoc/bruttomesso".

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL'89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM.

[8] R. Drechsler. Using word-level information in formal hardware verification. *Autom. Remote Control*, 65(6):963–977, 2004.

[9] B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.

[10] M. Ganai and A. Gupta. Completeness in SMT-based BMC for software programs. In *Design, Automation and Test in Europe (DATE'08)*, Munich, Germany, Mar. 2008.

[11] URL: http://www.icarus.com/eda/verilog.

[12] A. Kölbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In *DATE*, pages 196–201, 2009.

[13] S. Kundu, M. K. Ganai, and C. Wang. Contessa: Concurrency testing augmented with symbolic analysis. In *CAV*, pages 127–131, 2010.

[14] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *18th International Conference on Computer Aided Verification, CAV'06*, volume 4144 of *Lecture Notes in Computer Science*, pages 413–426. Springer, 2006.

[15] URL: http://www.smtcomp.org/2009.

[16] URL: http://opencores.org.

[17] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and F. Brewer. RTL sat simplification by boolean and interval arithmetic reasoning. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 297–302, 2005.

[18] URL: http://fmv.jku.at/precosat.
[19] URL: http://www.altera.com/support/examples/verilog/verilog.html.
[20] URL: http://www.verilog.com.
[21] Vis verification benchmarks. http://vlsi.colorado.edu/∼vis.
[22] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz. A normalization method for arithmetic data-path verification. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, volume 26, pages 1909–1922, 2007.
[23] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-brown. Satzilla: Portfolio-based algorithm selection for sat, 2008.
[24] URL: http://research.microsoft.com/en-us/um/redmond/projects/z3.
[25] Z. Zeng, P. Kalla, and M. Ciesielski. Lpsat: A unified approach to RTL satisfiability. In *In Proc. DATE*, pages 398–402, 2001.

Exploring and Categorizing Error Spaces using BMC and SMT

Tim King[1], Clark Barrett[1]
[1]New York University, taking|barrett@cs.nyu.edu

Abstract

We describe an abstract methodology for exploring and categorizing the space of error traces for a system using a procedure based on Satisfiability Modulo Theories and Bounded Model Checking. A key component required by the technique is a way to generalize an error trace into a category of error traces. We describe tools and techniques to support a human expert in this generalization task. Finally, we report on a case study in which the methodology is applied to a simple version of the Traffic Air and Collision Avoidance System.

1 Introduction

Finding traces that represent errors in hardware and software by the means of Bounded Model Checking (BMC) has been one of the great recent success stories in formal methods [4]. Both Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers have been used as effective engines for BMC [10]. While SAT techniques are more established, SMT solvers have the advantage of being able to reason natively at a higher level of abstraction, easing the modeling process and often leading to efficiency gains as well [5].

In this paper, we suggest a novel use of BMC: instead of searching for a single error trace, we develop a method for exploring and categorizing the space of all errors. Our proposed approach is to repeatedly complete the following steps: first, use SMT-based BMC to find an error trace; second, generalize the trace into a set of traces (we call this set a *category*); third, specify the category formally (using the language of the SMT solver); and finally, use the formal specification to exclude this category from the next iteration of the BMC search. If the system is finite, or the categories can be made sufficiently general, the process terminates with a complete categorization of all error traces together with a sample error trace for each category.

Such a method may be helpful in situations when a system is known to have many error traces (according to some specification) and the system designers believe these error traces to be sufficiently rare or benign (while changing the system is seen as costly) as to warrant not fixing the errors. By exploring and categorizing the error space, this conjecture can be tested and either confirmed (by verifying that all categories are non-problematic) or challenged (by finding a category of serious errors that were previously unknown). Even if no serious errors are found, the procedure can be seen as an aid in developing a more refined specification (e.g. the original specification can be extended with a formal characterization of "error" categories that are deemed non-critical).

A key step in our procedure is the generalization of an error trace into a category. This step is challenging because it must balance generality (which is desirable to ensure the procedure terminates relatively quickly) with meaningfulness (since each category should be limited to a closely related set of error traces). In this paper, we specifically and intentionally consider the case in which categories will be evaluated by a human

(i.e. no formal specification exists for what an "acceptable" error trace might be). This also motivates the need for keeping the number of categories to a minimum.

The paper is organized as follows. We begin with a review of SMT, BMC, and other necessary background information. We then explain the main algorithm for error space exploration and categorization. Next, we introduce a modeling language called `transmit`, which helps bridge the gap between the system model and the SMT back-end. We then describe results on a case study that motivated this work: a simplified version of the Traffic Air and Collision Avoidance System (TCAS). Finally, we conclude with a discussion of related and future work.

2 Preliminaries

We assume the reader is familiar with standard notions from many-sorted first order logic and the Satisfiability Modulo Theories (SMT) problem (see for example [12, 5]). We assume SMT-DECIDE is a model-generating algorithm solving the SMT problem for a theory $\mathfrak{T}$ with signature $\Sigma_{\mathfrak{T}}$. SMT-DECIDE takes a $\Sigma_{\mathfrak{T}}$-formula φ and returns $(\mathbf{sat}\ M)$ if φ is satisfiable (where M is a $\mathfrak{T}$-interpretation that satisfies φ) and $(\mathbf{unsat})$ if the formula is unsatisfiable.

Bounded Model Checking (BMC) is a verification technique for systems that works by considering finite traces of the system up to some maximum size. While limited in its ability to verify properties, it has been very effective at bug-finding. For our purposes, BMC refers to the process of: selecting a bound k on the number of system steps; creating a formula that represents execution of the system from the initial state through k transitions into an error state; and then using an SMT solver to decide whether this formula is satisfiable. If the formula is satisfiable, this indicates that the error state is reachable. Furthermore, if the SMT solver can provide information about the satisfying assignment, this can be used to generate a specific error trace. On the other hand, if the formula is unsatisfiable, this proves that the error cannot be reached in k steps.

Formally, given a background theory $\mathfrak{T}$, we will take as our system model triples of the form $(V, \mathcal{I}, \mathcal{T})$. Here, V is a set of *state* variables over sorts from $\Sigma_{\mathfrak{T}}$ that describe the state of the system. We call a $\Sigma_{\mathfrak{T}}$-formula, all of whose free variables are from V, a state formula. $\mathcal{I}$ is a state formula which is true exactly when the state variables take on values representing a valid initial state of the system. $\mathcal{T}$ is a $\Sigma_{\mathfrak{T}}$-formula whose free variables are from V and V' (a copy of V containing a variable x' for each variable $x \in V$) which is true iff the system can transition from a state represented by V to a state represented by V'. We assume that error states can also be described by state formulas (for simplicity, we consider only safety properties).

Besides V', we also define V_i to be a copy of V (containing variables x_i for each $x \in V$) for each $i \geq 0$. Also, for $i \geq 0$, the indexing operator $(\cdot)_i$ takes a state formula ϕ and produces a formula ϕ_i by replacing each occurrence of $x \in V$ by the corresponding variable $x_i \in V_i$. Similarly, the indexing operator applied to $\mathcal{T}$ produces a formula $\mathcal{T}_i$ obtained by replacing each occurrence of $x \in V$ with $x_i \in V_i$ and each occurrence of $x' \in V'$ with $x_{i+1} \in V_{i+1}$.

The unrolling function UNROLL takes as input a system and an unrolling depth k, and produces a formula that represents running the system for k steps from a valid initial state:

$$\text{UNROLL}(V, \mathcal{I}, \mathcal{T}, k) := \mathcal{I}_0 \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}_i.$$

A trace τ of length k is a $\mathfrak{T}$-interpretation satisfying a k-step unrolling of the system:

EXPLORE$(V, \mathcal{I}, \mathcal{T}, \mathcal{E}, k)$

1 $q_0 \leftarrow$ UNROLL$(V, \mathcal{I}, \mathcal{T}, \mathcal{E}, k)$
2 $i \leftarrow 0$
3 **while** SMT-DECIDE$(q_i) = (\textbf{sat}\ \epsilon_i)$
4 **do** $\mathcal{C}_i \leftarrow$ ANALYZE(ϵ_i)
5 $q_{i+1} \leftarrow q_i \wedge \neg \mathcal{C}_i$
6 $i \leftarrow i + 1$
7 **return** $[\mathcal{C}_0, \ldots \mathcal{C}_i], [\epsilon_0, \ldots, \epsilon_i]$

Figure 1: The EXPLORE procedure.

$$\tau \models \text{UNROLL}(V, \mathcal{I}, \mathcal{T}, k).$$

Given a state formula $\mathcal{E}$ describing an error state, an error trace, ϵ, is a system trace that additionally satisfies $\mathcal{E}_i$ for some i. We extend the unrolling function UNROLL to take as an additional input an error formula $\mathcal{E}$ and to produce a formula additionally requiring $\mathcal{E}$ to be satisfied within k transitions:

$$\text{UNROLL}(V, \mathcal{I}, \mathcal{T}, \mathcal{E}, k) := \mathcal{I}_0 \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}_i \wedge \bigvee_{i=0}^{k} \mathcal{E}_i.$$

We refer to the set of all error traces as the *error space* of the system.

3 Exploring the Error Space

Applications that employ BMC typically use it to generate a single error trace which is immediately reported to the user as a bug. The user then analyzes the trace and updates the system accordingly. In a scenario in which some errors may be deemed acceptable and modifying the system is considered to be expensive, this simple bug-finding and patch loop may no longer be appropriate. BMC can still be used to produce error traces, but a more complete picture of the error space is desirable in order to determine whether an update to the system is warranted.

The EXPLORE procedure employs BMC in a more general loop that explores and categorizes multiple error traces. A diagram of the procedure is shown in Fig. 1. EXPLORE takes as input a system, an error state, and an unrolling depth k. The procedure unrolls the system to obtain a formula q_0 that is satisfied if there are any error states reachable within k steps. q_i is sent to an SMT solver to decide if it is satisfiable (i is initially 0.) If q_i is satisfiable, the model-generating feature of the SMT solver (SMT-DECIDE) is used to obtain information about a satisfying interpretation of q_i. The satisfying interpretation ϵ_i is an error trace. The abstract procedure ANALYZE takes ϵ_i and generalizes it to a category formula $\mathcal{C}_i$, such that $\epsilon_i \models \mathcal{C}_i$. (We discuss different possible choices for ANALYZE in more detail at the end of the section and in Sec. 4.) To exclude the traces in $\mathcal{C}_i$ and discover a new satisfying error trace, q_i is conjoined with the negation of $\mathcal{C}_i$. The next iteration of the process then occurs, with the SMT solver being invoked with the new query q_{i+1}. If at some point q_i is unsatisfiable, then the categories include all the error traces possible for this k. In this case, the procedure terminates and outputs the set of categories and satisfying error traces, $[\mathcal{C}_0, \ldots \mathcal{C}_i], [\epsilon_0, \ldots, \epsilon_i]$.

EXPLORE is a straightforward generalization of a well-known solution for extracting multiple solutions of a single formula. Any solution implemented with multiple calls to an SMT solver must guarantee that the model returned in one iteration is excluded from the next. The easy way to do this is simply to assert the negation of some formula ϕ that is satisfied by the model. ϕ is an abstraction of the model. In EXPLORE, this process of abstracting the model and generating such a formula is captured by the ANALYZE procedure.

The choice of category formulas $\mathcal{C}_i$ introduced by ANALYZE must be done with care as this controls which future error traces are generated and also determines how quickly the procedure terminates. In the general presentation above, ANALYZE does not ensure that the different error traces represent *meaningful* distinctions or make progress. For example, ANALYZE could introduce the sequence of categories $x_0 = 1, x_0 = 2, \ldots$ restricting the initial value of a variable x to be a different constant each time (as long as each has a model), resulting in non-termination. On the other hand, suppose ANALYZE introduces the sequence $(x_0 \leq \frac{1}{2})$, $\neg(x_0 \leq \frac{1}{2})$. In this case, we do have termination, but there may be more than two substantially different kinds of bugs in the original system, whereas with this categorization, only 2 representative error traces will be produced. More (and possibly more efficient forms of) exhaustive case splitting can be done, but this may produce too many error traces.[1] While these models may be helpful for assisting other computations, if models are going to be validated by hand, the number presented to the user needs to be quite low. We mention all of this to emphasize that the value of the output of EXPLORE crucially depends on how ANALYZE guides the search through the error space, and also to highlight the difficulty of designing such a procedure.

4 Interactive Exploration of the Error Space

One of the main contributions of this paper is to evaluate an implementation of ANALYZE which uses an interactive approach. In particular, rather than trying to automate this step, we have investigated how best to support a system expert (the *analyst*) in analyzing the error traces and constructing the category formulas. This decision is principally motivated by the observation that the categories need to be meaningful and comprehensible to human evaluators, often taking into account domain-specific concepts, so that there may not be a good general mechanism for producing categories automatically.

Note that the analyst's task is not too different from what must be done by system developers when using BMC as a bug-finding tool. In particular, in order to determine the severity of and appropriate response to an error trace, a developer must thoroughly understand the trace, be able to abstract this understanding into a conceptual idea about what is wrong, and then apply this conceptual understanding to fix the problem. Here, the analyst must similarly understand the error trace and use this to create an abstract idea capturing the cause of the problem. The additional step required is to express this idea as a category formula.

There are several advantages of this approach over a simple use of BMC for bug-finding. First of all, under the assumption that there are many error traces to look at, it may not even be feasible to examine each one individually. Thus the step of categorization is crucial not just to save time but also to have any hope of covering all the traces. A second advantage is that if the analyst is able to capture the abstract concept behind each error trace, this ensures that each new error trace will represent a new conceptual problem. This is much more interesting and instructive than examining many error traces that are just slight variations of the same basic problem. Finally, this process is much more likely to lead the analyst to find rare and

[1] If ANALYZE consistently introduced the negation of the satisfying Boolean assignment to each atom in q_i, EXPLORE would be isomorphic to the SMT equivalent of ALLSAT [16], guaranteeing termination. There are a number of possible variations to this scheme, including predicate abstraction.

unexpected behaviors as they will work quickly to categorize (and thus eliminate) common and understood behaviors.

With a human analyst, the EXPLORE procedure can be summarized as a means of letting the analyst interactively search the error space of a system with the help of SMT-based BMC technology. The computer performs the difficult search for error traces on demand, while the analyst's job is to examine the error traces and produce category formulas for them. This is a paradigm similar to the one followed by users of interactive theorem provers: let the user focus on the big picture, and let the computer deal with the tedious details.

5 Transmit

As part of this work, we developed `transmit`, a language for quickly encoding BMC queries and categories for use with SMT.[2] `transmit` provides the analyst with tools to support the interactive EXPLORE procedure. `transmit` is also an appropriate language for specifying and testing prototypes. A small example of a `transmit` specification is given in Fig. 2 and explained in section 6.

Specifications in `transmit` are written by annotating S-Expressions in an underlying language, such as SMT-LIB v2. This annotated expression is reduced by `transmit` to an S-Expression in the underlying language using simple recursive top-down transformations. The primary constructs of `transmit` are: indexed state variables (`[$v]`), support for setting the index to the value of a constant expression (`[# c (.)]`), bounded repetition of an expression (`[#for start end (.)]`), and binding a parameter to a value (`[bind k c]`).

The specification of state predicates and transition relations for BMC problems can be expressed using `transmit` in a fairly straightforward definitional style. A typical use of `transmit` is to bind some parameters to constants (such as the unroll depth) and then pass the annotated formulas to `transmit` which produces an SMT query. While declarations of the system generally resemble those in more sophisticated high-level modeling languages such as SAL or UCLID, `transmit` specifications are written at a lower level: just above the level of SMT formulas. `transmit` can be thought of as a scripting language for designing SMT-LIB queries, giving the user a large degree of low-level control over the generated query.

`transmit` is designed to help support the interactive construction of categories during EXPLORE. The category formulas generated during EXPLORE are formulas over the first k states. It is desirable to make the specification of the categories hold for multiple choices of k and support basic temporal reasoning. `transmit`'s approach to this is to have the categories specified using the same underlying language as the system specification. Like system specifications, categories have access to the parameterization of the system, and can be parameterized on k as well. The categories are then given k in the same fashion as the system description, and are compiled by `transmit` into a well-defined first order logic formula C_i (over $\cup_{i=0}^{k} V_i$) which EXPLORE can use to make progress. `transmit` specifications given access to k are powerful enough to express Linear Temporal Logic safety properties [8] with past operators. This shows that basic temporal reasoning is feasible. By giving the user access to such a powerful specification scheme, we maximize their ability to interactively explore the space as they see fit.

To assist the analyst, `transmit` provides tools for parsing models output by a number of SMT solvers[3] into a pair of comma-separated-value files (one for system wide constants, and one for state variables). This format makes it easy to write short scripts that generate visualizations for the traces using tools such as gnuplot or R.

[2] An alpha version of `transmit` is available at `http://cs.nyu.edu/~taking`.

[3] Currently CVC3, CVC4 and Z3 are supported.

```
1  [bind k 2]
2  (set-logic QF_LRA)
3
4  {- System Paramters -}
5  (declare-fun MinFlowRate () Real)
6  (declare-fun Capacity () Real)
7
8  {- State Variables -}
9  [#for 0 k (declare-fun [$tank] () Real)]
10 [#for 0 k (assert (and (<= [$tank] Capacity) (>= [$tank] 0))) ]
11 [#for 0 k (declare-fun [$incoming] () Real)]
12 [#for 0 k (assert (> [$incoming] 0)) ]
13 [#for 0 k (declare-fun [$outgoing] () Real)]
14 [#for 0 k (assert (< [$outgoing] 0)) ]
15
16 {- Initial State -}
17 (assert (>= [# 0 [$tank]] (* (/ 1 2) Capacity)))
18
19 {- Transition relation -}
20 [#for 0 [- k 1] (assert (= [next [$tank]] (+ [$tank] [$incoming] [$outgoing])))]
21 [#for 0 [- k 1] (assert (ite (<= (* 2 [$tank]) Capacity)
22    (> [next [$incoming]] [$incoming]) (< [next [$incoming]] [$incoming])))]
23 [#for 0 k (assert (=> (>= (+ [$tank] MinFlowRate) 0) (<= [$outgoing] MinFlowRate)))]
24
25 {- Eventually the error formula is satsified. -}
26 (assert (or [#for 0 [- k 1] (> [$outgoing] MinFlowRate)]))
27
28 (check-sat)
```

Figure 2: `transmit` specification of a simple hybrid system.

`transmit` provides tools for facilitating the entire process of the interactive EXPLORE procedure. Doing this with a single tool helps the user maintain consistency between the system description, the categories, the query, and the satisfying interpretation.

6 Example using `transmit` and EXPLORE

This section describes an example of a `transmit` specification, and summarizes how a user might employ the EXPLORE procedure on this example. The example file is given in Fig. 2. This example models a simple hybrid system consisting of a water tank with an incoming nozzle whose rate is controllable. The liquid is continually flowing out of the tank, and the flow must be kept above a certain rate, i.e. there is an error if the amount of outgoing liquid is below some threshold.

`transmit` annotations use square brackets and LISP style S-expressions to annotate the formula. Expressions wrapped in `{-` and `-}` are comments in `transmit` (following notation from Haskell). This example is built on top of SMT-LIB v2, and compiles to an SMT-LIB v2 query. Line 1 in the example binds k to the constant 2. Line 2 is the header for the SMT query. Line 5 declares the variable `MinFlowRate`, the amount that should leave the tank every cycle. Line 6 declares the variable `Capacity`. Lines 9-10 declare a state variable `$tank` and assert that its value is always between 0 and `Capacity`. To see how the `#for` construct expands, the `transmit` output for these lines is shown below:

```
(declare-fun |tank_000| () Real)
(declare-fun |tank_001| () Real)
(declare-fun |tank_002| () Real)
```

```
(assert (and (<= |tank_000| Capacity) (>= |tank_000| 0)))
(assert (and (<= |tank_001| Capacity) (>= |tank_001| 0)))
(assert (and (<= |tank_002| Capacity) (>= |tank_002| 0)))
```

Lines 11-14 similarly declare the state variables `[$incoming]` and `[$outgoing]` and assert that the variables are respectively always positive and negative. Line 17 declares that the initial value of `[$tank]` (`[# 0 [$tank]]`) is at least half of `Capacity`. Lines 20-23 specify the transition relation. The first part of this is that the next value of `$tank` (`[next [$tank]]`) is equal to the sum of the current values of `$tank`, `$incoming` and `$outgoing`. The `transmit` output for this is 2 lines, the first of which is:

```
(assert (= |tank_001| (+ |tank_000| |incoming_000| |outgoing_000|)))
```

Lines 21-22 give a basic rule for increasing the incoming rate if the tank is at least half empty and decreasing the rate otherwise. Line 23 specifies that if `[$tank]` has at least the minimum flow rate currently in it, then at least this amount flows out. Line 26 is a statement of the error condition: at some point, less than `MinFlowRate` flows out of the tank. The `transmit` output for line 26 is:

```
(assert (or (> |outgoing_000| MinFlowRate) (> |outgoing_001| MinFlowRate)))
```

Finally, the file is ended by the SMT `(check-sat)` command.

We used the EXPLORE procedure on this specification, and generated two categories, after which the loop terminated with an unsatisfiable result. Our first category was motivated by the observation that the relationship between `MinFlowRate` and `Capacity` is undefined, so it is possible that `Capacity` is actually smaller than the magnitude of `MinFlowRate` (which is negative), meaning that `(>= (+ [$tank] MinFlowRate) 0)` would always be false. We used the following formula as a generalization of this category of errors:

```
(assert (< (+ Capacity (* 2 MinFlowRate)) 0))
```

Our second category addresses the main problem of the system. At any point, `$outgoing` can be sufficiently negative and `$incoming` sufficiently close to zero that as a result, the value of `$tank` in the next state is less than the magnitude of `MinFlowRate`. We introduced a category for this problem that captures exactly these cases: `(<= (+ [$tank] [$incoming] [$outgoing] MinFlowRate) 0)`. The negation of these two categories together with the original formula is unsatisfiable and EXPLORE terminates.

7 Case Study

The Traffic Air and Collision Avoidance System (TCAS) is a currently deployed collision avoidance system for aircraft [15]. The system provides pilots independent tracking of other aircraft in the local airspace and in emergency situations provides Resolution Advisories (RAs) to the pilots on how to avoid likely collisions. TCAS's RAs are considered a means of defense-in-depth for when normal air traffic management's separation procedures have broken down. Due to TCAS's inherent safety critical nature [1], it has been the subject of a number formal studies in the past [14, 17, 7, 18]. Other aircraft collision avoidance systems have been studied in detail as well [19].

TCAS treats the planes as points and attempts to keep the points sufficiently far apart so as to avoid Near Mid-Air Collisions (NMACs). An NMAC is defined as a situation in which two planes have a horizontal separation (`range`) less than the constant `NMACw` and a vertical separation less than the constant `NMACh`.[4]

[4] We used `NMACw` $= 500$ft and `NMACh` $= 100$ft.

Geometrically, this means that an NMAC occurs if a second plane enters a cylinder centered around the first plane. This over-approximation avoids having to model complex and mostly irrelevant plane and helicopter geometries that more accurate modeling of mid-air collisions would require. TCAS runs a fixed protocol every second during which it both checks its sensors and performs *sense selection* (deciding whether to issue an RA and if so what RA is selected). Non-linear floating point arithmetic calculations are used for estimating position, velocity, and the time of closest horizontal distance (TCA), as well as for determining when to issue an RA and when to stop issuing the RA.

TCAS is an excellent case study for our approach because it is a system that needs to be better understood but which is very difficult and costly to change. Furthermore, if we define any scenario that leads to an NMAC to be an error trace, it is clear that errors cannot always be avoided (a malicious pilot could always cause an NMAC for example). The goal of TCAS is to avoid NMACs in reasonable scenarios. However, it is not clear how to evaluate the success of this goal.

We applied our technique to a simplified version of TCAS called Tiny TCAS. Tiny TCAS was developed at MIT Lincoln Laboratory for the purpose of experimenting with formal techniques.[5] Tiny TCAS restricts its attention to the case when one plane is equipped with Tiny TCAS and a single intruder is equipped only with a transponder (which communicates its position and velocity). Tiny TCAS assumes its variables are real numbers (ignoring approximations and errors introduced by floating point representations). Tiny TCAS contains non-linear real arithmetic constraints for projecting positions in the future and calculating when an RA can be released.

While some existing SMT solvers do have limited support for non-linear real arithmetic[9, 3], there are no currently available solvers able to analyze the Tiny TCAS model without modification.[6] Since Tiny TCAS already makes many simplifying assumptions, we added one additional simplification (holding constant the horizontal rate at which the aircraft are approaching each other), which allowed us to obtain a linear model. Formulas generated from the model then fit within the SMT-LIB logic QF_LRA.

Using a Transmit model and the EXPLORE procedure, we generated five categories of system failure for Tiny TCAS. An automatically generated visualization of an example trace from each category is shown in Fig. 3. An informal description of the categories is given below. The visualizations are automatically generated from error traces using a collection of simple scripts. Each figure shows the altitude of the two planes over time. The blue line is the intruder, and the black line is the plane equipped with Tiny TCAS. The large orange dots represent an NMAC. The first and last vertical green lines are the first and last time the horizontal range is small enough for an NMAC to occur (labeled entry and exit). The middle green vertical line labeled TCA is the time of closest approach or minimum horizontal range. We found automatically generated visualizations like this to be *the key analytical tool* in categorizing error traces. Other formal studies of TCAS have noted the importance of generating visualizations as well [7].

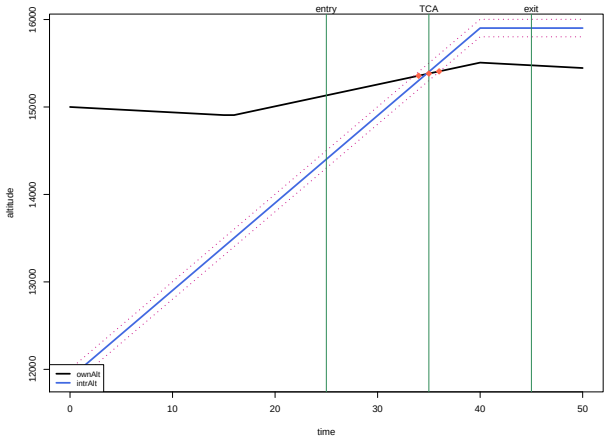
Before explaining the categories, we need the additional concept of a *projected NMAC*. By extrapolating based on the current position and velocities of the planes, the future paths of both planes can be estimated. A projected NMAC exists if an NMAC will occur based on these extrapolated paths.

The intuition behind the categories is as follows:

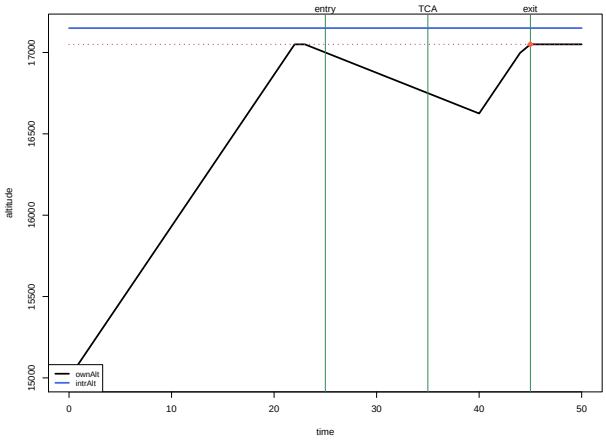
Doomed There is a projected NMAC from the beginning. Furthermore, even if an RA is issued, the plane cannot climb or descend fast enough to avoid an NMAC. This is the only category that was anticipated by the designers. (See Fig. 3a.)

[5] We are working with MIT Lincoln Laboratory to make the description of Tiny TCAS available, but it is not publicly available yet.

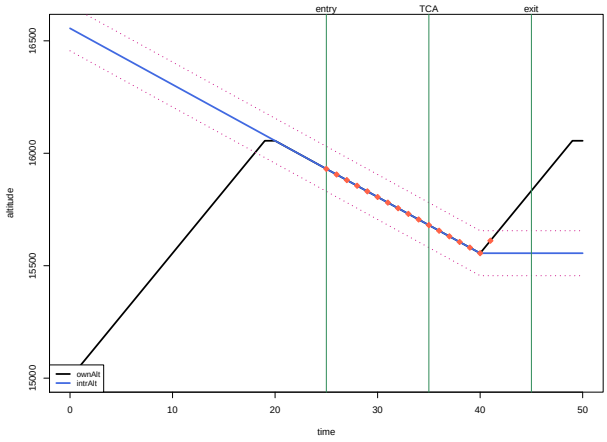
[6] An integration of interval constraint propagation and Simplex has been done within OpenSMT [13, 6]. Unfortunately at the time of this writing, this tool is unavailable.



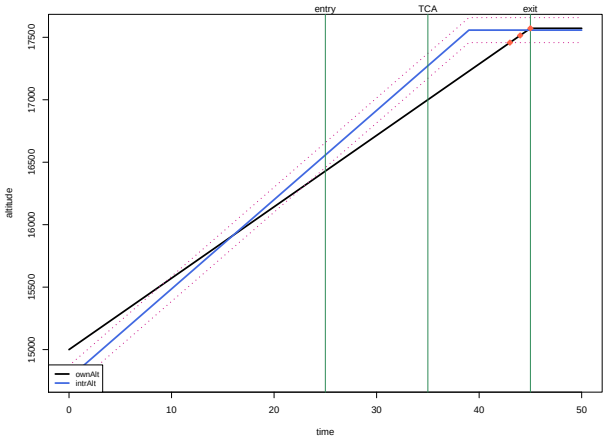
(a) Doomed



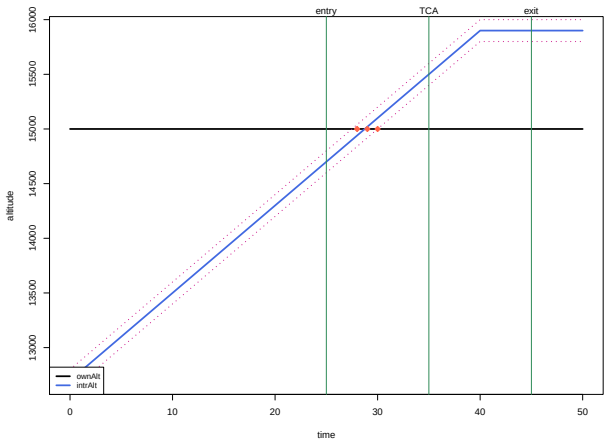
(b) Release



(c) Locked



(d) LockThenLevel



(e) Lazy

Figure 3: Automatically generated visualizations of categories for Tiny TCAS.

	Time (in s)			Memory (in MB)		
k	CVC3	Z3	CVC4	CVC3	Z3	CVC4
20	3.82	3.77	2.65	306	60	73
40	MO	598.48	12.51	MO	486	114
60	MO	47.85	20.52	MO	707	185
80	MO	43.45	21.50	MO	899	327
100	MO	121.91	98.10	MO	1720	331

Table 1: Wall clock time and maximum memory for CVC3, Z3, and CVC4 generating a model for Tiny TCAS with the constraint ¬Doomed ∧ ¬Release ∧ ¬Locked with `rangeRateMag` $= 0.016$. MO = Memory Out ($> 6GB$)

Locked Because of a projected NMAC, Tiny TCAS issues an RA. However, the trajectory projected by following this RA still leads to an NMAC. (See Fig. 3c.)

LockThenLevel Because of a projected NMAC, Tiny TCAS issues an RA. The new projected trajectory does not contain an NMAC. However, the intruder then changes its altitude rate, resulting in an NMAC. (See Fig. 3d.)

Lazy Tiny TCAS does not issue an RA despite an NMAC being projected. This is caused because the criteria for detecting collision threats are unsound. There are two important sub-cases depending on whether the TCA is being estimated correctly or not. (See Fig. 3e.)

Release Because of a projected NMAC, Tiny TCAS issues an RA, resulting in a new path on which an NMAC is not projected. When it appears safe, Tiny TCAS releases the RA. The pilot then changes the altitude rate in response to the RA being released. This then either directly results in an NMAC, or an additional change in altitude by the intruder results in an NMAC. (See Fig. 3b.)

All of the previously mentioned categories are expressible as safety properties. They are not, however, *easily* expressible as state properties, as they require a significant amount of information about the past. The system is augmented with witness variables that capture a sufficient amount of history to express the category. An example of a witness variable is the time an intruder levels off, which is used as a part of projecting the paths. To avoid introducing non-linearity, case splits are done by `transmit`. This also shows the advantage of using a low level tool like `transmit`.

While challenging for current SMT solvers, we have been able to use the EXPLORE procedure effectively to analyze Tiny TCAS. Table 1 shows the running time and memory consumption for a particular formula from our analysis using three SMT solvers and using different values for k.[7] The query is immediately after the categories Doomed, Release and Locked have been discovered, and these categories are being excluded. The SMT solvers represented are CVC3, Z3, and CVC4.[8] To the best of our knowledge, these are the only 3 SMT solvers that can handle the rewriting of the quasi non-linearity in the constraints correctly. CVC3 runs out of memory on every $k > 20$. This is due to the high memory consumption of the Fourier-Motzkin decision procedure for QF_LRA[20]. Z3 and CVC4 both use variants of the simplex method [11], and have significantly better memory performance.

[7] These experiments were run on an a 2.66GHz Intel Core2 Quad with 8GB memory.

[8] We used Z3 version 2.3 (for Linux), CVC4 version "svn co -r1780 `https://subversive.cims.nyu.edu/cvc4/cvc4/branches/arithmetic/preprocess`" with "–rewrite-arithmetic-equalities –enable-arithmetic-propagation" enabled, and CVC3 2.2 with the flag "+model".

8 Related Work

Our use of BMC is quite similar to that found in [2]. Both approaches focus on hybrid systems, reduce the problem to a single SMT query, and find violations of safety properties. Closest to our work on Tiny TCAS is the work presented in [7, 18]. This work focuses on techniques for proving that an alert is always issued to the pilot on all error traces. The scenario considered there is parallel runway approaches where an intruder deviates from a normal approach by banking. Their work proves the existence of conditions under which an aspect of TCAS is guaranteed to issue an alert. Tiny TCAS and our work focuses on both the case where alerts are issued and not issued, as well as errors that come about as part of conflict resolution. Other well known formal analysis on TCAS has focused on guaranteeing conditional safety in the case where both planes are equipped with TCAS [17].

9 Conclusion and Future Work

We have proposed a novel technique for efficient interactive exploration of the error space of a system. The procedure uses SMT-based BMC to generate error traces, and then relies on the user to guide the generation of additional error traces by generalizing and then excluding the current trace. This interactive approach allows the user to quickly find and understand multiple sources of system failure for complex fixed systems. We have used this approach to analyze Tiny TCAS, and have succeeded in identifying four additional error categories beyond those anticipated by the system designer. This shows the potential use and reasonableness of this approach.

The most serious limitation of our work is in the handling of non-linearity. We had to resort to simplification of the model in order to obtain linear (or nearly-linear) formulas. We plan on using Tiny TCAS as a motivating example for developing better techniques for solving quantifier free non-linear real arithmetic. Another interesting possibility is mixing in automated techniques to heuristically help the ANALYZE component of the system. Abstract interpretation techniques for trace abstraction seem the mostly likely candidate for success.

References

[1] Investigation report AX001-1-2/02, May 2004.

[2] AUDEMARD, G., BOZZANO, M., CIMATTI, A., AND SEBASTIANI, R. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science 119*, 2 (Mar. 2005), 17–32.

[3] BARRETT, C., AND TINELLI, C. CVC3. In *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*. Springer, Berlin, 2007, ch. 34, pp. 298–302.

[4] BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Cleaveland, Ed., vol. 1579 of *Lecture Notes in Computer Science*. Springer, Berlin, Mar. 1999, ch. 14, pp. 193–207.

[5] BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009.

[6] BRUTTOMESSO, R., PEK, E., SHARYGINA, N., AND TSITOVICH, A. The OpenSMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds.,

vol. 6015 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010, ch. 12, pp. 150–153.

[7] CARREÑO, V., AND MUÑOZ, C. Aircraft trajectory modeling and alerting algorithm verification. In *Theorem Proving in Higher Order Logics*, M. Aagaard and J. Harrison, Eds., vol. 1869 of *Lecture Notes in Computer Science*. Springer, Berlin, 2000, ch. 6, pp. 90–105.

[8] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, Jan. 1999.

[9] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. 24, pp. 337–340.

[10] DE MOURA, L., AND BJØRNER, N. Bugs, moles and skeletons: Symbolic reasoning for software development. In *Automated Reasoning*, J. Giesl and R. Hähnle, Eds., vol. 6173 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010, ch. 34, pp. 400–411.

[11] DUTERTRE, B., AND DE MOURA, L. A fast Linear-Arithmetic solver for DPLL(t). In *Computer Aided Verification*, T. Ball and R. Jones, Eds., vol. 4144 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 81–94.

[12] ENDERTON, H., AND ENDERTON, H. B. *A Mathematical Introduction to Logic, Second Edition*, 2 ed. Academic Press, Jan. 2001.

[13] GAO, S., GANAI, M., IVANCIC, F., GUPTA, A., SANKARANARAYANAN, S., AND CLARKE, E. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic. *FMCAD* (2010).

[14] IEEE. *On the formal verification of the TCAS conflict resolution algorithms* (1997), vol. 2.

[15] KUCHAR, J. K., AND DRUMM, A. C. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal 16*, 2 (2007), 277–296.

[16] LAHIRI, S., NIEUWENHUIS, R., AND OLIVERAS, A. SMT techniques for fast predicate abstraction. In *Computer Aided Verification*, T. Ball and R. Jones, Eds., vol. 4144 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006, ch. 39, pp. 424–437.

[17] LIVADAS, C., LYGEROS, J., AND LYNCH, N. A. High-level modeling and analysis of the traffic alert and collision avoidance system (TCAS). *Proceedings of the IEEE 88*, 7 (July 2000), 926–948.

[18] MUÑOZ, C., CARREÑO, V., DOWEK, G., AND BUTLER, R. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer (STTT) 4*, 3 (May 2003), 371–380.

[19] PLATZER, A., AND CLARKE, E. Formal verification of curved flight collision avoidance maneuvers: A case study. In *FM 2009: Formal Methods*, A. Cavalcanti and D. Dams, Eds., vol. 5850 of *Lecture Notes in Computer Science*. Springer, Berlin, 2009, ch. 35, pp. 547–562.

[20] SCHRIJVER, A. *Theory of Linear and Integer Programming*. Wiley, June 1998.

A Theory of C-Style Memory Allocation*

Stephan Falke, Florian Merz, and Carsten Sinz

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{stephan.falke, florian.merz, carsten.sinz}@kit.edu
http://verialg.iti.kit.edu

Abstract. This paper introduces the theory $\mathcal{T}_{\mathcal{H}}$ for reasoning about the correctness of memory access operations in the context of a C-style heap memory. The proposed approach makes a clear distinction between reasoning about the values stored in memory and checking whether access to a specific memory location is allowed. The theory provides support for `malloc` and `free` and is presented in the form of axioms that can be converted into conditional rewrite rules. It is also shown how $\mathcal{T}_{\mathcal{H}}$ can be used in a bounded model checker for C programs.

1 Introduction

Reasoning about memory access operations is an important part of many program verification tasks. Memory access checks can, e.g., be used to detect heap or stack buffer overflows which may be exploited by malware in attacks. In general, accessing unallocated memory can result in unpredictable program behavior, loss of data, or program crashes.

Whereas several approaches for formalizing computer memory have been presented in the past (see, e.g., [1, 2, 6, 7, 11–14, 17, 18]), models of heap (or stack) memory access control are not as widespread. This is in particular true for weakly-typed programming languages such as C.

This paper develops the theory $\mathcal{T}_{\mathcal{H}}$ for reasoning about validity of heap memory access operations. $\mathcal{T}_{\mathcal{H}}$ is suitable for a C-like memory management system using function calls to `malloc` and `free` for allocating and deallocating memory on the heap. The formalization of $\mathcal{T}_{\mathcal{H}}$ has similarities to the theory of arrays $\mathcal{T}_{\mathcal{A}}$ which is governed by McCarthy's axioms for array read and write operations (sometimes also called read-*over*-write *axioms*)

$$\begin{aligned} p = q &\Rightarrow \mathsf{read}(\mathsf{write}(a, p, x), q) = x \\ p \neq q &\Rightarrow \mathsf{read}(\mathsf{write}(a, p, x), q) = \mathsf{read}(a, q) \end{aligned}$$

These axioms state that writing the value x into an array a at index p and subsequently reading a's value at index q results in the value x if indices p and q are identical. Otherwise, the read operation is not influenced by the preceding write operation. Arrays are often used to model the content of computer memory. In the programming language C, memory can be regarded as a large array of byte values.

In [16], we have extended $\mathcal{T}_{\mathcal{A}}$ with capabilities for reasoning about the correctness of memory access operations by adding suitable global constraints formalizing heap properties and memory access correctness predicates. There are two major drawbacks to this approach. First, the approach does not perform local reasoning but requires knowledge about all past heap-modifying operations. This global view does not lend itself very well to modular reasoning. Second, the approach does not provide a "separation of concerns", i.e., memory access control is intermixed with read and write operations. `malloc` and `free` modify the state

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

of the memory allocation system, but do not modify the memory content in any of the allocated memory blocks. Memory write operations, on the other hand, modify the content of allocated memory blocks, but do not change the state of the memory allocation system. Memory accesses and their correctness are thus separate concepts. Because of this, memory content and memory allocation state should be represented by different objects since this makes it possible to reason about them separately.

2 Background

We first recall McCarthy's theory of arrays $\mathcal{T}_{\mathcal{A}}$.

Sorts	E : elements I : indices A : arrays
Functions	$\mathsf{read} : A \times I \to E$ $\mathsf{write} : A \times I \times E \to A$
Axioms	$p = q \;\Rightarrow\; \mathsf{read}(\mathsf{write}(a,p,x),q) = x$ $p \neq q \;\Rightarrow\; \mathsf{read}(\mathsf{write}(a,p,x),q) = \mathsf{read}(a,q)$

Objects of sort A denote arrays, i.e., maps from indices of type I to elements of type E. The write function is used to store an element in an array. Its counter-part, the read function, is used to retrieve an element from an array.

In SMT-solvers for $\mathcal{T}_{\mathcal{A}}$, the read-over-write axioms are typically applied from left to right using the *if-then-else* operator ITE, i.e., a term $\mathsf{read}(\mathsf{write}(a,p,x),q)$ is replaced by $\mathsf{ITE}(p{=}q, x, \mathsf{read}(a,q))$. After this transformation has been applied exhaustively, only read operations remain, which can then be treated as uninterpreted functions. The resulting formula can—if needed—be further transformed into pure equality logic using Ackermann's construction: for all array variables a, let Q_a be the set of all index arguments that occur in a read operation for a. Then, each occurrence of $\mathsf{read}(a,q)$ is replaced by a fresh variable A_q, and further (consistency) constraints of the form $q_1 = q_2 \;\Rightarrow\; A_{q_1} = A_{q_2}$ for all $q_1, q_2 \in Q_a$ are added as constraints to the formula. An alternative way to deal with McCarthy's axioms was presented in [4], adding instances of this axiom lazily (on demand) in a refinement loop.

3 The Theory $\mathcal{T}_{\mathcal{H}}$

This section gives the signature and axioms of the theory $\mathcal{T}_{\mathcal{H}}$.

Sorts	I : indices (pointers) S : sizes H : allocation system states
Functions	$\varepsilon : \to H$ $\mathsf{malloc} : H \times I \times S \to H$ $\mathsf{free} : H \times I \to H$ $\mathsf{mallocsize} : H \times I \to S$
Predicates	$\mathsf{accessible} : H \times I \times S$ $\mathsf{freeable} : H \times I$ $\mathsf{mallocable} : H \times I \times S$

ε denotes an "empty" heap object, i.e., a heap to which no memory allocation or deallocation operations have been applied. $\mathsf{malloc}(h, p, s)$ denotes the heap obtained from h by allocating a memory block of size s starting at address p (if the allocation is possible, i.e., if the block does not overlap with previously allocated blocks; otherwise the heap state is not modified). Accesses to this memory region are valid in the new heap. $\mathsf{free}(h, p)$ denotes the heap obtained from h by freeing the memory block starting at address p (if it is currently allocated; otherwise the heap state is not modified). Accesses to this memory region are invalid in the new heap. $\mathsf{mallocsize}(h, p)$ returns the size s if p is the first address of a memory region $[p, p + s)$ that is currently allocated in h.

The theory $\mathcal{T}_{\mathcal{H}}$ does not allow equality tests between objects of sort H. Thus, extensionality axioms for the equality of heap states are not needed.

The predicate $\mathsf{accessible}(h, p, s)$, the main predicate of $\mathcal{T}_{\mathcal{H}}$, is used for checking validity of memory read and write operations. It determines whether access to the memory region $[p, p + s)$ is valid in the heap h, i.e., whether it falls completely within a currently allocated memory region. $\mathsf{freeable}(h, p)$ determines if p is the first address of a currently allocated memory region. In this case, the memory region pointed to by p can safely be deallocated. Finally, $\mathsf{mallocable}(h, p, s)$ determines whether the memory region $[p, p + s)$ can be allocated in h, i.e., does not interfere with any other currently allocated memory region.

In order to simplify presentation, we restrict ourselves to $I = S = \mathbb{N}$ in the following. Alternatively, fixed-width bitvectors could be used (and we do so in our implementation).[1]

Auxiliary Predicates. For memory regions $[p, p + s)$ and $[q, q + t)$, the predicate $\mathsf{disjoint}(p, s, q, t)$ determines whether the regions are disjoint:

$$\mathsf{disjoint}(p, s, q, t) := p + s \leq q \vee q + t \leq p$$

The predicate $\mathsf{contained}(p, s, q, t)$ determines whether the memory region $[q, q + t)$ is completely contained in the region $[p, p + s)$:

$$\mathsf{contained}(p, s, q, t) := p \leq q \wedge q + t \leq p + s$$

Axioms for mallocable. Recall that the intended semantics of $\mathsf{mallocable}(h, p, s)$ is that the region $[p, p+s)$ can be allocated in h. The exact meaning of this is explained in the following. First, $\mathcal{T}_{\mathcal{H}}$ assumes that mallocs of size zero are not allowed.[2] Thus,

$$\mathsf{mallocable}_{\mathsf{size}}(h, p, s) \Leftrightarrow s \neq 0$$

Additionally, it needs to be ensured that distinct mallocs do not allocate overlapping regions of the heap. There are several ways to formalize this requirement. In a first step, we use a simplified formalization which achieves the non-overlapping property by enforcing that a malloc always returns an address that is larger than all addresses used in previous mallocs (a more general formalization will be presented in Section 4). This is stated by

$$\mathsf{mallocable}_{\mathsf{top}}(h, p, s) \Leftrightarrow p \geq \mathsf{heaptop}(h)$$

[1] Fixed-width bitvectors complicate the presentation due to overflow effects in bitvector arithmetic.

[2] The C standard states that in this case the result is implementation-defined. Specific ways of how this is handled in concrete implementations can easily be modeled in theories that extend $\mathcal{T}_{\mathcal{H}}$.

This axiom makes use of the additional function symbol $\mathsf{heaptop} : H \to I$ which is formalized by

$$\begin{aligned}&\mathsf{heaptop}(\varepsilon) = 0\\&\mathsf{heaptop}(\mathsf{free}(h,p)) = \mathsf{heaptop}(h)\\&\mathsf{heaptop}(\mathsf{malloc}(h,p,s)) = p+s\end{aligned}$$

In the definition of $\mathsf{heaptop}(\varepsilon)$, a different constant that more accurately reflects the lowest address used for heap memory on a system can be used instead of 0. The predicate mallocable is now defined as

$$\mathsf{mallocable}(h,p,s) := \mathsf{mallocable}_{\mathsf{size}}(h,p,s) \wedge \mathsf{mallocable}_{\mathsf{top}}(h,p,s)$$

Axioms for freeable. The intended semantics of $\mathsf{freeable}(h,p)$ is that p is the first address of a memory region that is currently allocated in h. This semantics is captured by the following axioms:[3]

$$\begin{aligned}&&\mathsf{freeable}(\varepsilon,q) &\Leftrightarrow \bot\\\mathsf{mallocable}(h,p,s) \wedge p = q &\Rightarrow& \mathsf{freeable}(\mathsf{malloc}(h,p,s),q) &\Leftrightarrow \top\\\neg(\mathsf{mallocable}(h,p,s) \wedge p = q) &\Rightarrow& \mathsf{freeable}(\mathsf{malloc}(h,p,s),q) &\Leftrightarrow \mathsf{freeable}(h,q)\\p = q &\Rightarrow& \mathsf{freeable}(\mathsf{free}(h,p),q) &\Leftrightarrow \bot\\p \neq q &\Rightarrow& \mathsf{freeable}(\mathsf{free}(h,p),q) &\Leftrightarrow \mathsf{freeable}(h,q)\end{aligned}$$

Notice that, for each possible first argument of freeable (i.e., ε, malloc, or free), the conditions on the left side of the implications cover all possible cases. This means that exactly one equivalence (on the right hand side of the implication) is usable under any circumstances. This observation also holds for the axioms in the following paragraphs.

Axioms for mallocsize. $\mathsf{mallocsize}(h,p)$ denotes the size of the currently allocated memory region which starts at p (if such a region exists; otherwise $\mathsf{mallocsize}(h,p)$ is zero):

$$\begin{aligned}&&\mathsf{mallocsize}(\varepsilon,q) &= 0\\\mathsf{freeable}(h,p) \wedge p = q &\Rightarrow& \mathsf{mallocsize}(\mathsf{free}(h,p),q) &= 0\\\neg(\mathsf{freeable}(h,p) \wedge p = q) &\Rightarrow& \mathsf{mallocsize}(\mathsf{free}(h,p),q) &= \mathsf{mallocsize}(h,q)\\\mathsf{mallocable}(h,p,s) \wedge p = q &\Rightarrow& \mathsf{mallocsize}(\mathsf{malloc}(h,p,s),q) &= s\\\neg(\mathsf{mallocable}(h,p,s) \wedge p = q) &\Rightarrow& \mathsf{mallocsize}(\mathsf{malloc}(h,p,s),q) &= \mathsf{mallocsize}(h,q)\end{aligned}$$

This function will be used in the axioms for accessible below.

[3] For all axiom groups stated below, it would also be possible to add further, more complex "axioms" that can be derived from the stated axioms (e.g., $p \neq q \wedge \mathsf{contained}(p,s,q,1) \Rightarrow \mathsf{freeable}(\mathsf{malloc}(h,p,s),q) \Leftrightarrow \bot$, which states that a free operation with an address "in the middle" of an allocated block is not valid). How these derived "axioms" affect the runtime of the implementation will be investigated in future work.

Axioms for accessible. Finally, we present the axioms for accessible. Recall that $\mathsf{accessible}(h, p, s)$ determines whether the region $[p, p + s)$ is completely contained within an allocated memory region.

$$
\begin{aligned}
& & \mathsf{accessible}(\varepsilon, p, s) &\Leftrightarrow \bot \\
\mathsf{mallocable}(h, p, s) \wedge \mathsf{contained}(p, s, q, t) &\Rightarrow & \mathsf{accessible}(\mathsf{malloc}(h, p, s), q, t) &\Leftrightarrow \top \\
\neg(\mathsf{mallocable}(h, p, s) \wedge \mathsf{contained}(p, s, q, t)) &\Rightarrow & \mathsf{accessible}(\mathsf{malloc}(h, p, s), q, t) & \\
& & &\Leftrightarrow \mathsf{accessible}(h, q, t) \\
\neg\mathsf{freeable}(h, p) &\Rightarrow & \mathsf{accessible}(\mathsf{free}(h, p), q, t) & \\
& & &\Leftrightarrow \mathsf{accessible}(h, q, t) \\
\mathsf{freeable}(h, p) \wedge \mathsf{disjoint}(p, \mathsf{mallocsize}(h, p), q, t) &\Rightarrow & \mathsf{accessible}(\mathsf{free}(h, p), q, t) & \\
& & &\Leftrightarrow \mathsf{accessible}(h, q, t) \\
\mathsf{freeable}(h, p) \wedge \neg\mathsf{disjoint}(p, \mathsf{mallocsize}(h, p), q, t) &\Rightarrow & \mathsf{accessible}(\mathsf{free}(h, p), q, t) &\Leftrightarrow \bot
\end{aligned}
$$

4 A Generalized Version of mallocable

Figure 1 contains refined axioms for mallocable. Here, $\mathsf{mallocable}_{\mathsf{fit}}(h, p, s)$ is true iff $[p, p + s)$ is disjoint from all currently allocated memory regions. The most complex axioms are the mallocable-over-free axioms (1)–(3) that are concerned with partial overlaps of a freed memory region with a memory region whose mallocability is to be determined. The different possible overlap situations are depicted in Fig. 1. Then, mallocable itself is defined by

$$\mathsf{mallocable}(h, p, s) := \mathsf{mallocable}_{\mathsf{size}}(h, p, s) \wedge \mathsf{mallocable}_{\mathsf{fit}}(h, p, s)$$

5 Extensions of the Theory

In order to be able to use the memory model for the verification of C programs, some peculiarities of the C programming language have to be taken into account. This is mostly related to the special role of `NULL`-pointers in C.

$\mathsf{malloc}(h, 0, s)$: The `malloc`-function may return `NULL` to indicate that the memory allocation could not be performed, for example because of an out-of-memory situation. Notice that the heap allocation state is not altered in this situation. To take this into account, mallocable can be replaced by $\mathsf{mallocable}'$, which is defined by

$$\mathsf{mallocable}'(h, p, s) := (p \neq 0) \wedge \mathsf{mallocable}(h, p, s)$$

Furthermore, $\mathsf{heaptop}(\varepsilon)$ needs to be changed to a non-zero constant.

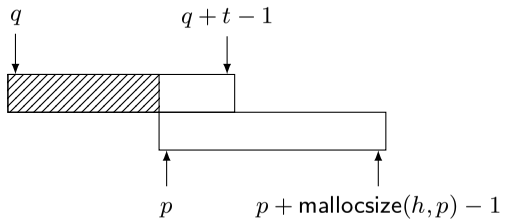
$\mathsf{free}(h, 0)$: Passing `NULL` to `free` is explicitly allowed in the C standard, but is specified to have no effect. To take this into account, $\mathsf{freeable}'$, defined by

$$\mathsf{freeable}'(h, p) := (p = 0) \vee \mathsf{freeable}(h, p)$$

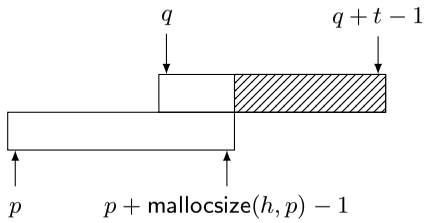
can be used instead of freeable for checking the correctness of free operations.

$$
\begin{array}{rcll}
 & & \mathsf{mallocable}_{\mathsf{fit}}(\varepsilon, p, s) \Leftrightarrow \top & \\
\mathsf{mallocable}_{\mathsf{fit}}(h, p, s) \wedge \neg\mathsf{disjoint}(p, s, q, t) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{malloc}(h, p, s), q, t) \Leftrightarrow \bot & \\
\neg(\mathsf{mallocable}_{\mathsf{fit}}(h, p, s) \wedge \neg\mathsf{disjoint}(p, s, q, t)) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{malloc}(h, p, s), q, t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h, q, t) & \\
\neg\mathsf{freeable}(h, p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h, p), q, t) \Leftrightarrow \top & \\
\mathsf{freeable}(h, p) \wedge \mathsf{contained}(p, \mathsf{mallocsize}(h, p), q, t) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h, p), q, t) \Leftrightarrow \top & \\
\mathsf{freeable}(h, p) \wedge \mathsf{disjoint}(p, \mathsf{mallocsize}(h, p), q, t) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h, p), q, t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h, q, t) & \\
\mathsf{freeable}(h, p) \wedge q < p \wedge q + t > p \wedge q + t \leq p + \mathsf{mallocsize}(h, p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h, p), q, t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h, q, p - q) & (1) \\
\mathsf{freeable}(h, p) \wedge q \geq p \wedge q < p + \mathsf{mallocsize}(h, p) \wedge q + t > p + \mathsf{mallocsize}(h, p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h, p), q, t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h, p + \mathsf{mallocsize}(h, p), & \\
 & & \qquad q + t - (p + \mathsf{mallocsize}(h, p))) & (2) \\
\mathsf{freeable}(h, p) \wedge q < p \wedge q + t > p + \mathsf{mallocsize}(h, p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h, p), q, t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h, q, p - q) \wedge & \\
 & & \qquad \mathsf{mallocable}_{\mathsf{fit}}(h, p + \mathsf{mallocsize}(h, p), & \\
 & & \qquad\qquad q + t - (p + \mathsf{mallocsize}(h, p))) & (3)
\end{array}
$$

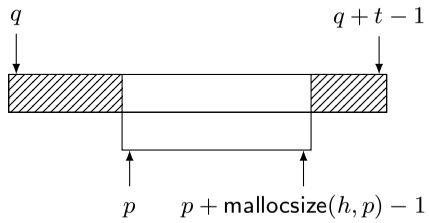
(a) Axioms



(b) Illustration for axiom (1)



(c) Illustration for axiom (2)



(d) Illustration for axiom (3)

Fig. 1. Axioms and illustrations for $\mathsf{mallocable}_{\mathsf{fit}}$.

6 Implementation

We have implemented $\mathcal{T}_{\mathcal{H}}$ in our software bounded model checking tool LLBMC as an alternative to the combined theory approach presented in [16]. Since current SMT solvers do not support $\mathcal{T}_{\mathcal{H}}$ (yet?), we apply the axioms in a pre-processing step before passing the resulting formula to an SMT solver. This pre-processing is done similarly to the case of $\mathcal{T}_{\mathcal{A}}$ as discussed in Section 2:

1. The equalities or logical equivalences in the axioms are oriented from left to right, turning them into conditional rewrite rules.
2. ITE-terms (possibly nested) are used in order to replace instances of left-hand sides by instances of right-hand sides. In order to prevent a blow-up of the formula, newly created ITE-terms are immediately simplified.

In LLBMC, mallocable and the corresponding axioms are not needed since suitable non-overlapping assumptions (see [16]) ensure that mallocable is always true.

Example 1. In this example we show that the formula

$$\mathsf{accessible}(\mathsf{free}(\mathsf{malloc}(\varepsilon, x, 1), x), x, 1) \tag{4}$$

is unsatisfiable using the above pre-processing and additional formula simplifications. Using the accessible-over-free axioms and simplifications of the introduced disjoint-subformula, (4) is equivalent to

$$\mathsf{ITE}(\mathsf{freeable}(\mathsf{malloc}(\varepsilon, x, 1), x), \bot, \mathsf{accessible}(\mathsf{malloc}(\varepsilon, x, 1), x, 1)) \tag{5}$$

The freeable-over-malloc axioms imply that the predicate $\mathsf{freeable}(\mathsf{malloc}(\varepsilon, x, 1), x)$ is equivalent to the predicate $\mathsf{ITE}(\mathsf{mallocable}(\varepsilon, x, 1), \top, \mathsf{freeable}(\varepsilon, x))$. Next, the subformula $\mathsf{mallocable}(\varepsilon, x, 1)$ is simplified to $\top$. Thus, (5) is equivalent to

$$\mathsf{ITE}(\mathsf{ITE}(\top, \top, \mathsf{freeable}(\varepsilon, x)), \bot, \mathsf{accessible}(\mathsf{malloc}(\varepsilon, x, 1), x, 1)) \tag{6}$$

Using ITE-simplifications, (6) is simplified to $\bot$, thus showing unsatisfiability of the original formula. $\Diamond$

7 Evaluation

We have evaluated LLBMC using the implementation of $\mathcal{T}_{\mathcal{H}}$ as described in Section 6 and the implementation of the approach from [16]. In [16], the $\mathcal{T}_{\mathcal{H}}$ predicates accessible, freeable, and mallocable are used as well. In contrast to $\mathcal{T}_{\mathcal{H}}$, however, the encoding of $\mathsf{accessible}(h, p, s)$ iterates over all mallocs that took place when obtaining the heap state h and have not been deallocated since then. $\mathsf{accessible}(h, p, s)$ is then encoded as a disjunction over these mallocs, where each disjunct checks whether the access operation falls within the memory block that is allocated by the malloc.

The evaluation has been performed on a collection of 97 small to medium-sized C programs from various sources. The largest part of the evaluated benchmarks was selected from the NEC Laboratories America benchmark suite[4], the Run Time Error Detection Test Suites[5], and the WCET benchmark selection[6]. Of these, only those benchmarks using dynamic heap memory allocation were included.

After unrolling of loops and inlinng of function calls, an average of 95.32 memory allocations per benchmark remained. The benchmark with the largest number of memory allocations was an algorithm for the flattening of a tree datastructure. This benchmark contained a total of 6930 memory allocations.

[4] Available at http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
[5] Available at http://rted.public.iastate.edu/
[6] Available at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Example 2. The following (artificial) example illustrates the difference between $\mathcal{T}_{\mathcal{H}}$ and [16].

```
1  int main()
2  {
3      int i;
4      int *p = NULL;
5
6      for (i = 0; i < N; ++i) {
7          p = malloc(4 * sizeof(int));
8      }
9
10     return *(p + 3);
11 }
```

Using $\mathcal{T}_{\mathcal{H}}$, the validity of the memory access operation in line 10 can easily be established by considering the last malloc in the heap state history. Using the approach from [16], on the other hand, builds a disjunction of N accessible predicates, one for each malloc in the loop. ◊

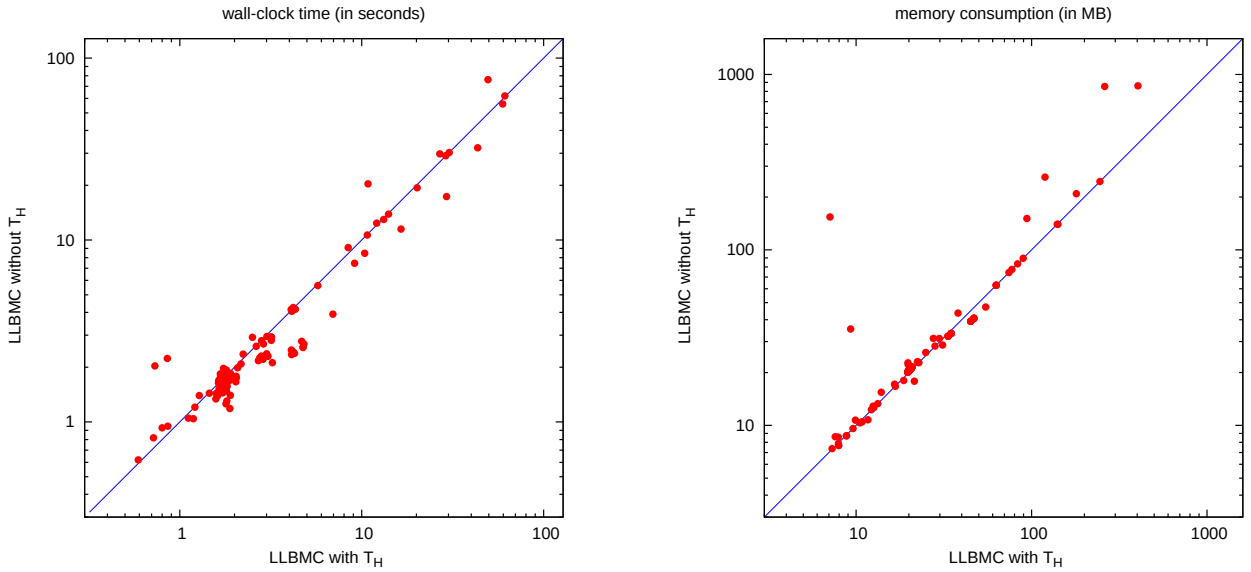


Fig. 2. Scatter plots of the run-time and memory consumption of LLBMC comparing the implementation of $\mathcal{T}_{\mathcal{H}}$ (x-axis) to the implementation following [16] (y-axis).

Scatter plots of the results are given in Fig. 2. The run-times of $\mathcal{T}_{\mathcal{H}}$ and [16] are roughly comparable. The same is true for memory consumption, but $\mathcal{T}_{\mathcal{H}}$ has a significantly lower ionsumption than [16] in certain cases. Table 1 contains a detailed comparison for selected benchmarks.

8 Related Work

Several low-level memory models[7] for C-like languages have been proposed in the past ([1, 2, 6, 7, 11–14, 17, 18]). However, they do not emphasize memory protection or ignore it completely.

[7] In a low-level memory model the memory is not much more than an array of bytes and suitable disjointness or consistency conditions are stated explicitly.

benchmark name	#mallocs/ #frees	#accessible	time SSV	time $\mathcal{T}_\mathcal{H}$	memory SSV	memory $\mathcal{T}_\mathcal{H}$
sparsemem	129/51	8374	76.2	49.5	861	404
plenty-of-mallocs	333/0	2	2.0	0.7	154	7
binary-tree	127/127	3048	7.5	9.1	150	94
flatten-trees	6930/0	42420	29.8	26.9	854	261
inplace-reverse	100/100	1800	20.4	10.8	260	119
wcet-bsort100	3/0	120204	12.4	12.1	246	246
wcet-statemate	106/0	2816	2.2	0.9	35	9

Table 1. Comparison of local ($\mathcal{T}_\mathcal{H}$) and global (SSV, see[16]) memory access formalizations on selected benchmarks. Reported times are wall-clock times in seconds; memory consumption is given in MBs.

Tuch *at al.* [18, 17] discuss a typed memory model in the context of interactive theorem proving with the proof assistant `Isabelle/HOL`. It is shown that this typed memory model is sound with respect to the untyped memory model assumed by C.

The memory model presented by Leroy and Blazy [13] is similar to our model and considers read, write, malloc, and free operations. While the disjointness of memory blocks allocated by separate mallocs is guaranteed, no such separation for accesses performed within the same memory block is ensured (e.g., accesses to different members of a structure). Leroy and Blazy prove properties of their memory model using the proof assistant `Coq` (such as semantic preservation of compiler passes). Cohen *et al.* [7] introduce a typed memory model similar to [18] for a C-like toy programming language and show that this typed memory model is sound with respect to the untyped memory model assumed by C. They support pointer arithmetic and memory access (read and write operations) at arbitrary locations in the memory, but do not consider memory protection (malloc and free operations). Mehta and Nipkow [14] present a mechanism to reason about pointer-based programs. Gast [11] gives a formalism for reasoning about memory layouts of C programs. In both cases, proof obligations are formulated in Hoare logic and verified using `Isabelle/HOL`. The memory model used in `Havoc` is presented in [6]. `Havoc` uses a reachability predicate based on the memory model in order to reason about heap-based data structures, but does not support memory protection.

Böhme and Moskal describe several typed heap encodings used by `VCC2` and `VCC3`. A memory model that is suitable for verification using separation logic is presented in [2], while a separation-based approach for deductive verification in `Caduceus` is given in [12]. Neither paper considers memory protection.

`KLEE` [5], a symbolic execution engine developed by Cadar *et al.*, shares the use of `LLVM`'s intermediate representation with our tool. `KLEE` uses an untyped, segmented memory model, where each object is represented by a separate array in the SMT solver `STP` [10]. How memory access correctness is modeled is not explicitly mentioned, though. `CUTE` [15] is a tool that combines symbolic and concrete execution in an approach called *concolic testing*. Their memory model uses fixed addresses for memory objects plus a global variable to store the next free address available for allocation. From what is published, it is not clear how they handle memory allocation. In general, symbolic execution requires techniques similar to the ones presented here. E.g., accessing an array of pointers at a "symbolic index" requires some kind of case distinction.

9 Conclusions and Future Work

We have presented $\mathcal{T}_\mathcal{H}$, a theory of heap memory allocation, that closely matches the semantics of `malloc` and `free` in C. Furthermore, we have shown how the theory's axioms can be applied as conditional

rewrite rules to reduce a problem from $\mathcal{T}_\mathcal{H}$ to a problem that can be solved by current SMT solvers such as `Boolector` [3] or `Z3` [8].

An Evaluation in the software bounded model checking tool `LLBMC` shows that even though application of $\mathcal{T}_\mathcal{H}$ eliminates the disadvantages of the approach presented in [16] by applying local simplifications to the formula, it does not impose a performance penalty in comparison to that approach.

As future work based on the results presented in this paper, we intend to combine spatially related accessible statements and hope to be able to reduce size and complexity of the generated SMT formula this way. Furthermore, we are planning to develop an approach based on lemmas-on-demand [9, 4] for solving $\mathcal{T}_\mathcal{H}$ formulas. A further possibility is to use SMT solvers that support quantified axioms (such as `Z3` [8]).

In the long term, we hope to be able to use the work presented in this paper as groundwork for a more modular software bounded model checking approach. For this, we intend to translate each function separately into our intermediate logic representation and apply syntactic and semantic rewriting on these functions. Only after this simplification has been performed, the final formula is created and passed on to the SMT solver. $\mathcal{T}_\mathcal{H}$ represents an important step towards this goal.

References

1. Sascha Böhme and Michał Moskal. Heaps and data structures: A challenge for automated provers. In *Proc. CADE 2011*, 2011. To appear.
2. Matko Botincan, Matthew Parkinson, and Wolfram Schulte. Separation logic verification of C programs with an SMT solver. *ENTCS*, 254:5–23, 2009.
3. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. TACAS 2009*, volume 5505 of *LNCS*, pages 174–177, 2009.
4. Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6:165–201, 2009.
5. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI 2008*, pages 209–224, 2008.
6. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A low-level memory model and an accompanying reachability predicate. *STTT*, 11(2):105–116, 2009.
7. Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *ENTCS*, 254:85–103, 2009.
8. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS 2008*, volume 4963 of *LNCS*, pages 337–340, 2008.
9. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Proc. SAT 2002*, 2002.
10. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV 2007*, volume 4590 of *LNCS*, pages 519–531, 2007.
11. Holger Gast. Reasoning about memory layouts. In *Proc. FM 2009*, volume 5850 of *LNCS*, pages 628–643, 2009.
12. Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Proc. HAV 2007*, pages 81–93, 2007.
13. Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.
14. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *IC*, 199(1–2):200–227, 2005.
15. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proc. FSE 2005*, pages 263–272, 2005.
16. Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *Proc. SSV 2010*, 2010.
17. Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *JAR*, 42(2–4):125–187, 2009.
18. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Proc. POPL 2007*, pages 97–108, 2007.