SMT 2014
12th International Workshop on Satisfiability Modulo Theories

Vienna, Austria, July 17-18, 2014

Affiliated with
26th Int'l. Conf. on Computer Aided Verification (CAV 2014),
7th Int'l Joint Conf. on Automated Reasoning (IJCAR 2014), and
17th Int'l Conf. on Theory and Applications of
Satisfiability Testing (SAT 2014).

Preface

This volume contains the papers presented at the 12th edition of the International Workshop on Satisfiability Modulo Theories (SMT 2014). The workshop was held on July 17th and 18th 2014 as part of the Vienna Summer of Logic (VSL 2014), in association with the 26th International Conference on Computer Aided Verification (CAV 2014), the 7th International Joint Conference on Automated Reasoning (IJCAR 2014), and the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014).

The workshop is the main annual event of the SMT community, where both researchers and users of SMT technology meet and discuss new theoretical ideas, implementation and evaluation techniques, as well as applications. Like in previous editions of the workshop, this year we invited submissions in three categories: extended abstracts, to present preliminary reports of work in progress; original papers, to describe original and mature research; and presentation-only papers, to provide additional access to important developments, recently published or submitted elsewhere and which SMT Workshop attendees may be unaware of. We received 13 submissions and each of them was reviewed by three program committee members. Due to the quality of and interest in the submissions, and in keeping with the desire to encourage presentation and discussion of work in progress, we were able to accept 11 contributions for presentation at the workshop: 2 original papers, 4 extended abstracts, and 5 presentation-only papers. Furthermore, the program included two invited talks, by Clark Barrett from New York University and Guillaume Melquiond from Inria. We would like to thank the authors, the invited speakers, the program committee, and the reviewers for their work and contributions to the workshop. We thank the CAV, IJCAR, SAT, and VSL organizers for their support and for hosting the workshop, and the EasyChair team for the availability of the EasyChair Conference System.

July, 2014

Philipp Rümmer
Christoph M. Wintersteiger

Table of Contents

Program Committee

Martin Brain	University of Oxford
Roberto Bruttomesso	Atrenta
Bruno Dutertre	SRI International
Pascal Fontaine	Inria, Loria, University of Lorraine
Malay Ganai	NEC Labs America
Sicun Gao	Carnegie Mellon University
Amit Goel	Calypto Design Systems
Alberto Griggio	FBK-IRST
Jochen Hoenicke	University of Freiburg
Dejan Jovanović	SRI International
Albert Oliveras	Technical University of Catalonia
Philipp Rümmer	Uppsala University
Christoph Sticksel	The University of Iowa
Cesare Tinelli	The University of Iowa
Tjark Weber	Uppsala University
Georg Weissenbacher	Vienna University of Technology
Thomas Wies	New York University
Christoph M. Wintersteiger	Microsoft Research

Additional Reviewers

Aleksandar Zeljić

Invited Talk

SMT: Where do we go from here?

Clark Barrett

New York University

There is no question that the last decade has been a remarkable success story for SMT. Yet, many challenges remain that are obstacles to unlocking the full potential of this technology. In this talk, I will take a look at what has brought SMT to this point, including some notable success stories. Then I will discuss some of the remaining challenges, both technical and non-technical, and suggest directions for addressing these challenges.

Speeding Up SMT-Based Quantitative Program Analysis

Daniel J. Fremont and Sanjit A. Seshia

University of California, Berkeley
dfremont@berkeley.edu
sseshia@eecs.berkeley.edu

Abstract

Quantitative program analysis involves computing numerical quantities about individual or collections of program executions. An example of such a computation is quantitative information flow analysis, where one estimates the amount of information leaked about secret data through a program's output channels. Such information can be quantified in several ways, including channel capacity and (Shannon) entropy. In this paper, we formalize a class of quantitative analysis problems defined over a weighted control flow graph of a loop-free program. These problems can be solved using a combination of path enumeration, SMT solving, and model counting. However, existing methods can only handle very small programs, primarily because the number of execution paths can be exponential in the program size. We show how path explosion can be mitigated in some practical cases by taking advantage of special branching structure and by novel algorithm design. We demonstrate our techniques by computing the channel capacities of the timing side-channels of two programs with extremely large numbers of paths.

1 Introduction

Quantitative program analysis involves computing numerical quantities that are functions of individual or collections of program executions. Examples of such problems include computing worst-case or average-case execution time of programs, and quantitative information flow, which seeks to compute the amount of information leaked by a program. Much of the work in this area has focused on extremal quantitative analysis problems — that is, problems of finding worst-case (or best-case) bounds on quantities. However, several problems involve not just finding extremal bounds but computing functions over multiple (or all) executions of a program. One such example, in the general area of quantitative information flow, is to estimate the entropy or channel capacity of a program's output channel. These quantitative analysis problems are computationally more challenging, since the number of executions (for terminating programs) can be very large, possibly exponentially many in the program size.

In this paper, we present a formalization and satisfiability modulo theories (SMT) based solution to a family of quantitative analysis questions for deterministic, terminating programs. The formalization is covered in detail in Section 2, but we present some basic intuition here. This family of problems can be defined over a weighted graph-based model of the program. More specifically, considering the program's control flow graph, one can ascribe weights to nodes or edges of the graph capturing the quantity of interest (execution time, number of bits leaked, memory used, etc.) for basic blocks. Then, to obtain the quantitative measure for a given program path, one sums up the weights along that path. Furthermore, in order to count the number of program inputs (and thus executions) corresponding to a program path, one can perform model counting on the formula encoding the path condition. Finally, to compute the quantity of interest (such as entropy or channel capacity) for the overall program, one combines the quantities and model counts obtained for all program paths using a prescribed formula.

The obvious limitation of the basic approach sketched above is that, for programs with substantial branching structure, the number of program paths (and thus, executions) can be exponential in the program size. We address this problem in the present paper with two ideas. First, we show how a certain type of "confluent" branching structure which often occurs in real programs can be exploited to gain significant performance enhancements. A common example of this branching structure is the presence of a conditional statement inside a for-loop, which leads to 2^N paths for N loop iterations. In this case, if the branches are proved to be "independent" of each other (by invoking an SMT solver), then one can perform model counting of individual branch conditions rather than of entire path conditions, and then cheaply aggregate those model counts. Secondly, to compute a quantity such as channel capacity, it is not necessary to derive the entire distribution of values over all paths. For this case, we give an efficient algorithm to compute all the values attained by a given quantity (e.g. execution time) over all possible paths — i.e., the support of the distribution — which runs in time polynomial in the sizes of the program and the support. Our algorithmic methods are particularly tuned to the analysis of timing side-channels in programs. Specifically, we apply our ideas to computing the channel capacity of timing side-channels for two standard programs which have far too many paths for previous techniques to handle.

Our techniques enable the use of SMT methods in a new application, namely quantitative program analyses such as assessing the feasibility of side-channel attacks. While SMT methods are used in other program verification problems with exponentially-large search spaces, naïve attempts to use them to compute statistics like those we consider do not circumvent path explosion. The optimizations that form our primary contributions are essential in making feasible the application of SMT to our domain.

To summarize, the main contributions of this paper include:

- a method for utilizing special branching structure to reduce the number of model counter invocations needed to compute the distribution of a class of quantitative measures from potentially exponential to linear in the size of the program, and
- an algorithm which exploits this structure to compute the support of such distributions in time polynomial in the size of the program and the support.

The rest of the paper is organized as follows. We present background material and problem definitions in Sec. 2. Algorithms and theoretical results are presented in Sec. 3. Experimental results are given in Sec. 4 and we conclude in Sec. 5.

2 Background and Problem Definition

We present some background material in Sec. 2.1 and the formal problem definitions in Sec. 2.2.

2.1 Preliminaries

We assume throughout that we are given a loop-free deterministic program F whose input is a set of bits I. Our running example for F will be the standard algorithm for modular exponentiation by repeated squaring, denoted `modexp`, where the base and modulus are fixed and the input is the exponent. Usually `modexp` is written with a loop that iterates once for each bit of the exponent. To make `modexp` loop-free we unroll its loop, yielding for a 2-bit exponent the program shown on the left of Figure 1. Lines 2–5 and 6–9 correspond to the two iterations of the loop.

```
1: r ← 1
2: if (e & 1) = 1 then
3:     r ← rb (mod m)
4: e ← e >> 1
5: b ← b^2 (mod m)
6: if (e & 1) = 1 then
7:     r ← rb (mod m)
8: e ← e >> 1
9: b ← b^2 (mod m)
10: return r
```

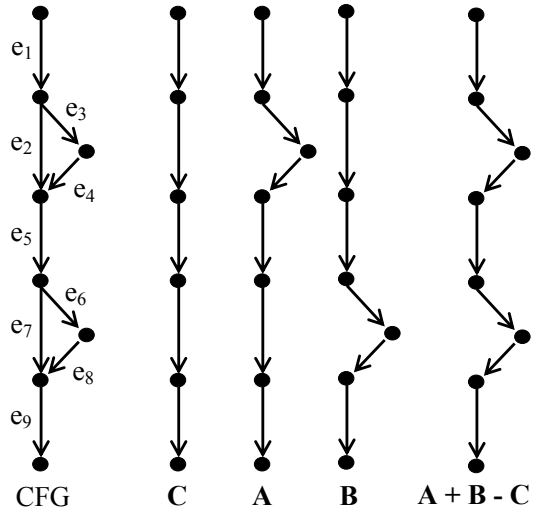


Figure 1: Unrolled pseudocode and CFG for `modexp`, computing $b^e \pmod m$ for a 2-bit exponent e. Paths **A**, **B**, and **C** form a basis, the remaining (rightmost) path being a linear combination of them.

To describe the execution paths of F we use the formalism introduced by McCabe [6]. Consider the control-flow graph (CFG) of F, where there is a vertex for each basic block, conditionals having two outgoing edges. For example, since 2-bit `modexp` has two conditionals, its CFG (shown in Figure 1) has two vertices with outdegree 2. We call such vertices *branch points*, and denote the set of them by B. Which edge out of a branch point $b \in B$ is taken depends on the truth of its *branch condition* C_b, the condition in the corresponding conditional statement. In Figure 1, the branch condition for the first branch point is $(e\&1) = 1$: if this holds, then edge e_3 is taken, and otherwise edge e_2 is taken. We model the finite-precision semantics of programs, variables being represented as bitvectors, so that the branch conditions can be expressed as bitvector SMT formulae. Since these conditions can depend on the result of prior computations (e.g. the second branch condition in Figure 1), the corresponding SMT formulae include constraints encoding how those computations proceed. Then each formula uniquely determines the truth of its branch condition given an assignment to the input bits. When necessary, these formulae can be bit-blasted into propositional SAT formulae for further analysis (e.g. model counting).

For convenience we add a dummy vertex to the CFG which has an incoming edge from all sink vertices. Since F is loop-free the CFG is a DAG, and each execution of F corresponds to a simple path from the source to the (now unique) sink. Given such a path P, we write $\mathrm{B}(P)$ for the set of branch points where P takes the right of the two outgoing edges, corresponding to making C_b true. If there are N edges then these paths can be viewed as vectors in $\{0,1\}^N$, where each coordinate specifies whether the corresponding edge is taken. For example, in Figure 1 path **A** corresponds to the vector $(1,0,1,1,1,1,0,0,1)$ under the given edge labeling. This representation allows us to speak meaningfully about linear combinations of paths, as long as the result is in $\{0,1\}^N$. A *basis* of the set of paths is defined by analogy to vector spaces to be a minimal set of paths from which all paths can be obtained by taking linear combinations. In Figure 1, the paths **A**, **B**, and **C** form a basis, as the only other path through the CFG can be

expressed as $\mathbf{A} + \mathbf{B} - \mathbf{C}$.

Now suppose we are given an integer weight for each basic block of F, or equivalently for each vertex of its CFG.[1] We define the *total weight* $\mathrm{wt}(P)$ of an execution path P of F to be the sum of the weights of all basic blocks along P. Note that we get the same value if the weight of each vertex is moved to all of its outgoing edges (obviously excluding the dummy sink), and we sum edge instead of vertex weights — thus $\mathrm{wt}(\cdot)$ is a linear function. Since F is deterministic, each input $x \in \{0,1\}^I$ triggers a unique execution path we denote $\mathrm{P}(x)$, and so has a well-defined total weight $\mathrm{wt}(x) = \mathrm{wt}(\mathrm{P}(x))$.

2.2 Problem Definition

We consider in this paper the following problems:

Problem 1. Picking $x \in \{0,1\}^I$ uniformly at random, what is the distribution of $\mathrm{wt}(x)$?

and the special case:

Problem 2. What is the support of the distribution of $\mathrm{wt}(x)$, i.e. what is the set $\mathrm{wt}(\{0,1\}^I) = \{\mathrm{wt}(x) \mid x \in \{0,1\}^I\}$?

One way to think about these problems is to view the weight of a basic block as some quantity or resource, say execution time or energy, that the block consumes when executed. Then Problem 1 is to find the distribution of the total execution time or energy consumption of the program.

Computing or estimating this distribution is useful in a range of applications (see [10]). We consider here a quantitative information flow (QIF) setting, with an adversary who tries to recover x from $\mathrm{wt}(x)$. In the example above, this would be a timing side-channel attack scenario where the adversary can only observe the total execution time of the program. Given the distribution of $\mathrm{wt}(x)$, we can compute any of the standard QIF metrics such as *channel capacity* or *Shannon entropy* measuring how much information is leaked about x. For deterministic programs, the channel capacity[2] is simply the (base 2) logarithm of the number of possible observed values [11]. Thus to compute the channel capacity we do not need to know the full distribution of $\mathrm{wt}(x)$, but only how many distinct values it can take — hence our isolation of Problem 2. As we will see, this special case can sometimes be solved much more rapidly than by computing the full distribution.

We note that the general problems above can be applied to a variety of different types of resources. On platforms where the execution time of a basic block is constant (i.e. not dependent on the state of the machine), they can be applied to timing analysis. The weights could also represent the size of memory allocations, or the number of writes to a stream or device. For all of these, solving Problems 1 and 2 could be useful for performance characterization and analysis of side-channel attacks.

3 Algorithms and Theoretical Results

The simplest approach to Problem 1 would be to execute program F on every $x \in \{0,1\}^I$, computing the total weight of the triggered path and eventually obtaining the entire map

[1] Note that our formalism and approach can be made to work with rational weights, but we focus here on applications for which integer weights suffice.

[2] Sometimes called the *conditional min-entropy* of x with respect to $\mathrm{wt}(x)$, since for deterministic programs with a uniform input distribution they are the same [11].

$x \mapsto \mathrm{wt}(x)$. This is obviously impractical when there are more than a few input bits, and is wasteful because often many inputs trigger the same execution path. A more refined approach is to enumerate all execution paths, and for each path compute how many inputs trigger it. This can be done by expressing the branch conditions corresponding to the path as a bitvector or propositional formula and applying a *model counter* [3] (this idea was used in [1] to count how many inputs led to a given output, although with a linear integer arithmetic model counter). If the number of paths is much less than $2^{|I|}$, as is often the case, this approach can be significantly more efficient than brute-force input enumeration. However, as noted above the number of paths can be exponential in the size of F, in which case this approach requires exponentially-many calls to the model counter and therefore is also impractical.

A prototypical example of path explosion is our running example `modexp`. For an N-bit exponent, there are N conditionals, and all possible combinations of these branches can be taken, so that there are 2^N execution paths. This makes model counting each path infeasible, but observe that the algorithm's branching structure has two special properties. First, the conditionals are *unnested*: the two paths leading from each conditional always converge prior to the next one. Second, the branch conditions are *independent*: they depend on different bits of the input. Below we show how we can use these properties to gain greater efficiency, yielding Algorithms 2 and 4 for Problems 1 and 2 respectively.

3.1 Unnested Conditionals

If F has no nested conditionals, its CFG has an "N-diamond" form like that shown in Figure 1 (the number of basic blocks within and between the "diamonds" can vary, of course — in particular, we do not assume that the "else" branch of a conditional is empty, as is the case for `modexp`). This type of structure naturally arises when unrolling a loop with a conditional in the body, as indeed is the case for `modexp`. Verifying that there are no nested conditionals is a simple matter of traversing the CFG.

With unnested conditionals, there is a one-to-one correspondence between execution paths and subsets of B, given by $P \mapsto \mathrm{B}(P)$. For any $b \in B$, we write B_b for the path which takes the left edge at every branch point except b (i.e. makes every branch condition false except for that of b — of course it is possible that no input triggers this path). We write B_{none} for the path which always takes the left edge at each branch point. For example, in Figure 1 if the conditionals on lines 2 and 6 correspond to branch points a and b respectively, then $\mathbf{A} = B_a$, $\mathbf{B} = B_b$, and $\mathbf{C} = B_{\mathrm{none}}$. In general, B_{none} together with the paths B_b form a basis for the set of all paths. In fact, for any path P it is easy to see that

$$P = \left(\sum_{c \in \mathrm{B}(P)} B_c \right) - (|\mathrm{B}(P)| - 1)\, B_{\mathrm{none}} \ . \tag{1}$$

This representation of paths will be useful momentarily.

3.2 Independence

Recall that an input variable of a Boolean function is a *support variable* if the function actually depends on it, i.e. the two cofactors of the function with respect to the variable are not equivalent. For each branch point $b \in B$, let $S_b \subseteq I$ be the set of input bits which are support variables of C_b. We make the following definition:

Definition 1. Two conditionals $b, c \in B$ are *independent* if $S_b \cap S_c = \emptyset$.

Independence simply means that there are no common support variables, so that the truth of one condition can be set independently of the truth of the other.

To compute the supports of the branch conditions and check independence, the simplest method is to iterate through all the input bits, checking for each one whether the cofactors of the branch condition with respect to it are inequivalent using an SMT query in the usual way. This can be substantially streamlined by doing a simple dependency analysis of the branch condition in the source of F, to determine which input variables are involved in its computation. Then only input bits which are part of those variables need be tested (for example, in Figure 1 both branch conditions depend only on the input variable e, and if there were other input variables the bits making them up could be ignored). This procedure is outlined as Algorithm 1. Note that as indicated in Sec. 2.1, the formula ϕ computed in line 6 encodes the semantics of F so that the truth of C_b (equivalently, the satisfiability of ϕ) is uniquely determined by an assignment to the input bits. For lack of space, the proofs of Lemma 1 and the other lemmas in this section are deferred to the Appendix found in the full version of this paper.

Algorithm 1 FindConditionSupports(F)

```
 1: Compute CFG of F and identify branch points B
 2: if there are nested conditionals then
 3:   return FAILURE
 4: for all b ∈ B do
 5:   S_b ← ∅   // these are global variables
 6:   φ ← SMT formula representing C_b
 7:   V ← input bits appearing in φ
 8:   for all v ∈ V do
 9:     if the cofactors of C_b w.r.t. v are not equivalent then
10:       S_b ← S_b ∪ {v}
11: return SUCCESS
```

Lemma 1. *Algorithm 1 computes the supports S_b correctly, and given an SMT oracle runs in time polynomial in $|F|$ and $|I|$.*

If all of the conditionals of F are pairwise independent, then I can be partitioned into the pairwise disjoint sets S_b and the set of remaining bits which we write S_{none}. For any $b \in B$, the truth of C_b depends only on the variables in S_b, and we denote by T_b the number of assignments to those variables which make C_b true. Then we have the following formula for the probability of a path:

Lemma 2. *Picking $i \in \{0,1\}^I$ uniformly at random, for any path P, the probability that the path corresponding to input i is P is given by*

$$\Pr\left[\mathrm{P}(i) = P\right] = \left[2^{|S_{\text{none}}|}\left(\prod_{b \in \mathrm{B}(P)} T_b\right)\left(\prod_{b \in B \setminus \mathrm{B}(P)} \left(2^{|S_b|} - T_b\right)\right)\right] / 2^{|I|} \ .$$

Lemma 2 allows us to compute the probability of any path as a simple product if we know the quantities T_b. Each of these in turn can be computed with a single call to a model counter, as done in Algorithm 2.

Theorem 1. *Algorithm 2 correctly solves Problem 1, and given SMT and model counter oracles runs in time polynomial in $|F|$, $|I|$, and the number of execution paths of F. The model counter is only queried $|B|$ times.*

Algorithm 2 FindWeightDistribution($F, weights$)

```
1: if FindConditionSupports(F) = FAILURE then
2:     return FAILURE
3: if the sets S_b are not pairwise disjoint then
4:     return FAILURE
5: for all b ∈ B do
6:     T_b ← model count of C_b over the variables in S_b
7: dist ← constant zero function
8: for all execution paths P do
9:     p ← probability of P from Lemma 2
10:    dist ← dist[wt(P) ↦ dist(wt(P)) + p]
11: return dist
```

Proof. Follows from Lemmas 1 and 2. □

Algorithm 2 improves on path enumeration by using one invocation of the model counter per branch point, instead of one invocation per path. In total the algorithm may still take exponential time, since we need to compute the product of Lemma 2 for each path, but if model counting is expensive there is a substantial savings.

Further savings are possible if we restrict ourselves to Problem 2. For this, we want to compute the possible values of $\mathrm{wt}(x)$ for all inputs x. This is identical to the set of possible values $\mathrm{wt}(P)$ for all *feasible* paths P (the paths that are executed by some input). Thus, we do not need to know the probability associated with each individual path, but only which paths are feasible and which are not. Lemma 2 implies that all paths are feasible (unless some $T_b = 0$ or $T_b = 2^{|S_b|}$, corresponding to a conditional which is identically false or true; then $S_b = \emptyset$, so we can detect and eliminate such trivial conditionals), and this leads to

Lemma 3. *Let D be the multiset of differences $\mathrm{wt}(B_b) - \mathrm{wt}(B_{\mathrm{none}})$ for $b \in B$. Then the possible values of $\mathrm{wt}(i)$ over all inputs $i \in \{0,1\}^I$ are the possible values of $\mathrm{wt}(B_{\mathrm{none}}) + D^+$, where D^+ is the set of sums of submultisets of D.*

To use Lemma 3 to solve Problem 2, we must find the set D^+. The brute-force approach of enumerating all submultisets is obviously impractical unless D is very small. We cannot hope to do better than exponential time in the worst case[3], since D^+ can be exponentially larger than D. However, in many practical situations D^+ is not too much larger than D. This is because the paths B_b often have similar weights, so the variation $V = \max D - \min D$ is small and we can apply the following lemma:

Lemma 4. *If $V = \max D - \min D$, then $|D^+| = O(V\,|D|^2)$.*

Small differences between weights are exploited by Algorithm 3, which as shown in the Appendix computes D^+ in $O(|D|\,|D^+|)$ time. By Lemma 4, the algorithm's runtime is $O(|D| \cdot V|D|^2) = O(V\,|D|^3)$, so it is very efficient when V is small. The essential idea of the algorithm is to handle one element $x \in D$ at a time, keeping a list of possible sums found so far sorted so that updating it with the new sums possible using x is a linear-time operation. For simplicity we only show how positive $x \in D$ are handled, but see the analysis in the Appendix for the general case.

[3] Although we note that for channel capacity analysis we only need $|D^+|$ and not D^+ itself, and there could be a faster (potentially even polynomial-time) algorithm to find this value.

Algorithm 3 SubmultisetSums(D)

```
 1: sums ← (0)
 2: for all x ∈ D do
 3:    newSums ← (sums[0])
 4:    i → 1   // index of next element of sums to add to newSums
 5:    for all y ∈ sums do
 6:       z ← x + y
 7:       while i < len(sums) and sums[i] < z do
 8:          newSums.append(sums[i])
 9:          i ← i + 1
10:       newSums.append(z)
11:       if i < len(sums) and sums[i] = z then
12:          i ← i + 1
13:    sums ← newSums
14: return sums
```

Using Algorithm 3 together with Lemma 3 gives an efficient algorithm to solve Problem 2, outlined as Algorithm 4. This algorithm has runtime polynomial in the size of its input and output.

Algorithm 4 FindPossibleWeights($F, weights$)

```
 1: if FindConditionSupports(F) = FAILURE then
 2:    return FAILURE
 3: if the sets S_b are not pairwise disjoint then
 4:    return FAILURE
 5: Eliminate branch points with S_b = ∅ (trivial conditionals)
 6: D ← empty multiset
 7: for all b ∈ B do
 8:    d ← wt(B_b) − wt(B_none)
 9:    D ← D ∪ {d}
10: D+ ← SubmultisetSums(D)
11: return wt(B_none) + D+
```

Theorem 2. *Algorithm 4 solves Problem 2 correctly, and given an SMT oracle runs in time polynomial in $|F|$, $|I|$, and $\left|\mathrm{wt}(\{0,1\}^I)\right|$.*

Proof. Clear from Lemmas 1 and 3, and the analysis of Algorithm 3 (see the Appendix). □

3.3 More General Program Structure

As presented above, our algorithms are restricted to loop-free programs which have only unnested, independent conditionals. However, our techniques are still helpful in analyzing a large class of more general programs. Loops with a bounded number of iterations can be unrolled. Unrolling the common program structure consisting of a for-loop with a conditional in the body yields a loop-free program with unnested conditionals. If the conditionals are pairwise independent, as in the `modexp` example, our methods can be directly applied. If the number of dependent conditionals, say D, is nonzero but relatively small, then each of the 2^D assignments

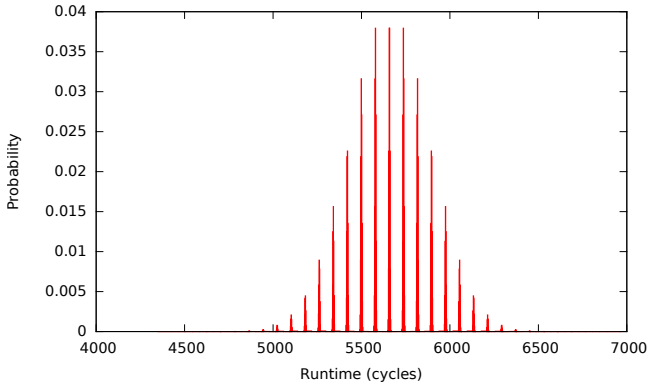


Figure 2: Timing distribution of 32-bit `modexp`, computed with Algorithm 2.

to these conditionals can be checked for feasibility with an SMT query, and the remaining conditionals can be handled using our algorithms. If many conditionals are dependent then checking all possibilities requires an exponential amount of work, but we can efficiently handle a limited failure of independence. An example where this is the case is the Mersenne Twister example we discuss in Sec. 4, where 2 out of 624 conditionals are dependent. A small level of conditional nesting can be handled in a similar way. In general, when analyzing a program with complex branching structure, our methods can be applied to those regions of the program which satisfy our requirements. Such regions do frequently occur in real-world programs, and thus our techniques are useful in practice.

4 Experiments

As mentioned in Sec. 2, Problem 2 subsumes the computation of the channel capacity of the timing side-channel on a platform where basic blocks have constant runtimes. To demonstrate the effectiveness of our techniques, we use them to compute the timing channel capacities of two real-world programs on the *PTARM* simulator [4]. The tool *GameTime* [9] was used to generate SMT formulae representing the programs, and to interface with the simulator to perform the timing measurements of the basis paths. SMT formulae for testing cofactor equivalence were generated and solved using *Z3* [2]. Model counting was done by using Z3 to convert SMT queries to propositional formulae, which were then given to the model counter *Cachet* [8]. Raw data from our experiments can be obtained at `http://math.berkeley.edu/~dfremont/SMT2014Data/`.

The first program tested was the `modexp` program already described above, using a 32-bit exponent. With 2^{32} paths, enumerating and model counting all paths is clearly infeasible. Our new approach was quite fast: finding the branch supports, model counting[4], and running Algorithm 3 took only a few seconds, yielding a timing channel capacity of just over 8 bits. In fact, although the number of paths is very large, the per-path cost of Algorithm 2 is so low that we were able to compute `modexp`'s entire timing distribution with it in 23 hours (effectively analyzing more than 50,000 paths per second). The distribution is shown in Figure 2.

[4]We note that for this program, each branch condition had only a single support variable, and thus we have $T_b = 1$ automatically without needing to do model counting.

The second program we tested was the state update function of the widely-used pseudorandom number generator the Mersenne Twister [5]. We tested an implementation of the most common variant, `MT19937`, which is available at [7]. On every 624th query to the generator, `MT19937` performs a nontrivial updating of its internal state, an array of 624 32-bit integers. We analyzed the program to see how much information about this state is leaked by the time needed to do the update. The relevant portion of the code has 2^{624} paths and thus would be completely impossible to analyze using path enumeration. With our techniques the analysis became feasible: finding the branch supports took 54 minutes, while Algorithm 3 took only 0.2 seconds because there was a high level of uniformity across the path timings. The channel capacity was computed to be around 9.3 bits. We note that among the 624 branch conditions there are two which are not independent. Thus all four truth assignments to these conditions needed to be checked for feasibility before applying our techniques to the remaining 622 conditionals.

5 Conclusions

We presented a formalization of certain quantitative program analysis problems that are defined over a weighted control-flow graph representation. These problems are concerned with understanding how a quantitative property of a program is distributed over the space of program paths, and computing metrics over this distribution. These computations rely on the ability to solve a set of satisfiability (SAT/SMT) and model counting problems. Previous work along these lines has only been applicable to small programs with very few conditionals, since it typically depends on enumerating all execution paths and the number of these can be exponential in the size of the program. We investigated how in certain situations where the number of paths is indeed exponential, special branching structure can be exploited to gain efficiency. When the conditionals are unnested and independent, we showed how the number of expensive model counting calls can be reduced to be linear in the size of the program, leaving only a very fast product computation to be done for each path. Furthermore, a special case of the general problem, which for example is sufficient for the computation of side-channel capacities, can be solved avoiding exponential path enumeration entirely. Finally, we showed the practicality of our methods by using them to compute the timing side-channel capacities of two commonly-used programs with very large numbers of paths.

Acknowledgements

Daniel wishes to thank Jon Kotker, Rohit Sinha, and Zach Wasson for providing assistance with some of the tools used in our experiments, and Garvit Juniwal for a useful discussion of Algorithm 3 and Lemma 4. The authors also thank the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy*, pages 141–153. IEEE, 2009.

[2] L. de Moura and N. Bjørner. Z3. http://z3.codeplex.com/.

[3] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. *Handbook of Satisfiability*, pages 633–654, 2009.

[4] E. A. Lee and S. A. Edwards. PTARM simulator. http://chess.eecs.berkeley.edu/pret/src/ptarm-1.0/ptarm_simulator.html.

[5] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[6] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[7] T. Nishimura and M. Matsumoto. Implementation of MT19937. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/MTARCOK/mt19937ar-cok.c.

[8] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[9] S. A. Seshia and J. Kotker. GameTime: A toolkit for timing analysis of software. In *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[10] S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):55:1–55:27, 2012.

[11] G. Smith. On the foundations of quantitative information flow. In *Foundations of Software Science and Computational Structures*, pages 288–302. Springer, 2009.

Multi-solver Support in Symbolic Execution

Hristina Palikareva and Cristian Cadar

Department of Computing, Imperial College London
London, United Kingdom
{h.palikareva, c.cadar}@imperial.ac.uk

Abstract

In this talk, we will present the results reported in our CAV 2013 paper [6] on integrating support for multiple SMT solvers in the dynamic symbolic execution engine `KLEE` [2]. In particular, we will outline the key characteristics of the SMT queries generated during symbolic execution, introduce an extension of `KLEE` that uses a number of state-of-the-art SMT solvers (`Boolector` [1], `STP` [4] and `Z3` [3]) through the `metaSMT` [5] solver framework, and compare the solvers' performance when run on large sets of `QF_ABV` queries obtained during the symbolic execution of real-world software. In addition, we will discuss several options for designing a parallel portfolio solver for symbolic execution tools.

References

[1] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS'09*.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08*.

[4] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*.

[5] Finn Haedicke, Stefan Frehse, Görschwin Fey, Daniel Große, and Rolf Drechsler. metaSMT: Focus on your application not on solver integration. In *DIFTS'12*.

[6] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *CAV'13*. `http://srg.doc.ic.ac.uk/files/papers/klee-multisolver-cav-13.pdf`.

Protocol Log Analysis with Constraint Programming (Work in progress)

Mats Carlsson[1], Olga Grinchtein[2] and Justin Pearson[3]

[1] SICS, Stockholm, Sweden
Mats.Carlsson@sics.se
[2] Ericsson AB, Stockholm, Sweden
olga.grinchtein@ericsson.com
[3] Uppsala University, Uppsala,Sweden
justin.pearson@it.uu.se

Abstract

Testing a telecommunication protocol often requires protocol log analysis. A protocol log is a sequence of messages with timestamps. Protocol log analysis involves checking that the content of messages and timestamps are correct with respect to the protocol specification. We model the protocol specification using constraint programming (MiniZinc), and we present an approach where a constraint solver is used to perform protocol log analysis. Our case study is the Public Warning System service, which is a part of the Long Term Evolution (LTE) 4G standard.

1 Introduction

In this paper we investigate the use of constraint programming to implement a part of a test harness for equipment involved in the Long Term Evolution (LTE) 4G standard [8, 2] in particular the broadcast of public warning messages [3]. The protocol includes a number of messages with complex timing requirements between them. The main novelty is that we use constraint programming [11] to directly model the protocol and to implement a test harness directly. Further, we believe that the protocol itself has independent interest as useful case study for other formal modelling approaches.

In our previous work [5] we presented an approach where we were testing an existing test harness written in Java. We generated protocol logs to test the existing Java implementation in order to find errors in the implementation. We created a model of the protocol in constraint programming in SICStus Prolog [7] and used the solutions of the constraint program to generate protocol logs. The model was then modified to produce protocol logs that were nearly correct, that is we injected faults, and these nearly correct logs were used to test if the test harness could spot errors in the protocol logs.

However, another approach can be applied in order to check that protocol log contains correct messages with correct timing: Our new approach is to use constraint solver to analyze logs directly, and hence implement the test harness using a constraint solver.

In this work, we model a part of the protocol directly in the MiniZinc [9] language (see Section 2). This approach requires a script that reads the protocol log, creates arrays of MiniZinc variables, and assigns values to the variables according to the information provided in the log.

There are a number of advantages of using MiniZinc and constraint programming: first it was very easy to translate the required parts of the telecommunication specification [3] directly into

MiniZinc; these MiniZinc specifications are automatically translated into a constraint program that can be used to test protocol logs for correctness directly; the MiniZinc specification is a declarative specification of the protocol behaviour rather than the procedural implementation that was used in the existing Java implementation of the checker; and finally the part of the protocol modelled here already provides more functionality and requires three times less lines in MiniZinc than existing Java code, and adding more functionality to the MiniZinc implementation is simply of adding more constraints.

The rest of this paper is structured as follows: in section 2 we give a very brief overview of constraint programming and MiniZinc; in section 3 we give the necessary telecommunication background to understand the case study; and in section 4 we give in some detail the constraint model that is required to test the protocol logs for correctness.

2 MiniZinc and Constraint Programming

Constraint Programming [11] (CP) is a framework for modelling and solving combinatorial problems such as verification and optimization tasks. A constraint problem is specified as a set of variables that have to be assigned values so that the given constraints on these variables are satisfied, and optionally so that a given objective function is minimised or maximised. Constraint solving is based on the constructive search for such an assignment. Constraint propagation plays an important role: a constraint is not only a declarative modelling device, but has an associated propagator, which is an algorithm to prune the search space by removing values that cannot participate in a solution to that constraint. The removal can trigger other propagators, and this process continues to fixpoint, at which time the next assignment choice must be made. A distinguishing feature of CP is the use of global constraints [11, 6]. They capture commonly occurring combinatorial patterns such as constraints on sequences, constraints on order, and constraints on placement of objects and tasks in space and time, to name a few.

MiniZinc [9] is a constraint modelling language, which has gained popularity recently due to its high expressivity and large number of available solvers. The MiniZinc language is a superset of SMT over quantifier-free formulas with linear arithmetic [10]. It also contains many useful modelling abstractions such as quantifiers, sets, arrays and a rich set of global constraints. MiniZinc is compiled into FlatZinc, a constraint solving language which specifies a set of built-in constraints that a constraint solver must support. The compilation process is based on flattening by introducing auxiliary variables, substituting them for nested subexpressions, and selecting the appropriate FlatZinc constraints. Common sub-expression elimination plays an important role as well. All the constraints presented in this paper are shown in a form that is very close to their MiniZinc version.

3 Public Warning System for LTE

In our case study we use a constraint solver to test the Public Warning System (PWS). The Public Warning System is a technology that broadcast Warning Notifications to multiple users in case of disasters or other emergencies.

3.1 E-UTRAN architecture

LTE (Long Term Evolution) [8] is the global standard for the fourth generation of mobile networks (4G). Radio Access of LTE is called evolved UMTS Terrestrial Radio Access Network

(E-UTRAN)[2]. A E-UTRAN consists of eNodeBs (eNBs), which is just another name for radio base stations. Our setup consists of an eNB, a simulated Mobility Management Entity (MME) that forwards PWS messages to the eNB, and some simulated User Equipment (UE). The functions of these entities are described in more detail below.

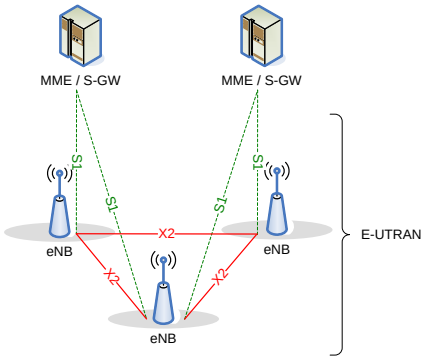


Figure 1: This figure is from 3GPP TS 36.300

An eNB connects to User Equipment via the air interface. The eNBs may be interconnected with each other by means of the X2 interface. The eNBs are also connected by means of the S1 interface to the EPC (Evolved Packet Core), more specifically to the MME (Mobility Management Entity) by means of the S1-MME interface, and to the Serving Gateway (S-GW) by means of the S1-U interface [2]. The functions of eNBs include radio resource management; IP header compression and encryption, selection of MME at UE attachment; routing of user plane data towards S-GW; scheduling and transmission of paging messages and broadcast information; and measurement and reporting configuration for mobility and scheduling [8]. An eNB is responsible for the scheduling and transmission of PWS messages received from MME. The MME performs mobility management; security control; distribution of paging messages; ciphering and integrity protection of signaling; and provides support for PWS message transmission. S-GW is responsible for packet routing and forwarding.

3.2 ETWS

Earthquake and Tsunami warning system (ETWS) is a part of PWS that delivers Primary and Secondary Warning Notifications to the UEs within an area where Warning Notifications are broadcast [3]. We show in Figure 2 the network structure of PWS architecture.

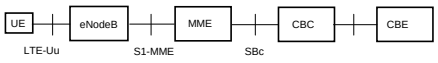


Figure 2: This figure is from 3GPP TS 23.041

The Cell broadcast Entity (CBE) can be located at content provider and sends messages to the Cell Broadcast Center. The Cell Broadcast Center (CBC) is part of EPC and connected to the MME.

The CBE sends emergency information to the CBC. The CBC identifies which MMEs need to be contacted and sends a Write-Replace Warning Request message containing the warning message to be broadcast to the MMEs. The MME sends a Write-Replace Warning Confirm message that indicates to the CBC that the MME has started to distribute the warning message to eNBs. The MME forwards Write-Replace Warning Request to eNBs in the delivery area. The eNB determines the cells in which the message is to be broadcast based on information received from MME [4]. If a Warning Type IE (information element) is included in a Write-Replace Warning Request message, then the eNB broadcasts a Primary Notification. If Warning Message Contents IE is included in a Write-Replace Warning Request message, then the eNB schedules a broadcast of the warning message according to the value of Repetition Period IE (`rPer`) and Number of Broadcasts Requested IE (`NumberofBroadcastRequested`) [1]. To inform UE about presence of an ETWS primary notification and/or ETWS secondary notification, a paging message is used. UE attempts to read paging at least once every defaultPagingCycle (`dPC`). If UE receives a Paging message including ETWS-indication, then it starts receiving ETWS primary notification or ETWS secondary notification according to schedulingInfoList contained in SystemInformationBlockType1 (`SIB1`). ETWS primary notification is contained in SystemInformationBlockType10 (`SIB10`) and ETWS secondary notification is contained in SystemInformationBlockType11 (`SIB11`). `SIB10` and `SIB11` are transmitted in System Information (SI) messages with different periodicity. If secondary notification contains a large message, then it is divided in several segments, which are transmitted in System Information messages.

4 Modelling of ETWS notifications acquisition by UE

Our models describe how UE acquires ETWS notifications sent by an eNB after receiving one Write-Replace Warning Request message from the MME. In Section 4.1 we give an overview of a model presented in [5]. In Section 4.2 we present a model in MiniZinc that analyzes protocol logs, and we compare MiniZinc model with a model from [5]. In Table 1 we present a description of parameters used in models.

4.1 Model for generation of protocol logs

ETWS requires testing that the paging messages, `SIB1`, `SIB10` and `SIB11` are transmitted correctly by the eNB. These messages appear in a UE protocol log. To test functionality automatically, the test harness initiates transmission of Write-Replace Warning Request messages by the MME simulator; configures the UE simulator and initiate logging; configures the eNB; and captures and reads a UE protocol log. The use of the MiniZinc model simply requires plugging into an existing framework that captures and reads the protocol logs.

It is useful to understand our previous work [5] where the goal was to generate UE protocol logs for ETWS, which consists of sequences of messages with timestamps, where different types of errors are introduced. To do this, we defined a model in SICStus Prolog consisting of constraints on arrays of timestamps and message contents, and based on solutions provided by SICStus Prolog we generated UE protocol logs. The constraints specified ordering constraints between messages; constraints on the number of messages of a certain type and content; and temporal constraints on the timestamps. The constraint that defines time difference between

Table 1: Parameters in the models

`delay`	Time difference between time when eNodeB starts to transmit primary notification and/or secondary notification and the time when UE reads first paging message.
`PagPN`	An array of timestamps of paging messages (also used in the model in [5]), which are transmitted every paging cycle. The size of the array is `ndPC`, which is configured in eNodeB.
`dPC`	The length of a paging cycle.
`PagSN`	An array of timestamps of paging messages (also used in the model in [5]), which are transmitted every repetition period. The size of the array is `nBR` = `NumberofBroadcastRequested` + 1.
`PagLog`	An array of timestamps of paging messages from the log. The size of the array is `nPagLog`.
`rPer`	The length of a repetition period.
`SIB1SIB10Time`	An array of timestamps of `SIB1` messages during paging cycles (used only in the model in [5]).
`SIB1SIB10Type`	An array of values from 0 to 3 that indicate whether `SIB1` messages contain schedulingInfoList for `SIB10` and/or `SIB11` (used only in the model in [5]).
`SIB1SIB11Time`	An array of timestamps of `SIB1` messages during repetition periods (used only in the model in [5]).
`SIB1SIB11Type`	An array of values from 0 to 3 that indicate whether `SIB1` messages contain schedulingInfoList for `SIB10` and/or `SIB11` (used only in the model in [5]).
`SIB1TimeLog`	An array of timestamps of `SIB1` messages from the log. The size of the array is `nSIB1Log`.
`SIB1TypeLog`	An array of values from 0 to 3 that indicate whether `SIB1` messages contain schedulingInfoList for `SIB10` and/or `SIB11`. The size of the array is `nSIB1Log`.
`SIB10Time`	An array of timestamps of System Information messages with `SIB10` (used only in the model in [5]).
`SIB10TimeLog`	An array of timestamps of System Information messages with `SIB10` from the log.
`SIB11Time`	An array of timestamps of System Information messages with `SIB11` (used only in the model in [5]).
`SIB11TimeLog`	An array of timestamps of System Information messages with `SIB11` from the log. The size of the array is `nSIB11Log`.
siPer	Periodicity of `SIB11`.
`nSeg`	Number of segments in a secondary notification.
`SIB11Segment`	An array of segment numbers of `SIB11` (used only in the model in [5]).
`SIB11SegmentLog`	An array of segment numbers of `SIB11`. The size of the array is `nSIB11Log`.

two consecutive paging messages transmitted every repetition period is

$$\begin{array}{l}(\forall 1 \leq i \leq \texttt{nBR} - 1) \\ (\texttt{PagSN}_{i+1} - \texttt{PagSN}_i = \lfloor \texttt{rPer}/\texttt{dPC} \rfloor \cdot \texttt{dPC}) \\ \vee \\ (\texttt{PagSN}_{i+1} - \texttt{PagSN}_i = (\lfloor \texttt{rPer}/\texttt{dPC} \rfloor + 1) \cdot \texttt{dPC})\end{array} \quad (1)$$

where $\texttt{PagSN}_i$ is ith element in the array PagSN. The constraint that guarantees that there is at least one paging message every repetition period is

$$\begin{array}{l}(\forall 2 \leq i \leq \texttt{nBR}) \\ (i-1) \cdot \texttt{rPer} - \texttt{dPC} < \texttt{PagSN}_i - \texttt{PagSN}_1 < (i-1) \cdot \texttt{rPer} + \texttt{dPC}\end{array} \quad (2)$$

We have also array PagPN of timestamps for paging messages which are transmitted every dPC.

Timestamps for SIB10 and SIB11 are elements of two-dimensional arrays, since several messages can be transmitted during the same paging cycle or repetition period. The constraint that defines that there are n System Information messages with SIB11 during every repetition period is

$$(\forall 1 \leq i \leq \texttt{nBR} - 1)(\forall 1 \leq j \leq n)\texttt{PagSN}_i < \texttt{SIB11Time}_{i,j} < \texttt{PagSN}_{i+1} \quad (3)$$

where SIB11Time is a two-dimensional array of timestamps of System Information messages with SIB11. It can be that UE reads different number of SIB11 during different repetition periods, but since we were interested in incorrect behaviour, we did not not model in [5] all possible correct behaviours.

Secondary notification can come in one or several segments. $\texttt{SIB11Segment}_{ij}$ contains the segment number of SIB11 with timestamp $\texttt{SIB11Time}_{i,j}$. The UE should read every segment at least once during every repetition period.

$$(\forall 0 \leq i < \texttt{nSeg})(\forall 1 \leq j \leq \texttt{nBR} - 1)(\exists 1 \leq k \leq n)\texttt{SIB11Segment}_{j,k} = i \quad (4)$$

We also constrain the time difference between two consecutive SIB10 received by UE in the same paging cycle and two consecutive SIB11 received by UE in the same repetition period. The constraint on two consecutive SIB11 received by UE is

$$\begin{array}{l}\forall(1 \leq i \leq \texttt{nBR} - 1)\forall(1 \leq j \leq n - 1) \\ (\texttt{SIB11Time}_{i,j+1} - \texttt{SIB11Time}_{i,j} > 0 \\ \wedge \\ \texttt{SIB11Time}_{i,j+1} - \texttt{SIB11Time}_{i,j} \mod \texttt{siPer} = 0 \\ \wedge \\ ((\texttt{SIB11Time}_{i,j+1} - \texttt{SIB11Time}_{i,j})/\texttt{siPer}) \mod \texttt{nSeg} = \\ (\texttt{SIB11Segment}_{i,j+1} - \texttt{SIB11Segment}_{i,j}) \mod \texttt{nSeg})\end{array} \quad (5)$$

The model contains parameters that represent timestamps and content of SIB1 messages. SIB1SIB11Time is a array of timestamps of SIB1 messages during repetition periods.

SIB1SIB11Type is array of values from 0 to 3 that indicates whether SIB1 contains

schedulingInfoList for SIB10 and/or SIB11. Then we post a constraint

$$
\begin{aligned}
&\forall(1 \leq i \leq \texttt{nBR}) \\
&((\texttt{SIB1SIB11Time}_i \leq \texttt{PagPN}_{\texttt{ndPC}} \wedge \texttt{SIB1SIB11Time}_i \leq \texttt{PagSN}_{\texttt{nBR}} \wedge \\
&\qquad \texttt{SIB1SIB11Type}_i = 1) \\
&\bigvee \\
&(\texttt{SIB1SIB11Time}_i > \texttt{PagPN}_{\texttt{ndPC}} \wedge \texttt{SIB1SIB11Time}_i \leq \texttt{PagSN}_{\texttt{nBR}} \wedge \\
&\qquad \texttt{SIB1SIB11Type}_i = 2) \\
&\bigvee \\
&(\texttt{SIB1SIB11Time}_i \leq \texttt{PagPN}_{\texttt{ndPC}} \wedge \texttt{SIB1SIB11Time}_i > \texttt{PagSN}_{\texttt{nBR}} \wedge \\
&\qquad \texttt{SIB1SIB11Type}_i = 3) \\
&\bigvee \\
&(\texttt{SIB1SIB11Time}_i > \texttt{PagPN}_{\texttt{ndPC}} \wedge \texttt{SIB1SIB11Time}_i > \texttt{PagSN}_{\texttt{nBR}} \wedge \\
&\qquad \texttt{SIB1SIB11Type}_i = 0)) \qquad (6)
\end{aligned}
$$

4.2 Model for protocol log analysis

In this section we present our new approach to use a constraint solver to find incorrect behaviour in protocol logs, by using a MiniZinc model of the correct behaviour of the protocol. There are some differences between a model in our previous work [5], outlined in Section 4.1 and the MiniZinc model here.

In Section 4.1 we had arrays PagPN and PagSN of paging messages. We keep the arrays in the MiniZinc model, but we introduce additional array PagLog of paging messages. PagLog contains timestamps of all paging messages from the log, and we use a constraint solver to check which paging message can be primary notification messages, and which can be secondary notification message. If a paging message is not first in the log we do not assign a value to $\texttt{PagLog}_1$ and add the constraint $\texttt{PagLog}_1 > 0$, otherwise we assign value 0 to $\texttt{PagLog}_1$. Then

$$(\forall 2 \leq i \leq \texttt{nSIB11Log})\texttt{PagLog}_i = \texttt{PagLog}_1 + \delta_i^{pag}, \qquad (7)$$

where δ_i^{pag} is difference between timestamp of ith paging message in the log and timestamp of the first paging message in the log. As in [5] we define constraints on PagPN and PagSN to model possible time differences between paging messages, where $\texttt{PagPN}_1 = 0$ and $\texttt{PagSN}_1 = 0$. Then we check if there is a correspondence between PagLog, PagPN and PagSN.

In Section 4.1 we had the constraint (1) that defines time difference between two consecutive paging messages transmitted every repetition period, and the constraint (2) that guarantees that there is at least one paging message every repetition period. However, the exact sequence of timestamps of paging messages which are transmitted every repetition period can be captured by the constraint

$$
\begin{aligned}
&(\forall 2 \leq i \leq \texttt{nBR}) \\
&((\texttt{rPer} \cdot (i-1) - \texttt{delay}) \mod \texttt{dPC} = 0 \rightarrow \\
&\texttt{PagSN}_i = \texttt{rPer} \cdot (i-1) - \texttt{delay}) \\
&\bigwedge \\
&((\texttt{rPer} \cdot (i-1) - \texttt{delay}) \mod \texttt{dPC} \neq 0 \rightarrow \\
&\texttt{PagSN}_i = (((\texttt{rPer} \cdot (i-1) - \texttt{delay}) / \texttt{dPC}) + 1) \cdot \texttt{dPC}) \qquad (8)
\end{aligned}
$$

We did not have constraint (8) in [5] since delay can be any value between 0 and dPC and test harness does not make checks based on (8). However, in the case when test harness is

implemented as a constraint solver, (8) can be used to check that there is a value for `delay` such that the sequence of timestamps of paging messages from the log is a valid sequence.

The constraint that defines that among paging messages from the log there are messages that correspond to paging messages of primary notification with correct timestamps is

$$(\forall 1 \leq i \leq \texttt{ndPC})((\exists 1 \leq j \leq \texttt{nPagLog})\texttt{PagLog}_j = \texttt{PagPN}_i) \leftrightarrow \texttt{PagPNinc}_i = 0, \tag{9}$$

where $\texttt{PagPNinc}_i$ is a boolean variable which indicates that there is paging message in the log which corresponds to $\texttt{PagPN}_i$.

The constraint that defines that among paging messages from the log there are messages that correspond to paging messages of secondary notification with correct timestamps is

$$(\forall 1 \leq i \leq \texttt{nBR})((\exists 1 \leq j \leq \texttt{nPagLog})\texttt{PagLog}_j = \texttt{PagSN}_i) \leftrightarrow \texttt{PagSNinc}_i = 0, \tag{10}$$

where $\texttt{PagSNinc}_i$ is a boolean variable which indicates that there is paging message in the log which corresponds to $\texttt{PagSN}_i$.

We also have the constraint

$$\begin{aligned}&(\forall 1 \leq i \leq \texttt{nPagLog})\\ &\quad(((\forall 1 \leq j \leq \texttt{ndPC})\texttt{PagLog}_i \neq \texttt{PagPN}_j) \wedge ((\forall 1 \leq j \leq \texttt{nBR})\texttt{PagLog}_i \neq \texttt{PagSN}_j))\\ &\quad\leftrightarrow \texttt{Paginc}_i = 1,\end{aligned} \tag{11}$$

where $\texttt{Paginc}_i$ is a boolean variable which indicates that ith paging message does not correspond to paging message of a primary or a secondary notification.

We first check that MiniZinc can find solution such that sum of elements of `PagPNinc`, `PagSNinc` and `Paginc` is equal to 0. If there is a solution we keep all constraints, but if MiniZinc does not find a solution we remove all constraints on `PagLog`.

Then we add constraints on a content and timestamps of `SIB1`, `SIB10` and `SIB11` messages. If a paging message is the first message in the log, then

$$(\forall 1 \leq i \leq \texttt{nSIB11Log})\texttt{SIB11TimeLog}_i = \delta_i^p, \tag{12}$$

where δ_i^p is difference between a timestamp of ith SIB11 message in log and a timestamp of first paging message in log.

If a paging message is not the first message in the log, then we have a variable $\texttt{SIB11TimeLog}_1$ that represent timestamp of first `SIB11` message in the log and

$$(\forall 2 \leq i \leq \texttt{nSIB11Log})\texttt{SIB11TimeLog}_i = \texttt{SIB11TimeLog}_1 + \delta_i^s, \tag{13}$$

where δ_i^s is difference between a timestamp of ith `SIB11` message in the log and a timestamp of first `SIB11` in the log. If a `SIB10` message is the first message in the log then $\texttt{SIB11TimeLog}_1 = \delta^{s10}$, where δ^{s10} is time difference between first `SIB10` message and first `SIB11` message. If a `SIB1` message is the first message in the log then $\texttt{SIB11TimeLog}_1 = \delta^{s1}$, where δ^{s1} is time difference between first `SIB1` message and first `SIB11` message.

We assign values to `SIB1TimeLog` and `SIB10TimeLog` using the same approach. We also assign values to `SIB1TypeLog` and `SIB11SegmentLog`.

The UE should read every segment at least once during every repetition period. Similar to (4), we have

$$\begin{aligned}&(\forall 2 \leq i \leq \texttt{nBR})(\forall 1 \leq k \leq \texttt{nSeg})\\ &\qquad((\exists 1 \leq j \leq \texttt{nSIB11Log})\texttt{SIB11SegmentLog}_j = k \wedge\\ &\qquad\qquad\texttt{PagSN}_{i-1} < \texttt{SIB11TimeLog}_j < \texttt{PagSN}_i)\\ &\qquad\leftrightarrow \texttt{PagSNSegmentinc}_{i-1,k} = 0,\end{aligned} \tag{14}$$

where $\texttt{PagSNSegmentinc}_{i,k}$ is a boolean variable which indicates that there is kth segment of secondary notification during ith repetition period.

Similar to (5) we constraint the time difference between to consecutive SIB11 messages

$$\begin{aligned}(\forall 2 \leq i \leq \texttt{nSIB11Log}) \\ ((\texttt{SIB11TimeLog}_i - \texttt{SIB11TimeLog}_{i-1}) \bmod \texttt{siPer} = 0 \wedge \\ ((\texttt{SIB11TimeLog}_i - \texttt{SIB11TimeLog}_{i-1})/\texttt{siPer}) \bmod \texttt{nSeg} = \\ (\texttt{SIB11SegmentLog}_i - \texttt{SIB11SegmentLog}_{i-1} + \texttt{nSeg}) \bmod \texttt{nSeg}) \\ \leftrightarrow \texttt{SIB11TimeLoginc}_i = 0, \end{aligned} \tag{15}$$

where $\texttt{SIB11TimeLoginc}_i$ is a boolean variable which indicates that the timestamp of the ith SIB11 message is correct.

We check that there are no SIB11 messages after the last paging message of secondary notification

$$((\exists 1 \leq i \leq \texttt{nSIB11Log})\texttt{SIB11TimeLog}_i > \texttt{PagSN}_{\texttt{nBR}}) \leftrightarrow \texttt{SIB11afterpaginc} = 1 \tag{16}$$

where SIB11afterpaginc indicates that there is SIB11 message after last paging message of secondary notification.

We have also constraints on the timestamps of SIB10 messages.

In the previous section we had two lists of timestamps of SIB1 messages. Since in protocol log it can be difficult to differentiate between which SIB1 is after paging for primary notification and which SIB1 is after paging for secondary notification, we create one array SIB1TimeLog of timestamps of SIB1 messages in MiniZinc. SIB1TypeLog is list of values from 0 to 3 that indicates whether SIB1 contains schedulingInfoList for SIB10 and/or SIB11. Similar to (6) we have the constraint for SIB1TypeLog.

We minimize sum of all "inc" boolean parameters and we use "inc" parameters to indicate errors in the log.

5 Conclusion

We think that using MiniZinc for protocol log analysis is a promising approach, since it is easy to model the protocol in MiniZinc and a constraint solver can easily handle complex requirements on time stamps. In comparison with [5], we do not need to generate random values for parameters, since we have a solution, that is values from protocol log. Since we have a solution, constraint solver can handle bigger domains of parameters than in [5]. As a future work we plan to extend the model to be able to capture behaviour in UE after receiving several Write-Replace Warning Request messages from MME and to integrate constraint solver into automation environment.

References

[1] 3GPP. Evolved universal terrestrial radio access (E-UTRA) ; S1 application protocol (S1AP). TS 36.413, 3rd Generation Partnership Project (3GPP).

[2] 3GPP. General packet radio service (GPRS) enhancements for evolved universal terrestrial radio access network (E-UTRAN) access. TS 23.401, 3rd Generation Partnership Project (3GPP).

[3] 3GPP. Public warning system (PWS) requirements. TS 22.268, 3rd Generation Partnership Project (3GPP).

[4] 3GPP. Technical realization of cell broadcast service (CBS). TS 23.041, 3rd Generation Partnership Project (3GPP).

[5] Kenneth Balck, Olga Grinchtein, and Justin Pearson. Model-based protocol log generation for testing a telecommunication test harness using clp. In *DATE*, 2014.

[6] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present, and future. *Constraints*, 12(1):21–62, March 2007. The current working version of the catalogue is at `http://www.emn.fr/z-info/sdemasse/aux/doc/catalog.pdf`.

[7] Mats Carlsson, Greger Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *PLILP 1997*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[8] SM Chadchan and CB Akki. 3GPP LTE/SAE: An overview. *International Journal of Computer and Electrical Engineering*, 2(5):806–814, 2010.

[9] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, pages 529–543, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[11] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

Reasoning About Set Comprehensions*

Edmund S. L. Lam[1] and Iliano Cervesato[1]

Carnegie Mellon University
sllam@qatar.cmu.edu, iliano@cmu.edu

Abstract

Set comprehension is a mathematical notation for defining sets on the basis of a property of their members. Although set comprehension is widely used in mathematics and some programming languages, direct support for reasoning about it is still not readily available in state-of-the-art SMT solvers. This paper presents a technique for translating formulas which express constraints involving set comprehensions into first-order formulas that can be verified by off-the-shelf SMT solvers. More specifically, we have developed a lightweight Python library that extends the popular Z3 SMT solver with the ability to reason about the satisfiability of set comprehension patterns. This technique is general and can be deployed in a broad range of SMT solvers.

1 Introduction

Reasoning about sets is a frequently occurring exercise when conducting automated verification of programs and algorithms. While some recent work has focused on automated reasoning about set (and multiset) cardinality constraints [6, 5] and about arithmetic aggregate computations [4], little research has been conducted on developing automated tools for reasoning about explicit *set comprehensions*. Set comprehension is a mathematical notation for defining sets on the basis of a property of their members. Although set comprehension is widely used in mathematics and some programming languages, direct support for reasoning about set comprehension is still not readily available in state-of-the-art SMT solvers. In this work, we consider the task of verifying the satisfiability of quantifier-free formulas in the language of set comprehension over some standard theory Th, supported by typical modern SMT solvers. We denote the resulting language as $SC(Th)$. Examples of such base theory Th include linear arithmetic over integers or real numbers, and user-defined theories over constructed data types (e.g., tuples, finite lists). In this paper, we will demonstrate our techniques on the theory of linear arithmetic over the integers (LIA). A typical problem in $SC(LIA)$ is given by the following equality:

$$\{10, 20, 30\} \doteq \{x * 10 \mid x < 4\}_{x \dot{\in} X}$$

It asks whether there are instances of the free variables in this formula (here, just the set variable X) so that the sets on the two sides are equal (we use $\doteq$ to denote set equality). The left-hand side is the extensional set $\{10, 20, 30\}$. The expression on the right-hand side is a set comprehension: it specifies the set of all the values of the expression $x*10$ where the values of x are drawn from X and are constrained by $x < 4$. The variable x is bound in the comprehension. Set comprehensions determine sets intensionally on the basis of transformations such as $x * 10$ and guards like $x < 4$. A solution to the satisfiability problem for this equality is a model $\mathcal{S}$ such that

$$\mathcal{S} \models_{SC(LIA)} \{10, 20, 30\} \doteq \{x * 10 \mid x < 4\}_{x \dot{\in} X}$$

*This paper was made possible by grant NPRP 09-667-1-100, *Effective Programming for Large Distributed Ensembles*, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

In particular, the model $\mathcal{S}$ carries information on an acceptable value of the set variable X — in this example, $X = \{1, 2, 3\}$ is the smallest solution, with $X = \{1, 2, 3, 4\}$ being another one.

Our goal is to reduce satisfiability in $SC(Th)$ to satisfiability in Th extended with an uninterpreted sort for sets and an uninterpreted binary predicate $\dot{\in}$, which encodes set membership. To achieve this, we translate a set formula of interest S in $SC(Th)$ into a formula $[\![S]\!]$ of this resulting theory, which we denote by $U{+}Th$ ($\underline{Th}$ with $\underline{U}$ninterpreted symbols). Specifically, each appearance of a set comprehension or of a standard set operation in S, is mapped to a fresh variable together with formulas in the language of $U{+}Th$. These formulas ensure that this variable is associated with the exact elements of the original set structure. Then, we verify that S is satisfiable in $SC(Th)$ by checking that $[\![S]\!]$ has a model $\mathcal{M}$ in $U{+}Th$, i.e., that $\mathcal{M} \models_{U+Th} [\![S]\!]$. For a wide range of formulas, this verification step can be carried out using a state-of-the-art high-performance SMT solver such as Z3 [1]. Finally, we lift $\mathcal{M}$ to a model $\mathcal{S}$ of S, such that $\mathcal{S} \models_{SC(Th)} S$. Even though this operation is still undecidable in general [2], it is nonetheless decidable for many instances of $SC(LIA)$ formulas and can be implemented as an effective satisfiability test operation. For instance, our work in [3] uses a conservative implementation of this test operation to determine satisfiability of finite set comprehension formulas that would guarantee safety of certain compiler optimizations.

To illustrate this technique, here is the (slightly simplified) encoding $[\![S]\!]$ where S is the formula from our earlier example:

$$[\![S]\!] \;=\; \begin{cases} \forall z.\ z \mathrel{\dot\in} X_2 \leftrightarrow z \mathrel{\dot\in} X_3 & - \ F_1 : X_2 = X_3 \\ \forall y.\ y \mathrel{\dot\in} X_2 \leftrightarrow (y \doteq 10 \vee y \doteq 20 \vee y \doteq 30) & - \ F_2 : X_2 = \{10, 20, 30\} \\ \forall x.\ (x * 10 \mathrel{\dot\in} X_3) \leftrightarrow (x \mathrel{\dot\in} X \wedge x < 4) & - \ F_3 : X_3 = \{x * 10 \mid x < 4\}_{x \dot\in X} \end{cases}$$

The first formula, F_1, states that X_2 and X_3 are extensionally equal. The second formula, F_2, constrains the members of X_2 to be the integers 10, 20 or 30. Hence X_2 is a precise representation of the extensional set $\{10, 20, 30\}$ in $U{+}LIA$. The third formula, F_3, restricts X_3 so that, for each element x in X that is larger than 4, $x * 10$ must be a member of X_3. Hence F_3 constrains the behavior of X_3 to that of $\{x * 10 \mid x < 4\}_{x \dot\in X}$ in $U{+}LIA$.

The rest of the paper is organized as follows: Section 2 introduces our source and target languages, $SC(LIA)$ and $U{+}LIA$. Section 3 defines the encoding of $SC(LIA)$ formulas into $U{+}LIA$ formulas. Section 4 presents a conservative satisfiability test operation for $SC(LIA)$ formulas. Section 5 describes our prototype implementation of this operation, a lightweight Python library that extends Z3. Section 6 discusses our conclusions and future work.

2 Notations, Term Languages and Models

In this section, we introduce meta-notation that we will use throughout this paper and we define the source and target term languages $SC(LIA)$ and $U{+}LIA$.

In discussions, we write $\bar{o}$ to denote finite sets of syntactic objects o, with $\emptyset$ for the empty set. We write $\{\bar{o}, o\}$ for the extension of set $\bar{o}$ with syntactic object o, omitting the brackets when no ambiguity arises. Meta-level set membership is denoted as $o \in \bar{o}$. We write meta-level set comprehension as $\{o : \bar{o} \mid \Phi(o)\}$, which represents the set containing all objects o from $\bar{o}$ that satisfy the condition $\Phi(o)$. We write $[o'/x]o$ for the simultaneous replacement within object o of all occurrences of variable x with o'. When traversing binding constructs and quantifiers, substitutions implicitly α-rename variables to avoid capture. These are all meta-level notations, not to be confused with the syntactic objects in our source language, which also features sets.

Figure 1 defines our source language $SC(LIA)$ and the target language $U{+}LIA$. Terms in $SC(LIA)$ are either arithmetic expressions, written t, or set expressions, denoted s. Arithmetic

Variables x, X Values v

$SC(LIA)$: Set Comprehensions over Linear Integer Arithmetic

Arithmetic Term	t	$::= x \mid v \mid t \ op \ t$
Arithmetic Formula	T	$::= t \doteq t \mid t < t \mid \neg T \mid T \wedge T$
Set Term	s	$::= X \mid \{\bar{t}\} \mid \{t \mid T\}_{x \dot{\in} s} \mid s \cup s \mid s \cap s \mid s \setminus s$
Set Formula	S	$::= t \dot{\in} s \mid s \doteq s \mid s \subseteq s \mid \neg S \mid S \wedge S$

U+LIA: Linear Integer Arithmetic and Uninterpreted Sets

Arithmetic Term	t	$::= x \mid v \mid t \ op \ t$
Arithmetic Formula	T	$::= t \doteq t \mid t < t$
Uninterpreted Set Term	s	$::= X$
Uninterpreted Set Formula	S	$::= t \dot{\in} s$
Formula	F, C	$::= S \mid T \mid \neg F \mid F \wedge F \mid \exists x.F \mid \forall x.F$

Figure 1: Object Languages: $SC(LIA)$ and U+LIA

expressions correspond to integers and set expressions to sets of integers. An arithmetic expression is either a base variable x, a number v in $\mathbb{Z}$, or a binary arithmetic operation $t \ op \ t$ supported in the theory of linear integer arithmetic (e.g., $+$ or $*$). A set expression is either a set variable X, an extensional set $\{\bar{t}\}$, the union, intersection or difference between two sets, or a set comprehension $\{t \mid T\}_{x \dot{\in} s}$. The empty set $\{\}$ is the special case of an extensional set $\{\bar{t}\}$ where $\bar{t}$ is empty. In a set comprehension, we refer to t as the *comprehension pattern*, to T as the *comprehension guard*, to x as the *binding variable*, and to s as the *comprehension domain*. The scope of the binding variable x is t and T. Where useful for clarity, we explicitly annotate this dependency by writing t and T as $t(x)$ and $T(x)$. Formulas T that appear within set comprehensions are quantifier-free arithmetic formulas over the integer domain. Atomic arithmetic formulas include equality, written $t_1 \doteq t_2$, and other standard predicates, for example $t_1 < t_2$. The arithmetic formulas T in comprehension guards combine them using the standard Boolean connectives — we display a minimal set consisting of $\neg$ and $\wedge$ in Figure 1 but will freely use the full set of Boolean connectives in our examples. The set formulas of $SC(LIA)$, denoted S, are the Boolean combination of set membership $t \dot{\in} s$, set equality $s_1 \doteq s_2$ and the subset relation $s_1 \subseteq s_2$.

The language of U+LIA resembles $SC(LIA)$, with two major syntactic differences: first, it dispenses with all set expressions except for set variables X; second, U+LIA formulas allow the use of quantifiers over base variables.

A model is a structure $\mathcal{M}$ that satisfies the axioms of a theory Th and a formula F. This is denoted by $\mathcal{M} \models_{Th} F$. A model $\mathcal{M}$ contains mappings from the variables of F to their respective instantiations that satisfy F in Th. The lookup operation $\mathcal{M}(x)$ denotes the instantiated value that x is mapped to. We write $\mathcal{M}_{x \mapsto v}$ for the model that maps x to v and is otherwise identical to $\mathcal{M}$. For each predicate symbol p that appears in Th or F, the model $\mathcal{M}$ also contains mappings from p to the set of all valid instances of the predicate relation. The lookup operation $\mathcal{M}(p)$ denotes the set of valid relation instances of p. For simplicity, we assume predicates have the same arity throughout Th and F. Interpretations of the (sub)formula(s) of F and terms t that appear in F are obtained from $\mathcal{M}$ by the operations $\mathcal{M}[\![F]\!]$ and $\mathcal{M}[\![t]\!]$. Unsatisfiability of a formula F is denoted by $\not\models_{Th} F$.

A model in our source language $SC(LIA)$ is denoted by $\mathcal{S}$ and its satisfiability judgment is

denoted by $\mathcal{S} \models_{SC(LIA)} S$ where S is an $SC(LIA)$ formula. We require that $SC(LIA)$ contain the theory of linear integer arithmetic and general set theory. A model $\mathcal{S}$ contains mappings for integer variables x to integer values v and set variables X to extensional sets $\{\bar{t}\}$. The sets of all integers and sets that appear in $\mathcal{S}$ are denoted by $dom_{\mathbb{Z}}(\mathcal{S})$ and $dom_{\mathbb{S}}(\mathcal{S})$ respectively. We will omit standard definitions of integer term interpretation $\mathcal{S}[\![t]\!]$ and integer formula satisfiability $\mathcal{S} \models_{SC(LIA)} T$. The interpretation $\mathcal{S}[\![s]\!]$ of set terms s is defined as follows:

$$\begin{array}{rcl}
\mathcal{S}[\![X]\!] & = & \mathcal{S}(X) \\
\mathcal{S}[\![\{\bar{t}, t\}]\!] & = & \{\mathcal{S}[\![\bar{t}]\!], \mathcal{S}[\![t]\!]\} \\
\mathcal{S}[\![\{\}]\!] & = & \emptyset \\
\mathcal{S}[\![\{t(x) \mid T(x)\}_{x \dot{\in} s}]\!] & = & \{\mathcal{S}_{x \mapsto a}[\![t(x)]\!] : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s]\!] \text{ and } \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} T(x)\} \\
\mathcal{S}[\![s_1 \cup s_2]\!] & = & \{a : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s_1]\!] \text{ or } a \in \mathcal{S}[\![s_2]\!]\} \\
\mathcal{S}[\![s_1 \cap s_2]\!] & = & \{a : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s_1]\!] \text{ and } a \in \mathcal{S}[\![s_2]\!]\} \\
\mathcal{S}[\![s_1 \setminus s_2]\!] & = & \{a : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s_1]\!] \text{ and } a \notin \mathcal{S}[\![s_2]\!]\}
\end{array}$$

This definition is inductive and makes use of $\mathcal{S}[\![t]\!]$ and $\mathcal{S} \models_{SC(LIA)} T$. The interpretation of a set comprehension $\mathcal{S}[\![\{t(x) \mid T(x)\}_{x \dot{\in} s}]\!]$ is defined as the set of all (and only) the interpretations of the comprehension pattern $t(x)$ with the binding variable x mapped to the integers a in the interpretation of the comprehension domain s such that the corresponding instance of the comprehension guard $T(x)$ is satisfiable under $\mathcal{S}$. The other cases are standard.

Satisfiability for set formulas, $\mathcal{S} \models_{SC(LIA)} S$, is defined inductively over set formulas and on top of $\mathcal{S}[\![t]\!]$, $\mathcal{S}[\![S]\!]$ and $\mathcal{S} \models_{SC(LIA)} T$. The key cases of this definition are as follows:

$$\begin{array}{lll}
\mathcal{S} \models_{SC(LIA)} t \dot{\in} s & \textit{iff} & \mathcal{S}[\![t]\!] \in \mathcal{S}[\![s]\!] \\
\mathcal{S} \models_{SC(LIA)} s_1 \doteq s_2 & \textit{iff} & \text{for all } a \in dom_{\mathbb{Z}}(\mathcal{S}), \\
 & & \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_1 \text{ iff } \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_2 \\
\mathcal{S} \models_{SC(LIA)} s_1 \subseteq s_2 & \textit{iff} & \text{for all } a \in dom_{\mathbb{Z}}(\mathcal{S}), \\
 & & \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_1 \text{ only if } \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_2
\end{array}$$

We omitted the cases for the logical connectives whose interpretations are standard. Note that membership $\dot{\in}$ is an interpreted predicate symbol in *U+LIA*: $t \dot{\in} s$ is satisfiable in $\mathcal{S}$ if and only if the interpretation of t is actually a member of the interpretation of s. Since $\dot{\in}$ is interpreted, $\mathcal{S}$ does not contain any mappings for $\dot{\in}$ (i.e., $\mathcal{S}(\dot{\in}) = \bot$). Satisfiability of equality, $s_1 \doteq s_2$ is defined in terms of the membership relation: s_1 and s_2 are equal if and only if they contain the same integers. In a similar manner, satisfiability of the subset relation $s_1 \subseteq s_2$ is defined in terms of membership $\dot{\in}$.

Models of our target language *U+LIA* are denoted by $\mathcal{M}$, and the satisfiability judgment is denoted by $\mathcal{M} \models_{U+LIA} F$. We require that *U+LIA* contains the theory of linear integer arithmetic and an uninterpreted domain for sets. Unlike $\mathcal{S}$, $\mathcal{M}$ contains no mappings for set variables, since the set domain is uninterpreted. However, $\mathcal{M}(\dot{\in})$ maps to the set of pairs $\langle \mathcal{M}[\![t]\!], X \rangle$ for every valid relation $t \dot{\in} X$ from F. The definition of $\mathcal{M}[\![t]\!]$ and $\mathcal{M} \models_{U+LIA} F$ are standard and therefore omitted.

3 Encoding $SC(LIA)$ into U+LIA

In this section, we define an encoding of $SC(LIA)$ terms and formulas into *U+LIA* terms and formulas. This encoding is given by two translation functions, the set term encoding $\lceil\!\cdot\!\rceil^{set}$ and set formula encoding functions $\lceil\!\cdot\!\rceil^{form}$.

$$
\begin{aligned}
[\![X]\!]^{set} &= X \ \rhd\ \emptyset \ ;\ \top \\
[\![\{\bar{t}\}]\!]^{set} &= \begin{cases} X \ \rhd\ \{X\} \ ;\ \forall x.\, x \mathrel{\dot\in} X \leftrightarrow mem(\bar{t}, x) \\ \quad \text{where} \quad mem(\{\bar{t}, t\}, x) \ = \ x \doteq [\![t]\!] \vee mem(\bar{t}, x) \\ \qquad\qquad mem(\emptyset, x) \ = \ \bot \end{cases} \\
[\![s_1 \ op \ s_2]\!]^{set} &= \begin{cases} X \ \rhd\ \mathcal{V}_1, \mathcal{V}_2, X \ ;\ C_1 \wedge C_2 \wedge constr(op) \\ \quad \text{where} \quad [\![s_1]\!]^{set} = xs_1 \ \rhd\ \mathcal{V}_1 \ ;\ C_1 \\ \qquad\qquad [\![s_2]\!]^{set} = xs_2 \ \rhd\ \mathcal{V}_2 \ ;\ C_2 \\ \qquad\qquad constr(\cup) = \forall x.\, x \mathrel{\dot\in} X \leftrightarrow x \mathrel{\dot\in} X_1 \vee x \mathrel{\dot\in} X_2 \\ \qquad\qquad constr(\cap) = \forall x.\, x \mathrel{\dot\in} X \leftrightarrow x \mathrel{\dot\in} X_1 \wedge x \mathrel{\dot\in} X_2 \\ \qquad\qquad constr(\setminus) = \forall x.\, x \mathrel{\dot\in} X \leftrightarrow x \mathrel{\dot\in} X_1 \wedge \neg(x \mathrel{\dot\in} X_2) \end{cases} \\
[\![\{t \mid T\}_{x \mathrel{\dot\in} s}]\!]^{set} &= \begin{cases} X \ \rhd\ \mathcal{V}, X \ ;\ C_{dom} \wedge C_{max} \wedge C_{rg} \\ \quad \text{where} \quad [\![s]\!]^{set} \ = \ X' \ \rhd\ \mathcal{V} \ ;\ C_{dom} \\ \qquad\qquad C_{max} \ = \ \forall x.\, (x \mathrel{\dot\in} X' \wedge [\![T]\!]) \rightarrow [\![t]\!] \mathrel{\dot\in} X \\ \qquad\qquad C_{rg} \ = \ \forall z.\, z \mathrel{\dot\in} X \rightarrow \exists x.\, (z \doteq [\![t]\!] \wedge x \mathrel{\dot\in} X' \wedge [\![T]\!]) \end{cases} \\
& \qquad\qquad [\![t]\!] = t \qquad\qquad [\![T]\!] = T
\end{aligned}
$$

Figure 2: Set Term Encoding into *U+LIA*: $[\![s]\!]^{set} = X \ \rhd\ \mathcal{V} \ ;\ C$

The *set term encoding* of a set expression s is given by a triple $(X, \mathcal{V}, C)$ that we write $[\![s]\!]^{term} = X \ \rhd\ \mathcal{V} \ ;\ C$. Here, X is a fresh variable associated with s (unless s is already a variable), $\mathcal{V}$ is a collection of uninterpreted set variables, and C is a *U+LIA* formula. We refer to X as the *representative variable* of s, C as the *extensional context formula* and $\mathcal{V}$ as the *set variable signature*. The context formula C constrains X to capture the extensional behavior of s, while $\mathcal{V}$ contains all local set variables in C created while encoding s. We require that $\mathcal{V}$ contain no duplicates and we rely on implicit α-renaming to enforce this constraint. This provides an implicit mechanism to guarantee unique assignments of set variables during encoding.

Figure 2 defines the set term encoding operation. In the case of an extensional set $\bar{t} = \{t_1, \ldots, t_n\}$, the extensional context formula $\forall x.\, x \mathrel{\dot\in} X \leftrightarrow (x \doteq t_1 \vee \ldots \vee x \doteq t_n)$ constrains representative variable X to contain all and only the elements that appear in $\bar{t}$. We encode all $SC(LIA)$ terms t_i in $\bar{t}$ by means of the encoding operation $[\![t_i]\!]$. For the element terms we consider in this paper, it is sufficient to define their encoding as the identity function. The entry $[\![s_1 \ op \ s_2]\!]^{set}$ covers the cases for set union, intersection and difference. For each, we first derive encodings for s_1 and s_2 to obtain representative variables X_1 and X_2. The variable X is assigned to represent the set $s_1 \ op \ s_2$. This is done with the help of the context formula $constr(op)$: If op is the union operator $\cup$, the context formula state that any member x of X must be a member of either X_1 or X_2 and vice-versa. The cases where op is $\cap$ and $\setminus$ are similar. Each of these context formulas constrains the extensional behavior X to that of the union, intersection or set difference of s_1 and s_2 respectively. Finally, $[\![\{t \mid T\}_{x \mathrel{\dot\in} s}]\!]^{set}$ encodes the set comprehension as follows: The comprehension domain s is encoded into X' with the constraints of s captured by C_{dom}. The comprehension $\{t \mid T\}_{x \mathrel{\dot\in} s}$ is encoded into X with its constraints captured by context formulas C_{max} and C_{rg}. The *comprehension maximality* condition C_{max} imposes the constraint that every member x of the comprehension domain X'

$$
\begin{aligned}
[\![t \mathrel{\dot\in} s]\!]^{form} &= [\![t]\!] \mathrel{\dot\in} X \ \rhd\ \mathcal{V}\ ;\ C \qquad \text{where} \quad [\![s]\!]^{set} = X \ \rhd\ \mathcal{V}\ ;\ C \\
[\![s_1 \doteq s_2]\!]^{form} &= \begin{cases} \forall x.\ (x \mathrel{\dot\in} X_1) \leftrightarrow (x \mathrel{\dot\in} X_2) \ \rhd\ \mathcal{V}_1, \mathcal{V}_2\ ;\ C_1 \wedge C_2 \\ \quad \text{where} \quad [\![s_1]\!]^{set} = X_1 \ \rhd\ \mathcal{V}_1\ ;\ C_1 \\ \qquad\quad\ [\![s_2]\!]^{set} = X_2 \ \rhd\ \mathcal{V}_2\ ;\ C_2 \end{cases} \\
[\![s_1 \subseteq s_2]\!]^{form} &= \begin{cases} \forall x.\ (x \mathrel{\dot\in} X_1) \rightarrow (x \mathrel{\dot\in} X_2) \ \rhd\ \mathcal{V}_1, \mathcal{V}_2\ ;\ C_1 \wedge C_2 \\ \quad \text{where} \quad [\![s_1]\!]^{set} = X_1 \ \rhd\ \mathcal{V}_1\ ;\ C_1 \\ \qquad\quad\ [\![s_2]\!]^{set} = X_2 \ \rhd\ \mathcal{V}_2\ ;\ C_2 \end{cases} \\
[\![S_1 \wedge S_2]\!]^{form} &= \begin{cases} F_1 \wedge F_2 \ \rhd\ \mathcal{V}_1, \mathcal{V}_2\ ;\ C_1 \wedge C_2 \\ \quad \text{where} \quad [\![S_1]\!]^{form} = F_1 \ \rhd\ \mathcal{V}_1\ ;\ C_1 \\ \qquad\quad\ [\![S_2]\!]^{form} = F_2 \ \rhd\ \mathcal{V}_2\ ;\ C_2 \end{cases} \\
[\![\neg S]\!]^{form} &= \neg F \ \rhd\ \mathcal{V}\ ;\ C \qquad \text{where} \qquad [\![S]\!]^{form} = F \ \rhd\ \mathcal{V}\ ;\ C
\end{aligned}
$$

Figure 3: Set Formula Encoding into *U+LIA*: $[\![S]\!]^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$

that satisfies the comprehension guard T has its corresponding comprehension pattern t as a member of X. The *comprehension range restriction* C_{rg} imposes the constraint that for every member z of the set comprehension X, there exists some member x of the comprehension domain X' whose corresponding comprehension pattern t is equal to z and satisfies the comprehension guard T.

While it is tempting to replace the context formulas C_{max} and C_{rg} with the single implication formula $C_{comp} = \forall x.\ (x \mathrel{\dot\in} X' \wedge T(x)) \leftrightarrow t(x) \mathrel{\dot\in} X$ (as done in our example in Section 1), C_{comp} would not accurately capture the behavior of set comprehensions with expressions $t(x)$ that are not injective functions. For instance, consider the comprehension $s_1 = \{x\%3 \mid \top\}_{x \mathrel{\dot\in} s_2}$, where % is the modulus operator. Assuming that X and X' are the representatives of set comprehension s_1 and set term s_2 respectively, the formula $C_{comp} = \forall x.\, x \mathrel{\dot\in} X' \leftrightarrow (x\%3) \mathrel{\dot\in} X$ is incorrect in the '$\leftarrow$' case: it demands any x such that $x\%3$ is a member of X must be a member of X', which is clearly not the behavior of the set comprehension. The combination of C_{max} and C_{rg} on the other hand, is sound as long as $t(x)$ is a total function.

Figure 3 defines the encoding of a set formula S, denoted by $[\![S]\!]^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$. It encodes a $SC(LIA)$ formula S into an *U+LIA* formula F under the extensional context formula C with the set variable signatures $\mathcal{V}$. We call F the *main encoding formula*. Similarly to the term encoding operation, the formula C specifies the constraints on uninterpreted set variables and $\mathcal{V}$ is the set of local set variables created by the encoding. The case $[\![t \mathrel{\dot\in} s]\!]^{form}$ encodes the membership relation by simply using the term encoding $[\![s]\!]^{set}$ of s. We then simply check that the encoding of t is in the representative variable X of $[\![s]\!]^{set}$. For $[\![s_1 \doteq s_2]\!]^{form}$, we encode s_1 and s_2 by applying the term encoding operation of Figure 3 to each, obtaining X_1 and X_2 as representatives of s_1 and s_2 respectively with extensional context C_1 and C_2. Following this, the equality constraint is represented in *U+LIA* as the formula $\forall x.\, x \mathrel{\dot\in} X_1 \leftrightarrow x \mathrel{\dot\in} X_2$, stating that all members of X_1 are members of X_2 and vice-versa. Encoding set equality in this manner exercises the axiom of extensionality of set theory, enforcing the extensional equality of sets with respect to the membership relation ($\dot\in$). The case $[\![s_1 \subseteq s_2]\!]^{form}$ is similar, except that it is encoded into a one-sided extensional membership 'equality' ($\rightarrow$). The case $[\![S_1 \wedge S_2]\!]^{form}$ defines the cases for the logical connectives $\wedge$, during which we traverse the respective sub-formulas and terms containing set terms and derive their encodings. The main formula is simply the $\wedge$ logical connective applied to the respective encoded formulas.

$$satCheck(S) = \begin{cases} \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor & \text{if } \lceil\!\lceil S \rceil\!\rceil^{form} = F \ \rhd\ \mathcal{V}\ ;\ C \text{ and } \mathcal{M} \models_{U+LIA} C \wedge F \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{array}{lrll} \text{where} & \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(x) & = & \mathcal{M}(x) \\ & \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(X) & = & \begin{cases} \{a : dom_{\mathbb{Z}}(\mathcal{M}) \mid \langle a, X\rangle \in \mathcal{M}(\dot\in)\} & \text{if } X \notin \mathcal{V} \\ \bot & \text{otherwise} \end{cases} \\ & \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(\dot\in) & = & \bot \end{array}$$

Figure 4: Satisfiability Checking for $SC(LIA)$ Formulas

The encoding of negation requires some care, and is the reason for separating the main encoding formula F and the extensional context formula C. Our definition in Figure 3 encodes the negated $SC(LIA)$ formula $\neg(\{1\} \subseteq \{1,2\})$ as follows:

$$\lceil\!\lceil \neg(\{1\} \subseteq \{1,2\}) \rceil\!\rceil^{form} = \neg(\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \ \rhd\ X_1, X_2\ ;\ X_1 = \{1\} \wedge X_2 = \{1,2\}$$

Here, we abbreviated the actual encoding of the extensional sets as $X_1 = \{1\}$ and $X_2 = \{1,2\}$ for to be succinct. The conjunction of the main formula and context formula (i.e., $\neg F \wedge C$) concisely captures the negated formula $\neg(\{1\} \subseteq \{1,2\})$, namely:

$$\neg(\{1\} \subseteq \{1,2\}) \quad \equiv \quad \neg(\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \wedge X_1 = \{1\} \wedge X_2 = \{1,2\}$$

Assume we had instead collapsed F and C into a single formula component, so that $\lceil\!\lceil S \rceil\!\rceil^{form} = F \wedge C \rhd \mathcal{V}$. Then we would have to translate the above formula as

$$\neg((\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \wedge X_1 \doteq \{1\} \wedge X_2 \doteq \{1,2\}) \rhd X_1, X_2$$

which is not the intended meaning of the negation as, propagating the negation,

$$\neg(\{1\} \subseteq \{1,2\}) \quad \not\equiv \quad \neg(\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \vee X_1 \neq \{1\} \vee X_2 \neq \{1,2\}$$

4 Satisfiability Testing for $SC(LIA)$

In this section, we define a satisfiability test operation of $SC(LIA)$ formulas on the encoding defined in the previous section. This operation, denoted by $satCheck(S)$, is defined in Figure 4. If the $SC(LIA)$ set formula S is satisfiable, this operation extracts a model for S from a model of its encoding $\lceil\!\lceil S \rceil\!\rceil^{form}$ in *U+LIA*. Such a model of $\lceil\!\lceil S \rceil\!\rceil^{form}$ can be obtained using an off-the-shelf SMT solver that supports the built-in theory of linear integer arithmetic and uninterpreted domains. More specifically, given $\lceil\!\lceil S \rceil\!\rceil^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$, the satisfiability of S is determined by checking the satisfiability of its *U+LIA* encoding: i.e., $\mathcal{M} \models_{U+LIA} C \wedge F$. Decidability of this operation therefore depends on the decidability of $\models_{U+LIA}$. If S is determined to be unsatisfiable, $satCheck(S)$ returns $\bot$. If S is determined to be satisfiable, it returns a *decoded model* $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$, which is an $SC(LIA)$ model inferred from $\mathcal{M}$. Since this operation is undecidable in general, it may not return a value at all for some formulas. The decode operator $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ is defined as follows: for integer variables x, $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$ simply maps x to $\mathcal{M}(x)$. For a set variable X however, $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(X)$ returns the set of all integers a such that $\langle a, X\rangle \in \mathcal{M}(\dot\in)$. The membership relation of $\mathcal{M}$ is stripped away in $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$, i.e., $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(\dot\in) = \bot$.

Lemma 1 determines the soundness of the term encoding: for any set term s, we can extract a corresponding interpretation of s from its encoding. Soundness for the formula encoding is

given by Theorem 2. It states that for some $SC(LIA)$ formula S with the encoding $\llbracket S \rrbracket^{form} = F \ \triangleright \ \mathcal{V} \ ; \ C$, if S is satisfiable under $SC(LIA)$, then $C \wedge F$ is satisfiable under U+LIA.

Lemma 1 (Soundness of the Term Encoding)**.** *Let s be a $SC(LIA)$ set term and $\llbracket s \rrbracket^{term} = X \ \triangleright \ \mathcal{V} \ ; \ C$ its encoding in U+LIA. For any $\mathcal{S}$ that contains an interpretation of s (i.e., $\mathcal{S}\llbracket s \rrbracket$), there exists $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C$ and $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$.*

Proof. The proof proceeds by structural induction on the encoding operation of the term s, with base cases X and $\{\bar{t}\}$. For the base case $\{\bar{t}\}$, we can infer that X contains all and only the values in $\bar{t}$ for the membership relation constraints ($x \ \dot{\in} \ X$, such that $x \in \bar{t}$) imposed by the context formula of the encoding of $\{\bar{t}\}$ (i.e., $\mathcal{S}\llbracket s \rrbracket = \bar{t}$). Hence, by definition of $\lfloor\!\lfloor . \rfloor\!\rfloor$, we show that $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$. For the inductive case $s_1 \ op \ s_2$ and $\{t \mid T\}_{x \dot{\in} s}$, we similarly show $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$ by observing that their respective interpretations in $\mathcal{S}$ correspond to sets built from the membership relations ($x \ \dot{\in} \ X$) captured by C. □

Theorem 2 (Soundness of the Formula Encoding)**.** *Let S be a $SC(LIA)$ formula S and $\llbracket S \rrbracket^{form} = F \ \triangleright \ \mathcal{V} \ ; \ C$ its encoding in U+LIA. For any $\mathcal{S}$ such that $\mathcal{S} \models_{SC(LIA)} S$, there exists $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C \wedge F$ and $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$.*

Proof. The proof proceeds by structural induction on the encoding operation of the formula S, with base cases $t \ \dot{\in} \ s$, $s_1 \doteq s_2$ and $s_1 \subseteq s_2$, whose proofs follow from the soundness of term encoding (Lemma 1). □

Lemma 3 and Theorem 4 state the converse of Lemma 1 and Theorem 2 respectively, hence providing a completeness guarantee for our encoding.

Lemma 3 (Completeness of the Term Encoding)**.** *Let s be a $SC(LIA)$ set term and $\llbracket s \rrbracket^{term} = X \ \triangleright \ \mathcal{V} \ ; \ C$ its encoding in U+LIA. For any $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C$, we have that $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor \models_{SC(LIA)} X \doteq s$.*

Proof. This proof is similar to that of Lemma 1. It proceeds by structural induction on the encoding of s. The main difference is that in each case, we show that interpretations of s obtained from $\mathcal{S}$ (i.e., $\mathcal{S}(X)$) corresponds to that of obtained from $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$, hence $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor = \mathcal{S}$. □

Theorem 4 (Completeness of the Formula Encoding)**.** *Let S be a $SC(LIA)$ formula and $\llbracket S \rrbracket^{form} = F \ \triangleright \ \mathcal{V} \ ; \ C$ its encoding in U+LIA. For all $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C \wedge F$, we have that $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor \models_{SC(LIA)} S$.*

Proof. Like the proof of Theorem 2, this proof proceeds by structural induction on the encoding of formula S. The proof for the base cases follows from Lemma 3. □

Corollary 1 states the soundness and completeness of the satisfiability test operation *satCheck*, and it follows from properties 1, 2, 3 and 4.

Corollary 1 (Soundness and Completeness of *satCheck*)**.** *For any $SC(LIA)$ formula S, we have the following:*

- *$satCheck(S) = \mathcal{S}$ if and only if $\mathcal{S} \models_{SC(LIA)} S$.*
- *$satCheck(S) = \bot$ if and only if $\not\models_{SC(LIA)} S$.*

5 Implementation

We implemented the above technique as a lightweight library with Z3 as the underlying SMT solver. This prototype is available for download at `https://github.com/sllam/pysetcomp`. Given an $SC(Th)$ formula, where Th is some theory that the Z3 SMT solver supports, our Python library implements the encoding operation $[\![S]\!]^{form} = F \ \triangleright \ \mathcal{V} \ ; \ C$. Formulas F and C are native Z3 formulas, hence $\mathcal{M} \models_{U+Th} F \wedge C$ is implemented by simply passing $F \wedge C$ to the Z3 satisfiability checking interfaces. Our prototype also includes a simple combinator library that provides the programmer with a convenient way to write $SC(Th)$ formulas.

The code that implement our example in Section 1 is as follows:

```
1  ## Initialize sorts and variables
2  I = z3.IntSort()
3  IntSet = mkSetSort( I )
4  x = z3.Int('x')
5  X = z3.Const('X', IntSet)
6
7  ## {10,20,30} ≐ { x*10 | x<4 }_{x ∈̇ X}
8  S = VSet(I,10,20,30) |Eq| Compre(x*10,x<4,x,X)
9
10 ## [[S]]^form = F ▷ V ; C
11 F, C, V = transForm( S )
12
13 ## M |=_{U+Th} F ∧ C
14 s = z3.Solver()
15 s.add(z3.And([F]+C))
16 print s.check()     -- Prints 'sat'
```

Z3's built-in operations are explicitly prepended by `z3`. Lines 2–5 initializes the Z3 sorts (data domains) and variables that we need in this example: `I` abbreviates the Z3 integer sort, `mkSetSort(I)` returns a representation of the sort of sets of integer. In line 4, a Z3 integer variable `x` is declared, while in line 5 a Z3 variable of the sort of sets of integer is declared. Line 8 implements the actual $SC(LIA)$ formula: `VSet(I,10,20,30)` builds the extensional set $\{$`10,20,30`$\}$, while `Compre(x*10,x<4,x,X)` builds the set comprehension $\{x * 10 \mid x < 4\}_{x \dot{\in} X}$. The infix operator `Eq` corresponds to the equality predicate $\doteq$. Line 11 implements the encoding of `S` into *U+LIA* formulas `F` and `C`, with the set of local variables `V` created by the encoding procedure. Finally, lines 14–15 implements the satisfiability test by feeding `F` $\wedge$ `C` to the Z3 satisfiability checker.

Our prototype works on more than just integers. The following example involves tuples of integers.

$$X \doteq \{\langle 1,8\rangle, \langle 2,5\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 4,8\rangle\} \wedge Y \doteq \{x * 10 \mid x \leq 3\}_{\langle x,y\rangle \dot{\in} X} \wedge x * 10 \dot{\in} Y \wedge \langle x,y\rangle \dot{\in} X$$

The corresponding code in our implementation is as follows:

```
1  IntPair = mkTupleSort( I, I )
2  tup = IntPair.tup
3  pi1 = IntPair.pi1
4
5  PairSet = mkSetSort( IntPair )
6  IntSet = mkSetSort( I )
7
8  X = z3.Const('X', PairSet)
```

```
9  Y = z3.Const('Y', IntSet)
10 xy = z3.Const('xy', IntPair)
11 x, y = z3.Ints('x y')
12
13 cs = [X |Eq| VSet(IntPair, tup(1,8), tup(2,5), tup(3,2), tup(3,4), tup(4,8))
14       ,Y |Eq| Compre(pi1(xy)*10, pi1(xy)<=3, xy, X)
15       ,(x * 10) |In| Y
16       ,tup(x,y) |In| X]
```

In this example, the set comprehension Y maps a set containing pairs of integers, into a set of integers, thus requiring our system to handle many-sorted encodings. Lines 1–3 define a new tuple sort of pairs of integers `IntPair`, with `tup` as its data constructor and `pi1` as a projection operator for the left tuple argument. Lines 5–6 declare new set sorts: `PairSet` is the sort of sets of integer pairs, while `IntSet` the sort of sets of integers. Lines 8–11 declare the variables of the respective sorts, while lines 13–16 define the actual formula. The infix binary relation `In` implements the membership relation $\dot{\in}$. Encoding `cs` with the `transForm` operation and feeding its output to the Z3 solver yields a satisfiable result, where a satisfiable instance of x and y can be extracted.

The implementation of the `In` operator is non-trivial and requires explanation. A call to `x |In| X` creates an instance of a Z3 function, named *mem*, that is interpreted by our encoding as the binary relation $x \dot{\in} X$ in Z3: this function maps the pair `x` and `X` to a Z3 Boolean value. Note that the two instances of `In` at lines 15–16 are of different sort: At line 15 we have an instance with the sort `Int*IntSet`, while at line 16 we have `IntPair*PairSet`. Since Z3 does not support parametric data types, our implementation must therefore map each instance to a different Z3 function symbol (here *mem_IntSet* and *mem_PairSet*) that are interpreted appropriately by our Python library. In order to provide a convenient interface that hides this mapping, we implement a lookup procedure inside the `In` operator that is akin to a simplified run-time version of type dictionary passing during type checking of Haskell type classes.

We have tested our implementation on a suite of set comprehension formulas of varying complexity. While most are rather small, in the future we intend to provide empirical results on more practical examples.

6 Conclusion and Future Work

In this paper, we reduced the satisfiability problem for formulas featuring comprehension and other set operations over a standard theory (linear integer arithmetic in our examples) to solving satisfiability constraints over this same theory augmented with a single uninterpreted sort. This technique allows the satisfiability of set-based formulas to be verified by a wide range of off-the-shelf SMT solvers that support the base theory. We have implemented a lightweight Python library that utilizes this encoding technique on the popular Z3 SMT solver. This implementation generalizes the encoding described here to a broad range of data types supported by Z3, which includes integer, real numbers, tuples, and finite lists.

In the future, we are interested in expanding our results to comprehensions over *multisets*. A commonly used representation for multisets relies on array maps, that associate each elements in the support domain to its multiplicity [5]. An adaptation of the technique presented in this paper that simply swaps our encoding of sets with this representation of multisets does not work however. To appreciate the added challenge, consider the task of verifying the following formula about multiset comprehensions:

$$M \quad = \quad X_1 \doteq \wr x\%3 \mid \top \int_{x \dot{\in} X_2}$$

where $\wr \cdot \int$ delimits multisets in the same way as $\{\cdot\}$ delimited sets in the rest of the paper. In the suggested encoding, the variables X_1 and X_2 would be implemented as array maps: for every element x in the domain, $X_1[x]$ gives us the number of times that x occurs in X_1 (its multiplicity), and similarly for X_2. A naive adaptation of our technique suggests the following encoding for the above formula:

$$[\![M]\!] = \begin{cases} \forall x, m.\ (X_2[x] = m \wedge m > 0) \rightarrow X_1[x\%3] = m \\ \forall z, m.\ (X_1[z] = m \wedge m > 0) \rightarrow \exists x.(z = x\%3 \wedge X_2[x] = m) \end{cases}$$

Although it would be adequate for injective operations on x (e.g., $x + 3$), this encoding fails for non-injective operations such as $x\%3$. For instance, the formula M is satisfied for $X_1 = \wr 0, 0, 0, 2 \int$ and $X_2 = \wr 3, 3, 6, 8 \int$, yet the above encoding yields an unsatisfiable formula. The problem is that we have multiple elements in X_2 that are mapped to the same element in X_1. Since $3\%3 = 0$ and $X_2[3] = 2$, the above encoding entails that $X_1[0] = 2$. However, $6\%3 = 0$ also, which means that, given that $X_2[6] = 1$, we have that $X_1[0] = 1$ too — an absurdity. We would need to combine these results by adding the multiplicities of all x that map to the same value in X_1: doing so for all x in X_2 such that $x\%3 = 0$ in this example would yield the expected result, $X_1[0] = 3$. This has proved to be non-trivial in Z3. A possible approach to reasoning about such arithmetic aggregate operations could be adapted from [4].

References

[1] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Joseph Y. Halpern. Presburger Arithmetic with Unary Predicates is Π^1_1 Complete. *Journal of Symbolic Logic*, 56:56–2, 1991.

[3] Edmund S. L. Lam and Iliano Cervesato. Constraint Handling Rules with Multiset Comprehension Patterns. In *11th Workshop on Constraint Handling Rules*, 2014.

[4] K. R. M. Leino and R. Monahan. Reasoning about Comprehensions with First-Order SMT Solvers. In *In Proc. of the 2009 ACM symposium on Applied Computing*, pages 615–622. ACM, 2009.

[5] R. Piskac and V. Kuncak. MUNCH — Automated Reasoner for Sets and Multisets. In *IJCAR'10*, volume 6173 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2010.

[6] P. Suter, R. Steiger, and V. Kuncak. Sets with Cardinality Constraints in Satisfiability Modulo Theories. In *Verification, Model Checking, and Abstract Interpretation*, pages 403–418. Springer, 2011.

Weakly Equivalent Arrays

Jürgen Christ and Jochen Hoenicke*

Department of Computer Science,
University of Freiburg
{christj,hoenicke}@informatik.uni-freiburg.de

Abstract

The (extensional) theory of arrays is widely used to model systems. Hence, efficient decision procedures are needed to model check such systems. Current decision procedures for the theory of arrays saturate the read-over-write and extensionality axioms originally proposed by McCarthy. Various filters are used to limit the number of axiom instantiations while preserving completeness. We present an algorithm that lazily instantiates lemmas based on *weak equivalence classes*. These lemmas are easier to interpolate as they only contain existing terms. We formally define weak equivalence and show correctness of the resulting decision procedure.

1 Introduction

Arrays are widely used to model parts of systems. In software model checking, for example, the heap of a program can be modelled by an array that represents the main memory. A software model checker using such a model can check for illegal accesses to memory or even memory leaks. While checking for illegal accesses can be done using only the axioms proposed by McCarthy, leak checking typically is done using extensionality. In this setting, extensionality is used to ensure that the memory after executing a program does not contain more allocated memory cells than it contained at the beginning of the program.

The theory of arrays was initially proposed by McCarthy [8]. It specifies two operations: (1) The store operation $a\langle i \lhd v\rangle$ creates a new array that stores at every index different from i the same value as array a and the value v at index i. (2) The select operation $a[i]$ retrieves the value of array a at position i. The theory is parametric in the index and element theories.

The store operation only modifies an array at one index. The values stored at other indices are not affected by this operation. Hence, the resulting array and the array used in the store operation are *weakly equal* in the sense that they differ only at finitely many indices. Current decision procedures do not fully exploit such dependencies between arrays. Instead, they use a series of instantiations of the axiom proposed by McCarthy to derive weak equivalences.

In this paper we present a new algorithm to decide the quantifier-free fragment of the theory of arrays. The decision procedure is based on the notion of *weak equivalence*, a property that combines equivalence reasoning with array dependencies. The new algorithm only produces a few new terms not present in the input formula during preprocessing. This is possible since the decision procedure does not instantiate the axiom proposed by McCarthy, but axioms derived from them.

*This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS)

Related Work Since the proposal of the theory of arrays by McCarthy [8] several decision procedures have been proposed. We can identify two basic branches: *rewrite-based* and *instantiation-based* techniques.

Armando et al. [1] used rewriting techniques to solve the theory of arrays. They showed how to construct simplification orderings to achieve completeness. The benchmarks used in this paper test specific properties of the array operators like commutativity of stores if the indices differ. While these benchmarks require a lot of instantiations of McCarthy's axioms, they are easy for the decision procedure presented in this paper since the properties tested by these benchmarks are properties satisfied by the weak equivalence relation presented in this paper.

Bruttomesso et al. [4] present a rewrite based decision procedure to reason about arrays. This approach exploits some key properties of the store operation that are also captured by the weak equivalence relation described in this paper. Contrary to our method, the rewrite based approach is not designed for Nelson–Oppen style theory combination and thus not easily integratable into an existing SMT solver. They extended the solver into an interpolating solver for computing quantifier-free interpolants. In contrast to our method their solver depends on the partitioning of the interpolation problem. We create a SMT proof without any knowledge of the partitioning and can use proof tree preserving interpolation [7], which only requires a procedure to interpolate the lemmas.

A decision procedure for the theory of arrays based on instantiating McCarthy's axioms is given by de Moura et al. [9]. The decision procedure saturates several rules that instantiate array axioms under certain conditions. Several filters are proposed to minimise the number of instantiations.

Closest to our work is the decision procedure published by Brummayer et al. [3]. Their decision procedure produces lemmas that can be derived from the axioms for the theory of arrays proposed by McCarthy. They consider the theory of arrays with bitvector indices and prove soundness and completeness of their approach in this setting. In contrast to our method, they do not allow free function symbols (i. e., the combination of the theory of arrays with the theory of uninterpreted function symbols) since they only consider a limited form of extensionality where the extensionality axiom is only instantiated for arrays a and b if the formula contains the literal $a \neq b$. We do not have this limitation, but add some requirements on the index theory that prevent the procedure presented in this paper from using the theory of bitvectors as index theory.

2 Notation

A first order theory consists of a signature Σ and a set of models $\mathbb{M}$. We assume the equality symbol $=$ with its usual interpretation is part of any signature. Every model contains for every sort interpreted by this model a non-empty domain and a mapping from constant or function symbol into the corresponding domain. A theory $\mathcal{T}$ is *stably infinite* if and only if every satisfiable quantifier-free formula is satisfied in a model of $\mathcal{T}$ with an infinite universe.

The theory of arrays $\mathcal{T}_A$ is parameterised by an index theory $\mathcal{T}_I$ and an element theory $\mathcal{T}_E$. The signature of $\mathcal{T}_A$ consists of the two functions $\cdot[\cdot]$ and $\cdot\langle\cdot \lhd \cdot\rangle$. Every model of the theory of arrays satisfies the select-over-store-axioms proposed by McCarthy [8]:

$$\forall a\, i\, v.\ a\langle i \lhd v\rangle[i] = v \qquad \text{(idx)}$$

$$\forall a\, i\, j\, v.\ i \neq j \implies a\langle i \lhd v\rangle[j] = a[j] \qquad \text{(read-over-write)}$$

Additionally we consider the extensional variant of the theory of arrays. Then, every model

has to satisfy the extensionality axiom:

$$\forall a\, b.\ a = b \vee \exists i.\ a[i] \neq b[i] \tag{ext}$$

We use a, b to denote array-valued variables, i, j, k to denote index variables, and v, w to denote element variables. Additionally we use subscripts to distinguish different variables. We use P to denote a path in a graph. A path in a graph is interpreted as a sequence of edges.

In the remainder of this paper, we consider quantifier-free $\mathcal{T}_A$-formulae. Furthermore we fix the index $\mathcal{T}_I$ to a stably infinite theory and the element theory $\mathcal{T}_E$ to a theory that contains at least two different values[1].

3 Towards a Nelson–Oppen-based Array Solver

Multiple theories are usually combined with a variant of the Nelson–Oppen combination procedure [10]. The procedure requires the participating theories to be stably infinite and to only share the equality symbol $=$.

The procedure first transforms the input such that every literal is *pure* with respect to the theories. Let $f(t)$ be a term in the input. If f is interpreted by theory $\mathcal{T}_1$ and t is interpreted by theory $\mathcal{T}_2$, then $f(t)$ is not pure. The first step of the Nelson–Oppen procedure then generates a fresh variable v, rewrites $f(t)$ into $f(v)$, and adds the definition $v = t$ as a new conjunct to the formula. The fresh variable is shared between theories $\mathcal{T}_1$ and $\mathcal{T}_2$. This step is repeated until all terms are pure. By abuse of notation, we name the shared variable after its defining term t, e. g., we use $a[i]$ to denote the shared variable that is defined as $a[i]$.

Let V be the set of fresh variables introduced in the first step of the combination procedure. The second step of the procedure tries to find an *arrangement* of V, i. e., an equivalence relation between variables in V such that $\mathcal{T}_1$ and $\mathcal{T}_2$ produce partial models that agree with this equivalence relation. Finding such an arrangement is typically done by propagating equalities or providing case split lemmas. In the following, we call this arrangement strong equivalence to distinguish it from weak equivalence defined in the next section. We write $a \sim b$ to denote that a and b are strongly equivalent, i.e., that in the current arrangement the shared variables a and b are equal.

For the theory of arrays, we consider every term of the form $\cdot\langle\cdot \lhd \cdot\rangle$ or $\cdot[\cdot]$ as being interpreted by the array theory. We consider all array terms, store, and select terms to be shared and thus they have to occur in the arrangement. Furthermore, every index term that appears in a store or select is considered shared between the array theory and the index theory. Then the goal is to find a suitable arrangement to these shared terms such that all theories agree on this arrangement.

For an array solver to be used in Nelson–Oppen combination we have to propagate equalities between shared array terms and shared select terms. Furthermore, the other theories have to propagate equalities between terms used as index in a select or store. In the remainder of this paper we will first show how to propagate equalities between select terms and afterwards deal with extensionality to propagate equalities between array-valued terms.

4 Weak Equivalences over Arrays

The theory of arrays has two constructors for arrays: array variables, and store terms $\cdot\langle\cdot \lhd \cdot\rangle$. Assuming quantifier-free input, we can only constrain the values of a finite number of indices.

[1] Note that $\mathcal{T}_A$ is stably infinite under these conditions.

These constraints can either be explicity like $a[i] = v$, or implicit like $a\langle i \lhd v\rangle$ where axiom (idx) produces the corresponding $a\langle i \lhd v\rangle[i] = v$. Hence, for quantifier-free input, arrays that are connected via a sequence of $\cdot\langle\cdot \lhd \cdot\rangle$ can only differ in finitely many positions. We call such arrays *weakly equivalent*. In this section we formally define weak equality and show how to exploit this to produce a decision procedure for the (extensional) theory of arrays.

Let $\mathcal{S}$ be the set of all terms of the form $\cdot\langle\cdot \lhd \cdot\rangle$ in the input formula and $\mathcal{A}$ be the set of all array-valued terms that are not in $\mathcal{S}$. Since $a\langle i \lhd v\rangle$ modifies a only at index i, these two arrays are guaranteed to be equal on all indices except on index i. We generalise this observation to chains of the form $\ldots\langle j \lhd w\rangle\langle i \lhd v\rangle$ to extract a set of indices for which two arrays might store different values.

Definition 1 (weak equivalence)**.** A *weak equivalence graph* G^W contains vertices $\mathcal{S} \cup \mathcal{A}$ and undirected edges defined as follows:

1. $a \leftrightarrow b$ if $a \sim b$, and
2. $a \overset{i}{\leftrightarrow} b$ if a has form $b\langle i \lhd \cdot\rangle$.

We write $a \overset{(P)}{\Leftrightarrow} b$ if there exists a path P between nodes a and b in G^W. In this case, we call a and b *weakly equal*. The weak equivalence class containing all elements that are weakly equal to a is defined as $\mathrm{WeakEQ}(a) := \{b \mid \exists P.\ a \overset{(P)}{\Leftrightarrow} b\}$.

For a path P we define $\mathrm{Stores}\,(P)$ as the set of all indices corresponding to edges of the form $\overset{\cdot}{\leftrightarrow}$, i. e., $\mathrm{Stores}\,(P) := \{i \mid \exists a\, b.\ a \overset{i}{\leftrightarrow} b \in P\}$.

Example 1. Consider the formula $a = b\langle j \lhd v\rangle \wedge b = c\langle i \lhd w\rangle \wedge d = e \wedge c[i] = w$. The weak equivalence graph for this example is shown in Figure 1. Note that the last conjunct is not important for the construction of the weak equivalence graph.

$$a \longleftrightarrow b\langle j \lhd v\rangle \overset{j}{\longleftrightarrow} b \longleftrightarrow c\langle i \lhd w\rangle \overset{i}{\longleftrightarrow} c$$
$$d \longleftrightarrow e$$

Figure 1: Weak Equivalence Graph for Example 1

We get two different weak equivalence classes. The first one contains the nodes a, $b\langle j \lhd v\rangle$, b, $c\langle i \lhd w\rangle$, and c. The second contains d and e. Note that d and e are actually strongly equivalent. Thus, they store the same value at every position. Let P denote the path from a to c in the weak equivalence graph. Then, $\mathrm{Stores}\,(P) = \{i, j\}$. Thus, arrays a and c can only differ in at most the values stored at the indices i and j.

If we want to know if $a[i]$ and $b[i]$ should be equal, we check if $a \overset{(P)}{\Leftrightarrow} b$ for a path P such that $i \notin \mathrm{Stores}\,(P)$. If this is the case, P witnesses the equivalence between the select terms.

Definition 2 (weak equivalence modulo i)**.** Two arrays a and b are *weakly equivalent modulo* i if and only if they are weakly equivalent and connected by a path that does not contain an edge of the form $\overset{j}{\leftrightarrow}$ where $j \sim i$. We denote weak equivalence modulo i by $a \approx_i b$ and define it as $a \approx_i b := \exists P.\ a \overset{(P)}{\Leftrightarrow} b \wedge \forall j \in \mathrm{Stores}\,(P)\,.\ j \not\sim i$.

Using this definition we can propagate equalities between shared selects if the arrays are weakly equivalent modulo the index of the select.

Lemma 1 (read-over-weakeq)**.** *Let $\sim$ be an arrangement satisfying the array axioms. Let $a[i]$ and $b[j]$ be two selects such that $i \sim j$ and $a \approx_i b$. Then, $a[i] \sim b[j]$ holds.*

Proof. We induct over the length of the path P witnessing $a \approx_i b$.

Base case. In this case, a and b are the same term. Hence $a[i] \sim b[j]$ holds by congruence.

Step case. Let the step from c to b be the last step of path P. By induction hypothesis we know that $a[i] \sim c[j]$ holds.

If the edge between c and b is due to a strong equivalence (i. e., $c \sim b$), then $c[j] \sim b[j]$ follows from congruence.

If the edge between c and b is of the form $c \overset{k}{\leftrightarrow} b$, then either c is $b\langle k \lhd \cdot \rangle$ or b is $c\langle k \lhd \cdot \rangle$. In both cases, we get the lemma $j = k \vee c[j] = b[j]$ from axiom (read-over-write). Since $j \sim i$ and $i \not\sim k$, we get $j \not\sim k$ and thus $c[j] \sim b[j]$. We conclude $a[i] \sim b[j]$ by transitivity. □

This lemma allows us to propagate equalities between shared selects. Note that it depends upon disequalities between index terms needed to ensure $a \approx_i b$.

If two arrays are weak equivalent modulo i they store the same value at the index i. The reverse is not necessarily true. Therefore, we define a weaker relation weak congruence modulo i.

Definition 3 (weak congruence modulo i)**.** Arrays a and b are *weak congruent modulo i* if and only if they are guaranteed to store the same value at index i. We denote weak congruence modulo i by $\sim_i$ and define $a \sim_i b := a \approx_i b \vee \exists a'\, b'\, j\, k.\, a \approx_i a' \wedge i \sim j \wedge a'[j] \sim b'[k] \wedge k \sim i \wedge b' \approx_i b$.

We use weak congruences to decide extensionality. Intuitively, if for all indices i the weak congruence modulo i $a \sim_i b$ holds, then $a = b$ should be propagated. But this naïve approach requires checking every index occurring in the formula. To minimise the number of indices we need to consider, we exploit the weak equivalence graph.

Lemma 2 (weakeq-ext)**.** *Let $\sim$ be an arrangement satisfying the array axioms. Let a and b be two arrays such that $a \overset{(P)}{\Leftrightarrow} b$ holds. If for all indices $i \in \mathrm{Stores}(P)$ we have $a \sim_i b$, then $a \sim b$ holds.*

Proof. Follows from Lemma 1, Definition 3 and (ext). □

5 A Decision Procedure Based on Weak Equivalences

Our decision procedure is based on weak equivalences and the Nelson–Oppen combination scheme. It propagates equalities between terms shared by multiple theories. We limit the propagation to shared array terms and array select terms.

The $\mathcal{T}_A$-formulae are preprocessed as follows. For every $a\langle i \lhd v \rangle$ contained in the input, we (1) instantiate the axiom (idx) and (2) add $a[i]$ to the set of terms contained in the input[2]. Thus, the preprocessing step adds at most two select operations for every store.

We propagate new equalities from weak equivalence relations and weak congruence relations based on lemmas 1 and 2. These relations depend on the arrangement $\sim$, which represents logical equality ($=$). We now define a function $\mathrm{Cond}(\cdot)$ that computes a condition (a conjunction of equalities and inequalities) under which a weak equivalence or weak congruence holds. To

[2]This can be achieved by adding the equality $a[i] = a[i]$.

denote the condition for a path that does not contain an edge labelled with index i we use $\mathrm{Cond}_i(\cdot)$. For an edge in the weak equivalence graph that represents an equality, the condition reflects this equality. For an edge that comes from a $\cdot\langle j \lhd \cdot\rangle$, no condition is needed. However, $\mathrm{Cond}_i(\cdot)$ should ensure that i does not occur on the path, so $i \neq j$ needs to hold.

$$\mathrm{Cond}(a \leftrightarrow b) := a = b \qquad \mathrm{Cond}_i(a \leftrightarrow b) := a = b$$
$$\mathrm{Cond}(a \overset{j}{\leftrightarrow} b) := \mathrm{true} \qquad \mathrm{Cond}_i(a \overset{j}{\leftrightarrow} b) := i \neq j$$

We can extend these definitions to paths by conjoining the conditions for all edges on that path. Then, we can compute $\mathrm{Cond}(a \approx_i b)$ using the path that witnesses $a \approx_i b$.

$$\mathrm{Cond}(a \approx_i b) := \mathrm{Cond}_i(P) \text{ where } a \overset{(P)}{\Leftrightarrow} b \wedge \forall j \in \mathrm{Stores}\,(P)\,.\; i \not\sim j$$

Finally, to define $\mathrm{Cond}(a \sim_i b)$, we use the definition of $\sim_i$.

$$\mathrm{Cond}(a \sim_i b) := \begin{cases} \mathrm{Cond}(a \approx_i b) & \text{if } a \approx_i b \\ \mathrm{Cond}(a \approx_i a') \wedge i = j \wedge a'[j] = b'[k] \\ \quad \wedge k = i \wedge \mathrm{Cond}(b' \approx_i b) & \text{if } \begin{array}{c} a \approx_i a' \wedge i \sim j \wedge a'[j] \sim b'[k] \\ \wedge\, k \sim i \wedge b' \approx_i b \end{array} \end{cases}$$

Example 2. Consider again the formula $a = b\langle j \lhd v\rangle \wedge b = c\langle i \lhd w\rangle \wedge d = e \wedge c[i] = w$ from Example 1 whose weak equivalence graph is shown in Figure 1. Assume $i \not\sim j$. Then we have $a \approx_i c\langle i \lhd w\rangle$ since no edge contains a label that is equivalent to i. We get $\mathrm{Cond}(a \approx_i c\langle i \lhd w\rangle) \equiv a = b\langle j \lhd v\rangle \wedge i \neq j \wedge b = c\langle i \lhd w\rangle$.

From Axiom (idx) we get $c\langle i \lhd w\rangle[i] = w$. With $c[i] = w$ we conclude $a \sim_i c$ since $a \approx_i c\langle i \lhd w\rangle$ and $c\langle i \lhd w\rangle[i] = c[i]$. We have $\mathrm{Cond}(a \sim_i c) \equiv \mathrm{Cond}(a \approx_i c\langle i \lhd w\rangle) \wedge c\langle i \lhd w\rangle[i] = c[i]$.

To decide the theory of arrays we define two rules to generate instances of array lemmas. We present the rules as inference rules. The rule is applicable if the current arrangement $\sim$ on the shared variables V satisfies the conditions above the line. The rule then generates a new (valid) lemma that can propagate an equality under the current arrangement.

The first rule is based on Lemma 1. Two select terms are equivalent if the indices of the selects are congruent and the arrays are weakly equivalent modulo that index. We only create this lemma if the select terms existed in the formula. Note that we create for select terms in the formula a shared variable with the same name in V.

$$\frac{a \approx_i b \qquad i \sim j \qquad a[i], b[j] \in V}{i \neq j \vee \neg\,\mathrm{Cond}(a \approx_i b) \vee a[i] = b[j]} \qquad \text{(read-over-weakeq)}$$

The next rule is based on Lemma 2 and used to propagate an equality between two extensionally equal array terms. Two arrays a and b have to be equal if there is a path P such that $a \overset{(P)}{\Leftrightarrow} b$ and for all $i \in \mathrm{Stores}\,(P)$, $a \sim_i b$ holds.

$$\frac{a \overset{(P)}{\Leftrightarrow} b \qquad \forall i \in \mathrm{Stores}\,(P)\,.\; a \sim_i b \qquad a, b \in V}{\neg\,\mathrm{Cond}(P) \vee \bigvee_{i \in \mathrm{Stores}(P)} \neg\,\mathrm{Cond}(a \sim_i b) \vee a = b} \qquad \text{(weakeq-ext)}$$

The resulting decision procedure is sound and complete for the existential theory of arrays assuming sound and complete decision procedures for the index and element theories.

Lemma 3 (soundness)**.** *Rules (read-over-weakeq) and (weakeq-ext) are sound.*

Proof. Soundness of the rules follows directly from the lemma with the corresponding name. □

Lemma 4 (completeness)**.** *The rules (read-over-weakeq) and (weakeq-ext) are complete.*

The proof of this lemma can be found in the extended version of this paper [5].

6 Restricting Instantiations

The preprocessor is the only component of our decision procedure that produces new select terms and thus might trigger new lemmas. These lemmas only generate new (dis-)equality literals between existing terms. Thus, reducing the number of select terms might reduce the number of lemmas generated by our decision procedure and speed up the procedure.

If the element theory is stably infinite we can omit the preprocessor step that adds for every $a\langle i \lhd v\rangle$ the select $a[i]$. Instead, we simply assume $a[i]$ to be different than any other $b[i]$ unless $a \sim b$. This method preserves soundness and completeness.

Lemma 5. *(soundness of modified procedure) The modified procedure is sound.*

Proof. Follows directly from Lemma 3 since it does not rely on the addition of $a[i]$ for every $a\langle i \lhd \cdot\rangle$. □

For the completeness lemma we take into account the fact that the element theory is stably infinite. Thus, if $a[i]$ is not present we use a fresh element in the value domain.

Lemma 6 (completeness of modified procedure)**.** *The modified procedure is complete.*

The proof of this lemma can be found in the extended version of this paper [5].

This optimisation enables us to limit the number of additional terms in the input. Since we only need to generate (read-over-weakeq) lemmas if the select terms in the conclusion are present after preprocessing, this optimisation also reduces the number of lemmas. Furthermore, it is widely applicable. In fact, the non-bitvector logics defined in the SMTLIB [2] only allow array sorts where the element theory is stably infinite. Thus, only the terms corresponding to instantiations of Axiom (idx) are required. In an actual implementation even these terms could be omitted (see [3]).

7 Implementation and Evaluation

We implemented the decision procedure described in this paper in our SMT solver SMTInterpol [6]. Besides the aforementioned preprocessing step that applies (idx) to every $\cdot\langle\cdot \lhd \cdot\rangle$ in the input, we also simplify $\mathcal{T}_A$-formulae by applying (read-over-write) if the index of the store and the index of the select are syntactically equal. Furthermore, we contract terms of the form $a\langle i \lhd v_2\rangle\langle i \lhd v_1\rangle$ to $a\langle i \lhd v_1\rangle$. We only add $a[i]$ to the set of terms contained in the formula if we have $a\langle i \lhd v\rangle$ in the input and the domain of v is finite.

We represent the weak equivalence relation and the weak equivalence modulo i relations in a forest structure, similarly to the representation of equivalence graph in congruence solvers [11]. Every node has an outgoing edge, and these edges build a spanning tree for every equivalence class. The edges point from a child node to the parent node. The root node of every tree has no outgoing edge and is the representative of its equivalence class.

We have to distinguish between strong equivalence, weak equivalence, and weak equivalence modulo i. The strong equivalence classes are already handled by the equality solver. In our implementation of the array solver we treat them as indivisible and create a single node for every strong equivalence class. To represent the weak equivalence relations the nodes have up to two outgoing edges, a primary p and a secondary s, see Figure 2. The edges come from a store operation and correspond to the edges $\overset{i}{\leftrightarrow}$ in the weak equivalence graph. The index of the primary edge is stored in the pi field. The primary edge points towards the representative

```
struct NODE
    p : NODE
    pi : INDEX
    s : NODE

GET-REP(n : NODE)
    if n.p = NIL then n
    else      GET-REP(n.p)

MAKE-REP(n : NODE)
    if n.p ≠ NIL then
        MAKE-REP(n.p)
        n.p.p := n
        n.p.pi := n.pi      } invert primary
        n.p := NIL          } edge
        MAKE-REP_i(n)
```

```
GET-REP_i(n : NODE, i : INDEX)
    if n.p = NIL then n
    elseif n.pi ≠ i then GET-REP_i(n.p, i)
    elseif n.s = NIL then n
    else      GET-REP_i(n.s, i)

MAKE-REP_i(n : NODE)
    if n.s ≠ NIL then
        if n.s.pi ≠ n.pi then
            n.s := n.s.p        } move towards
            MAKE-REP_i(n)       } representative
        else
            MAKE-REP_i(n.s)
            n.s.s := n.s        } invert
            n.s := NIL          } secondary edge
```

Figure 2: Data structure and functions to represent weak equivalence relations. A NODE structure is created for every strong equivalence class on arrays. It contains two outgoing edges p, s pointing towards the representative of the weak equivalence classes. The functions GET-REP and GET-REP$_i$ are used to find the representative of the weak equivalence (resp. weak equivalence modulo i) class. The functions MAKE-REP and MAKE-REP$_i$ invert the edges to make a node the representative of its weak equivalence classes.

of the weak equivalence class. Every primary edge p connects the node representing (the strong equivalence class of) a store $a\langle j \lhd v\rangle$ with the node representing a and the corresponding index in the pi field is j. Note, however, that the direction of the edge can be arbitrary, as we invert the edges during the execution of the algorithm. If the primary edge is missing the node is the representative of its weak equivalence class and of all its weak equivalence modulo i classes.

While the primary edge is enough to represent the weak equivalence relation we need another edge to represent weak equivalence modulo i. The representative of weak equivalence modulo i is also found by following the primary edges. However, if the store of the primary edge is on the index i, the secondary edge is followed instead. If the secondary edge is missing the node is the representative of its weak equivalence modulo i class.

The equivalence classes are represented as follows. Two arrays a and b are weakly equivalent iff GET-REP(a) = GET-REP(b) and $a \approx_i b$ iff GET-REP$_i(a, i)$ = GET-REP$_i(b, i)$.

The algorithm proceeds by inserting the store edges one by one, similarly to the algorithm presented in [11]. The algorithm that inserts a store edge is given in Figure 3. The algorithm first inverts the outgoing edges of one node to make it the representative of its weak equivalence class. If the other side of the store edge lies in a different weak equivalence classes, the store

```
ADD-SECONDARY(S : INDEX SET, a, b : NODE)
    if a = b then
        return
    if a.pi ∉ S ∧ GET-REP_i(a, a.pi) ≠ b then
        MAKE-REP_i(a)
        a.s := b
    ADD-SECONDARY(S ∪ {a.pi}, a.p, b)
```

```
ADD-STORE(a, b : NODE, i : INDEX)
    MAKE-REP(b)
    if GET-REP(a) = b then
        ADD-SECONDARY({i}, a, b)
    else
        b.p := a
        b.pi := i
```

Figure 3: The algorithm ADD-STORE adds a new store edge to the data structure updating the weak equivalence classes. In the else case a new primary edge is added to merge two disjoint weak equivalence classes. Otherwise, ADD-SECONDARY inserts new secondary edges to merge the necessary weak equivalence modulo i classes.

can be inserted as a new primary edge.

If the nodes are already weakly equivalent the procedure ADD-SECONDARY is called. This procedure follows the path from the other array a to the array b that was made the representative. For every node on this path it checks if a secondary edge needs to be added. If the primary edge of the node is labelled with a store on i, the algorithm first checks if the node is weakly equivalent modulo i with b due to the new store edge. This is the case if no store on i occurred on the path so far and the new store is also on an index different from i. We use the set S to collect these forbidden indices. Then if b is not already the representative of the weak equivalence modulo i class, the outgoing secondary edges are reversed and a new secondary edge is added.

The complexity of the procedure ADD-STORE is worst case quadratic in the size of the weak equivalence class. This stems from MAKE-REP$_i$ being linear in the size and being called a linear number of times. The overall complexity is cubic in the number of stores in the input formula. The space requirement, however, is only linear. In our current implementation in SMTInterpol this procedure was not a bottleneck so far. In SMTInterpol we also keep the stores that created the primary and secondary edge in the data-structure. This allows for computing the paths needed for lemma generation in linear time.

Example 3. Figure 4 shows an example of the data structure where the primary edges are labelled by the index of the corresponding store. This data structure represents only one weak equivalence class with the representative node 0. The resulting data structure after adding a store with index k between nodes 0 and 4 is shown on the right. Since nodes 0 and 4 were already in the same weak equivalence class, secondary edges were added.

These secondary edges are needed to connect the weak equivalence modulo i and modulo j classes. Figure 5(a) shows how the first secondary edge connects the two weak equivalence modulo i classes rooted at nodes 0 resp. 3. This is necessary since there is now a new path using the edge from 4 to 0. Note that no secondary edge is added to node 1, since nodes 1, 2, and 5 are still not weakly equivalent modulo i to the other nodes. Figure 5(b) shows the connection between the two weak equivalence modulo j classes rooted at nodes 0 resp. 2. The weak equivalence modulo j class rooted at node 6 is not affected by a new edge between nodes 0 and 4 since these nodes are on a different path.

We implemented this decision procedure in our SMT solver SMTInterpol [6] and tested it on the benchmarks from the QF_AX and QF_AUFLIA devisions of the SMTEVAL 2013 benchmarks. We solved all benchmarks in 1:32 resp. 10:45 minutes without running into a

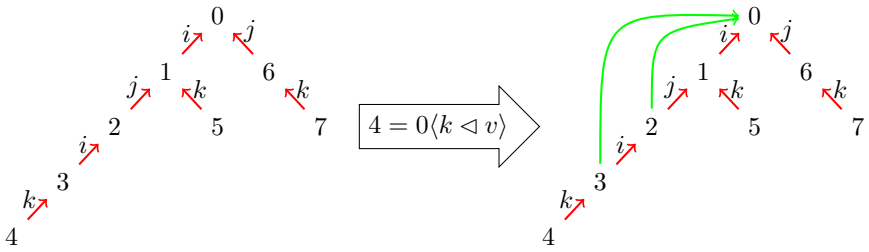
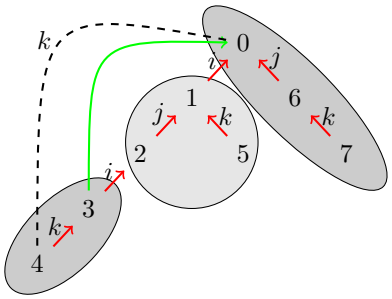
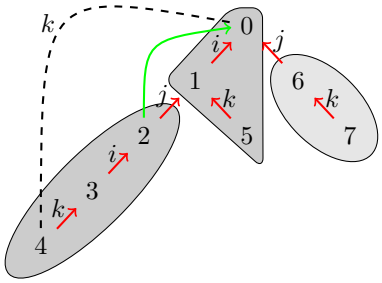


Figure 4: Weak equivalence classes represented by a graph using primary and secondary edges. The short direct edges are primary edges, the long bended edges are secondary edges. Each primary edge represents a store edge between the connected nodes and is labelled by the index of the store. The secondary edges in the right graph were created by a store edge on index k between node 0 and 4 as described in Example 3.



(a) Merging weak equivalence modulo i classes.



(b) Merging weak equivalence modulo j classes.

Figure 5: Secondary edges merge weak equivalence modulo i classes.

timeout of 10 minutes. According to the data from the SMTEVAL, no other solver was able to solve all benchmarks in these divisions. We defer an up-to-date comparison to the SMTCOMP 2014.

8 Conclusion and Future Work

We presented a new decision procedure for the extensional theory of arrays. This procedure exploits weak equalities to limit the number of axiom instantiations. The instantiations produced by the decision procedure presented in this paper can be restricted to terms already present in the input formula. Furthermore we discussed an implementation based on a graph structure similar to congruence closure graphs. This decision procedure is implemented in our SMT solver SMTInterpol [6]. We plan to implement a variant of the quantifier-free interpolation for arrays [4] based on the lemmas generated by this decision procedure. Since these lemmas only generate mixed equalities, proof tree preserving interpolation [7] can be used.

References

[1] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1) (2009)

[2] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: 2.0. In: SMT (2010)

[3] Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. JSAT 6(1-3), 165–201 (2009)

[4] Bruttomesso, R., Ghilardi, S., Ranise, S.: Quantifier-free interpolation of a theory of arrays. Logical Methods in Computer Science 8(2) (2012)

[5] Christ, J., Hoenicke, J.: Weakly equivalent arrays. CoRR abs/1405.6939 (2014)

[6] Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: SPIN. pp. 248–254 (2012)

[7] Christ, J., Hoenicke, J., Nutz, A.: Proof tree preserving interpolation. In: TACAS. pp. 124–138 (2013)

[8] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. pp. 21–28 (1962)

[9] de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD. pp. 45–52 (2009)

[10] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)

[11] Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: RTA. pp. 453–468. Springer (2005)

Decision Procedures for Flat Array Properties*

Francesco Alberti[1,3], Silvio Ghilardi[2], Natasha Sharygina[1]

[1] University of Lugano, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] VERIMAG, Grenoble, France

Abstract

We present new decidability results for quantified fragments of theories of arrays. Our decision procedures are fully declarative, parametric in the theories of indexes and elements and orthogonal with respect to known results. We also discuss applications to the analysis of programs handling arrays.

*The work of the first author was supported by Swiss National Science Foundation under grant no. P1TIP2_152261. This work has been published in the proceedings of *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 15-30, Springer, 2014. The original publication is available at `www.springerlink.com`.

Extending SMT-LIB v2 with λ-Terms and Polymorphism

Richard Bonichon, David Déharbe, and Cláudia Tavares

Universidade Federal do Rio Grande do Norte
Natal, Brazil
richard@dimap.ufrn.br, david@dimap.ufrn.br, claudia@ppgsc.ufrn.br

Abstract

This paper describes two syntactic extensions to SMT-LIB scripts: lambda-expressions and polymorphism. After extending the syntax to allow these expressions, we show how to update the typing rules of the SMT-LIB to check the validity of these new terms and commands. Since most SMT-solvers only deal with many-sorted first-order formulas, we detail a monomorphization mechanism to allow to use polymorphism in SMT-LIB syntax while retaining a monomorphic solver core.

1 Introduction

Dissemination of SMT-solvers requires they are powerful, trustable and open (i.e. easy to interface). The SMT-LIB format [BST10] is an initiative from the SMT community to address the last aspect, by offering both a common language to describe problems and a command language to interact with the solver.

The SMT-LIB format has evolved in the recent years from version 1.2 to version 2.0 to simplify the definition of the syntax for the part of the language that is responsible for problem descriptions (i.e. declaration of the signature and assertion of logic expressions) on the one hand, and to enrich the commands that allow third-party tools to interact with complying SMT solvers.

veriT is an open-source solver jointly developed at INRIA and UFRN. Early versions of the veriT solver implemented several extensions to version 1.2 of the SMT-LIB format: macro-definitions, λ-expressions and β-reduction. A later extension included polymorphic sorts, signatures and assertions. A successful application of these extensions has been to apply SMT solving to verify proof obligations stemming from set-based formalisms (namely, B and Event-B) using standard SMT-LIB logics (AUFLIA) [Dé10, DFGV12, Dé13].

The rationale of this application is essentially the following:

- A set is encoded as its characteristic predicate. For instance, $\{x \cdot 0 \leq x \leq 9\}$ is encoded as:

  (**lambda** ((x Int)) (**and** (<= 0 x) (<= x 9))).

- Set operations are encoded as higher-order functions. For instance, set intersection is encoded as a (polymorphic) macro called inter defined as:

  (**define**−**fun** (**par** (X) (inter (**lambda** ((f (X Bool)) (g (X Bool)) (x X)) (**and** (f x) (g x))))) ,

  and set membership by the macro named member and defined as:

  (**define**−**fun** (**par** (X) (member (**lambda** ((x X) (f (X Bool))) (f x))))) ,

  in both definitions, X denotes a type variable, its scope being the definition itself.

To handle such expressions, veriT implements a processor that performs macro-expansion and β-reduction steps as well as type inference. For instance, the formula $0 \in (\{x \cdot x \geq 0\} \cap \{x \cdot x \leq 0\})$ would be encoded as (member 0 (union (**lambda** (x Int) (>= x 0)) (**lambda** (x Int) (<= x 0)))), and the result of processing this expression is (**and** (>= 0 0) (<= 0 0)). In addition, this processor rewrites equalities between lambda expressions as universal quantifications. Once all steps have been applied, if the goal is not first-order logic, then veriT emits an error message and halts.

Some of these extensions were included in the SMT-LIB format: polymorphic sorts , though restricted to theory files, and macro-definition (named function definitions). In this paper, we thus discuss modifications to the SMT-LIB format 2.0 corresponding to the extensions to SMT-LIB format 1.2 that were implemented in the solver veriT. It is noteworthy that these modifications maintain backward compatibility with the existing definition of the SMT-LIB format. Also we discuss how to rewrite a problem expressed with the proposed extensions to plain SMT-LIB 2.0.

This paper is organized as follows. Sec. 2 presents the extensions made to SMT-LIB, introducing polymorphism at the SMT-LIB script level. This leads to an updated set of typing rules, mainly for SMT-LIB terms, which is discussed in Sec. 3. Using these rules, we detail a strategy to generate a monomorphic version of our target problem in Sec. 4. Finally, we discuss the pros and cons of our solution in the context of related work in Sec. 5 and detail ongoing and further work in Sec. 6.

2 Extensions to SMT-LIB

We propose two extensions to the SMT-LIB format:

- anonymous functions (i.e., λ-abstractions) and their applications;
- parametric polymorphism for assertions and function types.

The SMT-LIB already features two flavors of polymorphism:

- parametric polymorphism for sorts and function signatures, but only for background theories;
- ad-hoc polymorphism because functions can be overloaded.

The extension adds parametric polymorphism to assertions and function definitions and declarations. The additional introduction of λ gives a very functional, higher-order, flavor to this extended SMT-LIB. The typing rules presented in Sec. 3.1 even allow let-polymorphism inside the SMT-LIB.

However, λ-abstractions as we envision them will not add much expressiveness as we want to be able to get a first-order problem only through the application of β-reduction. Thus, we will not handle a reduced problem with residual higher-order or partial applications. We see the addition of λ-abstractions as a convenient mechanism to encode certain problems.

Also, the introduction of polymorphism at the syntactic level does not fundamentally change the expressive power available for SMT-LIB scripts. Indeed, the combination of type schemes to express background theories and overloading of functions permitted by the SMT-LIB standard already covers most functionalities which ML-style let-polymorphism permits. Nonetheless, we argue that polymorphism in scripts is syntactically more convenient than writing every ground instances of the expressions we are interested in.

The next section presents the concrete syntax for the two extensions mentioned above.

2.1 Syntax extensions for SMT-LIB

Anonymous functions are terms introduced using **lambda**, which becomes a keyword. Polymorphic terms reuse the same **par** keyword as the polymorphic elements of the SMT-LIB theories. The syntactic extensions are summarized in the BNF extract of Fig. 1.

```
⟨par_function_args⟩ ::= par (⟨symbol⟩+)

⟨par_command⟩ ::= (define-fun (⟨par_function_args⟩ (⟨symbol⟩
                       (⟨sorted_var⟩*) ⟨sort⟩ ⟨term⟩)))
                | (declare-fun (⟨par_function_args⟩ (⟨symbol⟩ (⟨sort⟩*) ⟨sort⟩)))
                | (assert (⟨par_function_args⟩ ⟨term⟩))
```

Figure 1: BNF extensions for SMT-LIB

Function types are allowed as the return type of functions, due to the inclusion of λ-term. The choice made is to declare a function type as a list of types. For example the function declaration (**declare-fun** f (Int Int) Int) is not the same as the function declaration (**declare-fun** f ((Int Int)) Int). The former is a function which expects two integers and returns an integer, the latter expects only one argument — a function from integer to integer — and returns an integer. Now, the only possible problem is to distinguish in a sort expression (X Y) between a sort meant to express a function type and a sort which is the application of a sort of arity ≥ 1 to its argument, like (Array Int). This is not a practical problem as sort arity is explicitly stated. Therefore, if the arity of X is greater than 1, we say it is a sort application, otherwise it is a function type.

3 Typing extended SMT-LIB

This section details the rules for typing extended SMT-LIB scripts. These rules extend the current set of rules for SMT-LIB[1].

Typing polymorphic terms is a necessary step towards uncovering monomorphic ground instances of polymorphic terms. Various type instantiations of the same polymorphic functions will lead in Sec. 4 to the generation of multiple monomorphic versions of the same function. Fortunately, ad-hoc polymorphism through overloading is permitted by the SMT-LIB standard.

3.1 Typing terms

The type of polymorphism we introduce in the extended SMT-LIB is ML-style prenex polymorphism [Pie02]. The typing rules of the system are described in Fig. 3. They can be seen as an adaptation of a Damas-Hindley-Milner [Mil78, Hin69] type system to the SMT-LIB syntax. These rules manipulates SMT-LIB sorts, type variables, tuple types and function types:

Definition 1 (Types)**.** *Let $\mathcal{S}$ be a set of sorts, $\mathcal{V}$ be a set of (type) variables. The set of well-formed types $\mathbb{T}$ is defined inductively as follows:*

1. *if $s \in \mathcal{S}$, then $s \in \mathbb{T}$*

[1] Detailed in Section 4.2.2 of the SMT-LIB Standard [BST10]

```
(set-logic AUFLIA)

(define-fun (par (X) (empty ((e X)) Bool false )))

(define-fun (par (Y)
    (insert ((e Y) (s (Y Bool))) (Y Bool)
        (lambda ((e1 Y)) (or (= e e1) (s e1))))))

(declare-fun (par (Z) (flat (((Z Bool) Bool)) (Z Bool))))

(assert (= (flat empty) empty))

(assert (par (X)
        (forall ((ss ((X Bool) Bool)))
        (= (flat (insert empty ss)) (flat ss)))))

(assert (par (X)
        (forall ((e X) (s (X Bool)) (ss ((X Bool) Bool)))
                (= (flat (insert (insert e s) ss))
                   (insert e (flat (insert s ss)))))))

(assert (forall ((s (Int Bool))) (= (flat (insert s empty)) s)))

(check-sat)
```

Figure 2: A polymorphic specification

2. *if* $v \in \mathcal{V}$, *then* $v \in \mathbb{T}$
3. *if* $T_1 \in \mathbb{T}, T_2 \in \mathbb{T}$, $T_1 \times T_2 \in \mathbb{T}$
4. *if* $T_1 \in \mathbb{T}, T_2 \in \mathbb{T}$, $T_1 \to T_2 \in \mathbb{T}$

Notations. Type variables will be denoted by α. $\bar{\alpha}$ represent a set of type variables (possibly empty). A type is said to be *ground* if it has no type variables. A *type substitution* is a mapping from type variables to types (ground or not). The application of a type substitution is denoted using $:=$ and extended to variable sets. Hence $T[\bar{\alpha} := \bar{T}']$ substitutes in T every member of $\bar{\alpha}$ by a corresponding type in $\bar{T}$. A *typing environment* Γ is a mapping from identifiers to types. Universal quantification over type variables is denoted $\forall_{\mathbb{T}}$, in order to separate this quantification from the universal quantifier ranging over term variable. For terms, x or x_i will stand for variables, t or t_i will stand for any term.

Typing rules. The rules of Fig. 3 use two additional functions:

- a function to compute the set of free type variables of a type, denoted fv_T;
- a generalization function Gen, which helps compute the most general type possible for a **let**-bound variable. This function is defined as follows:

$$\mathrm{Gen}(T, \Gamma) = \forall_{\mathbb{T}}(\mathrm{fv}_T(T) \backslash \mathrm{fv}_T(\Gamma)).T$$

The rules distinguish between curried functions ($\to$ type), where partial application is allowed, and uncurried functions, where the arguments are treated as a tuple, thus disallowing partial application. Curried functions come from explicit λ-abstractions whereas the SMT-LIB's **define−fun** define functions which can only be totally applied.

$$\frac{\Gamma(x) = \forall_{\mathbb{T}}\bar{\alpha}.T}{\Gamma \vdash x : T[\bar{\alpha} := \bar{T'}]}\ \textsc{Ax}_{\forall} \qquad \frac{}{\Gamma \vdash x \text{ as } T : T}\ \textsc{Ax}_{\textsc{as}}$$

$$\frac{\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash t : \mathsf{bool} \qquad Q \in \{\forall, \exists\}}{\Gamma \vdash Q\ x_1 \ldots x_n t : \mathsf{bool}}\ \textsc{Qua}$$

$$\frac{\Gamma \vdash x_1 : T_1 \qquad \ldots \qquad \Gamma \vdash x_n : T_n \qquad \Gamma \vdash t : T}{\Gamma \vdash \lambda x_1 \ldots x_n.t : T_1 \to \ldots \to T_n \to T}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \ldots \qquad \Gamma \vdash t_n : T_n \qquad \Gamma \vdash f : T_1 \to \ldots T_n \to T}{\Gamma \vdash f\ t_1\ \ldots\ t_n : T}\ \textsc{App}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \ldots \qquad \Gamma \vdash t_n : T_n \qquad \Gamma, x_1 : \mathrm{Gen}(T_1, \Gamma), \ldots, x_n : \mathrm{Gen}(T_n, \Gamma) \vdash t : T}{\Gamma \vdash \text{let } ((x_1\ t_1) \ldots (x_n\ t_n))\ t : T}\ \textsc{Let}$$

Figure 3: Typing rules

3.2 Typing commands

SMT-LIB commands can — and often do — change the global typing environment by introducing new function names. Some commands can change the typing environment by introducing new sorts, new variable names and new functions. In particular, assertions have to be type checked to verify that the term being asserted is indeed a boolean.

Some commands, such as declaring or defining new sorts can also have an indirect influence on the typing environment, as they modify the set of well-formed types. In this section, we suppose that checking the well-formedness of types has been done prior to typing terms.

A SMT-LIB script is formalized as an ordered set of commands $\mathcal{C}$. We represent a program as a couple $\langle \Gamma, \mathcal{C} \rangle$ where Γ represents the current typing environment, initially empty. In the rules below, we only detail commands whose syntax have been changed.

Note that FUNDEF rule follows the transformation explained in the SMT-LIB Standard (p.59).

$$\frac{\langle \Gamma, \{\textbf{define-fun}\ \forall_{\mathbb{T}}\bar{\alpha}(f\ ((x_1\ T_1) \ldots (x_n\ T_n))\ T\ t)\} \cup \mathcal{C} \rangle}{\langle \Gamma \cup \{f : \forall_{\mathbb{T}}\bar{\alpha}.T_1 \times \ldots \times T_n \to T\}, \{\textbf{assert}\ \forall_{\mathbb{T}}\bar{\alpha}\ ((\forall x_1 : T_1 \ldots x_n : T_n\ f\ x_1 \ldots x_n) = t)\} \cup \mathcal{C} \rangle}\ \textsc{fundef}$$

$$\frac{\langle \Gamma, \{\textbf{declare-fun}\ \forall_{\mathbb{T}}\bar{\alpha}(f\ (T_1 \ldots T_n)\ T)\} \cup \mathcal{C} \rangle}{\langle \Gamma \cup \{f : \forall_{\mathbb{T}}\bar{\alpha}.T_1 \times \ldots \times T_n \to T\}, \mathcal{C} \rangle}\ \textsc{fundec}$$

$$\frac{\langle \Gamma, \{\textbf{assert}\ \forall_{\mathbb{T}}\bar{\alpha}\ t\} \cup \mathcal{C} \rangle \qquad \Gamma \vdash t : \mathsf{bool}}{\langle \Gamma, \mathcal{C} \rangle}\ \textsc{Assert}$$

The typing system is used in particular to find monomorphic occurrences of polymorphic terms. In the next section, we will use it in a monomorphization procedure.

4 Monomorphization

Once we have checked that a set of commands is well-typed, we need to generate a SMT-LIB-compatible version of it, since we do not want to have to change the core of the solver. It means first that λ-terms must be eliminated and second, that the problem must be monomorphized. The elimination of λ-terms has been discussed in Sec. 1 and uses β-reduction. Only if the problem is still first-order after this elimination can we then try to compute a monomorphic version of it. Otherwise we will simply discard it.

Bobot and Paskevich [BP11] have shown the undecidability of computing a minimal monomorphic set formulas equivalent to an original set of polymorphic formulas. However, we still aim to present here a monomorphization method for polymorphic formulas. If needed, it should compute an over-approximation of the minimal monomorphic set. Our implied goal is that we hope monomorphization will be *good enough* in practice for most of our problems. The proposed monomorphization is expected to be sound with respect to the original polymorphic types. Hence, the unsatisfiability of the newly generated problem implies the unsatisfiability of the original problem but its satisfiability does not in general imply the satisfiability of the original problem.

Monomorphization example. Let us detail the example of Fig. 2 to show what we would like to achieve on this specific case. On this example, monomorphization is expected to fail, so that we can also explain how the procedure works and how we deal with failure. In the example, three function signatures are defined, where type variables are implicitly universally quantified:

- emptyset: $\alpha_e \to \mathbb{B}$
- insert: $((\alpha_i \times (\alpha_i \to \mathbb{B})) \to \alpha_i \to \mathbb{B})$
- flat: $(\alpha_f \to \mathbb{B} \to \mathbb{B} \to \alpha_f \to \mathbb{B})$

The first step identifies polymorphic and monomorphic instances of terms through typing. In the example, we have three polymorphic formulas. These formulas are detailed below. For the sake of readability, we write type annotations only for co-domains, according to the initial function signatures, and hide annotations for term variables.

$$\mathsf{flat}^{\langle\alpha\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle\alpha\to\mathbb{B}\to\mathbb{B}\rangle}) = \mathsf{emptyset}^{\langle\alpha\to\mathbb{B}\rangle} \tag{4.1a}$$

$$\forall ss : \alpha\to\mathbb{B}\to\mathbb{B} \quad (\mathsf{flat}^{\langle\alpha\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\alpha\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle\alpha\to\mathbb{B}\rangle}, ss)) = \mathsf{flat}^{\langle\alpha\to\mathbb{B}\rangle}(ss)) \tag{4.1b}$$

$$\forall e : \alpha, s : \alpha\to\mathbb{B}, ss : \alpha\to\mathbb{B}\to\mathbb{B} \quad (\mathsf{flat}^{\langle\alpha\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\alpha\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\alpha\to\mathbb{B}\rangle}(e, s), ss)) = \mathsf{insert}^{\langle\alpha\to\mathbb{B}\rangle}(e, \mathsf{flat}^{\langle\alpha\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\alpha\to\mathbb{B}\to\mathbb{B}\rangle}(s, ss)))) \tag{4.1c}$$

In Fig. 2, the unique monomorphic assertion is:

$$\forall s : \mathbb{Z}\to\mathbb{B}\ (\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(s, \mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle})) = s)$$

Such monomorphic assertions drive the procedure. Basically, they provide the set of ground types that forms the basis for the generation of monomorphic instances of polymorphic functions.

The second step consists in computing the set of new terms derived from the injection of monomorphic elements. Using type substitutions, we can generate monomorphic specializations $\mathcal{F} = \{\mathsf{emptyset}_{[\alpha_e:=\mathbb{Z}\to\mathbb{B}]}, \mathsf{insert}_{[\alpha_i:=\mathbb{Z}\to\mathbb{B}]}, \mathsf{flat}_{[\alpha_f:=\mathbb{Z}]}\}$ of the polymorphic functions. We try to substitute the polymorphic occurrences of the three terms emptyset, insert, flat by their monomorphic counterpart whenever we can in the polymorphic terms 4.1a, 4.1b and 4.1c, using a leftmost-innermost strategy. This generates the following new set of monomorphic assertions:

$$\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}) = \mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_e}:=\mathbb{Z}]} \tag{4.2a}$$

$$\forall ss : \mathbb{Z}\to\mathbb{B}\to\mathbb{B} \quad (\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_e}:=\mathbb{Z}]}, ss)) = \mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(ss)) \tag{4.2b}$$

$$\forall e : \mathbb{Z}, s : \mathbb{Z}\to\mathbb{B}, ss : \mathbb{Z}\to\mathbb{B}\to\mathbb{B} \quad (\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_i}:=\mathbb{Z}]}(e, s), ss)) = \mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_i}=\mathbb{Z}]}(e, \mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(s, ss)))) \tag{4.2c}$$

The procedure uses all monomorphic terms, and thus generates a number of new monomorphic assertions. At this point, one full pass of the procedure has been executed. This is repeated while new monomorphic type instances are created. For example, the problem after the first full pass now has uncovered new possible monomorphizations: $\mathsf{emptyset}_{[\mathbb{Z}/\alpha_e]}$ (in 4.2a and 4.2b) and $\mathsf{insert}_{[\mathbb{Z}/\alpha_i]}$ (in 4.2c).

The procedure thus stops on a final problem if it does not uncover any more monomorphic instances. There, only two things can happen:

- if polymorphic terms are only found in their original place (i.e. as subterms of the original problem) then we have found a monomorphic expression of the original polymorphic problem, if we remove the original polymorphic assertions and functions from the final problem.
- if polymorphic terms are still present, then we declare that we have failed to find a monomorphic expression of the original problem and halt there.

On the example, the procedural steps we have applied will never terminate because in the term (flat emptyset) = emptyset, a new monomorphic type can be inferred for the leftmost emptyset at any instantiation of the rightmost emptyset , which will be fed to the rightmost one, thus looping forever.

Monomorphization fixpoint. The proposed monomorphization procedure is now summarized. Let T represent the set of term occurrences of the original problem. Initially, the sets of monomorphic and polymorphic term occurrences are empty.

1. Apply type inference and divide the terms in T into two sets M and P of monomorphic and polymorphic term occurrences. If they are the same as the previous M and P, we have reached a fixpoint and can stop. Otherwise, go to step 2.
2. Let $M = \{m_1, m_2, \ldots m_n\}$ and $P = \{p_1, p_2, \ldots p_m\}$. For each (m_i, p_j), such that $m_i \in M$, $p_j \in P$ and $m_i = p_j$ (i.e. m_i and p_j are two occurrences of the same term), substitute the

polymorphic p_j by its monomorphic m_i in the term t where p_j occurs as subterm, thus deriving a new term $t_{ij} = t[p_j := m_i]$.

At the end of this step, we will have a new set of terms from the various possible pairings between monomorphic and polymorphic occurrences of the same term, generating a new set of term occurrences T'. Let the new problem be represented by $T \cup T'$.

Non-termination. The fact that our procedure is possibly non-terminating is mitigated by the fact that, in practice, we impose restrictions on time or in this case, on the number of full passes. However, we would like to be able to guess possibly infinite expansions because we know we will not be able to guarantee a sound and complete monomorphization. To this effect, we have conjectured the following criterion, which depends on unification [Rob65]:

Conjecture 1 (Non-termination criterion)**.** *Let t and s be terms and s_1 and s_2 be two occurrences of s in the term t. Let T_1 be the type inferred for s_1 and T_2 the one of s_2. If T_1 and T_2* cannot *be unified* because of a failing occur check *then the monomorphization procedure will not terminate.*

5 Related work

The use of polymorphic logics on top of many-sorted or mono-sorted logics has received specific attention in the last few years.

In the context of the Caduceus and Why [FM07, BFMP11], Couchot and Lescuyer [CL07] describe how to translate ML-style polymorphic formulas into untyped and multi-sorted versions of the original problem.

This method is refined by Bobot and Paskevich [BP11] who show a 3-staged treatment of polymorphic formulas to translate them into many-sorted versions, including various possibilities for the last translating step. Their proposals particularly take care of protecting data types which are known to be handled by decision procedures by the targeted SMT-solvers. This work is further detailed in Bobot's thesis [Bob11].

Leino and Rümmer [LR10] consider the higher-order polymorphic specification language of the Boogie2 tool [Lei08] that has to be translated to SMT-solvers which in general do not handle polymorphism. They present two translations, one using type guards, the other adding types as further function arguments.

These two last approaches already present various advanced techniques to translate polymorphism formulas for many-sorted SMT-solvers, each one coming from their experience and needs, but do not tackle monomorphization. In the case of Bobot and Paskevich, their proof of the undecidability of the monomorphization makes clear the reason why, while Leino and Rümmer leave it as a possible further optimization.

Bobot *et al.* [BCCL08] have added built-in support for polymorphism inside the Alt-Ergo prover [BCC+08]. Supporting polymorphic types at the solver level would indeed simplify the addition of polymorphism at the specification level. We chose to keep things separated (for now).

There is also a large body of work on the translation of typed higher-order-logic into untyped first-order logic. In the context of using automation to help discharge proofs Hurd [Hur03] or Meng and Paulson [MP08] can however rely on the type-checking capabilities of higher-order provers to verify automated but untyped first-order proofs. Therefore they can even use unsound translations and leave to the higher-order prover the task of checking the soundness of

the proof. In order to use SMT-solvers inside Isabelle/HOL [Pau94], the monomorphization step of Blanchette *et al.* [BBP11] bears a strong similarity to our proposal.

6 Conclusion and further work

We have presented a summary of two backward compatible syntactic extensions made to the SMT-LIB standard: λ-terms and polymorphism. We have shown how to deal with λ-terms through β-reduction. We also have presented how we attempt to generate a monomorphic version of our polymorphic problem using a fixpoint-like procedure. This procedure heavily uses a Damas-Hindley-Milner like type inference algorithm to discover relevant monomorphic instances of polymorphic terms.

The support for the proposed syntactic extensions is currently being implemented in the development version of veriT. λ-terms are already supported and there is preliminary support for polymorphic SMT-LIB scripts. We are currently testing the monomorphization process to see how it behaves in practice.

The preservation of satisfiability means that, even in the case of a non-terminating monomorphization, we could devise a strategy to correctly use an unsatisfiable result at any step of the monomorphization process. Indeed, this would mean that we have found an unsatisfiable subset of the initial problem.

This strategy would be similar to depth-first *iterative deepening* [Kor85], which has been heavily used in provers based on the tableau method [Smu95, DGHP99]. In tableaux, this is used to generate possible term instantiations for universally quantified variables in order to find a model refuting the original formula. In our case, after each deeper monomorphization step, the SMT-solver would get to try a partial and new monomorphic problem containing only a subset of possible type instantiations. If it can prove the unsatisfiability of this new problem, we can stop. If not, we can — and should to preserve completeness — continue. In practice, this iterative procedure will be bounded either by a time limit or by a given number of steps.

We believe this work is a step towards a more generalized use of polymorphism in the context of SMT-solvers. These efforts might converge into an Alt-Ergo-like solution for provers supporting SMT-LIB, indeed building polymorphism support *in* the prover and not only at the syntactic level.

Acknowledgments. We warmly thank the anonymous reviewers for their helpful feedback and constructive criticism, which already show promises of future fruitful discussions about the subject of this paper.

References

[BBP11] Jasmin C. Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Børner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.

[BCC+08] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo Automated Theorem Prover, 2008. `http://alt-ergo.lri.fr/`.

[BCCL08] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[Bob11] François Bobot. *Logique de séparation et vérification déductive*. Phd thesis, Université Paris-Sud, December 2011.

[BP11] François Bobot and Andrey Paskevich. Expressing polymorphic types in a many-sorted language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2011.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[CL07] Jean-François Couchot and Stéphane Lescuyer. Handling Polymorphism in Automated Deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[Dé10] David Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In *Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2010.

[Dé13] David Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78(3):310–316, 2013.

[DFGV12] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT Solvers for Rodin. In *Proc. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer, 2012.

[DGHP99] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*. Springer, 1999.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[Hin69] Roger Hindley. The principle type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.

[Hur03] Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[Kor85] Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.*, 27(1):97–109, 1985.

[Lei08] K. Rustan. M. Leino. *This is Boogie 2*, 2008.

[LR10] K. Rustan M. Leino and Philipp Rümmer. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[MP08] Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[Pau94] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Smu95] R.M. Smullyan. *First-order Logic*. Dover books on advanced mathematics. Dover, 1995.

Invited Talk

Automating the Verification of Floating-point Algorithms

Guillaume Melquiond

Inria Saclay – Île-de-France & LRI, CNRS UMR 8623, Université Paris-Sud
Centre Universitaire d'Orsay, Bâtiment 650 (PCRI), Orsay, F-91405

Floating-point numbers are limited both in range and in precision, yet they are widely used as a way to implement computations on real numbers. Thus arithmetic operations introduce small errors which might be amplified during subsequent computations and cause inaccuracies. As such, proving the correctness of a floating-point algorithm usually entails verifying that the computed results are still close enough to some ideal values, despite the method error and the round-off errors. The traditional way to tackle such a verification is to perform an error analysis, possibly using automated tools.

Unfortunately, when it comes to the low-level functions found in mathematical libraries, the floating-point code is usually so contrived that this approach falls short. Indeed, just knowing the code is no longer sufficient to verify it, one also has to know the mathematical reasons that led to choosing this code in the first place. This excludes any hope of full automation, yet automated tools are sorely needed, if only because performing a pen-and-paper proof of such functions is long, tedious, and error-prone.

This talk will show some issues specific to the verification of the floating-point functions of a mathematical library, and some methods for solving them automatically. These methods will be exemplified using Gappa, a tool dedicated to proving the logical formulas that arise during the verification of small yet complicated floating-point algorithms. This tool is based on interval arithmetic, expression rewriting, and theorem saturation. For increased confidence, the tool also generates formal proofs which can be verified by the Coq proof assistant.

Leveraging Linear and Mixed Integer Programming for SMT

Tim King[1], Clark Barrett[1], and Cesare Tinelli[2]

[1] New York University
[2] The University of Iowa

Abstract

SMT solvers combine SAT reasoning with specialized theory solvers to either find a feasible solution to a set of constraints or prove that no such solution exists. Linear programming (LP) solvers come from the tradition of optimization, and are designed to find feasible solutions that are optimal with respect to some optimization function. Typical LP solvers are designed to solve large systems quickly using floating point arithmetic. Because floating point arithmetic is inexact, rounding errors can lead to incorrect results, making inexact solvers inappropriate for direct use in theorem proving. Previous efforts to leverage such solvers in the context of SMT have concluded that in addition to being potentially unsound, such solvers are too heavyweight to compete in the context of SMT. In this paper, we describe a technique for integrating LP solvers that dramatically improves the performance of SMT solvers without compromising correctness. These techniques have been implemented using the SMT solver CVC4 and the LP solver GLPK. Experiments show that this implementation outperforms other state-of-the-art SMT solvers on the QF_LRA SMT-LIB benchmarks and is competitive on the QF_LIA benchmarks.

raSAT: SMT for Polynomial Inequality (extended abstract)

To Van Khanh[1], Vu Xuan Tung[2], and Mizuhito Ogawa[2]

[1] University of Engineering and Technology,
Vietnam National University, Hanoi
khanhtv@vnu.edu.vn
[2] Japan Advanced Institute of Science and Technology
tungvx@jaist.ac.jp, mizuhito@jaist.ac.jp

Abstract

This paper presents an iterative approximation refinement, called raSATloop, which solves a system of polynomial inequalities on real numbers. The approximation scheme consists of interval arithmetic (over-approximation, aiming to decide UNSAT) and testing (under-approximation, aiming to decide SAT). If both of them fail to decide, input intervals are refined by decomposition, a refinement step in raSATloop.

The SMT solver raSAT implements the raSATloop, on top of the miniSAT 2.2, with backend theories in Ocaml. The raSATloop is not only a simple framework, but also allows us to design mutually refining strategies, e.g., the result of interval arithmetic refines both test data generation and next refinements, and the result of testing refines next refinements. We discuss three strategy design choices: *dependency* to set priority among atomic polynomial constraints, *sensitivity* to set priority among variables, and *UNSAT core* for reducing learned clauses and incremental UNSAT detection.

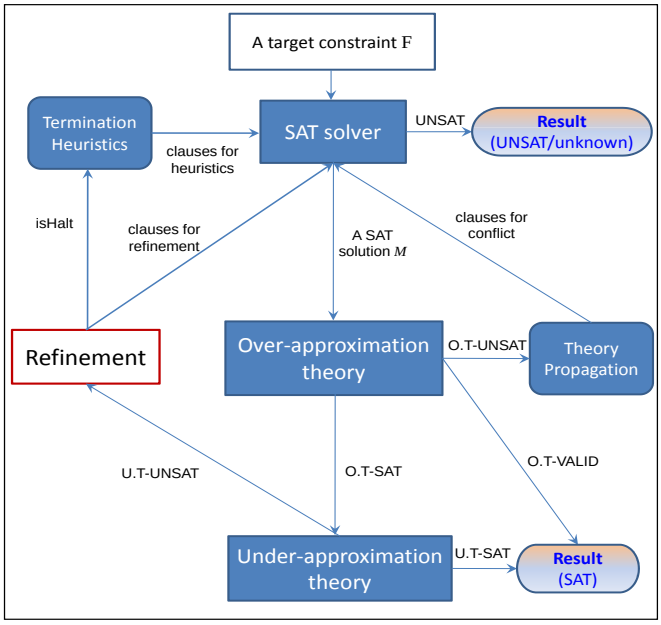


Figure 1: raSATloop

Better Answers to Real Questions

Marek Košta[1], Thomas Sturm[1] and Andreas Dolzmann[2]

[1] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
mkosta@mpi-inf.mpg.de, sturm@mpi-inf.mpg.de
[2] Leibniz-Zentrum für Informatik, 66041 Saarbrücken, Germany
andreas.dolzmann@dagstuhl.de

Abstract

We consider existential problems over the reals. Extended quantifier elimination generalizes the concept of regular quantifier elimination by providing in addition answers, which are descriptions of possible assignments for the quantified variables. Implementations of extended quantifier elimination via virtual substitution have been successfully applied to various problems in science and engineering. So far, the answers produced by these implementations included infinitesimal and infinite numbers, which are hard to interpret in practice. We introduce here a post-processing procedure to convert, for fixed parameters, all answers into standard real numbers. The relevance of our procedure is demonstrated by applications of our implementation to various examples from the literature, where it significantly improves the quality of the results.

Towards Conflict-Driven Learning for Virtual Substitution

Konstantin Korovin[1], Marek Košta[2] and Thomas Sturm[2]

[1] The University of Manchester, UK
korovin@cs.man.ac.uk
[2] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
mkosta@mpi-inf.mpg.de, sturm@mpi-inf.mpg.de

Abstract

We consider SMT-solving for linear real arithmetic. Inspired by related work for the Fourier–Motzkin method, we combine virtual substitution with learning strategies. For the first time, we present virtual substitution—including our learning strategies—as a formal calculus. We prove soundness and completeness for that calculus. Some standard linear programming benchmarks computed with an experimental implementation of our calculus show that the integration of learning techniques into virtual substitution gives rise to considerable speedups. Our implementation is open-source and freely available.

Author Index