SMT Workshop 2013
11th International Workshop on Satisfiability Modulo Theories

Helsinki, Finland, July 8th and 9th, 2013

Affiliated with the 16th International Conference on
Theory and Applications of Satisfiability Testing (SAT 2013)

`http://smt2013.fbk.eu`

Informal Proceedings

Preface

This volume contains the papers presented at the eleventh edition of the International Workshop on Satisfiability Modulo Theories (SMT-2013). The workshop was held on July 8th and 9th, 2013, in Helsinki, Finland, in association with the Sixteenth International Conference on on Theory and Applications of Satisfiability Testing (SAT 2013).

The Workshop is certainly the main annual event of the SMT community, where both researchers and users of SMT technology can meet and discuss new theoretical ideas, implementation and evaluation techniques, and 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, that SMT Workshop attendees may be unaware of. We received 12 papers. Each submission 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 works in progress, we were able to accept all contributions for presentation at the workshop: 5 original papers, 4 extended abstracts/works in progress, and 3 presentation-only papers. The program included also two invited talks, by Sylvain Conchon from Université Paris-Sud and Thomas Sturm from the Max Planck Institüt für Informatik or Saarbrücken.

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 also the SAT organizers for their support and for hosting the workshop, and the EasyChair team for the availability of the EasyChair Conference System.

June 2013 Roberto Bruttomesso and Alberto Griggio

Table of Contents

Cubicle: Design and Implementation of an SMT based Model Checker for Parameterized Systems

Sylvain Conchon
LRI, Université Paris-Sud
sylvain.conchon@lri.fr

Abstract

The automatic verification of safety properties like mutual exclusion or cache coherence for multi-core or distributed systems is very challenging. Consider for instance that, in the Stanford FLASH multiprocessor architecture, the transition system describing the cache coherence protocol has already more than 67 million states when just four processors are in competition. On these large problems, efficient state-of-the-art model checkers reach their limits in both time and memory consumption. In particular, all model checkers fail to prove the safety of FLASH for more than five processes. In this talk, I will present Cubicle, a new SMT based model checker that is able to automatically prove a protocol like FLASH.

Cubicle is based on the model checking modulo theories (MCMT) framework introduced by Ghilardi and Ranise. It is used to verify safety properties of array-based systems. This is a syntactically restricted class of parameterized transition systems with states represented as arrays indexed by an arbitrary number of processes. The kernel of Cubicle is a new symbolic backward reachability procedure with approximations and backtracking using Satisfiabilty Modulo Theories.

Cubicle is written in OCaml. Its SMT solver is a tightly integrated, lightweight and enhanced version of Alt-Ergo. Cubicle is available at http://cubicle.lri.fr.

Effective Quantifier Elimination and Decision - Theory, Implementations, Applications, Perspectives

Thomas Sturm
MPI für Informatik, Saarbrücken, Germany
sturm@mpi-inf.mpg.de

Abstract

Effective quantifier elimination procedures for first-order theories provide a powerful tool for generically solving a wide range of problems based on logical specifications. In contrast to general first-order provers, quantifier elimination procedures are based on a fixed set of admissible logical symbols with an implicitly fixed semantics. This admits to make use of sub-algorithms from symbolic computation. The talk gives an overview of several theories that admit quantifier elimination and for which there are implementations available in our open-source software Redlog. The focus is on real quantifier elimination and its applications and on the integers. We are furthermore going to sketch recent work on an incomplete decision procedure for the existential fragment of the reals, which is quite different from SMT approaches. It is our hope to stimulate discussions and to contribute to closing gaps between scientific communities.

Efficiently Solving Bit-Vector Problems Using Model Checkers

Andreas Fröhlich, Gergely Kovásznai, Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria*

Abstract

Bit-precise reasoning is essential in many applications of Satisfiability Modulo Theories (SMT). Most approaches for solving quantifier-free fixed-size bit-vector logics (QF_BV) rely on bit-blasting. In previous work, we have shown that bit-blasting is not polynomial in general [19], and later proposed QF_BV$_{\ll 1}$, a class of bit-vector problems that is PSPACE-complete [15]. In this paper, we give examples of how to create (polynomial) SMV specifications out of QF_BV$_{\ll 1}$ formulas. We then use various model checkers to solve those problems and give detailed experimental results. Our results show that BDD-based model checkers outperform current SMT solvers by several orders of magnitude on our benchmarks. Unrolling and using SAT-based model checking turns out to be the same as bit-blasting and gives worse results. In addition to this, our approach allows us to easily generate new challenging benchmarks for SMT solvers as well as for model checkers.

1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for fixed-sized bit-vector logics are Boolector [6], MathSAT [8], STP [16], Z3 [11], and Yices [12]. All these solvers rely on *bit-blasting* in order to translate bit-vector formulas into propositional logic (SAT). The result is then checked by a SAT solver.

In practice, e.g. in the SMT-LIB [1], the BTOR [7], and the Z3 format, the bit-widths in bit-vector formulas are encoded as binary, decimal, or hexadecimal numbers, i.e., a *logarithmic encoding* is used. In [19], we proved that the encoding of bit-widths affects the complexity of the decision problem of bit-vector logics. In particular, logarithmic encoding makes the quantifier-free fragment QF_BV NEXPTIME-complete.[1] Thus, bit-blasting is *not polynomial* in general. Consider the following example (in SMT2 syntax):

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(declare-fun z () (_ BitVec 1000000))
(assert (= z (bvadd x y)))
(assert (= z (bvshl x (_ bv1 1000000))))
(assert (distinct x y))
```

This formula verifies that for an arbitrary bit-vector x of bit-width one million, there exists no bit-vector $y \neq x$ with $x + y = x \ll 1$. Written to a file, this formula can be encoded with 225 bytes. Using the SMT solver Boolector (even with all rewritings switched on), bit-blasting

*This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE).

[1] In [19], we introduced the notation QF_BV1 resp. QF_BV2 for QF_BV using a *unary* resp. a *logarithmic*, actually without loss of generality, *binary encoding*. In this paper, QF_BV will always refer to the logarithmic/binary case.

produces a circuit of size 129 MB encoded in the actually rather compact AIGER format. Tseitin transformation results in a CNF in DIMACS format of size 843 MB.

In related work [20], we tried to avoid this growth in size by giving a translation from QF_BV to EPR and then using iProver to solve the problem. In most cases, this approach turned out to perform worse than Boolector on the original instance. Since QF_BV is NExpTime-complete, it is not clear if it is possible to solve the general case more efficiently. However, the given example only uses *addition*, *shift by one* and *equality*. In [15], we showed that this kind of formulas can be expressed by QF_BV$_{\ll 1}$, a subset of QF_BV which turned out to be PSpace-complete. In order to prove this, we gave a polynomial translation from QF_BV$_{\ll 1}$ to Sequential Circuits, similar to the one for linear arithmetic on *non-fixed-size* bit-vectors proposed in [22, 23].

In this paper, we show how model checkers can be used to solve *fixed-size* bit-vector problems of this class. In contrast to [15] which provided the theoretical background, we now focus on experimental evaluation and analyze the potential benefits for efficiently solving bit-vector formulas. First, in Sec. 2, we provide a short overview of our translation as described in [15] and give some examples to show how we used this concept to convert SMT2 files to SMV. In Sec. 3, we then describe some benchmarks that we generated to evaluate the performance of various model checkers compared to state-of-the-art SMT solvers with support for fixed-sized bit-vector logics. On most of our benchmarks, BDD-based model checkers turn out to be faster by several orders of magnitude. We provide experimental data and discuss the results in detail. Finally, in Sec. 4, we conclude the paper and discuss further topics for future work.

2 QF_BV$_{\ll 1}$ to SMV

In [22, 23], the authors gave a polynomial translation for linear arithmetic on *non-fixed-size* bit-vectors (QFPAbit) into Sequential Circuits. In contrast to [22, 23], we focus on *fixed-size* bit-vectors but share the goal of avoiding the exponential explosion due to explicit state representation as for example used in MONA [18]. We adapted this translation in [15] to deal with fixed-size bit-vectors and extended it by various other operators like *shift by one* and *indexing*.

Given a bit-vector formula $\Phi \in$ QF_BV$_{\ll 1}$ without nested equalities. Let n be a bit-width, $x^{[n]}, y^{[n]}$ denote bit-vector variables, $c^{[n]}$ a bit-vector constant, and $t_1^{[n]}, t_2^{[n]}$ bit-vector terms only containing bit-vector variables and bitwise operations. Following [22, 23], we assume w.l.o.g that Φ only consists of the following types of *atoms*: $t_1^{[n]} = t_2^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \ll 1^{[n]}$. It is easy to check that any QF_BV$_{\ll 1}$ formula can be written like this with only a linear growth in the number of original variables.

We encode each atom in Φ separately into an atomic Sequential Circuit. The encoding itself is straightforward in most cases. A concrete example translating QF_BV to SMV is given after the theoretic part of this section. Compared to [22, 23], we have to consider the fact that all bit-vectors have a fixed bit-width.

Let n_{max} be the maximal bit-width of all bit-vectors in the formula. We construct an additional Sequential Circuit representing a counter. The counter initially is set to 0 and is incremented by 1 in each clock cycle. A counter like this can be realized with $\lceil \log_2(n_{max}) \rceil$ latches, i.e. polynomially in the size of Φ.

Now, for each atomic Sequential Circuit, we add a check whether the value of the counter reached the bit-width n of the bit-vector variables corresponding to the input streams of the circuit. Once this is the case, the individual circuit does not change its output value anymore.

Since $n_{max} \geq n$, this will always hold at some point.[2]

Finally, after constructing all atomic circuits, their outputs are combined by logical gates following the Boolean structure of Φ. Other operators, such as *addition* or *indexing*, can either be replaced by *shift by one* in a preprocessing step or directly encoded into a Sequential Circuit [15].

We now show the translation for the motivational example given in Sec. 1 to the concrete SMV-format. First of all, a counter for the bit-width of the variables has to be introduced. This can be done using logarithmic many variables:

```
init(counter_bit0) := FALSE;
next(counter_bit0) := counter_bit0 xor (TRUE);
init(counter_bit1) := FALSE;
next(counter_bit1) := counter_bit1 xor (counter_bit0);
...
init(counter_bit19) := FALSE;
next(counter_bit19) := counter_bit19 xor (counter_bit0 & ... & counter_bit18);
```

We then keep track of whether the counter already reached the value of a certain bit-width.[3] This variable later serves as a guard for all atoms containing variables of the given bit-width:

```
init(counter_gte_1000000) := FALSE;
next(counter_gte_1000000) := counter_gte_1000000 |
  (counter_bit0 & counter_bit1 & ... & !counter_bit6 & ... & counter_bit19);
```

After introducing those helper variables, the actual formula can now be translated. The *distinct* operator is first replaced by negation of an *equality*. The translation to SMV then is straightforward:

```
init(atom_equal) := TRUE;
next(atom_equal) := case
  counter_gte_1000000 : atom_equal;
  TRUE : atom_equal & (x <-> y);
esac;
```

For translating *addition*, two atoms have to be introduced since the carry bit has to be remembered in the next step:

```
init(atom_add) := TRUE;
next(atom_add) := case
  counter_gte_1000000 : atom_add;
  TRUE : atom_add & (z <-> (x xor y xor atom_cin));
esac;

init(atom_cin) := FALSE;
next(atom_cin) := case
  counter_gte_1000000 : atom_cin;
  TRUE : atom_add & ((x & y) | (x & atom_cin) | (y & atom_cin));
esac;
```

[2] In contrast to [22], we assume that the input streams for all variables start with the least significant bit.

[3] The counter bits in the *next*-statement correspond to the binary representation of $n - 1$ (i.e. $999999_{10} = 11110100001000111111_2$ in our example).

The *shift* operator can be translated in a very similar way but will not be given here explicitely to keep the example short. Another way would be to replace $(x \ll 1)$ by $(x + x)$ in the preprocessing step.

Finally, the specification is defined by the logical combination of the individual atoms and additionally respecting the bit-width:

```
AG(!counter_gte_1000000 | !atom_add | !atom_shift | atom_equal)
```

We also implemented our translation including various operators in a tool called BV2SMV. Binaries and source code are available for download at [9].

3 Experiments

We first describe our benchmark sets. We generated six different sets of QF_BV formulas in SMT2 format. All sets of benchmarks consist of 32 instances each and have two attributes: First, all benchmark sets are *not bit-width bounded* [15]. Because of this, bit-blasting is known to be exponential in general. Second, all benchmarks only contain *bitwise operators*, *addition*, *subtraction*, *shift by one*, *indexing* and *relational operators*. This ensures that a polynomial translation to SMV exists. The different instances in a particular set of benchmarks only differ in the bit-width of their variables and constants. The bit-widths n of the individual instances are of the form $n = 2^i$ and $n = 1.5 \cdot 2^i$ with $i \in \{5, \ldots, 20\}$ for all six sets. All benchmarks will be submitted to the `QF_BV` category of SMT-LIB.

`QF_BV/froehlichkovasznaibiere/ndist.a.`n: We verify that, for two bit-vector variables $x^{[n]}, y^{[n]}$, it holds that $x^{[n]} < y^{[n]}$ implies $(x^{[n]} + 1^{[n]}) \leq y^{[n]}$. The instances are unsatisfiable and use *addition* and *unsigned less/greater than operators*.

`QF_BV/froehlichkovasznaibiere/ndist.b.`n: We give a counter-example (due to overflow) to the claim that, for two bit-vector variables $x^{[n]}, y^{[n]}$, it holds that $(x^{[n]} + 1^{[n]}) \leq y^{[n]}$ implies $x^{[n]} < y^{[n]}$. The instances are satisfiable and use *addition* and *unsigned less/greater than or equal operators*.

`QF_BV/froehlichkovasznaibiere/power2bit.`n: We verify that, for a bit-vector variable $x^{[n]} = 2^j$, it is not possible for two different bits to be both set to 1. The instances are unsatisfiable and use *indexing*, *subtraction*, *bitwise operators*, and *(in)equality*.

`QF_BV/froehlichkovasznaibiere/power2eq.`n: We verify that, for two bit-vector variables $x^{[n]} = 2^j, y^{[n]} = 2^k$, with a certain identical bit set to 1, the bit-vectors cannot be distinct. The instances are unsatisfiable and use *indexing*, *subtraction*, *bitwise operators*, and *(in)equality*.

`QF_BV/froehlichkovasznaibiere/power2sum.`n: We verify that, for two bit-vector variables $x^{[n]} = 2^j, y^{[n]} = 2^k$, with $j \neq k$, $x^{[n]} + y^{[n]}$ cannot be a power of 2. The instances are unsatisfiable and use *addition*, *subtraction*, *bitwise operators*, and *(in)equality*.

`QF_BV/froehlichkovasznaibiere/shift1add.`n: We verify that for an arbitrary bit-vector $x^{[n]}$, there exists no bit-vector $y^{[n]} \neq x^{[n]}$ with $(x^{[n]} + y^{[n]}) = (x^{[n]} \ll 1)$. The instances are unsatisfiable and use *addition*, *shift by one*, and *(in)equality*. The example used throughout the paper is part of this benchmark family.

Out of the benchmark instances in SMT2 format, we generated SMV instances by using BV2SMV and the flattening tool smvflatten.[4] We used the state-of-the-art SMT solvers Boolector, MathSAT, Z3, and STP on the SMT2 instances, and NuSMV [10] on the corresponding SMV instances. In order to involve state-of-the-art model checkers like Tip [13] and IImc[5] (that

[4] `http://fmv.jku.at/smvflatten/`
[5] `http://ecee.colorado.edu/wpmu/iimc/`

uses techniques described in [2, 3]), we also converted all the SMV instances to AIGER format by using the translation tool smvtoaig that is part of the AIGER distribution.

All our experiments were run on the same cluster and with the same setup as the latest Hardware Model Checking Competition (HWMCC'12).[6] More precisely, we used a 32-node cluster with Intel Quad Core 2.6 GHz processors and 8 GB RAM. The wall clock time limit was set to 900 seconds and the memory limit to 7 GB. Each solver had full access to one node (4 cores).

In total, we used 19 different solvers (resp. configurations) on 6 different benchmark sets each consisting of 32 instances, yielding a total of 3648 runs. All our results are available on our web page at [9] together with generation scripts for all benchmarks in SMT2 format and our tool BV2SMV.

Tab. 1 provides an overview of the total number of solved instances and the average runtime (in seconds) and space requirement (in megabytes) on the solved instances. For BMC solvers, we used the knowledge that the counters in the generated specifications only allow the atomic circuits to change their value in the first number of steps equal to the bit-width n of the original SMT2 formula. We therefore set the bound for unrolling to be equal to $n + 1$ and, whenever a BMC solver reached the bound without timeout or out-of-memory, counted the instance to be shown unsatisfiable.

The solvers were executed with default settings if not stated otherwise explicitely. However, in some exceptional cases, we intentionally used some promising or interesting strategies. For instance, in Tab. 1, `Tip-BMC` references Tip using BMC-based strategy. Since we expected and later experienced that BDD-based techniques perform particularly well on our benchmarks, we intended to test model checkers with BDD-based strategies, those which offer such an option. Note that NuSMV uses BDD-based forward reachability analysis by default. We also tested NuSMV with backward reachability analysis, referenced by `NuSMV-bw`. IImc also offers BDD-based solving strategy, with both forward resp. backward reachability analysis; we reference IImc with default settings resp. with BDD-based forward resp. backward reachability analysis as `IImc` resp. `IImc-BDD-fw` resp. `IImc-BDD-bw`.

	STP	Boolector	MathSAT5	Z3	IImc-BDD-bw	NuSMV-bw	IImc-BDD-fw	IImc	NuSMV	Blimc[‡]	Tip-BMC[‡]	Aigbmc[‡]	Tip
solved	147	146	127	123	**192**	189	185	172	170	147	130	99	93
sat	23	**32**	13	23	**32**	29	**32**	**32**	27	9	31	21	17
unsat	124	114	114	100	**160**	**160**	153	140	143	138	99	78	76
time	206	190	310	171	12	30	79	132	148	233	266	295	496
space	1063	805	587	2180	8	24	9	74	38	95	1142	2073	6

Table 1: Overall results for all solvers

Apart from those in Tab. 1, we tested other models checkers as well, all submitted to HWMCC'12. We excluded some of them due to uncertain results: (a) Super_prove2 and Simple_sat, which employ ABC with improved strategies, produced discrepancies on some satisfiable

[6] `http://fmv.jku.at/hwmcc12/`

[‡] Versions submitted to HWMCC'12.

instances; (b) PdTrav, on some instances, threw exception about syntactical error in input.

In total, `IImc-BDD-bw` clearly performs best as it can solve all instances. Backward reachability analysis seems to produce better results than forward reachability for BDD-based model checkers in general. While this applies especially to unsatisfiable instances, `NuSMV-bw` only performs slightly better than `NuSMV` on the satisfiable ones. Interestingly, `Boolector` also gives very good results for the satisfiable instances. As expected, in particular the average space requirement of all SMT solvers is very large.

Fig. 1, 2, and 3 provide a detailed overview of the runtimes and space requirements of various solvers on the individual benchmark sets. We chose `Boolector` and `STP` representing the SMT solver class and `NuSMV`, `NuSMV-bw`, `IImc`, `IImc-BDD-bw`, and `Tip-BMC` as model checkers. Please consider that sampling memory is imprecise in case of low runtime, causing noise on the plots that show memory consumption.

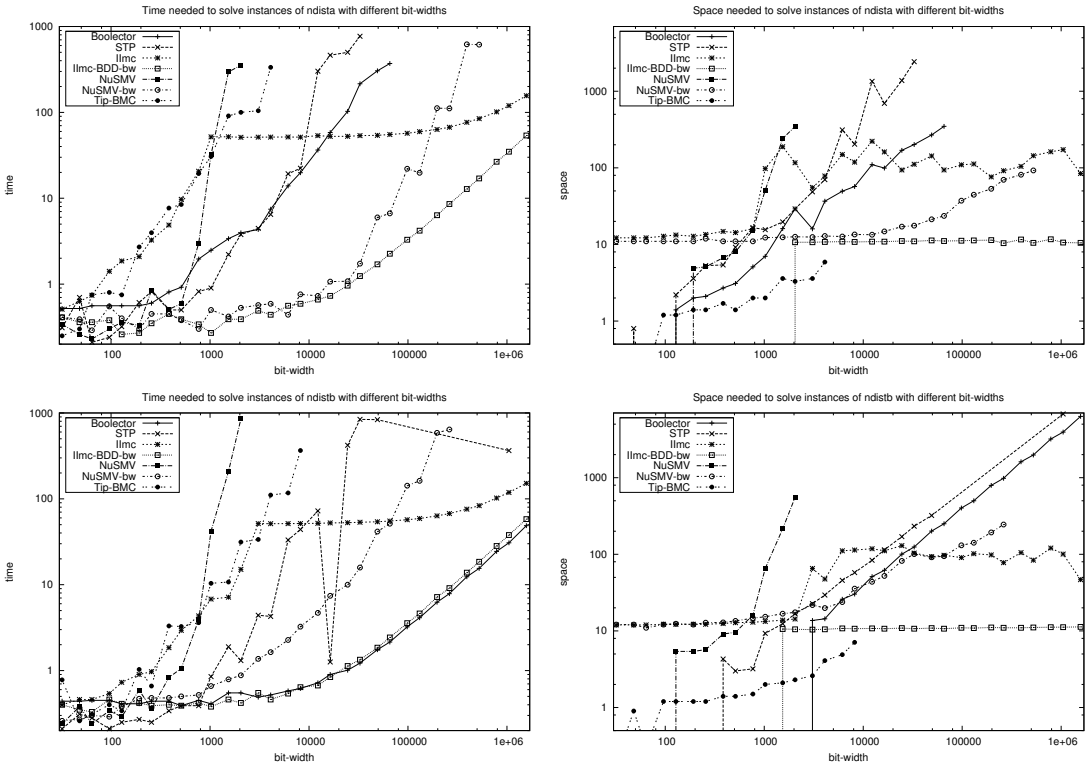


Figure 1: Detailed results of the `ndist.a` and `ndist.b` benchmark sets.

Fig. 1 shows the results of the solvers on the `ndist.a` and `ndist.b` benchmark sets. On the `ndist.a` instances, all BDD-based model checkers clearly outperform both SMT solvers considering time and space. `Tip-BMC` performs very similar to the SMT solvers. This is not surprising since unrolling up to a bound equal to the bit-width will in the end produce the same propositional formula as bit-blasting.

With `ndist.b` being satisfiable, SMT solvers show better runtimes while still requiring similar amounts of space. This can be explained by the fact that it is enough to guess the correct assignment which might be found as a consequence of good heuristics and at the same time

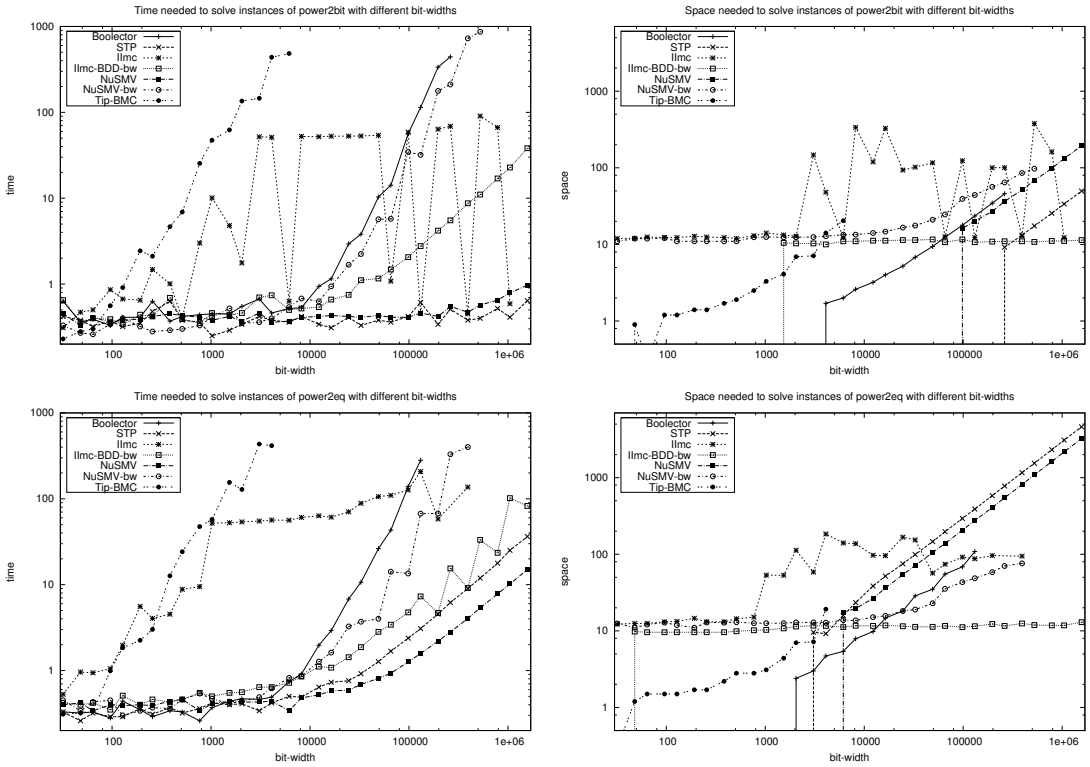


Figure 2: Detailed results of the `power2bit` and `power2eq` benchmark sets.

could cause the variation in the runtimes of `STP`. While backward reachability analysis seems to give a clear advantage on the unsatisfiable benchmark, it only slightly increases performance on the satisfiable one.

One interesting aspect in Fig. 2 is the fact that `STP` performs really well on both benchmarks. We suppose that this is connected to the fact that `power2bit` and `power2eq` both use indexing with relatively small indices. Interestingly, `Boolector` performs much worse on both instances. The good performance on this kind of formulas, therefore, does not seem to be a result of bit-blasting and applying SAT solvers but rather due to some special technique used in `STP`.

One might notice the typical shape of the runtime curves related to `IImc`: they start steep, but above a certain bit-width they show rather moderate ascent. The curves representing space consumption seem to grow slowly up to a certain point where, after a big jump, space usage almost seems to be fixed to a constant or, in some cases, even starts to decrease. We think that this strange behavior is due to the fact that `IImc` uses several scheduled approaches, such as IC3 [2], BMC, BDDs, etc. Probably due to the same fact, the `IImc` curves are even more hectic on the `power2bit` benchmark in Fig. 2. During our experiments we also tested IImc with IC3 strategy alone, resulting in timeouts on most instances. Therefore, we assume that above a certain bit-width IImc with default scheduling switches to BDDs, resulting in moderate ascent in memory consumption and runtime.

Probably Fig. 3 depicts most properly the distinction between BDD-based approaches and those which use SAT-based ones. Although SMT solvers and `Tip-BMC` time out quite soon on

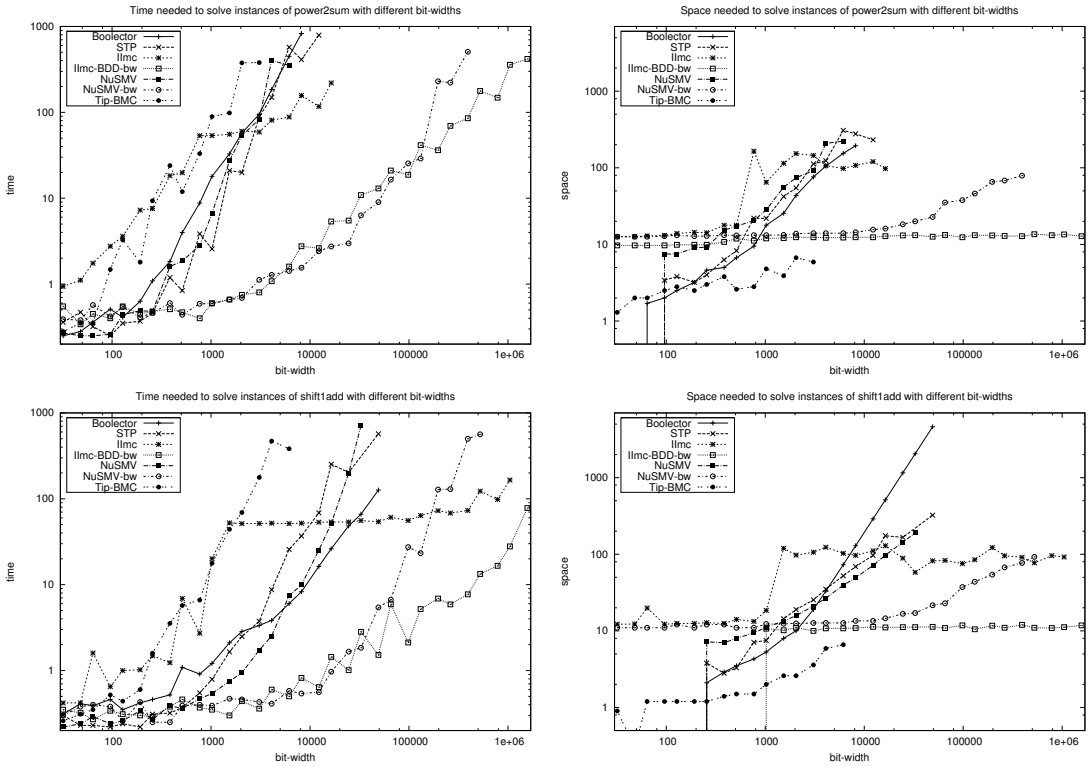


Figure 3: Detailed results of the `power2sum` and `shift1add` benchmark sets.

both problem sets, and, on the `power2sum` benchmark, the performance of `IImc` now is rather similar, BDD-based model checkers are able to deal even with very large bit-widths.

In general, looking at the runtimes, we can see that SMT solvers can compete well on instances with smaller bit-width, while BBD-based model checkers start to outperform their counter-parts with growing bit-width.

This effect becomes even stronger when we look at the space used during solving the formulas. Judging from the graphs, it might even be possible that the space requirement of BDD-based model checkers is logarithmic compared to that of SMT solvers. This could be the case due to the fact that SMT solvers apply bit-blasting, which is exponential for benchmarks that are not bit-width bounded, while our translation does not cause the problems to leave PSpace. However, this alone is not sufficient. BDD-based model checkers like NuSMV might create exponential sized BDDs nevertheless. More rigorous arguments or larger empirical analysis are needed.

4 Conclusion

In this paper, we efficiently solved quantifier-free bit-vector formulas using model checkers. While state-of-the-art SMT solvers usually apply bit-blasting to solve this kind of formulas, we already showed in previous work [19] that this can cause an exponential blowup in general. An approach for polynomially translating QF_BV to EPR exists [20] (as well as exponential

ones [14, 17]), but solving the resulting formulas also suffers from the NExpTime-completeness of EPR [20, 21]. Building on previous complexity results [15], however, we know that restricting QF_BV to only allowing *bitwise operators*, *shift by one*, *addition*, *subtraction*, *multiplication by constant*, *relational operators* and *indexing* leads to PSpace-completeness of the resulting logic. This allows us to polynomially translate bit-vector formulas to Sequential Circuits and use model checkers for reachability analysis.

In order to show the potential benefit of our approach, we created a set of benchmarks and used it to compare the performance of various model checkers on the translated instances to the one of current SMT solvers on the original files. We showed that on most of our problems, state-of-the-art model checkers like IImc and even older ones, such as NuSMV, performed better by several orders of magnitude considering runtime as well as space.

Our results also showed that BDD-based model checking techniques perform much better than SAT-based model checkers. This probably is the case because of the similarity between BMC and bit-blasting, and gives reason to investigate especially BDD-based solving techniques further.

Some of the best results were achieved by NuSMV. Considering the fact that NuSMV has seen relatively little development during the last years compared to current SMT solvers, this could lead to even better results if it is possible to improve the underlying techniques.

One of the main reasons we assume to be responsible for the good performance of model checkers on our benchmarks, is their better fit to the PSpace-nature of this problem class. Still, the resulting BDDs can of course be exponential in general.

While we did not pay special attention to the variable ordering during our translation, we ran NuSMV using `-dynamic` command, letting it figure out a good variable order during runtime. We also used the `-reorder` command to output the optimal variable order found by NuSMV and to look for patterns in it. When using this variable order in a second run instead of choosing the order dynamically, the runtimes usually decreased further.[7] Maybe our translation can be adapted using additional information to directly create variable orders that result in smaller BDDs. In order to do this, it might be interesting to look at the structure of the instances produced by our translation more closely. Especially the usage of counter definitions and constraints is similar throughout all formulas.

Sequential optimization techniques, such as those implemented in state-of-the-art model checkers like ABC [4], are useful even for bounded model checkers which otherwise only rely on unrolling. It is an interesting question whether it is possible to lift these techniques from model checking to bit-vector reasoning in combination or as a preprocessing step before bit-blasting.

Finally, only one model checker could solve all of our instances for the largest bit-widths. Constructing this kind of formulas, therefore, offers an easy way to provide challenging benchmarks for state-of-the-art SMT solvers and model checkers at the same time. For better solvers and future challenges, the difficulty of a problem can be adjusted by simply increasing the bit-width of the original SMT formula.

As a related classification problem, it will be interesting to investigate the complexity of Presburger arithmetic on fixed-size bit-vectors.[8] While the corresponding decision problem is known to be NP-complete for non-fixed-size bit-vectors, it is not clear whether we still remain in NP when considering fixed-size bit-vectors and whether translations as proposed in [5] are polynomial if a logarithmic encoding is used for the bit-widths.

[7] This is not included in our results since we did not analyze it in detail yet.
[8] The benchmark sets `ndist.a` and `ndist.b` are in this class.

References

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[2] Aaron R. Bradley. Sat-based model checking without unrolling. In *Proc. VMCAI'11*, pages 70–87, 2011.

[3] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Proc. FMCAD'11*, pages 144–153, 2011.

[4] Robert K. Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Proc. CAV'10*, pages 24–40, 2010.

[5] Raik Brinkmann and Rolf Drechsler. Rtl-datapath verification using integer linear programming. In *Proc. ASP-DAC'02*, 2002.

[6] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.

[7] Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: bit-precise modelling of word-level problems for model checking. In *Proc. BPR'08*, pages 33–38, 2008.

[8] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT SMT solver. In *Proc. CAV'08*, pages 299–303, 2008.

[9] BV2SMV project page. Website. `http://fmv.jku.at/bv2smv/`.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv version 2: An opensource tool for symbolic model checking. In *Proc. CAV02*, 2002.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proc. ETAPS'08*, pages 337–340, 2008.

[12] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[13] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[14] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In *FMCAD'10*, pages 137–144, 2010.

[15] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In *Proc. CSR'13 (to appear)*, 2013.

[16] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[17] Zurab Khasidashvili, Mahmoud Kinanah, and Andrei Voronkov. Verifying equivalence of memories using a first order logic theorem prover. In *FMCAD'09*, pages 128–135, 2009.

[18] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. Mona implementation secrets. In *Proc. CIAA'00*, pages 182–194, 2000.

[19] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proc. SMT'12*, pages 44–55, 2012.

[20] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Bv2epr: A tool for polynomially translating quantifier-free bit-vector formulas into epr. In *Proc. CADE'13 (to appear)*, 2013.

[21] Harry R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.

[22] Andrej Spielmann and Viktor Kuncak. On synthesis for unbounded bit-vector arithmetic. Technical report, EPFL, Lausanne, Switzerland, February 2012.

[23] Andrej Spielmann and Viktor Kuncak. Synthesis for unbounded bit-vector arithmetic. In *Proc. IJCAR'12*, volume 7364 of *LNCS*, pages 499–513, 2012.

A Difference Logic Formulation and SMT Solver for Timing-Driven Placement

Andrew Mihal

Tabula Inc., Santa Clara CA 95054, USA
amihal@tabula.com

Abstract

This paper presents a difference logic constraint satisfaction formulation and a custom SMT solver for programmable logic detailed placement problems. This problem domain is characterized by a large solution space with high space and time costs for generating constraints. To handle these problems efficiently, our solver features a dynamic clause generation callback interface to allow clauses to be added on demand during the search. The formulation and the solver both utilize concepts from static timing analysis to handle timing constraints. We show that the SMT approach provides better runtime and better quality of results than our previous Boolean SAT encoding.

1 Introduction

Semiconductor design automation tools have long made use of Boolean SAT for logic optimization, verification, test generation, and routing [13, 10, 5, 15, 21]. SAT-based placement has attracted less research interest. In the context of programmable logic, placers assign components from a netlist graph onto discrete sites on a programmable fabric. The result must satisfy a variety of constraints such as timing and routability. Devadas described placement via SAT-based bipartitioning in [4] but concluded that SAT solvers of the time were not yet powerful enough to solve more general 2-D placement problems.

Simulated annealing has been used effectively for placement [1], but this technique has disadvantages that can be addressed by a constraint satisfaction approach. Annealers make small perturbations to an initial placement (e.g. swapping the positions of two components) and accept or reject changes based on the total delta cost computed by a set of cost functions.

The simplicity of the move set makes it difficult for annealers to solve problems that require a coordinated change involving many netlist components. Such problems must be solved using a series of independent moves. The cost functions must be carefully crafted to ensure that partial progress towards the final goal is seen as gradual improvement. This property can be difficult to arrange, especially if the problem requires moving unrelated, non-violating components out of the way in order to make room for the components that are actually violating constraints.

A constraint satisfaction approach can directly address this weakness. Instead of searching for small changes that gradually approach an acceptable solution, a placer based on constraint satisfaction could rearrange a large number of components simultaneously such that the result satisfies all of the constraints. This strategy has the potential to outperform simulated annealing if it can be made efficient.

Scalability is a major technical obstacle to building a practical placer based on constraint satisfaction. In order for a sequential circuit to run at a specified clock rate, every state-to-state path in the netlist must have a total delay less than or equal to the target clock period. The number of such paths can be exponential in the size of the netlist.

Furthermore, each netlist edge has a number of placement options equal to the product of the number of candidate sites for the source and sink components. The routing delay between the source and sink sites on the fabric determines the edge delay, which in turn contributes to the path delays. Computing all of the fabric routing delays and generating the edge and path constraints is too expensive in runtime and memory to build a practical placer.

To solve this scalability problem, we use a two-part solution. First, concepts from static timing analysis are used to address the worst-case exponential scaling of path-based timing constraints. This formulation is presented in Section 3. Second, we use a dynamic clause generation approach to construct clauses lazily. We find that the solver is able to find a SAT or UNSAT result after exploring only a small fraction of the total search space, and therefore a majority of the clauses can be omitted entirely. This technique is presented in Section 4.

In the development of our detailed placer, our first approach was to use a purely Boolean formulation and solve it using a regular SAT solver. We explain this translation in Section 5 and discuss the shortcomings that motivated moving to an SMT solver that accepts difference logic constraints directly.

In Section 6, we present a custom difference logic SMT solver for the detailed placement problem domain. This solver specifically handles dynamic clause generation and the types of difference logic constraints that arise from our formulation of timing constraints based on static timing analysis. Section 7 gives experimental results comparing the difference logic formulation to our original purely Boolean SAT formulation. We also show the runtime improvement from dynamic clause generation. To get started, the next section briefly introduces the detailed placement problem for programmable logic.

2 Programmable Logic Detailed Placement

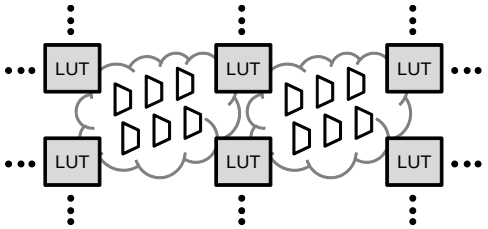
Programmable logic devices are implementation platforms for digital electronics that offer low up-front design costs and bypass the complexities of nanometer-scale transistor design. The general idea is that a prefabricated chip can be configured to implement any desired digital circuit by setting programmable bits on the chip.

Figure 1 shows an array of n-input lookup tables interconnected by a rich network of multiplexers. Each lookup table can implement any combinational function of n inputs by programming the 2^n bits of memory contained therein. Lookup tables are connected together by programming memory bits that drive the select signals of the multiplexers. A modern device may contain more than one million such lookup tables and often includes other resources such as flip-flops and memories. Kuon et al. [11] provide a survey of modern architectures.

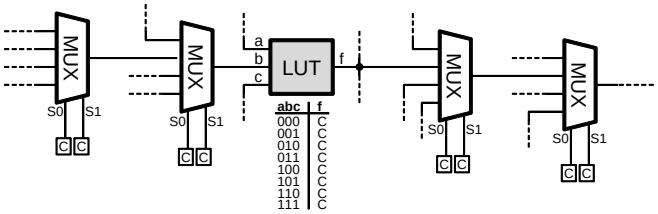
Engineers create a design for implementation on programmable logic by writing a register-transfer level circuit description in Verilog or VHDL. A suite of design automation tools provided by the programmable logic vendor compiles the design and produces the programmable bit values for the chip. The compilation process has four major phases: synthesis, global placement, detailed placement, and routing. Synthesis compiles the circuit description into an optimized netlist of lookup tables. Placement assigns the lookup tables in the netlist onto lookup tables on the physical chip. Routing configures the multiplexers so that the connections specified in the netlist are made on the chip. Each phase poses intriguing optimization challenges. Chen et al. [2] give a broad survey of the techniques commonly used.

This paper focuses on the detailed placement problem, and specifically on timing optimization. Detailed placers assume that the netlist already has a placement that satisfies some global optimization criteria but still has problems of a more local nature that need to be repaired. For example, a global placer could use an analytical algorithm with a continuous coordinate system and an abstract geometric model of routing delay. The detailed placer refines this placement by snapping components to overlap-free discrete placement sites using a more accurate routing delay model based on actual paths through the interconnect network.

In addition to providing a Verilog or VHDL circuit description, users also specify that the sequential logic must run at a particular clock frequency. To meet this constraint the placer must ensure that all state-to-state paths in the netlist are placed such that the worst path delay is less than or equal to the target clock period τ.



(a) Array of LUTs



(b) Programmable LUT Connections

Figure 1: Generic Programmable Logic Device Architecture

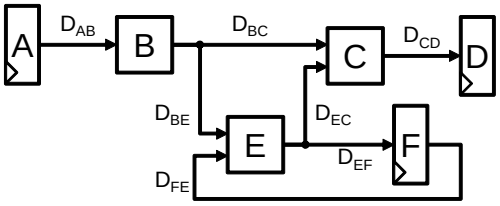


Figure 2: Netlist Annotated With Delays

Figure 2 shows a netlist with several such paths: $ABCD$, $ABECD$, $ABEF$, FEF, and $FECD$. Each path has one or more edge delays (e.g. D_{AB}) that are dependent on the placement sites for the netlist components. The components on path $ABCD$, for example, must be placed in a way that satisfies the constraint $D_{AB} + D_{BC} + D_{CD} \leq \tau$.

Paths that fail timing must be repaired by moving components to reduce routing delays or by using retiming to move logic across state elements. The detailed placer can assume that the global placer has already performed global delay optimization under an abstract model of routing delay. When the paths are reassessed using actual routing delays, some may violate timing. It is expected that only a minority of paths will require repair and that components will only have to move a relatively short distance away from their starting locations.

A complete placer has to consider other optimization criteria such as placement legality and routability. Our experimental results are based on our complete system which includes these issues, but we focus only on the timing constraints in this paper.

3 Problem Formulation

In this section we describe how concepts from static timing analysis can be used to efficiently encode path-based timing constraints. We begin with variables that encode the placement of netlist components

onto sites. The Boolean variable V_{AX} has the meaning that component A is placed on site X. These *placement variables* can be arranged into a sparse matrix where the components are the rows and the sites are the columns as follows:

	W	X	Y	Z	$\cdots$
A	V_{AW}	V_{AX}	V_{AY}		
B		V_{BX}	V_{BY}		
C			V_{CY}	V_{CZ}	
$\vdots$					$\ddots$

(1)

For each row in the matrix, an *exactlyOne* constraint (a standard *atMostOne* cardinality constraint combined with a standard Boolean clause) is added to ensure that each component gets a placement. For each column in the matrix, an *atMostOne* constraint is added to prevent placement overlaps.

Timing constraints for netlist edges follow this pattern:

$$(V_{AX} \wedge V_{BY}) \rightarrow (D_{AB} = d_{XY}) \tag{2}$$

This formula says that if netlist component A is placed on site X and component B is placed on site Y, then the delay on netlist edge AB is equal to the routing delay between sites X and Y. We assume there exists a timing model of the target architecture which provides these d_{XY} numbers. The number of clauses for each edge is equal to the product of the number of candidate sites for the source and sink components.

Timing constraints for netlist state-to-state paths follow this pattern:

$$D_{AB} + D_{BC} + D_{CD} + \ldots \leq \tau \tag{3}$$

To avoid making a constraint of this type for every path in the netlist, we rewrite equations 2 and 3 using concepts from static timing analysis. Static timing analysis computes an *arrival time* and a *required time* at each component in the netlist graph [9]. The arrival time is the maximum path length to state elements backwards through the transitive fanin of a component. This value represents the time it takes from the beginning of the clock cycle for the data to propagate through the circuit and to become valid at the output of the component. The arrival time at the output of a state element is defined to be zero.

The required time is the clock period τ minus the maximum path length to state elements forwards through the transitive fanout of a component. This value is the latest time at which the component output can become valid and still make it to the downstream state elements before the end of the clock cycle. The required time at the input of a state element is defined to be τ.

The *slack* of a component is the required time minus the arrival time. A component with negative slack is on a path that fails timing. Instead of making a separate timing constraint for each state-to-state path in the netlist, one can simply constrain each netlist component to have non-negative slack with a clause $\mathrm{Arr}_A \leq \mathrm{Req}_A$.

The arrival time Arr_A and required time Req_A are defined in terms of the immediate fanin and fanout components:

$$\mathrm{Arr}_A = \max_{\text{fanin } Fi} (\mathrm{Arr}_{Fi} + D_{FiA}) \tag{4}$$

$$\mathrm{Req}_A = \min_{\text{fanout } Fo} (\mathrm{Req}_{Fo} - D_{AFo}) \tag{5}$$

Using these formulas, Equation 2 can be rewritten as:

$$(V_{AX} \wedge V_{BY}) \rightarrow (\mathrm{Arr}_B - \mathrm{Arr}_A \geq d_{XY}) \\ \wedge (\mathrm{Req}_A - \mathrm{Req}_B \leq -d_{XY}) \tag{6}$$

The result is a difference logic formulation. The only non-Boolean variables are the arrival and required times Arr_A and Req_A for each netlist component.

This formulation does not require enumerating all state-to-state paths and avoids the worst-case exponential number of constraints. However, the number of constraints of the form of Equation 6 still grows with the product of the number of placement options for the source and sink components for each netlist edge. This is still an impractically large number of constraints. We address this scalability issue using dynamic clause generation.

4 Dynamic Clause Generation

Observe that the placement variable matrix (1) is mostly false due to the *exactlyOne* constraint on each row. Also, each clause $(V_{AX} \wedge V_{BY}) \rightarrow (\ldots)$ starts with two negative literals $(\overline{V_{AX}} + \overline{V_{BY}} + \ldots)$. A majority of the clauses are therefore trivially satisfied during the search.

Furthermore, the detailed placer is responsible for repairing only local constraint violations and does not attempt to make global changes to the initial placement. It is expected that only a minority of netlist components will have to be moved to accomplish this task. The solver is likely to find a SAT or UNSAT solution after attempting only a small fraction of the placement options V_{AX} for each component.

Therefore, while all of the clauses are theoretically necessary for correctness, in practice most of them do not affect the search. We can take advantage of this fact to reduce the working size of the problem. Clauses of the form $(\overline{V_{AX}} + \overline{V_{BY}} + \ldots)$ can be left out until the search actually enters a subspace where V_{AX} and V_{BY} are both true.

Our solver provides a callback method *decisionCallback* that is invoked after assigning a placement variable V_{AX} = T and after all unit propagations have completed without conflicts. The placer adds clauses starting with $(\overline{V_{AX}} + \ldots)$ during this callback. All fanin and fanout components of A are checked to see if they are placed in the solver state at the time of the callback. Then clauses related to fanin edges FiA and fanout edges AFo are generated.

The solver then processes the incoming clauses and makes its internal state consistent using a minimal amount of backtracking when necessary. The search can then continue in the subspace under V_{AX} = T as normal. The resulting behavior is the same as if the dynamic clauses had always been present, but with substantial runtime savings since most clauses are never generated.

Our approach is complementary to the two-watched-literal technique [14] in that it addresses clause generation runtime and memory footprint in addition to search runtime. Quantitative results are given in Section 7.

4.1 Related Dynamic Approaches

Eén and Sörensson describe a dynamic clause generation approach in [7] wherein the solver is restarted with additional clauses after a complete solution has been produced and examined. Our approach adds clauses while the solver is running and not only between invocations of an incremental solver.

The Lynx SAT solver includes a similar callback method for adding clauses after each propagation step [8]. That callback method examines the solver trail and adds clauses that conflict with the current assignment. In comparison, our approach allows clauses that do not conflict with the current assignment to be added as well.

Ohrimenko et al. [18] describe a purely Boolean encoding of difference logic constraints that includes lazy clause generation to reduce the number of Boolean clauses created for difference logic propagation. Our approach performs dynamic clause generation on a coarser level of abstraction. This is possible due to the natural subdivisions of the problem space and the low likelihood of actually searching the majority of these subspaces.

5 Boolean Formulation

Our first approach to constructing a detailed placer based on constraint satisfaction was to convert the difference logic timing constraints into a purely Boolean SAT problem [12]. We then added support for dynamic clause generation to MiniSAT version 1.12b to solve the instances [6]. This formulation is briefly described here because it is used as a baseline for measuring the benefits of our difference logic SMT solver.

The Boolean formulation uses a small-domain encoding variation due to Ohrimenko et al. [18]. The key idea is to define Boolean variables that represent upper and lower bounds on the difference logic variables instead of exact values of the variables. This approach is a natural fit for modeling arrival and required times and detecting negative slack.

For each netlist component A we create a number line subdivided into T discrete values representing times within the clock period τ. Each subdivision has an associated non-decision Boolean variable E_{At}. When true, this variable has the meaning that $\text{Req}_A \leq t\frac{\tau}{T}$. When false, this variable indicates that $\text{Arr}_A > t\frac{\tau}{T}$.

The clauses shown in Equation 6 are then rewritten using these Boolean variables:

$$(V_{AX} \wedge V_{BY}) \rightarrow \bigwedge_t (\overline{E_{At}} \rightarrow \overline{E_{B\lceil t+\Delta \rceil}}) \quad \text{where } \Delta = d_{XY}\frac{T}{\tau} \tag{7}$$

If the solver makes placement decisions that result in an arrival time becoming larger than a required time at any netlist component, then some variable E_{At} will be assigned to both true and false. Timing violations are therefore detected as ordinary Boolean conflicts.

A major drawback of the purely Boolean encoding of timing constraints is the quantization of time. Arrival and required times must be rounded conservatively to discrete number line variables E_{At}. On a long state-to-state path through the netlist, these rounding errors accumulate and over-constrain the problem. This is especially problematic when the target clock period τ approaches the maximum frequency supported by the target architecture. The conservative rounding forces the placer to find a solution that exceeds the actual target, but the architecture does not have interconnect routes that are fast enough. Consequently, placements that would actually be acceptable are rejected. The quantization of time can be partially addressed by increasing the number of subdivisions T, but this additional accuracy comes at the cost of increasing the number of clauses in the formulation.

The Boolean encoding is also computationally expensive. Boolean constraint propagation is used to mimic difference logic constraint propagation (e.g. addition and comparison of numerical variables). Both of these drawbacks can be addressed by using a difference logic SMT solver instead of a Boolean SAT solver.

6 A Dynamic Difference Logic Solver

Since dynamic clause generation is an essential component of our approach, we constructed a custom difference logic SMT solver instead of using a publicly available solver. Our design decisions for im-

plementing the solver are influenced by our experience building static timing analysis tools and prior work on DPLL(T) SMT solvers and difference logic solvers [3, 20, 16, 17].

The general software architecture of our solver follows MiniSAT but is entirely new code. Boolean literals are encoded as positive integers, with even numbers used for positive literals and odd numbers for negated literals. Difference logic literals are encoded as negative integers, with even numbers used for lower bounds and odd numbers used for upper bounds: The difference logic portions of the solver use floating-point math.

$\cdots$	-4	-3	-2	-1	0	1	2	3	$\cdots$
$\cdots$	$(DL_1 \geq)$	$(DL_1 \leq)$	$(DL_0 \geq)$	$(DL_0 \leq)$	B_0	$\overline{B_0}$	B_1	$\overline{B_1}$	$\cdots$

(8)

The placer creates one difference logic variable for each netlist component and one difference logic variable for each netlist edge. The solver is informed which variables represent components and which represent edges, and is also provided the graph connectivity of these variables. This graph structure is used during the propagation phase to perform t-propagations and deduce t-consequences.

Netlist edge delay clauses are of the form:

$$(V_{AX} \wedge V_{BY}) \rightarrow (D_{AB} \geq d_{XY}) \tag{9}$$

Propagation

The propagation phase is divided into three parts which are executed in priority order. First, all Boolean unit propagations are made until nothing else can be propagated. If any conflicts are encountered, the solver performs conflict analysis and nonlinear backtracking as in MiniSAT.

After Boolean unit propagations are finished, the solver performs all difference logic propagations in a single batch t-propagation step. This is the "lazy" approach described in [3].

The difference logic propagation mirrors the arrival and required time propagation used in static timing analysis. If any netlist component has a new lower bound, an update is scheduled to recompute lower bounds on all of its fanout components. Similarly, any component with a new upper bound schedules upper bound updates for all of its fanin components. A new lower bound on a netlist edge schedules lower bound updates on fanout components and upper bound updates on fanin components.

Upper and lower bounds are propagated forwards and backwards throughout the netlist graph until the system converges or until the Bellman-Ford iteration limit is reached which indicates that a positive-weight cycle has been discovered. If such a cycle is found or if at any time some difference logic variable has a lower bound greater than its upper bound, the propagation phase stops and the solver proceeds to conflict analysis.

The result of the difference logic propagation phase is that all netlist components have updated arrival and required times that are consistent with the known netlist edge delays. The propagation also computes new upper bounds on the delays of edges:

$$(\mathrm{Arr}_A \geq a \wedge \mathrm{Req}_B \leq b) \rightarrow (D_{AB} \leq b - a) \tag{10}$$

The solver uses these upper bounds to rule out future placement decisions that would result in edge delays that are too large. This feedback is one example of how t-consequences guide the Boolean part of the search.

Finally, after all Boolean and difference logic propagations have completed, the solver performs dynamic clause generation. The *decisionCallback* function is called for all new Boolean variables that have been assigned since the last round of dynamic clause generation. The placer adds clauses that are now relevant to the search.

The strict priority order of Boolean propagation, difference logic propagation, and dynamic clause generation reflects the runtime cost of each of these steps. The solver attempts to find conflicts using existing clauses before asking the placer to add new clauses. Thus no computation is wasted when the solver is destined to backtrack out of the current subtree. Also, the solver attempts to discover conflicts resulting from Boolean propagations before performing a more expensive t-propagation step.

Conflict Analysis

Conflicts are analyzed and learned clauses are generated using the 1-UIP approach as in MiniSAT. For each difference logic variable, the solver maintains two stacks of $\{value, reason, level\}$ tuples that store monotonically tightening upper and lower bounds on the variable's value. These stacks are searched to find the earliest decision level at which a difference logic literal became true or false. The matching stack entry contains the *reason* data necessary to build the implication graph and construct the 1-UIP learned clause.

The *reason* for a difference logic literal to be true or false can be one of two possibilities. If the literal was assigned as a result of a clause becoming unit, the *reason* field will refer to that clause. The conflict analysis algorithm can then continue unwinding that clause.

If the literal was assigned during difference logic propagation, the *reason* field will refer to a pair of difference logic node and edge literals that are responsible for the propagated value. For example, consider a netlist edge AB with delay $D_{AB} = 10.0$ and source node A with $\text{Arr}_A \geq 20.0$. The difference logic propagation would enqueue $\text{Arr}_B \geq 30.0$. If that literal is later involved in a conflict, the *reason* will appear to be the clause $(\overline{\text{Arr}_A \geq 20.0} + \overline{D_{AB} \geq 10.0} + \text{Arr}_B \geq 30.0)$. This clause is not in the clause database. It is a temporary object that is made so that the conflict analysis algorithm can unwind clausal implications and t-consequences identically.

7 Results

To measure the improvement of the difference logic formulation over the purely Boolean formulation, we ran a series of experiments placing netlists from the OpenCores database [19] using both formulations. Our custom difference logic solver is used in both cases. The purely Boolean problems use the formulation described in Section 5 which uses no difference logic variables.

These results are based on the complete detailed placement tool. In addition to the timing constraints described in this paper, our placer also considers placement legalization constraints and routability constraints. The decision variables are also more extensive than described here. The placer is able to explore lookup table pin permutations and alternative routing paths for netlist edges between the same source and sink component sites. Retiming can also be applied to move logic across state elements.

A search-and-repair strategy is used that breaks the entire placement problem into smaller subproblems that each try to repair a specific timing violation. Each subproblem contains a subset of the netlist with approximately 100 components in the neighborhood of a timing violation. This subproblem size is sufficient to solve complex violations that require coordinated motion of dozens of components in order to repair. Each SAT result repairs not only the targeted timing violation but also other violations that are coincidentally included in the neighborhood. The placer creates and solves a series of subproblems until all of the timing violations in the netlist have been repaired.

The runtime and frequency numbers in Table 1 are normalized to the Boolean formulation results and represent the total runtime of the tool. The difference logic formulation achieves higher frequency placements due to the increased accuracy of representing delays using floating point numbers in the solver. The pessimistic rounding required by the Boolean formulation over-constrains the problem. In some cases, the difference logic formulation is able to find a placement that achieves the highest

		Boolean Formulation				Difference Logic Formulation					
Design	Comps	Runtime	Freq	Bvars	Clauses	RT Match	RT Best	Freq	Bvars	DLvars	Clauses
mancala	287	1.0	1.0	146k	393k	0.47	5.09	1.29	13k	160	60k
rs_enc	1373	1.0	1.0	189k	59k	0.60	0.68	1.03	23k	467	55k
aeMB	3066	1.0	1.0	285k	233k	1.26	2.16	1.07	24k	411	81k
sha256	3283	1.0	1.0	283k	489k	0.71	1.01	1.03	28k	470	132k
aes	5236	1.0	1.0	233k	94k	0.57	0.79	1.03	29k	510	80k
warp	5560	1.0	1.0	283k	729k	0.66	11.20	1.71	24k	375	88k
r2000sc	5807	1.0	1.0	294k	238k	0.93	7.34	1.32	26k	398	86k
minimips	5855	1.0	1.0	231k	252k	0.39	1.03	1.07	27k	425	91k
fpu_double	10300	1.0	1.0	327k	126k	0.38	0.48	1.05	32k	327	78k

Table 1: Boolean Formulation vs. Difference Logic Formulation

		Boolean Formulation				Difference Logic Formulation			
		Dynamic		No Dynamic		Dynamic		No Dynamic	
Design	Comps	Runtime	Clauses	Runtime	Clauses	Runtime	Clauses	Runtime	Clauses
mancala	287	1.0	393k	13.8	76590k	0.40	60k	12.2	28545k
rs_enc	1352	1.0	59k	12.7	1029k	0.30	55k	48.2	16487k
aeMB	3045	1.0	233k	8.7	20116k	0.58	81k	15.7	20531k

Table 2: Average Instance Runtime and Clause Count

frequency possible on the fabric. This happens when the entire critical path in the netlist is placed using the shortest possible fabric connections.

For the difference logic formulation, the table lists an *RT Match* result that indicates how long it took to attain the same frequency as the Boolean formulation, and a *RT Best* runtime for the maximum achieved frequency. The difference logic formulation is able to attain an equivalent result in less runtime, and goes on to achieve frequencies that are out of reach of the Boolean formulation.

The difference logic formulation is also more efficient in the number of clauses and variables. Table 1 lists the average number of clauses and variables over all of the subproblems in each netlist, which are about the same size independent of the total netlist size. In the Boolean formulation, equation 7 expands into a large number of clauses that mimic arrival and required time propagation. These are unnecessary in the difference logic formulation, as are the Boolean variables used to represent quantized times.

Table 2 shows the average per-instance runtime and clause count with and without the dynamic clause generation technique. Due to the long runtimes, results are shown only for small netlists. For both the Boolean and difference logic formulations, dynamic clause generation leads to approximately an order of magnitude reduction in runtime for constructing and solving each instance.

8 Conclusion

Programmable logic detailed placement is a challenging problem domain that has not been previously solved with a constraint satisfaction approach. Using the techniques described in this paper, our formulation can be solved efficiently and used to construct a practical placement tool.

One of the keys to making the approach scalable is to take advantage of the natural subdivisions of the problem space and the fact that the solver is able to determine a SAT or UNSAT result without visiting a majority of the subspaces. This characteristic makes the problem domain amenable to on-demand dynamic clause generation. Other problem domains with the same characteristic should be able to enjoy the same benefits.

The SMT-based placer is currently deployed in a production placement tool at Tabula. It replaces the purely Boolean formulation that was solved with a version of MiniSAT extended to support dynamic

clause generation. That approach, in turn, replaced a placement tool based on simulated annealing. The constraint satisfaction approaches have provided considerable runtime and quality benefits and we are encouraged to explore this technology further.

References

[1] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement, and routing tool for FPGA research. In *Intl. Workshop on Field Programmable Logic and Applications*, pages 213–222, 1997.

[2] Deming Chen, Jason Cong, and Peichen Pan. FPGA design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):195–330, November 2006.

[3] Scott Cotton and Oded Maler. Fast and flexible difference constraint propagation for DPLL(T). In *9th Intl. Conference on Theory and Applications of Satisfiability Testing*, pages 170–183, August 2006.

[4] Srinivas Devadas. Optimal layout via Boolean satisfiability. In *IEEE International Conference on Computer-Aided Design*, pages 294–297, November 1989.

[5] Rolf Drechsler, Stephan Eggersglüß, Görschwin Fey, and Daniel Tille. *Test Pattern Generation using Boolean Proof Engines*. Springer, 2009.

[6] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[7] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *First International Workshop on Bounded Model Checking*, volume 89, pages 543–560, 2003.

[8] V. Ganesh, C. W. O'Donnell, M. Soos, S. Devadas, M. C. Rinard, and A. Solar-Lezama. Lynx: A programmatic SAT solver for the RNA-folding problem. In *Intl. Conf. on SAT*, pages 142–155, June 2012.

[9] T. I. Kirkpatrick and N. R. Clark. PERT as an aid to logic design. *IBM Journal of Research and Development*, 10(2):135–141, 1966.

[10] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD*, 21(12):1377–1394, December 2002.

[11] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, February 2008.

[12] Andrew Mihal and Steve Teig. A constraint satisfaction approach for programmable logic detailed placement. In *16th Intl. Conf. on Theory and Applications of Satisfiability Testing*, pages 208–223, July 2013.

[13] Alan Mishchenko, Robert Brayton, Jie-Hong Roland Jiang, and Stephen Jang. SAT-based logic optimization and resynthesis. In *International Workshop on Logic and Synthesis*, pages 358–364, May 2007.

[14] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[15] Gi-Joon Nam, Karem Sakallah, and Rob Rutenbar. Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based Boolean SAT. In *Intl. Symposium on FPGAs*, pages 167–175, 1999.

[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[17] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *17th International Conference on Computer Aided Verification*, July 2005.

[18] Olga Ohrimenko, Peter Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009.

[19] Various. OpenCores open source hardware IP cores, April 2013. `http://opencores.org`.

[20] C. Wang, F. Ivančić, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Proc. of the 12th Intl. Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 322–336, 2005.

[21] R. Glenn Wood and Rob Rutenbar. FPGA routing and routability estimation via Boolean satisfiability. *IEEE Transactions on VLSI*, 6(2), June 1998.

Handling Bit-Propagating Operations in Bit-Vector Reasoning

Alexander Nadel

Intel Corporation, P.O. Box 1659, Haifa 31015, Israel
alexander.nadel@intel.com

Abstract

Our aim is to improve bit-vector reasoning in modern SMT solvers. We enhance bit-vector preprocessing by introducing algorithms that explicitly handle an important class of bit-vector operations which we call *bit-propagating*. Such operations fulfill the following property: each output bit is either a bit of one of the inputs or a constant (0 or 1). We identified ten bit-propagating operations in the SMT-LIB 2.0 language; these operations are encountered frequently in practice. Our algorithms seek to improve the run-time of SMT solvers by simplifying the problem that is eventually provided to the underlying SAT solver. Empirical evaluation of our algorithms reveals a performance boost across a variety of SMT-LIB benchmark families.

1 Introduction

Bit-vector (abbr., BV) reasoning is widely used in practice. Over 48% out of more than 93,000 benchmarks in SMT-LIB [4] are either plain BV benchmarks or combine the BV theory with the theory of arrays (where 33% are plain BV benchmarks) [1]. Bit-vector reasoning is supported by a variety of solvers, such as Boolector [7], STP [12], Mathsat [8] and others. In the eager approach to BV solving (used by Boolector and STP, for example), the solver preprocesses the word-level formula, then translates the simplified formula to Conjunctive Normal Form (CNF) and solves it with a SAT solver. In this paper we identify an important class of BV operations and propose an efficient way of handling them in the preprocessor. We restrict further discussion in this paper to eager SMT solvers applied to solving bit-vector benchmarks that conform to the QF_BV logic syntax [2]. However, our results are applicable whenever BV reasoning is required.

The central notion of our paper is that of a *bit-propagating* operation. A bit-propagating operation fulfills the following property: each output bit is either a bit of one of the inputs or a constant 0 or 1 (a more precise definition appears in Section 3.1). We identified 10 bit-propagating operations amongst the 38 operations over bit-vectors supported in QF_BV (that is, the union of the 35 operations in the Fixed_Size_BitVectors theory and the 3 operations in the Core theory). The two basic bit-propagating operations are `concat` and `extract`, while the others comprise two rotation operations, `repeat`, and three shift operations (we consider the shift operations to be bit-propagating only when the shift is by a constant). A full list of bit-propagating operations is provided in Fig. 1. We found that bit-propagating operations appear frequently in practice. See Table 1 for more details.

As an example of a bit-propagating operation, consider the shift left operation `bvshl`. Assume that a bit-vector variable of width 4 $v = \left[v^{[3]}, v^{[2]}, v^{[1]}, v^{[0]}\right]$ ($v^{[0]}$ is the least significant bit) is shifted by the constant 3. The result would be $v = \left[v^{[0]}, 0, 0, 0\right]$. Clearly, our property holds: bits 0 through 2 of the output are constants, while bit 3 of the output is bit 0 of the input.

We assume that the SMT solver maintains a directed acyclic graph to represent the input formula, where each node correponds to either a bit-vector constant, an input bit-vector variable, or an internal bit-vector variable created as a result of applying an operation. We call a variable *bit-propagating* if it was created as a result of applying a bit-propagating operation.

The main observation behind our work is that given a nested application of bit-propagating operations, the bits of the resulting variable can always be expressed in terms of bits of non-bit-propagating variables and the constants 0 and 1. For example, assume that the following variables were created in the order specified, with the width being 4 bits for every variable except v_3:

1. $v_1 := \texttt{bvadd}(u_1, u_2)$ (bit-vector addition of two previously declared variables)
2. $v_2 := \texttt{bvshl}(v_1, 1)$ (a shift by a constant)
3. $v_3 := \texttt{extract}(v_2, 3, 2)$ (extracting bits 2 through 3)

Both v_2 and v_3 can be expressed in terms of the bits of v_1 and the constant 0. More specifically, we have $v_2 = \left[v_1^{[2]}, v_1^{[1]}, v_1^{[0]}, 0\right]$ and $v_3 = \left[v_1^{[2]}, v_1^{[1]}\right]$.

Our main idea is of associating each newly created variable with the so-called *Bit-Propagating Normal Form (BPNF)* which expresses the variable in terms of bits of non-bit-propagating variables and constants. BPNFs of all the variables are stored in a hash table. We propose a succint representation for BPNF in Section 3.1.

The idea of associating a normal form with bit-vector expressions over `concat` and `extract` operations is well known [9, 6, 5]. In particular, the concatenation normal form [6], detailed in [11], is largely similar to our BPNF. The added value of our proposal is that it extends the normal form to variables created with eight additional operations available in the modern SMT-LIB 2.0 language and integrates the normal-form-based reasoning into a modern SMT solver.

One advantage of maintaining a BPNF is that one can avoid creating new variables when bit-vector variables with identical BPNFs are created through different sequences of bit-propagating operations. Let us continue our example:

4. $v_4 := \texttt{repeat}(v_1, 2)$
5. $v_5 := \texttt{extract}(v_4, 6, 5)$

We have $v_4 = \left[v_1^{[3]}, v_1^{[2]}, v_1^{[1]}, v_1^{[0]}, v_1^{[3]}, v_1^{[2]}, v_1^{[1]}, v_1^{[0]}\right]$ and $v_5 = \left[v_1^{[2]}, v_1^{[1]}\right]$. Hence v_5 is identical to v_3. Imagine that the SMT solver is required to create an internal variable coresponding to the operation $\texttt{extract}(v_4, 6, 5)$. After calculating the BPNF and looking in the BPNF hash table, the solver can conclude that a new internal variable is not required, since v_3 can be used instead.

Another advantage of maintaining a BPNF is that when the formula is translated to CNF, we create new CNF variables only for non-bit-propagating word-level variables, since we can express all bits of the bit-propagating variables in terms of constants and bits of non-bit-propagating variables. Hence, maintaining BPNFs is expected to reduce the number of CNF variables and clauses, thus simplifying the problem for the SAT solver.

An alternative way of refraining from creating new CNF variables for the outputs of bit-propagating operations would simply be to reuse CNF variables during the bit-blasting stage. In our example, in order to represent v_3 in CNF, one could use the CNF variables created

to represent $v_1^{[2]}$ and $v_1^{[1]}$.[1] However, such an approach would not have the first advantage of maintaining a normal form we mentioned, that is, creating only one actual word-level variable for variables with identical BPNFs created through different sequences of bit-propagating operations. Our proposal has this advantage over any approach based on hashing individual bits.

We implemented our algorithms in Intel's new eager BV solver Hazel, whose architecture is largely similar to that of Boolector and STP, and tested their usefulness on benchmark families from SMT-LIB [3]. We show that applying our algorithms results in a performance boost across a variety of SMT-LIB families. We also show that on these families, Hazel is usually faster than the state-of-the-art academic solvers.

In what follows, Section 2 contains preliminaries. Section 3 describes the core algorithms we propose. Experimental results are provided in Section 4. Section 5 concludes our paper.

2 Preliminaries

We need to define notions related to the basics of BV reasoning.

Definition 1 (Bit, Bit-Vector Variable, Constant)**.** *A* bit *is a Boolean variable (it can be interpreted as 0 or 1). A* bit-vector variable v *of width* $|v|$ *is a sequence* $v = \left\{v^{[|v|-1]}, \ldots, v^{[1]}, v^{[0]}\right\}$*, where* $v^{[i]}$ *is a bit for each* $|v| > i \geq 0$*. The set of all bit-vectors is denoted by* $\mathcal{B}$*. A* constant *is a bit-vector variable, whose every bit is interpreted as 0 or 1. The set of all constants is denoted by* $\mathcal{C}$*.*

We will sometimes refer to bit-vector variables as either bit-vectors or variables. We consider bits to be bit-vectors of width 1. We denote by 0^w a constant of width w whose every bit is 0.

It is not hard to check that the domain of every operation supported in QF_BV is a cross product of one, two or three bit-vectors (we consider variables of sort Bool to be bit-vectors of width 1) and zero, one or two natural numbers, while the range comprises a bit-vector. We denote by $\mathcal{N}$ the set of natural numbers (including 0), and by $bits(v)$ the set of all bits of a bit-vector v. Formal definitions of a bit-propagating operation and a bit-range follow.

Definition 2 (Bit-Propagating Operation)**.** *An operation* $\omega : \underbrace{\mathcal{B} \times \mathcal{B} \times \ldots \mathcal{B}}_{1 \leq k \leq 3} \times \underbrace{\mathcal{N} \times \mathcal{N} \times \ldots \times \mathcal{N}}_{0 \leq l \leq 2} \rightarrow \mathcal{B}$ *is* bit-propagating *if for every application of* ω*:* $\omega(v_k \in \mathcal{B}, \ldots, v_1 \in \mathcal{B}, x_l \in \mathcal{N}, \ldots, x_1 \in \mathcal{N}) = u \in \mathcal{B}$*, for every* $|u| > i \geq 0$ *it holds that* $u^{[i]} \in \{0,1\} \cup bits(v_k) \cup \ldots \cup bits(v_2) \cup bits(v_1)$ *and that* $u^{[i]}$ *can be computed at the time the operation is applied.*

Definition 3 (Bit-Range)**.** *Let* v *be a bit-vector of width* n*. Then for every* $n > i \geq 0$ *and* $n > j \geq i$*, the sequence* $v^{[j:i]} = \left\{v^{[j]}, \ldots, v^{[i+1]}, v^{[i]}\right\}$ *is a* bit-range *of* v*.*

Fig. 1 provides the set of all bit-propagating operations in the SMT-LIB 2.0 language. The shift operations in the language (`bvshl`–shift left, `bvlshr`–logical shift right, and `bvashr`–arithmetic shift right) support shifting by an arbitrary bit-vector. However, we consider shifts to be bit-propagating only when the shift is by a constant, since otherwise the match between the output and the input bits is not known at the time the operation is applied. Hence, in Fig. 1, the shift operations receive a natural number as their second parameter. Note that all the operations can be expressed in terms of one or more applications of `extract` and/or

[1] Unfortunately, we are not aware of any publications containing details of bit-blasting algorithms applied by SMT solvers. To the best of our knowledge, some SMT solvers re-use CNF variables while handling the `extract` and `concat` operations during bit-blasting, but not the other eight bit-propagating operations we identified.

1. $\texttt{concat}(v_1 \in \mathcal{B}, v_2 \in \mathcal{B}) = \left\{v_1^{[|v_1|-1]}, \ldots, v_1^{[1]}, v_1^{[0]}, v_2^{[|v_2|-1]}, \ldots, v_2^{[1]}, v_2^{[0]}\right\}$
2. $\texttt{extract}(v \in \mathcal{B}, m \in \mathcal{N}, l \in \mathcal{N}) = v^{[m:l]}$, where $|v| > m, l \geq 0$ and $m \geq l$
3. $\texttt{repeat}(v \in B, s \in \mathcal{N}) = \texttt{concat}(\texttt{repeat}(v, s-1), v)$ for $s > 1$; $\texttt{repeat}(v \in B, 1) = v$; $\texttt{repeat}(v, 0)$ is undefined
4. $\texttt{zero\_extend}(v \in \mathcal{B}, s \in \mathcal{N}) = \texttt{concat}(0^s, v)$ for $s > 0$; $\texttt{zero\_extend}(v, 0) = v$
5. $\texttt{sign\_extend}(v \in \mathcal{B}, s \in \mathcal{N}) = \texttt{concat}(\texttt{repeat}(v^{[|v|-1]}, s), v)$ for $s > 0$; $\texttt{sign\_extend}(v, 0) = v$
6. $\texttt{bvshl}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{extract}(v, |v|-1-s, 0), 0^s)$ for $|v| > s > 0$; $\texttt{bvshl}(v \in B, 0) = v$; $\texttt{bvshl}(v, s \geq |v|) = 0^{|v|}$
7. $\texttt{bvlshr}(v \in \mathcal{B}, s \in N) = \texttt{concat}(0^s, \texttt{extract}(v, |v|-1, s))$ for $|v| > s > 0$; $\texttt{bvlshr}(v \in B, 0) = v$; $\texttt{bvlshr}(v, s \geq |v|) = 0^{|v|}$
8. $\texttt{bvashr}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{repeat}(v^{[|v|-1]}, s), \texttt{extract}(v, |v|-1, s))$ for $|v| > s > 0$; $\texttt{bvashr}(v \in B, 0) = v$; $\texttt{bvashr}(v, s \geq |v|) = \texttt{repeat}(v^{[|v|-1]}, s)$
9. $\texttt{rotate\_left}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{extract}(v, (|v|-1-s)\%|v|, 0), \texttt{extract}(v, |v|-1, (|v|-s)\%|v|))$ for $s : s\%|v| \neq 0$; $\texttt{rotate\_left}(v, s) = v$ for $s : s\%|v| = 0$
10. $\texttt{rotate\_right}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{extract}(v, (s-1)\%|v|, 0), \texttt{extract}(v, |v|-1, s\%|v|))$ for $s : s\%|v| \neq 0$; $\texttt{rotate\_right}(v, 0) = v$ for $s : s\%|v| = 0$

Figure 1: Bit-Propagating Bit-Vector Operations in the SMT-LIB 2.0 Language (% stands for the modulo operation)

concat. Note also that all the bit-propagating operations are equally applicable to bit-vector variables and constants.

3 Handling Bit-Propagating Operations

In this section we describe our algorithms for handling bit-propagating operations. Subsection 3.1 introduces the Bit-Propagating Normal Form (BPNF), while subsection 3.2 shows how to take advantage of BPNFs to boost the performance of the SMT solver.

3.1 Bit-Propagating Normal Form

We start with a definition of a segment, where a variable is *bit-propagating* if it was created as a result of applying a bit-propagating operation.

Definition 4 (Segment)**.** *A segment is either: (1) a bit-range of a non-bit-propagating variable, or (2) a constant. The set of all segments is denoted by* $\mathcal{S}$*.*

Bit-propagating variables can be expressed in terms of sequences of segments. Consider the example presented in Section 1. We would have: $v_2 = \left[v_1^{[2:0]}, 0^1\right]$, $v_3 = v_5 = \left[v_1^{[2:1]}\right]$ and

$v_4 = \left[v_1^{[3:0]}, v_1^{[3:0]}\right]$.

To define the standard form we need to make sure that adjacent segments are *merged*. For example, consider the variable v created by the following operation $v = \texttt{concat}(v^{[3:2]}, v^{[1:0]})$. The variable v could be expressed as any one of the following sequences of segments: $r_1 = \left[v^{[3:2]}, v^{[1:0]}\right]$, or $r_2 = \left[v^{[3:3]}, v^{[2:0]}\right]$, or $r_3 = \left[v^{[3:0]}\right]$. The last representation is the one that is desirable as the normal form. We formalize merge-related notions.

Definition 5 (Mergeable and Non-Mergeable Segments)**.** *Let $s_2, s_1 \in \mathcal{S}$ be two segments. The segments s_2 and s_1 (provided in that particular order) are* mergeable *iff one of the following conditions holds, otherwise they are* non-mergeable*:*

1. *Both s_2 and s_1 are constants, that is $s_2, s_1 \in \mathcal{C}$*
2. *Both s_2 and s_1 are bit-ranges, such that $s_2 = v^{[k:j+1]}$ and $s_1 = v^{[j:i]}$*

Definition 6 (Merge)**.** *Let $s_2, s_1 \in \mathcal{S}$ be two segments. The* merge *operation $\mathbb{M}(s_2, s_1)$ returns a sequence of one or two segments as follows:*

1. *If s_2 and s_1 are non-mergeable, $\mathbb{M}(s_2, s_1) = [s_2, s_1]$*
2. *If s_2 and s_1 are mergeable and are constants, $\mathbb{M}(s_2, s_1) = [\texttt{concat}(s_2, s_1)]$*
3. *If $s_2 = v^{[k:j+1]}$ and $s_1 = v^{[j:i]}$ are mergeable and are bit-ranges, $\mathbb{M}(s_2, s_1) = \left[v^{[k:i]}\right]$*

In our latest example (provided just before Def. 5), merging the two segments of r_1 and merging the two segments of r_2 results precisely in r_3 for both cases. We are now ready to introduce the Bit-Propagating Normal Form.

Definition 7 (Bit-Propagating Normal Form (BPNF))**.** *Given a bit-vector or a constant $t \in \mathcal{B} \cup \mathcal{C}$, the* bit-propagating normal form (BPNF) *$\Phi(t) = \left[\phi^t_{|\Phi(t)|-1} \in \mathcal{S}, \ldots, \phi^t_1 \in \mathcal{S}, \phi^t_0 \in \mathcal{S}\right]$ is a sequence of one or more segments, where for every $|\Phi(t)| - 2 > i \geq 0$, it holds that ϕ^t_{i+1} and ϕ^t_i are non-mergeable.*

In our example, we have $\Phi(v) = r_3$. Before presenting an algorithm for calculating the BPNF, we need some more definitions. We denote the number of bits in a segment $s \in S$ by $|s|$.

Definition 8 (Sub-segment)**.** *Let $s \in \mathcal{S}$ be a segment and i, j be numbers, such that $|s| > j, i \geq 0$ and $j \geq i$. Then the sub-segment $s^{[j:i]}$ is a new segment defined as follows:*

1. *If s is a constant $\left\{s_{|s|-1}, \ldots, s_1, s_0\right\}$, $s^{[j:i]} = \{s_j, \ldots, s_{i+1}, s_i\}$*
2. *If $s = v^{[k:l]}$ is a bit-range, $s^{[j:i]} = v^{[j+l:i+l]}$ (assuming $k \geq j + l$)*

It is not difficult to verify that a sub-segment is a segment. We will sometimes need to refer to the segment in $\Phi(v)$ of a bit of a given variable $v^{[i]}$ and the bit corresponding to $v^{[i]}$ in its segment.

Definition 9 (Segment Number, Segment Bit)**.** *Let $v^{[i]}$ be the i's bit of v. Let $s \geq 0$ be the largest number, such that $i \geq \sigma$, where $\sigma = \sum_{j=0}^{s-1} |\phi^v_j|$. Then, the* segment number *$sn(v^{[i]})$ and the* segment bit *$sb(v^{[i]})$ are defined as follows: $sn(v^{[i]}) = s$; $sb(v^{[i]}) = i - \sigma$.*

1. For a constant c: $\Phi(c) = [c]$.
2. For a non-bit-propagating variable v: $\Phi(v) = \left[v^{[|v|-1:0]}\right]$.
3. For a bit-propagating variable $v = \texttt{concat}(v_1, v_2)$: $\Phi(v) = \left[\phi^{v_2}_{|\Phi(v_2)|-1}, \ldots, \phi^{v_2}_2, \phi^{v_2}_1\right] \circ \mathbb{M}(\phi^{v_2}_0, \phi^{v_1}_{|\Phi(v_1)|-1}) \circ \left[\phi^{v_1}_{|\Phi(v_1)|-2}, \ldots, \phi^{v_1}_1, \phi^{v_1}_0\right]$
4. For a bit-propagating variable $v = \texttt{extract}(u, m, l)$: $\Phi(v) = \left[\phi^{u\left[\mathrm{sb}(u^{[m]}),0\right]}_{\mathrm{sn}(u^{[m]})}, \phi^{u}_{\mathrm{sn}(u^{[m]})-1}, \ldots, \phi^{u}_{\mathrm{sn}(u^{[l]})+1}, \phi^{u\left[|\phi^{u}_{\mathrm{sn}(u^{[l]})}|-1,\mathrm{sb}(u^{[l]})\right]}_{\mathrm{sn}(u^{[l]})}\right]$
5. For a bit-propagating variable v created by neither `concat` nor `extract`, create $\Phi(v)$ by reducing the operation to applications of `concat` and `extract` as presented in Fig. 1.

Figure 2: Algorithm for calculating the Bit-Propagating Normal Form (BPNF) for a variable v. The operator $\circ$ stands for concatenation of sequences.

For example, given $v_2 = \left[v_1^{[2:0]}, 0^1\right]$, we have $\mathrm{sn}(v_2^{[0]}) = 0$; $\mathrm{sn}(v_2^{[1]}) = \mathrm{sn}(v_2^{[2]}) = \mathrm{sn}(v_2^{[3]}) = 1$; $\mathrm{sb}(v_2^{[0]}) = 0$; $\mathrm{sb}(v_2^{[1]}) = 0$; $\mathrm{sb}(v_2^{[2]}) = 1$; $\mathrm{sb}(v_2^{[3]}) = 2$.

In our approach, the SMT solver creates the BPNF for each new constant, input variable and internal variable representing the result of an operation. The algorithm for calculating the BPNF is provided in Fig. 2. Calculating the BPNF for constants, non-bit-propagating variables, and variables associated with the `extract` operation is straightforward. Finding the BPNF for `concat` requires concatenating the BPNFs of the two operands, where the BPNFs of the new neighbour pair are merged. Due of space limitations, we omit the proof that the algorithm in Fig. 2 returns a BPNF.

3.2 Implementation

In this section we show how to take advantage of BPNFs to speed-up the SMT solver.

Hashing BPNFs. To ensure that variables with the same BPNF are not created more than once, we maintain a hash table with all the current BPNFs and their corresponding variables. Whenever an operation is applied by the user, the solver creates a BPNF for a variable representing that operation (an actual variable is *not* created at this stage). If the BPNF appears in the hash table, its corresponding variable is returned to the user, otherwise a new variable is created and returned to the user, and the hash table is updated accordingly. The overhead of creating and maintaining the hash table is negligible in practice.

Using BPNFs for Translating to CNF. A major goal in introducing BPNFs is decreasing the number of CNF variables and clauses. This is achieved by never introducing any CNF variables or CNF clauses to represent bit-propagating variables and operations. Instead, to represent a bit-propagating variable v in CNF, we use the CNF variables that represent the non-bit-propagating variables that appear in v's BPNF.

User-Given Threshold on the Number of Segments. Maintaining too many segments in a BPNF might inflate the memory and lead to a performance degradation (at least in theory), because the algorithm for calculating BPNFs in Fig. 2 is linear in the number of segments. Hence we allow the user to impose a threshold, T, on the maximal number of segments permitted in a BPNF. If the number of segments for a variable v is greater than T, the variable will be considered to be a non-bit-propagating variable by the algorithm. Hence its BPNF will contain $v^{[|v|-1:0]}$, and the soundness of the SMT solution with respect to the corresponding bit-propagating operation will be ensured by bit-blasting that operation to CNF. We analyze the empirical impact of experimenting with different T values in Section 4.

Rewriting `assert`-based variable definitions. The SMT-LIB 2.0 language allows the user to build formulas in various ways. One of the common ways to create a new variable corresponding to a new operation is to use the `declare-fun` command to create a fake input variable and then to assert (using the `assert` command) that the new variable is, in fact, the result of an operation over existing variables. For example, the following sequence creates a new variable v that is defined to be `repeat`$(u, 2)$:

i. (`declare-fun` u () (_ `BitVec` 32)); ii. (`declare-fun` v () (_ `BitVec` 64)); iii. (`assert` (= v (`repeat` u 2)).

Such a way of creating variables is incompatible with our algorithm for calculating BPNFs, since our algorithm would consider variables bound to operations to be non-bit-propagating input variables. In our example, instead of figuring that $\Phi(v)$ is $\left[u^{[31:0]}, u^{[31:0]}\right]$, the algorithm would consider v to be a non-bit-propagating input variable with $\Phi(v) = \left[v^{[63:0]}\right]$. To overcome this problem, the preprocessor must identify such cases and rewrite them into a BPNF-friendly dag-oriented representation. In our example, rewriting the last `assert` command into the following form solves the problem: (`define-fun` v () (_ `BitVec` 64) (`repeat` u 2)).

The preprocessing algorithm for carrying out such rewriting is straightforward. Its complexity is linear in the size of the problem, and the overhead is still low. Moreover, such an algorithm can be seen as a particular case of term substitution [11], which in any event is implemented in modern solvers and is known to be useful for BV reasoning [11, 10].

Constant Propagation. Constant propagation is known to boost the performance of SMT solvers, and hence it is commonly used [11, 10]. It is essential to make sure that constant propagation is applied to take full advantage of BPNF-based algorithms over three shift operations, because there exist cases where the second operands of shift operations are not constants originally, but become constants after constant propagation. Recall that our algorithms consider shifts to be bit-propagating operations only when the second parameter is a constant.

4 Experimental Results

We carried out a number of experiments over SMT-LIB benchmark families from the QF_BV category to demonstrate the usefulness of our algorithms.

In the first experiment, we measured the proportion of bit-propagating operations in all the families (with the exception of the mcm family, whose benchmarks do not always conform to QF_BV syntax). The results are displayed in Table 1. One can see that for 37 families, the proportion of bit-propagating operations is at least 5%.

In our second (and main) experiment we ran Hazel over these 37 families (with the exception of all the sub-families of sage except app10 and app6, since they have a huge number of

benchmarks) with different T values. Recall from Section 3.2 that T is a user-given threshold value that limits the number of segments allowed in a BPNF. Note that our BPNF-based algorithms are disabled when $T = 0$. Recall also that Hazel is Intel's new eager BV solver. For the experiments we used machines running Intel® Xeon® processors with 3Ghz CPU frequency and having 32Gb of memory. The time-out for all the experiments was 600 sec. Table 2 shows the results for all the families where Hazel's cumulative run-time was more than 1 second for at least one configuration. Benchmarks where all the configurations timed-out are not considered in the table. The time-out value was added to the run-time when a memory-out occured.

One can see that our algorithms result in a performance boost in the case of 14 families. More specifically, for these families there exists at least one configuration of Hazel with $T \neq 0$ that outperforms the configuration with $T = 0$. The speed-up is at least 30% for eight of the families. The performance boost is especially significant for the top three families. The family spear/openldap v2.3.35 can only be solved when our algorithms are applied and T is high enough. The family pipe can only be solved with the configuration $T = 10$, while we observe a solid performance boost of over 2x for the family brummayerbiere for non-0 configurations. The choice between $T = 10$ and $T = 1000$ is family-specific, while an additional experiment has shown that increasing T from 1000 to 100000 does not change the performance.

Table 3 shows the number of word-level operations before bit-blasting the formula to CNF, as well as the number of CNF clauses and CNF variables for the configuration with $T = 0$. It also shows the ratio by which these numbers are reduced for configurations with $T = 10$ and $T = 1000$ as compared to the configuration with $T = 0$. The main conclusions to be drawn from the table are as follows. First, the number of word-level operations is only slightly reduced or not reduced at all, while the number of CNF clauses and variables is usually reduced considerably. This hints that the contribution to performance of BPNF-based translation to CNF is higher than that of BPNF hashing. Second, in most cases, the reduction in the number of CNF clauses and variables translates to a performance boost. However, this correlation is not absolute. Consider the family brutomesso/core, where our algorithms exhibited their worst performance. The number of clauses and variables was considerably reduced for that family. The reason for the lack of correlation in this case is apparently related to the sensitivity of SAT solver heuristics to the problem representation. We leave the study of this phenomenon to future reasearch.

Finally, to demonstrate that Hazel can compete with academic state-of-the-art SMT solvers, we ran Hazel against the latest versions of Boolector [7] (version 1.5.118), STP [12] (version 1373M), and Mathsat 5 [8] (with the configuration applied at the SMT'12 competition) over the eight families where use of our algorithms resulted in a performance boost of at least 30%. See Table 4 for the results. Hazel outperforms the academic solvers on all but one family. In our experiments, model generation was enabled for all the solvers. When model generation is disabled, the only significant change in run-time is for Boolector over the family uclid/catchconv, where the run-time is reduced to 702 seconds. Hazel is still much faster on this family.

5 Conclusion

Our goal was to improve bit-vector reasoning in modern SMT solvers. We identified a family of ten *bit-propagating* bit-vector operations in the SMT-LIB 2.0 language that fulfill the following property: each output bit is either a bit of one of the inputs or a constant (0 or 1). We demonstrated that bit-propagating operations are encountered frequently in SMT-LIB benchmarks. We proposed dedicated algorithms for handling such operations during SMT preprocessing and confirmed their empirical usefulness over a variety of SMT-LIB benchmark families.

Table 1: The number of benchmarks and the proportion of bit-propagating operations are provided per each SMT-LIB family in the QF_BV category. The families are sorted, in descending order, according to the proportion of bit-propagating operations. The sub-families of asp are not shown since the proportion is zero for all asp benchmarks.

Family	#	Proportion	Family	#	Proportion
uclid/tcas	2	0.61	calypto	23	0.56
sage/app11	611	0.5	bruttomesso/core	672	0.49
bench_ab	285	0.44	sage/app10	51	0.42
bruttomesso/lfsr	240	0.38	brummayerbiere2	65	0.37
uum	8	0.36	crafted	21	0.35
check	5	0.35	sage/app6	245	0.34
sage/app12	5784	0.29	wienand-cav2008/Booth	6	0.25
pipe	1	0.25	stp_samples	426	0.24
sage/app2	1417	0.22	check2	6	0.21
sage/app7	8663	0.19	brummayerbiere	52	0.17
sage/app5	1103	0.16	sage/app9	3301	0.16
sage/app8	2756	0.16	sage/app1	2676	0.14
spear/openldap_v2.3.35	8	0.14	galois	4	0.14
bruttomesso/simple_processor	64	0.13	uclid_contrib_smtcomp09	7	0.13
spear/inn_v2.4.3	219	0.08	wienand-cav2008/Commute	6	0.08
spear/wget_v1.10.2	42	0.08	spear/samba_v3.0.24	1386	0.08
wienand-cav2008/Distrib	6	0.07	uclid/catchconv	414	0.07
spear/xinetd_v2.3.14	2	0.06	stp	1	0.05
brummayerbiere3	79	0.05	spear/zebra_v0.95a	9	0.048
rubik	7	0.04	spear/cvs_v1.11.22	29	0.03
brummayerbiere4	10	0.02	dwp_formulas	332	0.01
asp (23 sub-families)	501	0	gulwani-pldi08	6	0
tacas07	5	0	VS3	11	0

Table 2: The impact of BPNF-based algorithms. We show the run-time of Hazel in seconds corresponding to 3 different T values (0, 10, 1000), the speedups of configurations with $T \neq 0$ over configuration with $T = 0$, and the number of solved instances corresponding to the 3 different T values. The results are sorted by the maximal speed-up over the configuration with $T = 0$. Best run-times are highlighted.

	Run-time in Seconds			Time Ratio		Solved Instances		
Family	Hzl_0	Hzl_10	Hzl_10^3	10/0	10^3/0	Hzl_0	Hzl_10	Hzl_10^3
spear/openldap_v2.3.35	1800	600	**19**	3.000	96.774	5	7	8
pipe	600	**155**	600	3.872	1.000	0	1	0
brummayerbiere	1649	**709**	711	2.326	2.321	40	41	41
wienand-cav2008/Booth	43	**26**	**26**	1.643	1.626	2	2	2
uum	18	**12**	**12**	1.535	1.534	2	2	2
brutomesso/simple_processor	374	**266**	**266**	1.404	1.405	64	64	64
uclid_contrib_smtcomp09	226	200	**169**	1.130	1.340	7	7	7
uclid/catchconv	9	8	**7**	1.115	1.312	414	414	414
brummayerbiere3	2921	**2600**	2885	1.123	1.012	42	42	42
brutomesso/lfsr	8039	**7435**	7439	1.081	1.081	230	227	227
spear/samba_v3.0.24	3516	**3359**	3433	1.047	1.024	1386	1386	1386
spear/inn_v2.4.3	624	**607**	708	1.027	0.880	219	219	219
stp	11	**10**	11	1.023	1.000	1	1	1
spear/wget_v1.10.2	308	319	**306**	0.966	1.006	42	42	42
brummayerbiere2	**719**	801	799	0.899	0.901	32	33	33
calypto	**213**	254	253	0.838	0.843	11	11	11
brutomesso/core	**19355**	24307	24307	0.796	0.796	933	925	925

6 Acknowledgments

The author would like to thank Paul Inbar for editing the paper and the anonymous reviewers whose valuable comments helped the author improve it.

References

[1] Clark Barrett, Morgan Deters, Leonardo Mendonça de Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. *J. Autom. Reasoning*, 50(3):243–277, 2013.

Table 3: Hazel Statistics. The overall number of word-level operations, CNF clauses, and CNF variables are displayed for the configuration with $T = 0$ ('M' stands for millions). The reduction ratio of these parameters to the corresponding parameter for the configuration with $T = 0$ is displayed for the configurations with $T = 10$ and $T = 1000$. The families are sorted as in Table 2.

	0: Data Summary			**10: Reduction Ratio**			**1000: Reduction Ratio**		
Family	**Ops**	**Clss**	**Vars**	**Ops**	**Clss**	**Vars**	**Ops**	**Clss**	**Vars**
spear/openldap_v2.3.35	12469	3.86M	0.93M	1.0236	1.0986	1.2593	1.0236	1.1012	1.2661
pipe	1048	0.19M	92478	1.0029	2.7350	2.9764	1.0640	3.2986	3.6959
brummayerbiere	35351	14.5M	6.86M	1.0012	1.2373	1.2994	1.0012	1.2381	1.3004
wienand-cav2008/Booth	542	13471	3393	1	1.5158	1.3218	1	1.5199	1.3280
uum	1076	12605	5932	1	1.3812	1.3873	1	1.3812	1.3873
brutomesso/simple_processor	41736	1.66M	0.58M	1	1.3726	1.6326	1	1.3726	1.6326
uclid_contrib_smtcomp09	46156	1.84M	0.61M	1.0057	1.0584	1.0906	1.0068	1.0720	1.1125
uclid/catchconv	5.03M	63.5M	34.8M	1.0002	1.0711	1.0619	1.0002	1.0748	1.0631
brummayerbiere3	24289	2.46M	0.83M	1	1.0144	1.0137	1	1.0169	1.0175
brutomesso/lfsr	0.8M	71.9M	28.7M	1	1.2367	1.3153	1.0038	1.2399	1.3188
spear/samba_v3.0.24	9.4M	993M	338M	1	1.0322	1.0480	1	1.0380	1.0568
spear/inn_v2.4.3	0.11M	216M	46.4M	1.0048	1.0024	1.0055	1.0067	1.0040	1.0071
stp	0.32M	5.2M	3.39M	1	1.0025	1.0021	1	1.0025	1.0021
spear/wget_v1.10.2	15364	85.8M	19.6M	1	1.0006	1.0013	1.0017	1.0053	1.0019
brummayerbiere2	62564	71.9M	12.8M	1	1.3639	1.0038	1	1.3639	1.0038
calypto	11786	0.74M	0.29M	1.0448	1.3839	1.4321	1.0448	1.3963	1.4375
brutomesso/core	6.49M	106M	40.9M	1.0111	1.3442	1.4920	1.0370	1.3788	1.5262

Table 4: Comparing Hazel to Boolector, Mathsat, and STP. Best run-times are highlighted.

	Run-time in Seconds					Solved Instances				
Family	Btr	Mst	STP	Hzl_10	Hzl_10^3	Btr	Mst	STP	Hzl_10	Hzl_10^3
spear/openldap_v2.3.35	1924	1200	1204	600	**19**	5	6	6	7	8
pipe	325	600	600	**155**	600	1	0	0	1	0
brummayerbiere	907	4994	736	**709**	711	41	36	40	41	41
wienand-cav2008/Booth	21	**19**	45	26	26	2	2	2	2	2
uum	16	29	15	**12**	**12**	2	2	2	2	2
brutomesso/simple_processor	1908	22488	5156	**266**	**266**	64	29	59	64	64
uclid_contrib_smtcomp09	783	268	770	200	**169**	7	7	7	7	7
uclid/catchconv	9015	201	19	8	**7**	414	414	414	414	414

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. QF_BV Logic. http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2, 2010.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] 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.

[5] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC*, pages 522–527, 1998.

[6] Nikolaj Bjørner and Mark C. Pichora. Deciding Fixed and Non-fixed Size Bit-vectors. In *TACAS*, pages 376–392, 1998.

[7] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS*, pages 174–177, 2009.

[8] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, pages 93–107, 2013.

[9] David Cyrluk, M. Oliver Möller, and Harald Rueß. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *CAV*, pages 60–71, 1997.

[10] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. Dissertation, University of Trento, 2010.

[11] Vijay Ganesh, Sergey Berezin, and David L. Dill. A Decision Procedure for Fixed-Width Bit-Vectors. Technical report, Computer Science Department, Stanford University, April 2005.

[12] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, pages 519–531, 2007.

ddSMT: A Delta Debugger for the SMT-LIB v2 Format*

Aina Niemetz and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
http://fmv.jku.at/

Abstract

Delta debugging tools automatically minimize failure-inducing input and enable efficient localization of erroneous code. In particular when debugging complex verification backends such as SMT solvers, delta debuggers provide an effective debugging approach where other debugging techniques are infeasible due to the input formula size. In this paper, we present ddSMT, a delta debugger for the SMT-LIB v2 format, which supports all SMT-LIB v2 logics and in particular handles macros and scopes defined by the commands *push* and *pop*. We introduce its architecture and describe its workflow in detail.

1 Introduction

Delta debugging algorithms [1, 5, 6, 7, 9] based on the algorithms introduced in [8, 10] typically minimize failure-inducing input by omitting parts irrelevant to the original erroneous behaviour. The resulting simplified failure-inducing input represents a minimal configuration in the sense that all of its possible subsets are necessary to cause the test to fail. Sat Modulo Theories (SMT) solvers serve as a backend for various applications in the field of e.g. deductive software verification, model checking and automated test generation. These applications heavily rely on the correctness of the underlying SMT solver – a highly complex tool, where debugging faulty behaviour becomes increasingly difficult with respect to the input formula size and structure. Rather than manually tracing error paths in order to find the actual error location, delta debugging provides means to automatically minimize input for failing SMT solvers and enables solver developers to localize failure-inducing code in a time efficient manner. Further, as shown in [5], delta debugging in combination with fuzz testing is a particularly effective approach to uncover bugs in SMT solvers.

In 2009, deltaSMT, a delta debugger for quantifier-free logics of the previous SMT-LIB [3] version[1] developed by our group has been presented in [5]. It is tailored to the SMT-LIB v1 language, hence incompatible with SMT-LIB v2 [4], which is a major upgrade of its predecessor. Further, deltaSMT does not employ the original delta debugging algorithm proposed in [8], but exploits the hierarchical structure of the input formula similar to the hierarchical delta debugging approach described in [9]. Representing the input formula as a directed acyclic graph (DAG), deltaSMT tries to simplify nodes in a breadth first search (BFS) manner. Nodes are substituted one-by-one, depending on their sort, with either constant 0, constant 1, or one of their children. Unfortunately, this substitution approach is also one of the limitations of deltaSMT, as in the worst case, too many node-by-node substitution attempts (no matter if successful or unsuccessful) have a negative impact on the overall runtime. Further, we encountered various cases, where deltaSMT was struggling or even unable to simplify certain input files.

*This work was partially funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

[1] http://smtlib.cs.uiowa.edu/papers/format-v1.2-r06.08.30.pdf

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (assert (= x y ))
6  (push 1)
7    (define-sort sort2 () Bool)
8    (declare-fun x () sort2)
9    (declare-fun y () sort2)
10   (assert (and (as x Bool) (as y Bool)))
11   (assert (! (not (as x Bool)) :named z))
12   (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value ((let ((x 1) (y 1)) (= x y))))
17 (exit)
```

Figure 1: A simple example in SMT-LIB v2 format.

More recently and independently, an update of deltaSMT for SMT-LIB v2 by Pablo Federico Dobal and Pascal Fontaine has been released[2]. This version does not provide full SMT-LIB v2 support but syntactically extends the original tool for SMT-LIB v2 compliance, but without support for important new SMT-LIB v2 features such as quantifiers or *push* and *pop* commands. Note that in the following, we will refer to this update of deltaSMT as deltaSMT2.

In this paper we present ddSMT, a delta debugger for the SMT-LIB v2 format. It supports all SMT-LIB v2 logics. It is not based on deltaSMT, but tries to overcome its limitations with a different algorithmic approach, which we will introduce in detail in the following.

2 The Delta Debugger ddSMT

The delta debugger ddSMT is a tool for minimizing failure-inducing input in SMT-LIB v2 format based on the exit code of a given command (typically a call to an SMT solver) when executed on that input. It is implemented in Python 3 and not only supports all SMT-LIB v2 logics, but in particular handles macros (command *define-fun*), named annotations (attribute *:named*), and scopes defined by the commands *push* and *pop*. The tool is intended to be easy to maintain and extend and further provides a dedicated, modular and standalone SMT-LIB v2 parser, which particularly should be useful for prototyping other (Python) tools working on the SMT-LIB v2 language.

2.1 Architecture

One of the challenges introduced in v2 of the SMT-LIB language is the addition of the commands *push* and *pop*, which enables scoping of assertions, and sort and function declarations. Hence, SMT-LIB v2 distinguishes between local scoping of sorted variables and variable bindings (as defined by *forall*, *exists* and *let* terms) and global scoping as defined by the commands *push* and *pop*. Note that in the following, if distinction is needed, we refer to locally defined scopes as *term-level scopes*, and globally defined scopes as *command-level* scopes. Further note that ddSMT does not distinguish between actual functions and variables (or uninterpreted constants

[2] `http://www.verit-solver.org/veriT-toolsDownload.php`

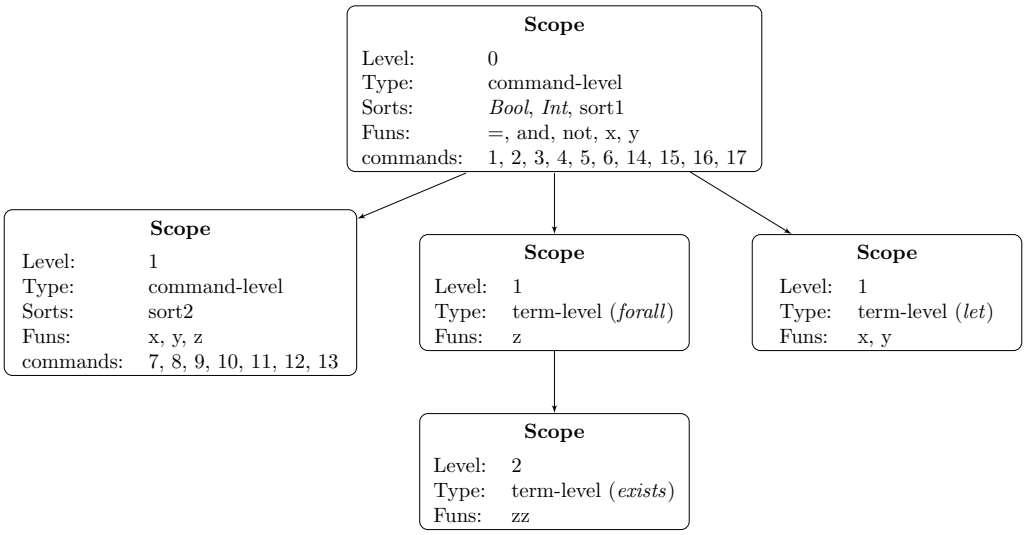


Figure 2: The basic internal structure of ddSMT given the Example in Figure 1.

in first order terminology) explicitly. Hence, in the following, if we do not make an explicit distinction, *function* may refer to either of them.

Internally the tool represents the given SMT-LIB v2 input as a tree of scopes. Each scope maintains a nesting level, a set of nested scopes, and a set of functions. Command-level scopes additionally maintain a set of commands and a set of sorts. Note that this structure enables a visibility handling of sorts and functions similar to related techniques in compiler construction, where a sorts (resp. functions) cache provides access to currently visible sorts (resp. functions) in constant time.

Example 1. To illustrate the basic internal structure of ddSMT as described above, consider the input file given in Fig. 1. As shown in Fig. 2, it defines two command-level scopes (the root scope at level 0 and the scope defined by given *push* and *pop* commands at level 1), and three term-level scopes defined by given *forall*, *exists* and *let* terms, respectively.

All sorts and functions defined at theory level are treated as being defined at level 0. Further, named annotations (attribute *:named*) are internally handled as if additionally a corresponding function definition had been given (in this particular case: `(define-fun z () Bool (not x))`). Commands are maintained by the scope they appear in, with a *push* command as the last command before a new scope is opened, and a *pop* command as the last command before the current scope is closed. Note that for better readability, we refer to the resp. commands in Fig. 2 by the line number they appear in Fig. 1.

In our example, the root scope maintains the predefined sorts *Bool* and *Int*, as well as the user-defined sort *sort1*. It further declares the predefined functions =, *and* and *not*, and the user-defined functions x and y (both of sort *sort1*). The command-level scope at level 1 maintains the user-defined sort *sort2*, and further declares functions x and y (both of sort *sort2*) and the named annotation z (of sort *Bool*). The term-level scope defined by *forall* at level 1 declares variable z of sort *Int*, whereas its nested term-level scope defined by *exists* at level 2 declares variable zz of sort *Int*. Finally, the term-level scope defined by *let* at level 1 declares variables x and y of sort *Int*.

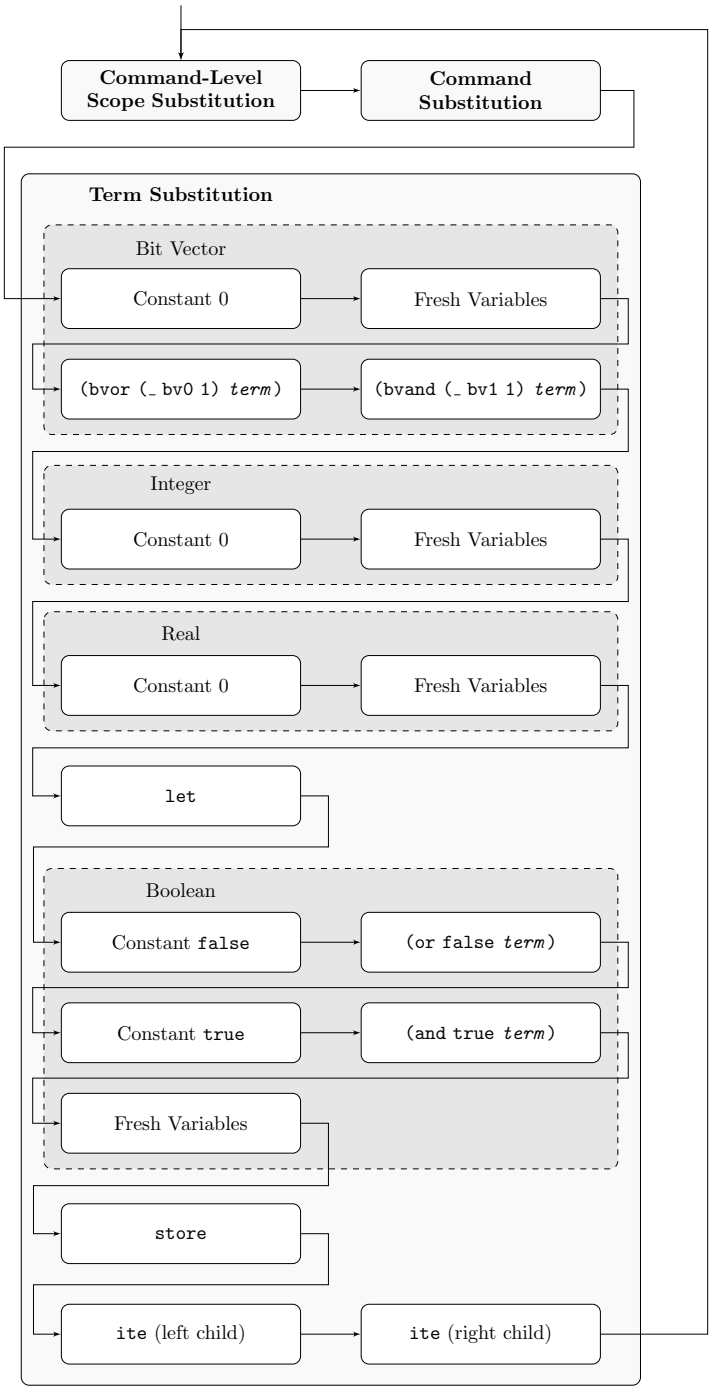


Figure 3: The general workflow and delta debugging phases of ddSMT .

2.2 General Workflow

The SMT-LIB v2 input is simplified by eliminating command-level scopes and commands, and substituting terms with simplified expressions. Note that *eliminating* scopes resp. commands refers to *substituting* nodes by *None* (the Python null object). In contrast to deltaSMT, ddSMT does not employ a hierarchical delta debugging approach on a BFS and node-by-node substitution base, but tries to exploit the strength of the original delta debugging algorithm (a divide-and-conquer strategy) as follows.

As illustrated in Fig. 3, ddSMT works in rounds, where each round is divided into several substitution phases. In each phase, nodes are first filtered and collected by a specific characteristic (e.g. nodes with a bit vector sort), and then substituted using a modified version of the original delta debugging algorithm as described in Fig. 8. The individual substitution phases are described as follows.

Command-level scope substitution Starting with the nested scopes of the root scope, command-level scopes are eliminated level-wise, in BFS manner, until a fixpoint is reached.

Command substitution After the command-level scope substitution phase, any command in any of the remaining command-level scopes irrelevant to the original failure-induced behaviour except the *set-logic* and *exit* commands, which are mandatory for starting and terminating SMT-LIB v2 scripts, is eliminated (while preserving the order of remaining commands) until a fixpoint is reached. Note that in the initial round, in order to prevent lots of likely unsuccessful test runs when eliminating e.g. *declare-fun* commands previous to term substitution, ddSMT considers *assert* commands only.

Further note that ddSMT does not ensure that the resulting simplified output is legal in the sense that e.g. variables must be declared previous to being used – the elimination of commands is solely tied to the exit code of the given command. This usually does not pose a problem though, as this kind of syntactically invalid input should be treated accordingly by a tool working on the SMT-LIB v2 language. In case the above behaviour poses a problem, e.g. when debugging parser related faulty behaviour, this can be easily handled by appropriate wrapper scripts (e.g. to check on specific solver output).

Term substitution Internally, ddSMT represents SMT-LIB v2 terms as DAGs with exactly one root. The SMT-LIB v2 format defines three commands with terms as arguments: *assert*, *define-fun* and *get-value*. Commands of each of these kinds are handled separately, with *define-fun* commands being processed prior to *assert* and *get-value* commands in order to prevent redundant substitution work due to the fact that functions defined via *define-fun* are usually referenced in *assert* and *get-value* commands multiple times. For each of these sets of commands, term substitution replaces terms w.r.t. their resp. sort (and other characteristics) in several steps as indicated in Fig. 3 until a fixpoint is reached. Note that individual steps (e.g. substitution of bit vector terms with constant 0) are defined by the characteristics of both the terms to be substituted and the substitution itself. Further note that steps depending on the SMT-LIB v2 logic in use are skipped if inapplicable (e.g. the substitution of *Real* terms with constant 0 or fresh variables if given logic is not a *Reals* logic). The individual steps are described as follows.

Initially, and depending on the SMT-LIB v2 logic in use, first bit vector, then *Int*, then *Real* terms are substituted with constant 0 and fresh variables, respectively. Additionally, if given formula is defined over the theory of *Fixed_Size_Bit_Vectors*, terms of the form `(bvor (_`

`bv0 1)` *term*`)` and `(bvand (_ bv1 1)` *term*`)` are replaced by their resp. child *term*. Next, *let* terms are replaced by their child term. After that, Boolean terms are substituted by constant *false*, constant *true*, and fresh variables, respectively. Subsequently, terms of the form `(or false` *term*`)` and `(and true` *term*`)` are replaced by their resp. child *term*. If the logic in use is an array logic, *store* terms are replaced by their child array terms. Finally, *ite* terms are substituted with their left and right child, respectively.

Note that in those steps of the term substitution phase, where terms are replaced with a simpler expression rather than one of their child terms (e.g. substituting Boolean terms with constant *false*), constant terms are skipped. Further note that each step of the term substitution phase (e.g. substituting bit vector terms with constant 0) is performed until a fixpoint is reached.

If any of the above substitution phases succeeded, i.e. if in any of the above phases, scopes, commands or terms have been eliminated or replaced successfully, ddSMT tries to iteratively simplify the current configuration even further until a fixpoint is reached.

Example 2. Continuing Ex. 1, consider the input file given in Fig. 1 and an executable failing on this input by not providing support for *get-value* commands as simulated by the Shell script given in Fig. 4. The input file is simplified by ddSMT in two rounds as follows.

In round one, first all redundant command-level scopes are eliminated. In this case, the scope defined by the *push* and *pop* commands in line 6 and 13 is redundant. The resulting simplified input is depicted in Fig. 5a. Next, all commands irrelevant to the failure-induced behaviour are successfully eliminated. As mentioned earlier, in the first round command substitution only considers *assert* commands. Hence, commands 5 and 6 (but not command 7, which is a *check-sat* command) are eliminated. The resulting simplified input is depicted in Fig. 5b. After command substitution, ddSMT subsequently performs term substitution on argument terms of *define-fun*, *assert* and *get-value* commands, in the order specified. As the current simplified input (as depicted in Fig. 5b) only contains a single *get-value* command, term substitution for *define-fun* and *assert* commands is skipped and the argument term of the *get-value* command at line 6 is the only one to be processed as follows. The original input in Fig. 1 is defined over the theory of *Ints* (but not over the theory of *Fixed_Size_Bit_Vectors* or *Reals*), hence all bit vector and *Reals* related steps are skipped. The *let* expression in line 6 contains two non-constant *Int* terms, `x` and `y`, which are first (and successfully) replaced by constant 0. The resulting simplified input is depicted in Fig. 6a. As no more non-constant *Int* terms remain, subsequent substitution with fresh variables is skipped. Next, the *let* term is successfully replaced by its child term (due to the fact that all occurrences of its variable bindings have been substituted by constant 0, previously). The resulting simplified input is depicted in Fig. 6b. Finally, the remaining non-constant Boolean term `(= 0 0)` is successfully replaced by constant *false*. As depicted in Fig. 6c, in the current simplified input the only remaining term (in the argument term of the *get-value* command at line 6) is a Boolean constant. Hence, all further term substitution steps operating on Boolean and *ite* terms are skipped and the first round concludes with the simplified input depicted in Fig. 6c.

In round two, the only successful substitution phase is command substitution, where commands 2, 3, and 4 are eliminated. The final result is depicted in Fig. 7.

2.3 `substitute`: The Delta Debugging Core Algorithm

The core of the actual delta debugging in ddSMT is the substitution algorithm described in Fig. 8. Command-level scopes and commands are substituted with *None*, whereas terms, depending

```
1  #!/bin/sh
2  if [ `grep -c "\<get-value\>" $1` -ne 0 ]; then exit 1 fi
3  exit 0
```

Figure 4: A simple Shell script simulating an executable failing on the input given in Fig. 1.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (assert (= x y))
6  (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
7  (check-sat)
8  (get-value ((let ((x 1) (y 1)) (= x y))))
9  (exit)
```

(a) The simplified input after command-level scope substitution.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value ((let ((x 1) (y 1)) (= x y))))
7  (exit)
```

(b) The simplified input after subsequent command substitution.

Figure 5: The input of Fig. 1 during the first substitution round in Ex. 2.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value ((let ((x 1) (y 1)) (= 0 0))))
7  (exit)
```

(a) The result of substituting non-constant *Int* terms with constant 0.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value ((= 0 0)))
7  (exit)
```

(b) The result of substituting the *let* term with its child term.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value (false))
7  (exit)
```

(c) The result of substituting the remaining Boolean term with constant *false*.

Figure 6: Continuing from Fig. 5, all three simplified inputs are the result of individual steps of term substitution in the first round.

```
1  (set-logic UFNIA)
2  (check-sat)
3  (get-value (false))
4  (exit)
```

Figure 7: The final result of simplifying the input of Fig. 1 in Ex. 2 after the second substitution round. In round two, commands 2, 3, and 4 are eliminated during command substitution.

```
1  def substitute (subst_fun, superset):
2      granularity = len(superset)
3      while granularity > 0:
4          nsubsets = len(superset) / granularity
5          subsets = split(superset, nsubsets)
6          for subset in subsets:
7              nsubsts = 0
8              for item in subset:
9                  if not item.is_substituted():
10                     item.substitute_with(subst_fun(item))
11                     nsubsts += 1
12             if nsubsts == 0:
13                 continue
14
15             dump (tmpfile)
16
17             if test():
18                 dump(outfile)
19                 subsets.delete(subset)
20             else:
21                 # reset substitutions of current subset
22                 restore_previous_state()
23         superset = subsets.flatten()
24         granularity = granularity / 2
```

Figure 8: The core substitution algorithm in ddSMT in Python-style pseudo code.

on their sort, are replaced by constant 0, *false*, *true*, fresh variables, or one of their children, respectively. Each substitution phase utilizes `substitute` as follows. Given a substitution function `subst_fun` and a set of nodes filtered by some specific filter criteria (e.g. nodes with a bit vector sort) as `superset`, this set is gradually split into `nsubsts` subsets, where the `granularity`, i.e. the number of items, of each subset initially starts at `len(superset)`. Note that this basically means that in a first attempt, all nodes of `superset` will be substituted. For each `subset` of these subsets, all items are substituted by the application of the substitution function `subst_fun` to the resp. item before issuing the original command (usually a call to an SMT solver) on the current configuration. If this (so called) test run succeeds, i.e. if the exit code of the current run matches the exit code of the original configuration, the current simplified input is stored for immediate reuse in `outfile`. Otherwise, all substitutions of the current `subset` are reset and we continue with the next subset.

Note that previously substituted nodes will be skipped. This is due to the fact that `superset` initially contains either the original node (if it is yet to be substituted) or its most current substitution.

	TS	Files	Red. [%]			Time [s]			Runs			Mem. [MB]		
			avg	min	max	avg	min	max	avg	min	max	avg	min	max
deltaSMT	1	2	0	0	0	257	14	500	4051	655	7446	113	108	117
	2	95	94.0	0	99.9	49	0.1	1738	599	5	7296	111	33	153
	3	5	66.6	0	93.8	12	3	34	608	262	1297	107	76	126
	4	53	99.6	98.8	99.9	8	0.6	20	463	4	852	128	52	142
	5	-	-	-	-	-	-	-	-	-	-	-	-	-
ddSMT	1	2	90.0	83.9	96.0	44	9	79	1412	782	2041	13	10	16
	2	95	94.7	68.2	99.9	92	0.1	1594	1499	2	3790	15	10	24
	3	5	80.4	66.8	87.2	23	14	35	1533	1171	1764	11	10	12
	4	53	99.8	99.3	99.9	57	1	246	431	13	1240	28	15	42
	5	5	97.4	95.7	98.3	12	5	16	247	215	371	39	10	59

Table 1: Comparison between deltaSMT (for SMT-LIB v1) and ddSMT on test sets (*TS*) 1 to 5. Test set 1 to 4 are randomly generated bit vector formulas originally given in SMT-LIB v1, test set 5 contains non-quantifier-free test cases for CVC4. *Red.* denotes the overall reduction in percent of the original file size, *Time* denotes the overall runtime in seconds, and *Mem.* denotes the maximum resident set size in MB.

3 Experimental Evaluation

Our delta debugger ddSMT has recently been released[3] under version 3 of the General Public License (GPLv3)[4] and is currently still a work in progress. Its parser is tested on the complete SMT-LIB v2 benchmark set (available at [3]) and the delta debugger itself has been tested on a wide range of crafted instances and SMT-LIB v2 benchmarks using simple shell scripts in place of actual solver calls in order to achieve a wider distribution over the SMT-LIB v2 logics. Additionally, ddSMT has further been applied to actual failure inducing test cases encountered during the development of our solver Boolector[5], as well as the open source SMT solver CVC4[6], a joint effort between the NYU and the University of Iowa.

As of May 23rd 2013, deltaSMT2, which we understand to be still work in progress, does not produce legal intermediate output for bit vector logics and is thus not able to simplify any of the test cases available for Boolector. Further, deltaSMT2 does not support non-quantifier-free logics such as AUFLIA or AUFLIRA and is hence not applicable to any of the test cases available for CVC4. Unfortunately, it was therefore not possible to evaluate the overall performance of ddSMT in comparison to deltaSMT2 . Instead, we translated quantifier-free SMT-LIB v1 input to SMT-LIB v2 and run deltaSMT-0.2 and ddSMT-0.96-beta on various sets of test cases (*TS*) as indicated in Table 1. Test sets 1 to 4 are randomly generated bit vector formulas originally given in SMT-LIB v1 and serve as test cases for Boolector, whereas test set 5 contains non-quantifier-free SMT-LIB v2 test cases for CVC4. Input reduction (*Red.*) is given in percent of the file size of the original input file, *Time* denotes the wall clock runtime in seconds, and *Mem.* indicates the maximum resident set size per run in MB. All experiments were performed on a 3.4 GHz Intel Core i7-2600 machine with 16GB RAM, running a 64 Bit Arch Linux OS.

Overall and even though the bit vector test cases where originally given in SMT-LIB v1 (i.e. they do not employ SMT-LIB v2 features such as e.g. *push* and *pop* commands, which could be fully exploited by ddSMT), our first results look promising. Even for test cases, where deltaSMT failed to simplify given input at all, ddSMT successfully achieved reductions by at least 81.1% of the original input file size. Note that except for the test cases denoted in Table 1,

[3] `http://fmv.jku.at/ddsmt`
[4] http://www.gnu.org/licenses
[5] `http://fmv.jku.at/boolector`
[6] http://cvc4.cs.nyu.edu

we currently still miss real test cases in SMT-LIB v2 logics other than QF_BV, QF_AX and QF_AUFBV. We therefore would like to encourage the SMT community to actually use ddSMT and thus further its development, and appreciate any comments, suggestions or bug reports.

4 Conclusion

In this paper, we introduced our delta debugger ddSMT, a tool for minimizing failure-inducing input in SMT-LIB v2 format. It supports all SMT-LIB v2 logics and in particular handles macros, named annotations, and scopes defined by the commands *push* and *pop*. Especially in combination with fuzz testing, ddSMT provides an effective approach to find and localize bugs in tools working on the SMT-LIB v2 language.

Recently, model-based delta-debugging (and fuzzing) in the context of testing and debugging verification backends was reported to be more effective than file based delta-debugging [2], in particular in combination with option resp. configuration fuzzing. Even though the delta-debugger ddSMT presented in this paper does not work on the API level of an SMT solver directly, we believe that the "programmatic nature" of the SMT-LIB v2 format using commands allows ddSMT to be equally effective.

In future work we will compare the effectiveness of API level delta-debugging with the approach presented in this paper. We further plan to evaluate ddSMT in combination with fuzzing SMT-LIB v2 input with command-level scopes.

We want to thank Morgan Deters for providing actual test cases for the SMT solver CVC4.

References

[1] Cyrille Artho. Iterative Delta Debugging. In Hana Chockler and Alan J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2008.

[2] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification backends. In *Proc. 7th Intl. Conf. on Tests & Proofs (TAP'13)*, LNCS, page 17 pages. Springer, 2013. To be published.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

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

[5] Robert Brummayer and Armin Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proc. 7th Intl. Workshop on Satisfiability Modulo Theories (SMT'09)*, page 5. ACM, 2009.

[6] Robert Brummayer and Matti Järvisalo. Testing and debugging techniques for answer set solver development. *TPLP*, 10(4-6):741–758, 2010.

[7] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated Testing and Debugging of SAT and QBF Solvers. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[8] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *ISSTA*, pages 135–145, 2000.

[9] Ghassan Misherghi and Zhendong Su. HDD: hierarchical Delta Debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 142–151. ACM, 2006.

[10] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

Z34Bio: An SMT-based Framework for Analyzing Biological Computation

Boyan Yordanov, Christoph M. Wintersteiger, Youssef Hamadi and Hillel Kugler

Microsoft Research, Cambridge, UK,
http://research.microsoft.com/z3-4biology

Abstract

The basic principles governing the development and function of living organisms remain only partially understood, despite significant progress in molecular and cellular biology and tremendous breakthroughs in experimental methods. The development of system-level, mechanistic, computational models has the potential to become a foundation for improving our understanding of natural biological systems, and for designing engineered biological systems with wide-ranging applications in nanomedicine, nanomaterials and computing. We describe Z34Bio (Z3 for Biology), a unified SMT-based framework for the automated analysis of natural and engineered biological systems. Z34Bio enables addressing important biological questions, and studying models more complex than previously possible. The framework provides a formalization of the semantics of several model classes used widely for biological systems, which we illustrate through the treatment of chemical reaction networks and Boolean networks. We present case-studies which we make available as SMT-LIB benchmarks, to enable comparison of different analysis techniques, and towards making this new domain accessible to the formal verification community.

1 Introduction

Many mechanisms and properties of biological systems remain only partially understood, thus limiting our understanding of natural living systems and processes. Recently, advanced experimental techniques have enabled the rational design and construction of biological systems, delineating a branch of biology as an engineering discipline, with potential applications in nanomedicine, nanomaterials and computing. However, understanding the system-level behavior of organisms or designing ones with specific behavior remains a major challenge for the engineering and the reverse engineering of biological systems.

Computational modeling, together with methods enabling the automated analysis of realistic models for diverse biological queries, can help address these challenges and tackle important questions related to *biological computation* - the information processing within living organisms. Along this direction, we introduce Z34Bio (Z3 for Biology) as a framework that allows flexible and scalable analysis of biological models using Satisfiability Modulo Theories (SMT)-based procedures. The framework provides a formalization of the semantics of several widely used formalisms in biological modeling, which we illustrate through the treatment of chemical reaction networks (CRNs) and Boolean networks (BNs), as well as combinations thereof. These formalisms are useful for describing DNA computing circuits (as well as more general biochemical mechanisms within natural systems) and biological interaction networks such as gene regulation networks (GRNs). We formalize the semantics of CRNs and BNs as transition systems, which we represent and analyze symbolically using SMT to allow flexible and convenient encoding of (possibly infinite-state) biological models.

The richness of the various SMT logics also allows us to express a range of important biological properties that are not easily captured by other specification formalisms. For instance,

we are able to formalize and study certain mass-conservation properties and the effect of gene knockouts on system dynamics. The availability of efficient decision procedures for some SMT logics such as uninterpreted functions and bit vectors (UFBV) with quantifiers [26] provides a foundation for the analysis of such questions, even for large and complex systems. While the Z3 theorem prover [9] is used in Z34Bio, arbitrary SMT solvers can be substituted in the framework through the SMT-LIB input language. In [28] we showed how SMT-based methods can be applied to engineered biological systems, and, more specifically, in DNA computing and synthetic biology. Here we present a framework supporting this approach accompanied by an online tool, extend it to allow modeling and reasoning about biological computation within living systems via Boolean networks, and provide support for hybrid models, composed of CRNs and BNs. We outline a number of case-studies illustrating the analysis of engineered DNA circuits and genetic regulatory networks (GRNs), which we curate and make available as SMT-LIB benchmarks, with the goal of improving the evaluation of existing SMT algorithms, helping in the development of new methods, and making this auspicious application domain more accessible to the SMT community.

2 Chemical Reaction Networks and Boolean Networks

In the field of DNA computing, which aims at engineering and understanding forms of computation performed by biological material (*e.g.*, reacting DNA strands), chemical reaction networks (CRNs) serve as models of circuits [25, 19]. More generally, CRNs are often used to describe a number of natural and engineered biochemical mechanisms. Here, we study such systems with single-molecule resolution, abstracting from the exact reaction kinetics (rates), thereby approximating probabilistic systems by non-deterministic ones. While certain information is not captured in this representation of the behavior of a CRN, it is a useful level of detail for various studies of DNA circuits, including cases where functional correctness is under investigation. Where studies of natural biological systems are concerned, this is often also a useful abstraction, when the rates of certain reactions are unknown and a precise measurement in a wet-lab is challenging.

We treat a CRN as a pair $(\mathcal{S}, \mathcal{R})$ of species (different DNA strands) and reactions where a reaction $r \in \mathcal{R}$ is a pair of multisets $r = (R_r, P_r)$ describing the reactants (inputs) and products (outputs) of r with their stoichiometries (the numbers of participating strands). We formalize the behavior of a CRN as the transition system $\mathcal{T} = (Q, T)$ where a state $q \in Q$ is a multiset of species, where $q(s)$ indicates how many strands of s are available in a state q, and T is the transition relation defined as $T(q, q') \leftrightarrow \bigvee_{r \in \mathcal{R}} [on(r, q) \wedge \bigwedge_{s \in \mathcal{S}} q'(s) = q(s) - R_r(s) + P_r(s)]$, where $on(r, q)$ is *true* if in state q there are enough molecules of each reactant of r for it to fire.

The complementarity of DNA sequences, dictated by the binding of Watson-Crick DNA base pairs (A-T and G-C), provides a mechanism for engineering chemical reaction networks using DNA. In this approach, various single and double-stranded DNA molecules are designated as chemical species. The binding, unbinding and displacement reactions possible between the complementary DNA domains (subsequences) of these species form the desired CRN structure. When specific computational operations are implemented using such a strategy, the resulting system is called a *DNA circuit* (see [19] and the references therein for additional details on the formalization and design of DNA circuits).

Figure 1 (left panel) shows a simple DNA circuit implementing a logical AND gate. The system is represented as a CRN with seven different species (A, B, C, Gate, GateA, GateB, GateAB) and four reactions, two of which are reversible as indicated by the bi-directional arrows. Species A and B represent the two system inputs, species Gate is the actual AND gate, and species C

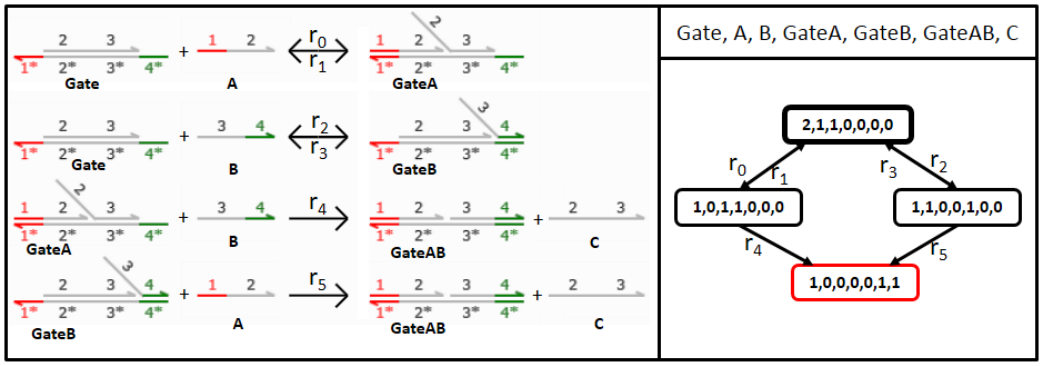


Figure 1: A simple DNA circuit implementing a logical AND gate. For each species (`Gate`,`A`,`B`,`GateA`,`GateB`,`GateAB`,`C`), domains labeled by $1, \ldots, 4$ represent different DNA sequences, while complementary sequences are denoted by $*$ (*e.g.* domains 1 and 1^* are complementary). The binding of complementary domains and the subsequent displacement of adjacent complementary sequences determines the possible chemical reactions $(r_0, \ldots, r_5)$ between the DNA species (left panel). The DNA circuit is represented as a transition system (right panel) where a state captures the number of molecules from each species and the initial state is highlighted using a thick black border. For this system, a state can be reached where no additional reactions are possible (shown with a red border), where computation terminates. For a single molecule of species `Gate`, the output `C` is produced at the end of the computation if and only if both input species `A` and `B` are present, which captures the required logical AND behavior.

is the output (all other species are intermediates). A state of the system captures the number of available molecules from each DNA species, which change as reactions take place, leading to the transition system representation in Figure 1 (right panel).

In some applications, it is sufficient to describe species more coarsely, using a small number of discrete levels of activity. This has proven to be a most useful abstraction, especially for analyzing the dynamics of species within gene regulatory networks (GRNs) [17] *e.g.* during the life-cycle of a cell or an organism. Unlike the biological engineering applications described above, the focus here is on understanding natural systems, and often only the species' presence or absence or the activity or inactivity of *genes* is tracked. A Boolean network is a popular representation of a GRN, which is given as a pair $(\mathcal{S}, \mathcal{F})$ of species and a set of update functions. We capture the behavior over time in the transition system $\mathcal{T} = (Q, T)$ where $Q = \mathbb{B}^{|\mathcal{S}|}$ and $q(s) \in \mathbb{B}$ indicates the presence or absence of s. The system's dynamics are defined by $\mathcal{F}$, which is a set of functions, one for each species, i.e., $f_s \in \mathcal{F}, f_s : \mathbb{B}^{|\mathcal{S}|} \to \mathbb{B}$ where the *synchronous*[1] transition relation $T(q, q') \leftrightarrow \left(\bigwedge_{s \in \mathcal{S}} q'(s) = f_s(q)\right)$ results in a deterministic, deadlock-free system. Despite the apparent simplicity of such models, they are tremendously useful in practice, because often we do not know the quantitative interactions within the system, and a precise measurement of levels of activation in a wet-lab experiment is challenging.

`Z34Bio` supports a natural combination of CRN and BN models, allowing for "localized" abstractions, *e.g.,* to simplify the analysis of parts of a system that do not require a model on the single-molecule level. These parts may be abstracted by a BN; Figure 3 shows an example of such a combined model, using the CRN component from Figure 1 and the Boolean network

[1] This means all species update at each time step, *asynchronous* updates are also supported by our framework, but not described here.

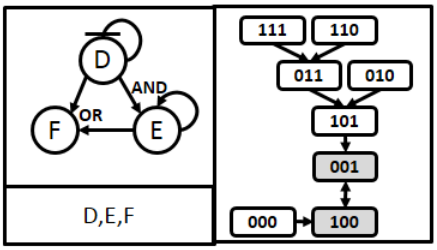


Figure 2: A Boolean network representing three species D,E, and F. The Boolean update functions are represented graphically (left panel) using pointed arrows (positive interaction), T-arrows (negative interaction), and the logical combination of inputs (*e.g.* the next state of species F is given by $F' = D \vee E$). The Boolean network is represented as a transition system (right panel) where all nodes are updated synchronously. This representation reveals that the system does not stabilize in a single state but instead reaches a cycle where the values of species D and F oscillate.

component from Figure 2 (modified to allow the interaction between the two components). In Z34Bio, we encode a BN as a single bit-vector, which leads to a compact representation, provides convenient bit-wise and arithmetic operations, while efficient decision procedures even when quantifier are used (SMT BV and UFBV), are also available [26]. We use integers to encode CRNs due to their potentially unbounded numbers of strands (or molecules) where this is required. When this is so, we first use Z34Bio in an attempt to prove the validity of mass-conservation constraints, providing us with bounds on the integer representation, thereby allowing the use of bit-vector encodings of appropriate size without sacrificing precision.

3 Analysis Strategies

The basic analysis strategy of Z34Bio is inspired by well-known model checking and deductive verification algorithms, most prominently Bounded Model Checking (BMC) and inductive invariants. We describe system behavior as constraints over a set of symbolic, finite paths of the transition system $\mathcal{T}$. A path is denoted as $\tau = \{q_0, \ldots, q_{K-1}\}$ where $\bigwedge_{i=0}^{K-2} T(q_i, q_{i+1})$ and $\tau[k] = q_k$ denotes the (symbolic) state at step k. Initial conditions of the system are described symbolically through constraints. Once all constraints describing the model as well as the property of interest are included, we encode them to a series of SMT queries, which allow us to find a model and to instantiate the abstract paths to concrete ones, or to report an unsatisfiable specification.

We use standard logical operations to construct formulas and enforce them for states. This allows us to find states with certain characteristics as (counter-) examples or prove their absence, and to test reachability and (certain) temporal properties over paths. For instance, stability and oscillations are studied in terms of paths with specific features. A *cycle* of length K is a path τ where $|\tau| = K$, $T(\tau[K-1], \tau[0])$ and no other states are repeated, and a *fixed point* is a cycle of length $K = 1$. For non-deterministic systems such as CRNs, a cycle τ may be *stable* or *unstable* resulting in persistent or (possibly) transient behavior, which is tested using a path τ' of length 2, such that $\forall\tau'.(\bigwedge_{i=0}^{K-2}(\tau[i] = \tau'[0]) \rightarrow (\tau[i+1] = \tau'[1])) \wedge ((\tau[K-1] = \tau'[0]) \rightarrow (\tau[0] = \tau'[1]))$, unsatisfiability of which indicates transient behavior.

General system analysis in the case of biology also includes the analysis of existing systems that occur in nature. While for some applications this can be viewed as the estimation of black-box behavior, typically, biologists propose models of natural behavior which are meant to explain observations made in labs and allow predicting the outcome of new experiments. As the models grow, the task of refuting a model, or verifying that a model indeed explains all known

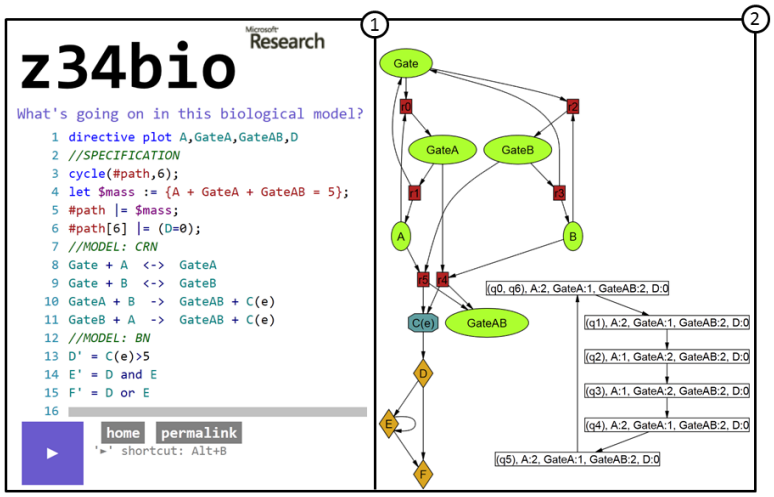


Figure 3: A screen shot illustrating the use of Z34Bio. A model (combined CRN and BN) and a specification (a mass-conservation property) are defined (1). When the "play" button is pressed, the model is visualized (Boolean, chemical and shared species are drawn as diamonds, ovals, and octagons, resp.). Interesting states and trajectories fulfilling the specification are found and displayed (2).

observations to a satisfying degree becomes harder. At the same time, the parts of the behavior of the models which are not covered by observations remain doubtful and it is imperative that new experiments for testing such models are performed to finally refine the theory. Therefore, the task of analyzing a biological model does not only include the establishment of invariants of such systems and studying their normal behavior, but also the investigation of perturbations and changes to the system behavior.

One instance of such a task is the analysis of *gene knockouts*, *i.e.,* the analysis of the system where one or more of the genes are permanently (or temporarily) disabled. Gene knockouts can occur naturally by acquiring a certain mutation in a gene, or induced by various experimental methods while studying model organisms (*e.g.*, fruit fly, worm). To automate the process of finding knockouts that effect a certain biological phenomena, and lead to interesting behavior, we augment the definition of a BN to include the bit-vector $ko \in \mathbb{B}^{|\mathcal{S}|}$, where $ko(s) \in \mathbb{B}$ indicates whether species s has been knocked out, in which case it is always inactive. Additional constraints, (*e.g.*, on the cardinality of ko) and properties regarding the desired behavior are specified and the missing information (specific gene knockouts) is obtained from the underlying SMT solver. To close the loop back to the biological science, note that "interesting" behavior in this type of analysis means that either a new experiment is suggested (when the system behaves in an unpredicted way), or that a problem in the model is identified because the model does not explain previous observations (where some gene may have been knocked out during a wet-lab experiment).

4 Applications

We implemented Z34Bio as an online analysis tool [29], providing basic user-interface and visualization capabilities (Figure 3). Analysis problems can be exported as SMT-LIB benchmarks

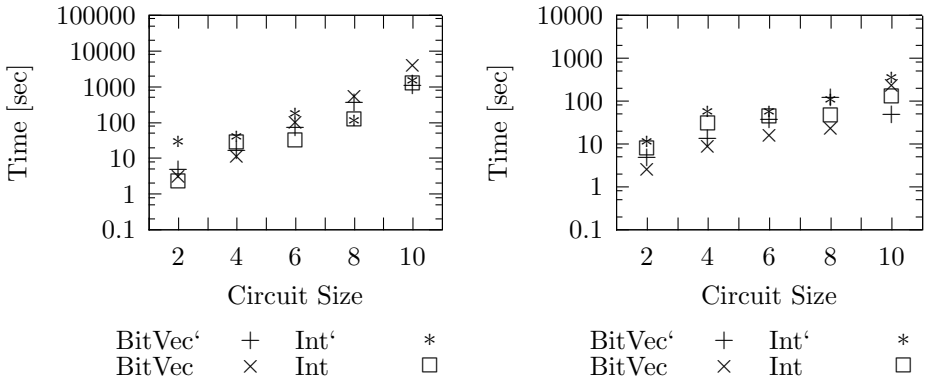


Figure 4: Computation times for the identification of traces of lengths up to $K = 100$ in the flawed transducer circuits such that a "good" state (left panel) or a "bad" state (right panel) is reached (note the difference in scales). BitVec‘ and BitVec (resp. Int‘ and Int) indicate a bit-vector (resp. integer) encoding with or without the additional mass-conservation constraints. Results from [28].

and processed offline. To illustrate some potential applications, we present a set of case-studies which are provided as benchmarks or can be explored interactively online. More details about these examples as well as additional challenging benchmarks are also available on our web-site [30]. The experimental results are obtained using the Z3 solver directly on the SMT-LIB benchmarks. All computation is performed on 2.5 Ghz Intel L5420 CPUs with a 2GB memory limit per benchmark.

The *transducer* DNA circuits described in [19, 28] are designed to convert all molecules of some chemical input to some output molecules (for an example of the structure of these models, see Figure 5). Computation terminates when a state with no possible reactions is reached but certain reactive species must also be fully consumed. First, we prove that a set of mass-conservation constraints hold for these systems, which capture the property that DNA is not created or degraded but only converted between species. Then, we use these constraints to prove that no "bad" terminal states (where reactive species remain) are possible for correct transducer circuits, but such states can be found for "faulty" designs. Finally, we show that "good" (resp. both "good" and "bad") states are reachable for correct (resp. "faulty") circuits using Bounded Model Checking [28]. Computational results from [28] shown in Figure 4, show that models take non-trivial runtimes, but are in a feasible range. The tradeoff between using bitvectors or integer representation is an interesting aspect for further exploration, as it seems each performs better for different models, sizes, and queries.

Besides the set of transducer circuits, we also applied our method to analyze a design of a DNA circuit that computes the square root of a 4-bit number [23, 4], which is one of the largest DNA computation circuits constructed experimentally. This system is designed to compute the square root of a number represented using DNA species. Our results indicate that the described methods can be applied to analyze functional correctness of systems of such complexity with large numbers of copies of the circuit operating in parallel.

Understanding the effects of gene knockouts on the dynamics of gene networks is an important biological question. Such GRN perturbations are often caused by mutations leading to a

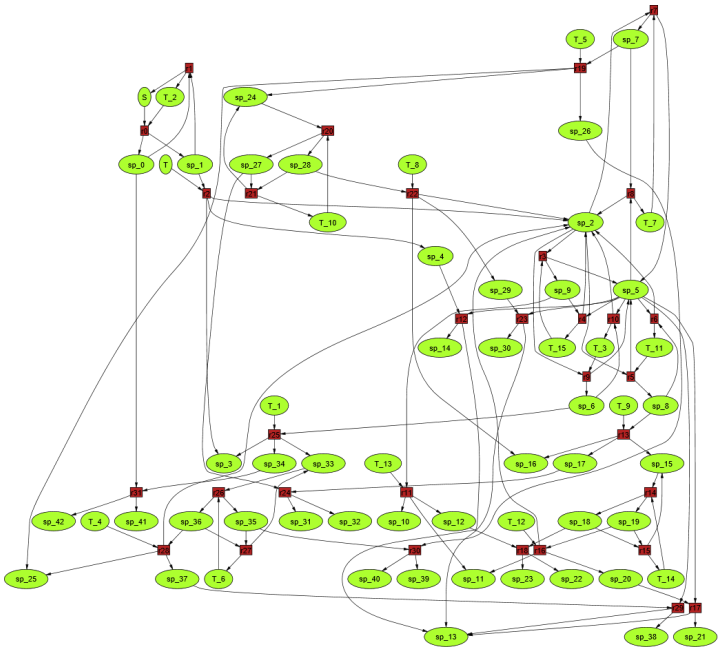


Figure 5: A chemical reaction network derived from a transducer DNA circuit.

number of diseases, while experimental techniques for introducing specific gene knockouts are also available, providing a strategy for the comparison of model predictions and experimental observations. For synchronous Boolean network models, stabilization is possible only when a fixed point exists, while oscillations are possible when a cycle of length $K > 1$ exists. We used bounded model checking to identify cycles of length up to $K = d$ (the recurrence diameter) and, while such a procedure is generally expensive, a short diameter is characteristic of some biological models. This is the case for the Boolean network models collected in [11] and the large-scale regulatory and metabolic network reconstruction studied in [24] (Table 1); for an example of the structure of such models, see Figure 6, which illustrates part of a fruit fly's GRN. A model's diameter also provides interesting information about the underlaying biological system, since it captures properties related to its response time.

To search for gene knockouts that influence stability, we first introduce the parameter ko and investigate how this affects the diameters d'. Next we find paths that originate in the same state but change the stability/oscillatory behavior depending on ko, providing the target set of knockouts (genes that must be knocked out to achieve the required behavior). Results in Table 1 show that we can effectively identify single and double mutations that effect system dynamics, and the method can be effective also for larger cardinality, providing a powerful way to investigate gene knockouts which is currently very challenging utilizing simulation methods. Overall, our framework has proven to be powerful enough to tackle important biological models, suggesting that SMT-based methods have the potential to play a significant role in this emerging field. Significant advances are still needed to allow biologists to analyze some of the systems they study, and we hope this work will inspire additional progress and development of methods towards this goal.

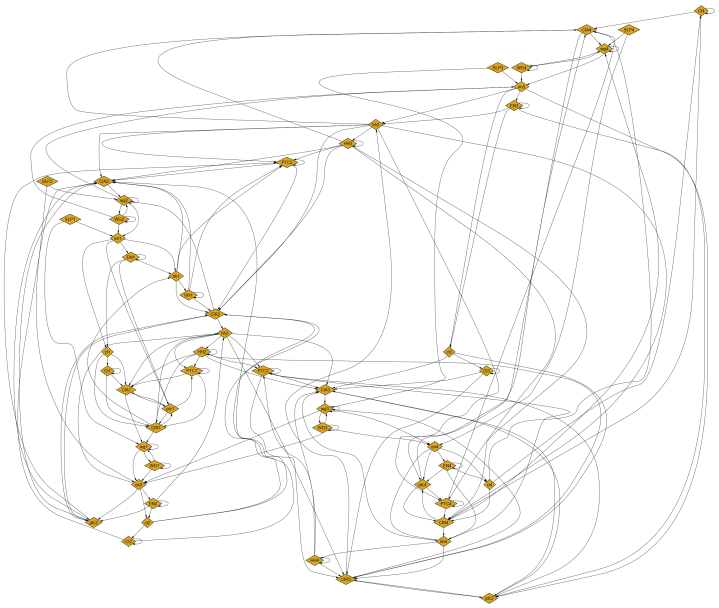


Figure 6: Boolean network model of Drosophila melanogaster's segment polarity genes. (Part of a fruit fly's gene regulatory network.)

Model name	$\|\mathcal{S}\|$	K	d	Behavior	Time	d'	Single knockouts	Time	Double knockouts	time
arabidopsis	15	10	10	stabil	0.12	10	LFY, SEP (2)	0.12	{EMF1,SEP},... (19)	2.17
budding yeast	12	18	18	stabil	0.20	18	none	0.5	none	0.5
drosophila	52	34	34	stabil	2.7	43	HH2 (1)	114.96	{SLP3,HH3},... (42)	929.1
fission yeast	10	6	6	stabil	0.09	6	none	0.1	none	0.08
mammalian	10	7	11	both	0.07	13	Rb,Cdc20,...(7)	0.75	{CycE,CycB},...(41)	2.997
tcr	40	6	18	both	0.26	22	CD8,CD45,...(9)	14.6	{CD45,IKK},... (345)	379.1
t helper	23	11	11	stabil	0.12	17	none	0.34	none	0.31
met. regulation	693	7	7	stabil	35.05	8	none	1154.4	none	1094.8

Table 1: Stability analysis and gene knockout identification for the Boolean network models from [11] and [24]. $|\mathcal{S}|$ is the number of species, d is the diameter (d' is the diameter when knockouts are allowed), K is the length of the shortest cycle when one exists or $K = d$ otherwise, and all times are in sec. Only some of the identified single and double knockouts are shown (total numbers in parentheses).

5 Related work and Future Directions

The simulation of biological models is now supported by many specialized tools (*e.g.* SimBiology by MathWorks), and has been used as an analysis strategy, for example, in [24] but is inherently incomplete and expensive for certain problems. As an alternative, the application of formal methods in the context of biology has already attracted attention [3], providing completeness and more rigorous formalizations of properties. For example, besides simulation capabilities, `Biocham` [13] allows the analysis of rule-based models [7] using temporal logics with numerical constraints, while deriving control strategies for large Boolean networks using CTL specifications and the NuSMV model checker is described in [20].

Such expressivity is not always sufficient - to capture notions of system stability an extension is introduced in `Anelope` [1]. Stability has also been studied through dedicated BDD-based

(GINsim [21]), SAT-based (BNS [11]) and modular (BMA [2]) algorithms for Boolean networks and their generalizations. Probabilistic model checking was used in [19] to study DNA circuits with properties involving probabilities and time, while Petri-nets analysis methods [15] also serve to study chemical reaction networks and therefore DNA circuits. Synthesis methods are developed in GNBox [6] based on Constraint Logic Programming (CLP) to uncover genetic network models from incomplete information while SMT-based approaches have been applied to computer-aided chemical synthesis planning [12] and scenario-based modeling in biology [18].

When a sufficient number of molecules is present, species concentrations can be described as continuous values ((*e.g.* using (non-linear) ODEs) [8]. Such systems, as well as other infinite-state, continuous and hybrid models used in biology, can be encoded into SMT directly but might require expensive (or incomplete) decision procedures. As an alternative, (conservative) finite transition system abstractions can be constructed (*e.g.* as in [27]), enabling the analysis and integration of infinite state systems within the framework described here. The application of formal methods to Petri Nets [5], which also describe chemical reaction networks, has been studied extensively and can provide useful analysis procedures, which can then be extended to other formalisms we consider through their common representation. Chemical reaction networks have also been studied at steady state using flux balance analysis (FBA) [22].

The available analysis tools often focus on a specific class of models and specifications, while so far the expressivity of SMT has not been fully exploited to allow a more general framework. By doing so, we handle logical, temporal and numerical constraints and can express certain stability properties, while the model-finding capabilities of SMT solvers enable us to seamlessly synthesize parts of the model. A number of extensions can be introduced immediately in our current framework (*e.g.* to capture more general genetic network models) but novel SMT procedures are required for others *e.g.* to allow analysis for probabilistic properties when chemical kinetics are considered [14, 16, 10].

Analysis of biological models is related to hardware and software verification in general. Due to the special nature of biological models it is often the case that traditional verification tools do not perform well on models of these systems, as they are highly concurrent and non-deterministic. A more thorough investigation of the differences between models in biology, software and hardware, especially on standardized realistic models (*e.g.*, through established SMT-LIB benchmarks which we initiate here) poses an interesting challenge for future work.

References

[1] Gustavo Arellano, Julián Argil, Eugenio Azpeitia, Mariana Benítez, Miguel Carrillo, Pedro Góngora, David A. Rosenblueth, and Elena R. Alvarez-Buyllaa. "Antelope": a hybrid-logic model checker for branching-time Boolean GRN analysis. *BMC Bioinformatics*, 12(1):490, 2011.

[2] David Benque, Sam Bourton, Caitlin Cockerton, Byron Cook, Jasmin Fisher, Samin Ishtiaq amd Nir Piterman, Alex Taylor, and Moshe Y. Vardi. BMA: Visual tool for modeling and analyzing biological networks. In *CAV*, volume 7358 of *LNCS*, pages 686–692. Springer, 2012.

[3] Miguel Carrillo, Pedro a Góngora, and David a Rosenblueth. An overview of existing modeling tools making use of model checking in the analysis of biochemical networks. *Frontiers in plant science*, 3(July):155, January 2012.

[4] Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Localized hybridization circuits. *DNA Computing and Molecular Programming (DNA17)*, pages 64–83, 2011.

[5] C. Chaouiya. Petri net modelling of biological networks. *Briefings in Bioinformatics*, 8(4):210–219, 2007.

[6] Fabien Corblin, Eric Fanchon, and Laurent Trilling. Applications of a formal approach to decipher discrete genetic networks. *BMC Bioinformatics*, 11(1):385, 2010.

[7] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Abstract interpretation of cellular signalling networks. In *VMCAI*, pages 83–97, 2008.

[8] H. de Jong. Modeling and Simulation of Genetic Regulatory Systems: A Literature Review. *Journal of Computational Biology*, 9(1):67–103, 2002.

[9] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[10] Christian Dehnert, Joost-Pieter Katoen, and David Parker. SMT-based bisimulation minimisation of markov models. In *VMCAI*, volume 7737 of *LNCS*, pages 28–47. Springer, 2013.

[11] Elena Dubrova and Maxim Teslenko. A SAT-Based Algorithm for Finding Attractors in Synchronous Boolean Networks. *IEEE/ACM TCBB*, 8:1393–1399, 2011.

[12] Rolf Fagerberg, Christoph Flamm, Daniel Merkle, and Philipp Peters. Exploring chemistry using SMT. In *PPCP*, LNCS, pages 900–915. Springer, 2012.

[13] Franois Fages and Sylvain Soliman. Formal cell biology in BIOCHAM. In *FMCSB*, 2008.

[14] Martin Fränzle, Holger Hermanns, and Tino Teige. Stochastic satisfiability modulo theory: A novel technique for the analysis of probabilistic hybrid systems. In *HSCC*, 2008.

[15] Monika Heiner, David Gilbert, and Robin Donaldson. Petri nets for systems and synthetic biology. *FMCSB*, 5016:215–264, 2008.

[16] David Henriques, João Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. Statistical model checking for markov decision processes. In *QEST*, pages 84–93. IEEE Computer Society, 2012.

[17] S A Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.

[18] H. Kugler, C. Plock, and A. Roberts. Synthesizing Biological Theories. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV'11)*, volume 6806, pages 579–584, 2011.

[19] Matthew Lakin, David Parker, Luca Cardelli, Marta Kwiatkowska, and Andrew Phillips. Design and analysis of DNA strand displacement devices using probabilistic model checking. *J. R. Soc. Interface*, 9(72):1470–85, July 2012.

[20] Christopher James Langmead and Sumit Kumar Jha. Symbolic approaches for finding control strategies in Boolean networks. *J. BCB*, 7(2):323–338, 2009.

[21] Aurélien Naldi, Denis Thieffry, and Claudine Chaouiya. Decision diagrams for the representation and analysis of logical models of genetic networks. In *CMSB*, 2007.

[22] J.D. Orth, I. Thiele, and B.O. Palsson. What is flux balance analysis? *Nat Biotech*, 28(3), 2010.

[23] Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–201, June 2011.

[24] Areejit Samal and Sanjay Jain. The regulatory network of E. coli metabolism as a Boolean dynamical system exhibits both homeostasis and flexibility of response. *BMC Systems Biology*, 2(1):21, 2008.

[25] Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–8, 2006.

[26] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. In *FMCAD*, pages 239–246, 2010.

[27] B. Yordanov and C. Belta. Formal analysis of discrete-time piecewise affine systems. *IEEE Transactions on Automatic Control*, 55(12):2834–2840, 2010.

[28] Boyan Yordanov, Christoph M. Wintersteiger, Youssef Hamadi, and Hillel Kugler. SMT-based analysis of biological computation. In *NFM*, volume 7871 of *LNCS*, pages 78–92. Springer, 2013.

[29] `Z34Bio`. Online at http://rise4fun.com/Z34Biology.

[30] `Z34Biology project website`. http://research.microsoft.com/en-us/projects/z3-4biology/.

SyMT: finding symmetries in SMT formulas

Work in progress

Carlos Areces[1], David Déharbe[2], Pascal Fontaine[3] and Ezequiel Orbe[1*]

[1] FaMAF, U. Nacional de Córdoba (Argentina)
[2] Universidade Federal do Rio Grande do Norte, (Brazil)
[3] Inria, U. of Lorraine (France)

Abstract

The QF_UF category of the SMT-LIB test set contains many formulas with symmetries, and breaking these symmetries results in an important speedup [8]. This paper presents SyMT, a tool to find and report symmetries in SMT formulas. SyMT is based on the reduction of the problem of detecting symmetries in formulas to finding automorphisms in a graph representation of these formulas. The output of SyMT may be used to improve SMT formulas to enforce the SMT solver to examine only one assignment out of many symmetric ones. We show that the classic propositional symmetry breaking technique can be lifted to SMT and yields a generic technique to break the symmetries found by SyMT.

Experiments on a large part of the SMT-LIB show that symmetries are pervasive in most categories.

1 Introduction

Consider a propositional formula $\varphi(p, q)$ with propositional variables p and q, and symmetric by permutation of p and q. Propositional symmetry breaking [12] eliminates symmetry, e.g. by adding clause $p \Rightarrow q$, since there is a Boolean model of $\varphi(p, q)$ if and only if there is a model such that $p \Rightarrow q$. Searching for models such that $p \wedge \neg q$ is unnecessary. Such transformation is a sound reduction of the search space for the SAT-solver.

Now, consider the first-order formula $\varphi(f(a) = f(b), a = b)$ with the standard interpretation of equality. It is clear that there exists no model such that $f(a) \neq f(b) \wedge a = b$ holds, although $f(a) = f(b) \wedge a \neq b$ is satisfiable; if $\varphi(p, q)$ has only models such that exactly one proposition in $\{p, q\}$ is true, $\varphi(f(a) = f(b), a = b) \wedge (f(a) = f(b) \Rightarrow a = b)$ is unsatisfiable. This simple example shows it is not sound to break symmetry of an SMT formula based on the symmetry of its propositional structure alone. Essentially the problem is that the abstraction does not take the theory into account. However, we show in the paper that it is sound to break symmetries stemming from permutation of uninterpreted symbols, similarly to what is done for propositional logic.

Previous results show [8] that exploiting symmetries in SMT formulas can lead to an impressive decrease in the size of the search space, and thus to a considerable increase in efficiency. Techniques described in [8] are, however, highly heuristic and vulnerable to formula rewriting. Graph automorphism detection algorithms [11, 9, 10] have been used to find symmetries in propositional formulas. Based on such techniques we developed SyMT, a tool to find and report symmetries in SMT formulas. In essence, the tool rewrites the SMT formula into a graph while preserving the syntactic symmetries. The resulting graph is suitable for input to graph

*This work has been partially supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS), and by CNPq/INRIA project SMT-SAVeS, and CNPq grant 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES).

isomorphism tools; in particular, there is no ordering on nodes' children. The output of SyMT provides to users the information they need to rewrite their SMT formulas so that they have no symmetries and are easier to solve. Such transformation is highly heuristic and domain-specific and it is the SMT user that is in the best position to realize it. In future work, we however plan to design of concrete generic symmetry breaking heuristics, that would provide further indication to users about how they can improve their formulas.

Outline. We first give a formal basis for symmetry breaking in SMT, then present our SyMT tool for detecting symmetries in SMT formulas. Section 3 also introduces the translation from SMT formulas to graphs. Some statistics on symmetry detection on a large part of the SMT-LIB [5] are given. They clearly show that (1) graph automorphism algorithms scale for SMT formulas, and (2) the SMT-LIB contain many highly symmetric formulas.

2 Symmetries in SMT

We assume knowledge of basic notions of permutation group theory, such as generators and cyclic forms. We use the standard notions of multi-sorted logic, term, formula, and interpretation commonly used in the context of SMT. A theory is a set of interpretations. Consider a finite set $\mathcal{S}$ of uninterpreted symbols (for constants, functions or predicates), and a bijective function σ on $\mathcal{S}$, that maps every symbol to a symbol of the same sort (i.e., arity and sorts of arguments and image should match). Function σ extends naturally to terms and formulas, and $t\sigma$ denotes σ applied to term or formula t, just like a higher-order substitution would, considering symbols in $\mathcal{S}$ as variables. σ can also be applied on an interpretation $\mathcal{I}$ to yield interpretation $\mathcal{I}\sigma$ similar to $\mathcal{I}$ except that $\mathcal{I}\sigma[s'] = \mathcal{I}[s]$ whenever $s\sigma = s'$. The identity function is denoted σ_I.

We say that σ is a symmetry for formula φ if $\varphi\sigma$ is syntactically equal to φ up to satisfiability preserving rewritings, e.g. using commutativity of some interpreted symbols. Notice that if σ is a symmetry for φ, so is any of its powers σ^i, and in particular σ^{-1} is also a symmetry of φ since there exists n such that $\sigma^n = \sigma_I$. The case where σ is its own inverse ($\sigma^2 = \sigma_I$) is a particular, though extremely frequent, case. It occurs when there is a group that contains all permutations of elements in a subset of $\mathcal{S}$. In our experiments on the SMT-LIB test bed, we have observed that most symmetry groups found have a set of generators that are their own inverse; in the following we will only consider such symmetries. Let us thus consider a symmetry σ such that $\sigma^2 = \sigma_I$ for a formula φ. For every interpretation $\mathcal{I}$ of φ we have $\mathcal{I}\sigma[\varphi] = \mathcal{I}[\varphi]$ (using straightforward structural induction). Consider now a set of atoms (not necessarily simple propositional variables) $p_1, \dots p_n$ and their image $q_1 = p_1\sigma, \dots q_n = p_n\sigma$. If φ is satisfiable in a model $\mathcal{M}$ then there exists a model of φ that furthermore satisfies the following formulas for $i \in \{1..n\}$:

$$\psi_i =_{\text{def}} \Big(\bigwedge_{1 \le j < i} p_j \equiv q_j \Big) \Rightarrow (p_i \Rightarrow q_i).$$

This model is indeed either $\mathcal{M}$ or $\mathcal{M}\sigma$. Assume k is the smallest value for which $\mathcal{M}[p_k] \neq \mathcal{M}[q_k]$, and consider ψ_k. If $\mathcal{M}[p_k] = \bot$ and $\mathcal{M}[q_k] = \top$ then $\mathcal{M}$ satisfies ψ_k, as well as all ψ_i with $i \neq k$. Now, if $\mathcal{M}[p_k] = \top$ and $\mathcal{M}[q_k] = \bot$ then $\mathcal{M}\sigma$ is a model of φ such that $\mathcal{M}\sigma[p_i] = \mathcal{M}\sigma[q_i]$ for $i < k$ and $\mathcal{M}\sigma[p_k] = \top$ and $\mathcal{M}\sigma[q_k] = \bot$. The model $\mathcal{M}\sigma$ of φ thus satisfies ψ_i for $i \in \{1..n\}$.

It is well known (see, e.g., [12]) that the formulas ψ_i can serve to break symmetry for propositional formulas. The above shows that this extends to SMT. This leaves out, however, many choices for the set of atoms p_i: the insight of the SMT user is usually necessary to make the best choice.

3 SyMT Implementation

SyMT is a command line tool implemented in C that detects symmetries in SMT formulas, taking into account the commutativity of conjunction, disjunction, addition, multiplication and equality. Given an input SMT formula, SyMT proceeds by creating a colored graph from it and then uses a graph automorphism component to detect the generators of the automorphism group of the colored graph. In particular, SyMT uses Saucy 3.0 [10] as the graph automorphism component. Integration with Saucy is done via Saucy's C API. SyMT also provides simplification capabilities on the input formulas, some of which involve using theory reasoning (and thus may unfortunately fail on large instances). Simplification of the input formula is important because it may uncover hidden symmetries and remove trivial symmetries, e.g., symmetries that do not involve uninterpreted symbols. Simplifications include simple rewriting, simplification of entailed literals, and some normalization of terms and formulas.

Example 1. *The command line and output of SyMT on a formula of the QF_UF category of SMT-LIB is as follows:*

```
./SyMT --enable-simp smt-lib2/QF_UF/NEQ/NEQ004_size4.smt2
(p7 p9)(c12 c13)
(c_3 c_1)
(c_2 c_1)
(c_0 c_1)
```

SyMT finds four generators for the symmetry group, and prints them in cyclic form. There is the full group of permutations for constants `c_0`*,* `c_1`*,* `c_2`*,* `c_3`*, generated by the last three generators, as well as a further symmetry that permutes unary predicate symbols* `p7` *and* `p9`*, as well as constant symbols* `c12` *and* `c13`*. This last symmetry was not detected with the heuristic techniques of [8].*

Reduction to the colored graph automorphism problem is the most successful technique for detecting symmetries in propositional formulas in clausal form, primarily due to the availability of efficient tools to detect graph automorphisms (e.g., [11, 9, 10]) that are fast and easy to integrate. Several reductions from propositional formulas to colored graphs have been proposed [6, 7, 1], all based on the same idea: to use the formula to construct a colored graph whose automorphism group is isomorphic to the symmetry group of the formula. Also, extensions to other logics, e.g., QBF [3] and modal logics [2], have been proposed, further showing the applicability of this technique. Nevertheless, as far as we know, there is no extension of this technique to the case of SMT formulas.

We now present the reduction algorithm to colored graphs for SMT formulas. The reduction is as a two-stage process. First, SyMT constructs the syntax direct acyclic graph of the formula with some additional nodes. Second, colors are introduced, to avoid spurious symmetries. Colors are represented as natural numbers. Let φ be an SMT formula. The colored graph $G(\varphi)$ is constructed recursively as follows (= and other predicates, and propositions are considered as functions and constants ranging over Booleans):

- **Graph Construction:**

  1. For each symbol, add a unique *symbol* node.
  2. For each (constant or propositional) term without argument, the *root* node is the symbol node introduced above.
  3. For each term $f(t_1, \ldots, t_n)$ of arity $n > 0$,

(a) Add a *root* node for $f(t_1, \ldots, t_n)$. Add an edge from the root node to the (unique) symbol node for f.

(b) If the function is commutative (e.g. $\wedge$, $\vee$, $\equiv$, $=$, $+$, $*$), add an edge from the root node to the root node of t_i ($i \in \{1..n\}$). Quantifiers, as commutative operators, are handled similarly (coloring discriminates the matrix).

(c) If the function is not commutative:

i. For each argument t_i, add an *argument node* and an edge from this node to the root node of t_i.

ii. Add an edge from the argument node of t_i to the argument node of t_{i+1} ($1 \leq i < n$). These edges represent the ordering of the arguments in $f(t_1, \ldots, t_n)$.

iii. Add an edge from the root node to the argument node of t_1.

- **Graph Coloring:**

1. Argument nodes are assigned a specific, unique color.
2. Uninterpreted symbol nodes and root nodes are assigned a color based on their sort (Boolean being considered as any other sort).
3. Each interpreted symbol node is assigned a unique color.

Example 2. *Consider formula* $\varphi = p(f(a,b)) \vee p(f(b,a)) \vee p(g(a,b)) \vee p(g(b,a))$*, where* p *is a unary predicate symbol and* f, g, a *and* b *are uninterpreted symbols. The associated colored graph,* $G(\varphi)$*, is shown in Figure 1 (colors are represented by numeric labels and node shapes in the figure).*

Theorem 1. *Let* φ *be an SMT formula and* $G(\varphi)$ *the colored graph constructed from it. Then, every automorphism of the graph* $G(\varphi)$ *is a symmetry of the formula* φ*.*

Proof sketch: This follows directly by structural induction and from the following observations:

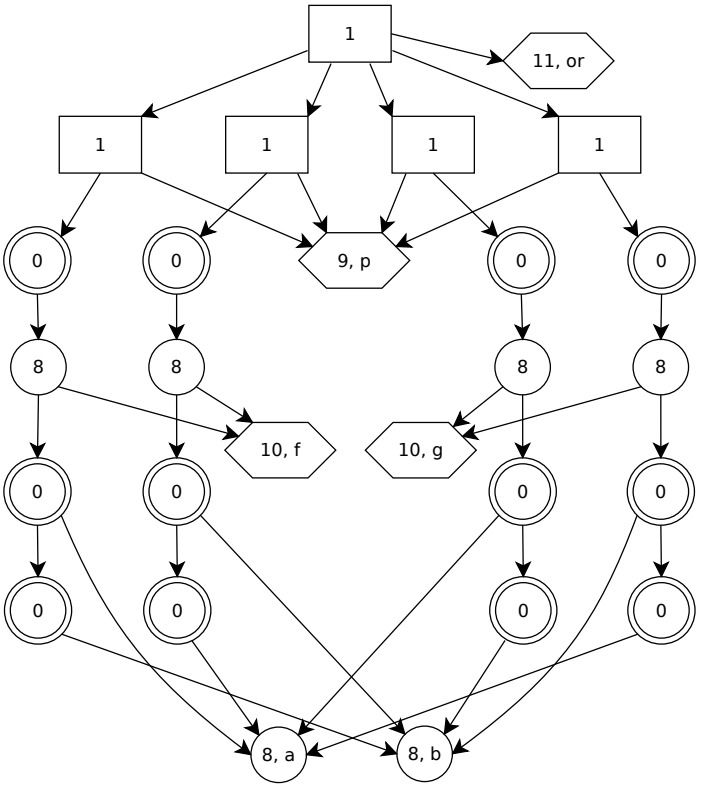
- The graph $G(\varphi)$ is the syntax directed acyclic graph of φ plus additional nodes.

- For terms, $f(t_1, \ldots, t_n)$ of arity > 0, the coloring of nodes, and the combination of root nodes and symbol nodes ensures that only symbols nodes of the same sort (i.e., same arity and same argument sorts) and with the same number of occurrences can be permuted.

- For terms without arguments (constants and predicates), the coloring of symbol nodes and the existence of argument nodes ensures that only symbols of the same sort and occurring the same number of times in the same argument positions can be permuted.

Finally, to reconstruct a formula symmetry from a graph automorphism, we just need to restrict the graph automorphism to symbol nodes. □

Notice that the converse of Theorem 1 is not true: the proposed graph construction does not find all the symmetries of the input formula. For example, consider the formula $\varphi = f(a,b) \wedge g(c,d)$ where a, d are of some sort and b, c of another sort (with appropriate sorts for f and g. The permutation $\sigma = (f\ g)(a\ c)(b\ d)$ is a symmetry of φ. Nevertheless, we can not detect it in the graph $G(\varphi)$. This is due to the fact that symbol nodes are colored based on the symbol sort, and this prevent the automorphism component from detecting permutations involving symbols of different sorts. Nevertheless, from a practical point of view, symmetries involving symbols of different sorts are rather unnatural and do not arise often.



Generator 1: $(a\ b)$
Generator 2: $(f\ g)$

Figure 1: Graph representation of $p(f(a,b)) \vee p(f(b,a)) \vee p(g(a,b)) \vee p(g(b,a))$.

4 Symmetries in SMT-LIB

We test SyMT against 19 categories[1] from SMT-LIB [5] to investigate the existence of symmetries and evaluate the efficiency of our tool. All tests are run on an Intel Xeon X3440 with 16GB, using the four cores simultaneously and we report the cumulative core time (roughly 4 times the CPU time). Three different configurations of SyMT were tested. Configuration 1 has no simplification: the formula is parsed and converted to a graph for automorphism detection. Configuration 2 uses trivial syntactic simplifications. Configuration 3 enables stronger simplifications, using an SMT engine, e.g., simplification of atoms implied by unit clauses. Configuration 2 may fail (with no symmetry reported) because the simplification algorithm used is not linear with respect to the input formula. However it often reveals symmetries hidden by irrelevant garbage easily removed by the simplification procedure. Configuration 3 is likely to fail on very large formulas, but again, it may reveal hidden symmetries. Simplification sometimes reduces a formula to false, in which case no symmetry is reported. The timeout (relevant for configuration 2 and, foremost, for configuration 3) is set to 30 seconds.

Among the 19 analyzed categories, three (LRA, QF_UFLRA, QF_UFNRA) do not reveal symmetries with SyMT. Of the only five formulas in UFLRA, one has symmetries. The others 15 categories presented a significant number of symmetries in at least one of the tested

[1] Bit vectors are not supported by our parser.

Category	#Inst	#Sym[1]	#Sym[2]	#Sym[3]	#Sym[P]	Avg[GS]	Time
AUFLIA	6480	6212	6231	5941	6258	134.00	378.79
AUFLIRA	19917	15779	16475	12500	16476	1.08	9.13
AUFNIRA	989	985	985	923	985	1.00	0.41
QF_AUFLIA	1140	2	71	77	78	1.00	0.72
QF_AX	551	22	22	22	22	1.00	0.37
QF_IDL	1749	348	526	683	756	12745.43	327.95
QF_LIA	5938	728	1172	524	1200	104.55	486.19
QF_LRA	634	73	150	208	210	110.49	29.06
QF_NIA	530	169	169	168	169	5.92	3.92
QF_NRA	166	9	43	43	43	1.00	0.23
QF_RDL	204	0	0	24	24	0.00	10.13
QF_UF	6639	250	3638	375	3638	44.00	34.58
QF_UFIDL	431	19	175	186	189	1.00	2.70
QF_UFLIA	564	0	198	198	198	0.00	0.45
UFNIA	1796	1062	1061	1058	1070	47.08	543.26

Table 1: Symmetries in SMT-LIB

configurations. Table 1 summarizes the results obtained for these 15 categories. For each category we report the number of instances (#Inst), the number of instances that have symmetries for the various simplification configurations (#Sym[1], #Sym[2] and #Sym[3]), the number of instances that have symmetries in at least one of the configurations (#Sym[P]), the average logarithm in base 2 of the size of the symmetry group (Avg[GS]) for Configuration 1, and the total time in seconds required to analyze all the instances (Time) also for Configuration 1. It is clear from Table 1 that the SMT-LIB has many highly symmetric formulas, in most categories. The cumulative time required to build the graph and detect the symmetries is negligible in all categories. We do not output the times for other configurations since there are timeouts and time is dominantly spent in the simplification modules, so these numbers give little insight about symmetry detection itself. The above experiments are using Saucy as the graph isomorphism detection engine. We also investigated Bliss as an alternative, with similar results. Unfortunately, this alternative is currently unavailable for users because of license issues.

The current tool fails to find some symmetries in the QF_UF category although they are discovered with the heuristic techniques from [8]. We are investigating the issue.

5 Conclusions and future work

We presented SyMT, a tool to detect symmetries in SMT formulas. SyMT is based on the reduction of the symmetry detection problem to graph automorphism detection. We presented the corresponding graph construction algorithm and showed that symmetry detection scales on SMT formulas by providing experimental results on executions of the tool on many SMT-LIB categories. We also showed that propositional symmetry breaking can be lifted to the SMT case, which provides a simple symmetry breaking mechanism for SMT.

In future work we will address the issue of symmetry breaking. We want to study the structures of symmetry groups found by SyMT. A deeper understanding of these structures may provide useful information to develop generic symmetry breaking mechanisms. We also believe that, to fully exploit the presence of symmetries in formulas, *ad hoc*, application-tailored,

heuristics are also necessary. We will use SyMT to mine the SMT-LIB to find symmetries, and we will devise appropriate heuristics integrated into an SMT symmetry breaking pre-processor. We expect this will result in a significant speed up for solving the formulas in the repository, since our experiments show symmetries are pervasive in many SMT test sets. We plan to carry out a similar analysis on the TPTP library [13].

We are aware that symmetry breaking is essentially heuristic, and a compilation of *ad hoc* heuristics would not be a silver bullet: the expertise of the user is generally the best approach to break symmetries. The current version of SyMT already provides the SMT users with a simple, yet powerful, tool to detect symmetries.

The tool and its source are available for download under the BSD License at `http://www.veriT-solver.org/SyMT`. It uses the Saucy 3.0 source code, distributed under its own specific license.

Acknowledgements: We thank Stephan Merz for interesting discussions, Cesare Tinelli for encouraging to investigate further symmetries in SMT, and the anonymous reviewers for their comments. We are very grateful to the Saucy developers for their tool.

References

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[2] C. Areces, G. Hoffmann, and E. Orbe. Symmetries in modal logics: A coinductive approach. In *Proc. of the 7th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2012)*, Rio de Janeiro, September 2012.

[3] G. Audemard, B. Mazure, and L. Sais. Dealing with symmetries in quantified Boolean formulas. In *Proc. of SAT'04*, pages 257–262, 2004.

[4] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[6] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *Proc. of AAAI Workshop on Tractable Reasoning*, pages 17–22, 1992.

[7] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR*, pages 148–159. Morgan Kaufmann, 1996.

[8] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 222–236. Springer, 2011.

[9] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Brodat, Daniel Panario, and Robert Sedgewick, editors, *Proc. of the 9th Workshop on Algorithm Engineering and Experiments and the 4th Workshop on Analytic Algorithms and Combinatorics*. SIAM, 2007.

[10] Hadi Katebi, Karem Sakallah, and Igor Markov. Conflict anticipation in the search for graph automorphisms. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *LNCS*, pages 243–257. Springer, 2012.

[11] B. McKay. Nauty user's guide. Technical report, Australian National University, Computer Science Department, 1990.

[12] Karem Sakallah. Symmetry and satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, 2009.

[13] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

SMT Solvers for Malware Unpacking (Extended Abstract)

Ian Blumenfeld, Roberta Faux, Paul Li and Mark Raugas

CyberPoint International, Baltimore, Maryland, U.S.A.
iblumenfeld@cyberpointllc.com, rfaux@cyberpointllc.com, pli@cyberpointllc.com, mraugas@cyberpointllc.com

Abstract

In order to perform in-depth malware analysis, reverse engineers must first overcome advanced packing methods employed by malware authors. We investigated using an SMT solver for undoing some of the code obfuscation used by a particular packer. In this note we describe the issues, our approach, and preliminary results.

1 Introduction

The analysis of malicious packed binaries is a challenging problem in the world of cyber security. Some of the more advanced packers employ code virtualization and obfuscation techniques to deter the malware analyst from comprehending the packed malware. This note describes an application of SMT solvers to this problem, and presents some preliminary results against one particular packer.

SMT solvers have previously been applied to the unpacking problem. Rolles ([8], [9], [10] has described using an automated theorem prover to verify an optimization-based code deobfuscator written against a commercial virtualizing packer. We investigated using the SMT solver itself as the deobfuscator. While this approach is currently much slower than optimization-based methods, we believe there is still some value in it.

As packers employ more complex obfuscation techniques, SMT solvers may be better able to handle them than syntax-based methods. For example, if a future version uses complicated linear arithmetic for code obfuscation, an LA theory solver could be very useful. Another benefit of using an SMT solver is that any improvements to the solver are automatically incorporated into our system.

Our experiments were done against sample programs that were packed with Themida, a commercially available virtualizing packer [7]. However, we expect that our approach will be applicable to other virtualizing packers such as VMProtect [3] and Enigma [1].

2 The Packer Problem

A major challenge facing malware analysts is the existence of "packed" binaries. Having their origins in simple compression or encryption, binary packers employ a wide array of techniques to deter automated and manual analysis. A wide variety of advanced packers are available to authors of both malware and legitimate software.

2.1 Virtualization

The use of virtualizing packers is becoming increasingly common [11]. A *virtualizing packer* obfuscates sections of binary code by translating them into bytecode streams that are executed by a custom interpreter routine ("virtual processor", or "VM") embedded within the packed file [6]. The software author has various ways to select portions of his code to virtualize, for instance by inserting special macros into his source code. Typically each virtual instruction type is executed by a separate subroutine of the virtual processor. We refer to these subroutines as *handlers*. To analyze a virtualized section of

```
dispatch:
  lodsb             ; loads the next byte into AL,
                    ; then advances the bytecode pointer ESI
  add al, bl        ; bytecode obfuscation
  xor al, 0xe8      ;
  add al, 0x47      ;
  sub bl, al        ;

  movzx eax, al     ; zero-extend the handler index
  jmp [edi+eax*4]   ; jump to handler
                    ; in most cases the end of the
                    ; handler jumps back to "dispatch"
```

Listing 1: Deobfuscated `dispatch` routine

code, it may be necessary to understand how the bytecode is parsed by the virtual processor, and the functionality of each handler.

For our experiments, we focused on Themida's "CISC VM." In this case a main dispatch routine reads one byte from the bytecode stream, uses that to calculate an index into a table of approximately 170 handler addresses, and then jumps to the appropriate handler. In most cases the end of the handler jumps back to the dispatch routine. To deter "disassembly" of the bytecode instructions, the code of the VM (including the handlers) would be obfuscated in a randomized manner, and the order of the handers in the table randomized, in a per-file basis.

The obfuscation is done using a pattern-based approach that repeatedly rewrites ("de-optimize") single instructions into more complicated but equivalent sequences of instructions in a randomized fashion [11]. For instance, `push EAX` may be converted into either

```
sub ESP, 4
mov [ESP], EAX
```

or

```
push 0x1d9fa093
mov [ESP], EDX
mov [ESP], EAX
```

By repeating this process many times, a relatively short snippet of x86 can be transformed into a much longer sequence of instructions whose functionality is no longer apparent.

2.2 Obfuscation Constants

In the Themida CISC VM, the main dispatch routine as well as about 15% of the handlers use an additional "obfuscation constants" technique to make the process of disassembling bytecode into virtual instructions more difficult.

Listing 1 shows the dispatch routine (after deobfuscation) in a particular packed file. Instead of using a byte from the bytecode stream directly as the handler index, the byte is first passed through an obfuscation function involving four randomly selected arithmetic operations (selected out of the set $\{\text{add}, \text{sub}, \text{xor}\}$), two numeric constants, and the "state" register EBX, following the general pattern above. The exact operations and constants will not be obvious in the obfuscated code. (The original obfuscated version of the dispatch routine is shown in Appendix A.)

Note that the value of EBX is updated by the process, so the the exact operations and constants need to be determined in order to statically disassemble a bytecode stream (i.e. determine the sequence of handlers).

```
handler_push_imm32:
  lodsd                    ; copy four bytes from bytecode
                           ; stream into EAX, then increment
                           ; the bytecode pointer ESI by 4

  xor eax, ebx             ; obfuscation operations and constants
  xor eax, 0xdeadbeef
  sub eax, 0xcafebabe
  xor ebx, eax

  push eax                 ; handler body
```

Listing 2: A handler using two 32-bit obfuscation constants

Most handlers that take an immediate argument also use this technique. For example, the handler that pushes a 32-bit constant onto the stack might look like Listing 2 in a particular file (after removing x86 obfuscation).

In this case "lodsd" (load dword) is used instead of "lodsb" (load byte), and the obfuscation pattern now involves the 4-byte registers EAX and EBX. (There is also a "lodsw" variety involving AX and BX.) Again, the choice of operations and constants will not be obvious from looking at the obfuscated x86 code for the handler. However, they will need to be determined before we can statically disassemble the next bytecode instruction (since EBX is affected).

3 Methodology

Our general methodology was to locate obfuscated handlers in sample packed files, and attempt to identify each one by comparing it with a list of reference specifications. The list of reference handlers was compiled through a combination of outside literature and manual analysis aided by a custom symbolic rewriter. To compare an unknown handler against a reference handler, both are symbolically executed and the resulting machine states compared.

We now describe our method in more detail.

3.1 The Toolchain

In order to execute the handlers, we implemented a simple symbolic simulator written in Haskell, heavily relying on the open-source SBV (SMT-Based Verification) library [5]. SBV provides a Haskell API that works for several SMT solvers, specifically Z3, Yices, CVC4, and Boolector. It also supports generating SMT-LIB2 output for use with solvers that are not directly supported.

Using SBV, we are able to set up a data type representing the state of a machine that operates on symbolic values. We use a flat memory model implemented as function from 32-bit symbolic bit vectors to symbolic bytes. SBV updates the function as values are stored to it. The registers are simply a static length array of symbolic bit vectors, with specific array offsets corresponding to the x86 registers EAX, EDX, etc. The emulator transforms the machine state according to instructions written in a small custom intermediate language (IL). Our IL is RISC-like and resembles LLVM [2].

We wrote a translator that disassembles the handlers' x86 binary instructions and translates them into our IL. The disassembly process involved following unconditional jumps (inserted to complicate analysis), and then recognizing when the translator has reached the end of the handler (since the jump back into the dispatch routine initially looks just like the other jumps). Although the handlers and obfuscation patterns used only a restricted subset of the possible x86 instructions (e.g. no floating point operations were used), the translation of the required x86 instructions was quite labor intensive.

For our solver, we primarily used CVC4 [4], because it has a convenient API through the Haskell SBV library; it performed well in comparison with other solvers; and its license permits commercial use

3.2 Generating Equations

Once we have translated our unknown handler and a candidate reference handler into our IL, we apply them to a common initial symbolic machine state and then compare specific portions of the resulting final states using the SMT solver (through SBV). In both preparing the initial state and selecting which part of the final states to compare, it was helpful to be familiar with general properties of the Themida handlers.

For instance, many of the handlers that access memory do so in a small number of specific ways relative to the initial register values. For example, a subset of handlers reference memory via EDX, e.g. "push dword [EDX]". When preparing the initial machine state for this reference handler, we can start with zero memory and store a symbolic value x at the location addressed by the (either symbolic or concrete) initial EDX value value of EDX). However, to capture the semantics of other handlers it was useful to initialize memory using an uninterpreted function from symbolic 32-bit words to symbolic bytes. For example, consider the handler

```
dispatch:
  lodsd                 ; load 4 bytes at esi into eax,
                        ; then increment esi by 4

  op1 eax, ebx          ; unknown obfuscation pattern
  op2 eax, const1       ;
  op3 eax, const2       ;
  op4 eax, ebx          ;

  push dword [eax]
```

Listing 3: Unknown obfuscation operations

with four unknown operations in $\{\text{add}, \text{sub}, \text{xor}\}$ and two unknown 32-bit constants. In this case the address of the memory read is not known *a priori* in terms of the initial machine state.

After symbolic emulation, we compare only certain registers and regions of memory that are most likely to be (meaningfully) affected by most of the handlers. Indeed, a side effect of the extra obfuscation code is to overwrites the area of the stack before the terminal stack pointer with garbage data, so we had to take care to exclude that area of memory from our comparison. In effect, the obfuscated version of the handler is *not quite* equivalent to the original version. For example, the obfuscated version of the reference handler

```
  pop eax
  push [eax]
```

almost always pushes the same value on the stack as the original version—except when the source address of the push happens to be in the area affected by the obfuscation.

Similarly, knowledge of the reference handlers helps us set constraints on program inputs to avoid pointer aliasing problems. Besides using the stack, the handlers typically address memory at a small offset from EDI, ESI, or EDX, so we make sure those initial values are at a safe distance away from ESP (either by concretizing or adding symbolic constraints).

In addition, we were able to use the SMT solver to recover the unknown obfuscation operations and constants described in Section 2.2. For example, consider an obfuscated handler shown in Figure 3 in deobfuscated form, with four unknown obfuscation operations and two constants. Originally, to compare this handler against a particular reference handler, we take the fixed portion of the reference

handler and prepend it with each potential obfuscation pattern (choice of operations), using symbolic values as the constants. In each case we then set the inputs to some concrete values, and try to solve for a satisfying model for the constants. When we find a solution, we perform a secondary check on the specific obfuscation pattern and constant values, using symbolic values for the initial machine state. Later, noting that EBX not used by the constant portion of most of the reference handlers, we were able to speed up the identification process by first solving for the obfuscation pattern and constants independently of the reference handler, by comparing EBX and nothing else, and then looping over the possible reference handlers once the obfuscation pattern and constants have been identified.

4 Experimental Results

We have tested our techniques on a simple "Hello world" program packed with Themida. For each handler in the packed code, we try to identify it, by running an equivalence test against each file in a list of reference specifications. This list of reference handlers were identified by using a combination of a simple rewriter and manual analysis. For handlers that made use of an obfuscation constant as in Section 2.2, we first identified the obfuscating operations and constants and then compared to each handler using this technique.

We ran tests single threaded on a 2.26 GHz Intel Xeon processor. However the problem equivalence checking against each item in a list in embarrassingly parallelizable, and for different cases could run multi-threaded against different sections of the list to improve speed in the trivial way.

Our packed program contained 139 handlers that did not make use of obfuscation constants. We gave these a timeout period of 30 seconds for each comparison. Of these, our techniques successfully identified 124 of them, about 89%. Most of these finished in under 30 seconds whereas the longest successful identification took about 11 minutes. The remaining 15 failed to match any of the reference handlers. Figure 1 shows the timings for the 124 successful cases.

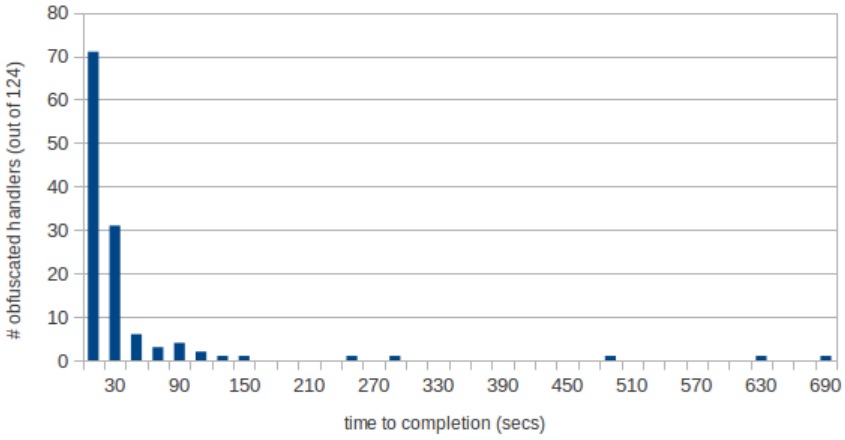


Figure 1: Time to identify simple handlers

In the case of handlers that made use of bytecode stream obfuscation we first performed a preliminary test to determine the obfuscation operations and constants. We timed these results. Afterward, we ran a secondary test, looping over the appropriate set of reference handlers once we determine the correct operations and constants. Once a matching handler was found, we stopped trying further patterns. The results from this are summarized in Figures 2 and 3.

Type	No. handlers	No. w/ solved patterns	No. matched
8-bit const.	13	11	10
16-bit const.	1	1	1
32-bit const.	15	12	12

Figure 2: Identifications of bytecode obfuscation patterns and handlers

Type	Median time to solve pattern	Median time to match
8-bit const.	102.11 s	63.28 s
16-bit const.	184.01 s	67.06 s
32-bit const.	91.39 s	52.59 s

Figure 3: Times to find bytecode obfuscation patterns and handlers

In the case of these handlers, the timings showed a large amount of variability. Identifying patterns and constants could take as little as just over a minute to as much as just over an hour. Identifications of the actual handler showed an even larger range, from about 8 seconds to just over an hour. Almost all fell close to the range of the medians given in Figure 3.

5 Conclusions

Certain x86 operations such as division occasionally resulted in comparisons that would run for a long time. We would like to investigate whether we may get a speedup by modeling the operation using an interpreted function symbol rather than using the more precise arithmetic model.

In the future, we plan to compare results using various SMT solvers. Initially our efforts have focused on using CVC4 however, Boolector has excellent performance on logical, bit-level operations which may be more effective on specific handlers. Additionally, as packers continue to evolve, one might expect the incorporation of API calls involving floating point operations. With the advent of Z3 currently implementing floating point arithmetic, it should be possible to add a floating point support to our capability.

The techniques described in this note should be applicable to other virtualizing packers. Constraint solving has traditionally been used in functional programming, yet there are a growing number of applications in malware analysis. As has been pointed out previously [10], SMT solvers are becoming a powerful tool for reverse engineers.

A Obfuscated `dispatch` routine

```
lodsb
xor al, bl
sub esp, 0x4
mov [esp], ebp
mov [esp], ebx
mov bl, 0x6c
shr bl, 0x7
xor bl, 0x70
xchg bl, ch
xchg ch, bh
not bh
xor ch, bh
xor bh, ch
xor ch, bh
xchg bl, ch
add bl, 0xde
push ecx
mov cl, 0xe3
shr cl, 0x7
xor cl, 0x6d
push eax
mov al, 0xa1
inc al
add al, 0xf2
add al, 0x72
xor cl, al
pop eax
dec cl
add cl, 0x1
xor cl, 0x63
add al, 0xfa
sub al, cl
sub al, 0xfa
pop ecx
add al, bl
push dword 0x15fb
mov [esp], ecx
mov ch, 0x68
push edx
push ecx
mov cl, 0xce
xchg cl, bh
not bh
xchg cl, bh
add cl, 0x5e
shl cl, 0x6
sub cl, 0x97
mov dh, cl
pop ecx
sub ch, dh
pop edx
not ch
sub ch, 0xaf
add ch, 0x48
push eax
mov al, 0xea
push ecx
mov cl, 0xba
xor al, cl
pop ecx
sub ch, al
pop eax
add al, ch
pop ecx
pop ebx
push word 0x235e
mov [esp], dx
push edx
mov edx, esp
push ecx
mov [esp], ebp
mov ebp, 0x4
add edx, ebp
mov ebp, [esp]
add esp, 0x4
sub edx, 0x4
xor edx, [esp]
xor [esp], edx
xor edx, [esp]
pop esp
mov [esp], ebx
push eax
push ecx
mov cl, 0xc5
mov al, cl
pop ecx
sub esp, 0x4
mov [esp], edx
mov dh, al
push ecx
mov ch, dh
mov bl, ch
pop ecx
push dword [esp]
pop edx
add esp, 0x4
pop eax
mov dh, bl
push dword [esp]
mov ebx, [esp]
add esp, 0x4
add esp, 0x4
add al, dh
mov dx, [esp]
add esp, 0x2
push ax
mov al, 0xfd
add bl, al
pop ax
sub bl, 0x5e
add bl, 0x1a
sub bl, al
push ecx
mov ch, 0x1a
sub bl, ch
pop ecx
push ecx
mov ch, 0x90
shr ch, 0x8
add ch, 0xff
push eax
push edx
mov dh, 0x4d
shl dh, 0x8
dec dh
not dh
or dh, 0x54
push cx
mov ch, 0xda
sub dh, 0x2b
sub dh, ch
push ebx
push ecx
mov cl, 0x7b
push eax
mov al, 0x63
dec al
not al
sub al, 0xed
mov bl, al
pop eax
add bl, cl
pop ecx
add dh, bl
pop ebx
pop cx
mov ah, dh
pop edx
and ch, ah
push dword [esp]
pop eax
push ebx
mov ebx, esp
add ebx, 0x4
add ebx, 0x4
push ebx
push dword [esp+0x4]
pop ebx
pop dword [esp]
pop esp
neg ch
add ch, 0x92
sub ch, 0xba
add bl, ch
pop ecx
push ax
mov al, 0xfd
sub bl, al
pop ax
movzx eax, al
jmp dword near [edi+eax*4]
```

References

[1] *The Enigma Protector*. `http://enigmaprotector.com`.

[2] *The LLVM compiler infrastructure*. `http://llvm.org`.

[3] *VMProtect Software*. `http://vmpsoft.com`.

[4] C. Barrett and C. Tinelli, *CVC4: the smt solver*. http://cvc4.cs.nyu.edu/web/.

[5] L. Erkok, *SMT based verification: Symbolic haskell theorem prover using SMT solving*. http://hackage.haskell.org/package/sbv, 2013.

[6] P. Ferrie, *Anti-unpacker tricks*, CARO Workshop, 2008.

[7] Oreans Technologies, *Themida*. http://www.oreans.com/themida.php.

[8] R. Rolles, *The case for semantics-based methods in reverse engineering*, RECON, 2012.

[9] ——, *Finding bugs in VMs with a theorem prover, round 1*. Rolf Rolles Blog, http://www.openrce.org/blog/view/1963/, January 2012.

[10] J. Vanegue, S. Heelan, and R. Rolles, *Smt solvers for software security*. USENIX Workshop on Offensive Technologies, 2012.

[11] Z. J. Wang, *Virtual machine protection technology and AV industry*, 2010. Microsoft Malware Protection Center.

Extending Proof Tree Preserving Interpolation to Sequences and Trees (Work In Progress)

Jürgen Christ
Albert-Ludwigs-Universität Freiburg
christj@informatik.uni-freiburg.de

Jochen Hoenicke
Albert-Ludwigs-Universität Freiburg
hoenicke@informatik.uni-freiburg.de

Abstract

We present ongoing work to extend proof tree preserving interpolation to inductive sequences and tree interpolants. We present an algorithm to compute tree interpolants inductively over a resolution proof. Correctness of the algorithm is justified by the concept of partial tree interpolants and the appropriate definition of a projection function for conjunctions of literals onto nodes of the tree. We observe great similarities between the interpolation rules for binary interpolation and those for tree interpolation.

1 Introduction

Craig interpolation is widely used in model checking [6, 13, 15]. Instead of binary interpolation [5], these techniques use inductive sequences or trees of interpolants. Tree interpolants arise from model-checking recursive and concurrent programs in a natural way. An execution of the program with procedures can be represented as a nested trace, where the statement after a procedure call has two predecessors, the return statement of the called procedure and the procedure call itself. To reason about correctness in a modular way requires combining the function summary with the intermediate assertion before the procedure call. This leads naturally to a tree-like structure [11, 12]. Tree interpolants are also useful to approximate function summaries for incremental update checking [17].

Similarly modular reasoning about concurrent programs need interference free proofs or assume-guarantee reasoning. The proof of an intermediate assertions can depend on the previous assertion of the same thread and the guarantees provided by the other threads. Thus, an unfolding of the parallel program has again a tree-like shape. Other uses of interpolants in model-checking are data-flow graph based method [8] which compute tree interpolants for an unfolded data-flow tree.

Although tree interpolants are widely used, only a few tools are able to produce them without the need for repeated applications of binary interpolation to different interpolation problems. The techniques used by these tools to ensure correctness of inductive sequences and trees of interpolants is not well documented.

In this paper, we extend the recently proposed technique of proof tree preserving interpolation [3, 4] to compute inductive sequences and trees of interpolants. The key idea of this technique is to define partial interpolants in the context of mixed literals that cannot be assigned to a partition of the input problem. This is achieved by introducing auxiliary variables and defining a projection function that splits mixed literals using auxiliary variables. We extend the projection function to trees of formulae and the notion of partial interpolants to partial tree interpolants. This allows us to compute tree interpolants inductively over the proof tree and is an essential step towards the correctness proof of the algorithm.

We show that tree interpolants can be computed for every node separately provided that some precautions are met for leaf interpolation. We give a rule that allows for inductively computing partial tree interpolants over a given proof. We observe that this rule for tree interpolation is identical to applying the rule from [3] for binary interpolation for every node separately.

Related Work. Only a few publications describe how to compute tree interpolants. Gupta et al. [10] describe how to solve a set of recursion-free Horn clauses over the theories of uninterpreted functions and linear real arithmetic. This corresponds directly to the tree interpolation problem for a conjunctive formula that does not contain negated equalities. They have stricter syntactic restrictions for the partial solutions and a rule for combining these, which is similar to our combination rule for partial interpolants. Our algorithm computes the same solutions when working on this fragment, however, we allow more input problems and our method is complete even for linear integer arithmetic.

The interpolating version of Z3 (iZ3) [1] can extract tree interpolants although there is no publication describing how it computes tree interpolants. iZ3 poses additional restrictions on the occurrence of symbols in the input and treats every non-constant function symbol as global symbol. In contrast to the method presented in this paper, iZ3 cannot interpolate linear real arithmetic and is incomplete on linear integer arithmetic.

2 Notation

We assume the usual notation and terminology of many sorted first-order logic. We consider the quantifier-free fragments of the theory of uninterpreted functions with atoms of the form $s_1 = s_2$ for two terms s_1 and s_2, and linear arithmetic over integer and reals. To allow for quantifier-free interpolation of integer arithmetic, we extend the signature with the functions $\lfloor \frac{\cdot}{k} \rfloor$ for all integers $k \geq 2$. By $\mathbb{Q}_\varepsilon$ we denote the rational numbers including an infinitesimal part [7], i. e., $\mathbb{Q}_\varepsilon = \mathbb{Q} \cup \{c - \varepsilon \mid c \in \mathbb{Q}\}$. We assume that the literals of linear arithmetic are normalised to the form $\sum_i c_i a_i \leq c$ where a_i are constant symbols, $c_i \in \mathbb{Z}$, and $c \in \mathbb{Z}$ (for integers) or $c \in \mathbb{Q}_\varepsilon$ (for reals). We use $t \leq c - \varepsilon$ to denote $t < c$. Note that in linear arithmetic the negation of an atom $\neg t \leq c$ can be expressed as $-t \leq -c - \varepsilon$.

For formulae we use the symbols F and I (for interpolants). By $symb(F)$ we denote the set of non-logical symbols occurring in F. We denote constant symbols by a, b, terms by s, t, numerical constants by c, variables by x, and set-valued variables by X. By $I[F]$ we denote a formula that contains the subformula F only positively. By $I(t)$ we denote a formula that contains a term t. Given two clauses $\ell \vee C_1$ and $\neg\ell \vee C_2$ (called antecedents), the resolution rule

$$\frac{\ell \vee C_1 \qquad \neg\ell \vee C_2}{C_1 \vee C_2}$$

concludes $C_1 \vee C_2$. In the context of SMT, a resolution proof is a derivation of the empty clause $\bot$ from the input clauses, theory lemmas, and theory combination clauses using only the resolution rule.

3 Proof Tree Preserving Interpolation

A binary interpolation problem consists of a pair of formulae (A, B). An interpolant exists if $A \wedge B$ is unsatisfiable and the theory admits interpolation. An interpolant I satisfies (i) $A \models I$, (ii) $B \wedge I$ is unsatisfiable, and (iii) $symb(I) \subseteq symb(A) \cap symb(B)$. In this section we briefly review proof tree preserving interpolation [4]. This technique extends the interpolation algorithms given by Pudlák [16] and McMillan [14] to mixed literals, i. e., literals containing symbols occurring only in formula A and symbols occurring only in formula B. For a given binary interpolation problem (A, B), proof tree preserving interpolation algorithms define two projection functions for every literal ℓ. The first projection, $\ell \lfloor A$, projects the literal onto A, the second, $\ell \lfloor B$ projects onto B. The algorithms from Pudlák and McMillan define the projection functions for non-mixed literals and require the invariant $\ell \leftrightarrow (\ell \lfloor A) \wedge (\ell \lfloor B)$. Usually, one of the projections is ℓ and the other is $\top$. The projection is extended to conjunctions of literals.

If ℓ is mixed, we cannot split the literal into two conjuncts such that the first conjunct only contains symbols occurring in A and the second conjunct only contains symbols occurring in B. We extended the projection function to mixed literals by introducing *auxiliary variables*. The invariant satisfied by our projection function is the existential closure $\ell \leftrightarrow \exists x.\ ((\ell \lfloor A) \wedge (\ell \lfloor B))$ of the invariant above. For example, the mixed literal $a \leq b$ may be split into $a \leq x$ and $x \leq b$ using the auxiliary variable x. The auxiliary variable is used to connect the projections to the A and B part, similarly (but orthogonal) to Nelson-Oppen theory combination.

To compute an interpolant from a resolution proof, we compute *partial interpolants* for every node in the resolution proof: Given a clause C in the resolution proof of $A \wedge B$, a partial interpolant I_C is an interpolant for $(A \wedge (\neg C \lfloor A), B \wedge (\neg C \lfloor B))$. Computation of partial interpolants differs between input clauses, theory lemmas, and results of resolution steps.

Since input clauses do not contain mixed literals, we can use the usual syntactic rules to compute partial interpolants [14]. Unfortunately, for theory lemmas the situation is different. These lemmas are the source of mixed literals in SMT proofs and, hence, need special procedures to compute partial interpolants. We compute an interpolant for $(\neg C \lfloor A, \neg C \lfloor B)$, i. e., an interpolant of the theory conflict (the negation of the theory lemma) projected onto A and B. The conflict is interpolated using a theory specific interpolator. Note that the auxiliary variables introduced during projection of mixed literals may occur in the interpolant.

The algorithms from Pudlák and McMillan give rules to compute a partial interpolant for the consequence of a resolution step given partial interpolants for the antecedents. The resulting partial interpolant is either a conjunction, a disjunction, or a multiplexer depending on whether the pivot literal occurs only in A, only in B, or in both. We extend these rules to handle mixed literals. The partial interpolants for clauses containing mixed literals contain the auxiliary variables introduced by the projection function only in specific syntactically restricted sub-formulae. The structure of these formulae makes it possible to remove the auxiliary variable once the corresponding mixed literal is the pivot of a resolution step. Details are out of the scope of this paper and can be found in [3, 4]. In this paper we will give slightly different syntactic restrictions in Section 5.4 and define the interpolation rules in Section 5.5.

4 Tree Interpolation

A tree interpolation problem is specified by a (directed) tree $T = (V,E)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges (pointing from child to parent node), and $L : V \to \mathit{Formula}$ is a labelling function that assigns a formula to every node in the tree. With $st(v) := \{w \mid (w,v) \in E^*\}$ (where E^* is the reflexive transitive closure of E) we denote the set of nodes in the subtree with the root v. A solution to the tree interpolation problem exists if the conjunction $\bigwedge_{v \in V} L(v)$ is unsatisfiable and the theories involved support interpolation. A labelling function $I : V \to \mathit{Formula}$ is called a *tree interpolant* [1] for T and L if the following properties hold:

1. $I(v_r) \equiv \bot$ where v_r is the root of T,
2. $(\bigwedge_{(w,v) \in E} I(w)) \wedge L(v) \models I(v)$ for all $v \in V$,
3. for every node v, all symbols in $I(v)$ occur both inside the subtree rooted at v and outside this subtree, i. e., $\mathit{symb}(I(v)) \subseteq (\bigcup_{w \in st(v)} \mathit{symb}(L(w))) \cap (\bigcup_{w' \notin st(v)} \mathit{symb}(L(w')))$.

By $lca(v,w)$, we denote the least common ancestor of v and w, i. e., the first common node in the tree that is encountered when traversing the tree towards the root from v and w. Obviously, for any pair of nodes v and w, the least common ancestor $lca(v,w)$ is unique. Every non-empty set of nodes has a unique least common ancestor, which can be computed by repeatedly applying the binary lca function.

We define for every node $v \in V$ the set of symbols $symb(v)$, that we could add to $L(v)$ without changing the symbol condition in the definition of tree interpolants. For a symbol a occurring in two nodes $v_1, v_2 \in V$, i. e., $a \in symb(L(v_1)) \cap symb(L(v_2))$, we add a to $symb(v)$ for every node v on the path from v_1 or v_2 to their least common ancestor $lca(v_1, v_2)$.

Lemma 1. *Replacing $symb(L(w))$ with $symb(w)$ does not change Condition 3. of tree interpolants. In particular*

$$\bigcup_{w \in st(v)} symb(L(w)) = \bigcup_{w \in st(v)} symb(w) \quad \text{and} \quad \bigcup_{w' \notin st(v)} symb(L(w')) = \bigcup_{w' \notin st(v)} symb(w').$$

Inductive Sequences and Tree Interpolation

Given a sequence of n formulae $F_1, \ldots, F_n$ such that $\bigwedge_{i=1}^{n} F_i$ is unsatisfiable, an inductive sequence of interpolants is a sequence of $n+1$ formulae $I_0, \ldots, I_n$ such that (i) $I_0 \equiv \top$, (ii) $I_n \equiv \bot$, (iii) $I_{i-1} \wedge F_i \models I_i$ for $1 \leq i \leq n$, and (iv) $symb(I_i) \subseteq (\bigcup_{j=1}^{i} symb(F_j)) \cap (\bigcup_{j=i+1}^{n} symb(F_j))$ for $1 \leq i \leq n$. Such a sequence can be computed either by repeatedly computing interpolants according to condition (iii), or by carefully extracting all $n+1$ interpolants for this sequence from one proof.

Theorem 1 (Sequence Interpolation is Tree Interpolation)**.** *Inductive sequences of interpolants are a special case of tree interpolants.*

Since we can recast a sequence interpolation problem as a tree interpolation problem we will only extend proof tree preserving interpolation to tree interpolation. The extension to sequences is left to the reader.

5 Adapting Proof Tree Preserving Interpolation to Tree Interpolation

To adapt proof tree preserving interpolation to tree interpolation we have to adapt the projection function used in binary interpolation to trees. Furthermore, we have to show that the interpolating resolution rules are still valid, i. e., that the interpolants computed by these rules satisfy the properties of partial tree interpolants. Throughout this section, let $T = (V, E)$ and L be a tree interpolation problem and $v, v_c, v_p \in V$ be nodes in the tree.

The projection function $\ell \downarrow v$ projects a literal ℓ onto the node $v \in V$ of the tree defining the interpolation problem. As in [4], we introduce auxiliary variables x for mixed literals ℓ. For tree interpolation, we introduce a fresh variable for each node $v \in V$ where the literal is mixed. The auxiliary variables introduced for a node are shared with the parent node; for each edge $(v_c, v_p) \in E$, the projection $\ell \downarrow v_p$ also contains the auxiliary variables of node v_c. The projection of a literal ℓ with the auxiliary variables $\vec{x}$ must satisfy two conditions. First, the conjunction of the projections of a literal ℓ onto every node in the tree $\bigwedge_{v \in V} \ell \downarrow v$ is equivalent to ℓ, i. e.,

$$\ell \iff \exists \vec{x}. \bigwedge_{v \in V} \ell \downarrow v \text{ where } \vec{x} \text{ is the set of auxiliary variables introduced for } \ell.$$

Second, $\ell \downarrow v$ must only contain theory symbols, symbols from $symb(v)$, and the auxiliary variables introduced for ℓ and v or the children of v.

Our algorithm computes a tree interpolant from the resolution proof by computing a partial tree interpolant for every clause C occurring in the proof tree. Partial tree interpolants are defined using the projection function as follows.

Definition 1 (Partial Tree Interpolant)**.** A *partial tree interpolant* for a clause C is a tree interpolant for T and L' where $L'(v) = L(v) \wedge (\neg C \downharpoonright v)$ for $v \in V$.

Obviously, a partial tree interpolant of the empty clause is a tree interpolant of T and L. Note that for an intermediate clause C a partial tree interpolant I may contain the auxiliary variables of the literals occurring in the clause C. To be more precise, for a node $v \in V$ the interpolant $I(v)$ may contain the auxiliary variables introduced for ℓ and v.

5.1 Adapting Propositional Interpolation Algorithms

In this section we show how the rules for propositional interpolation [16, 14] can be adapted to tree interpolation. Since we only consider the propositional case, we assume no literal is mixed. In this case we do not introduce any auxiliary variables. The projection $\ell \downharpoonright v$ is either ℓ or $\top$ and there must be at least one node $v \in V$ with $\ell \downharpoonright v = \ell$. These conditions guarantee $\ell \iff \bigwedge_{v \in V} \ell \downharpoonright v$. McMillan's and Pudlák's algorithm only differ in the projection function. For Pudlák's algorithm we set $\ell \downharpoonright v = \ell$ if and only if $\ell \in \mathit{symb}(v)$. For McMillan's algorithm we set $\ell \downharpoonright v = \ell$ only for the least common ancestor of the nodes $v \in V$ with $\ell \in \mathit{symb}(v)$.

Tree interpolants will be computed recursively over the resolution proof of the conjunction of the labels of the tree. As in the binary case, we devise special rules to compute partial tree interpolants for leaves of the proof tree. We compute partial tree interpolants for the resolution steps using the following rule.

$$\frac{\ell \vee C_1 : I_1 \qquad \neg\ell \vee C_2 : I_2}{C_1 \vee C_2 : I_3}, \text{ where } I_3(v) = \begin{cases} I_1(v) \vee I_2(v) & \text{if } \ell \downharpoonright v' = \top \text{ for all } v' \notin \mathit{st}(v) \\ I_1(v) \wedge I_2(v) & \text{if } \ell \downharpoonright v' = \top \text{ for all } v' \in \mathit{st}(v) \\ (I_1(v) \vee \ell) \wedge & \\ (I_2(v) \vee \neg\ell) & \text{otherwise} \end{cases}$$

The rule above can be interpreted as applying Pudlák's resp. McMillan's algorithm for each node separately. The condition $\ell \downharpoonright v' = \top$ for all $v' \notin \mathit{st}(v)$ means that the literal does not occur outside the subtree of v, i. e., the literal is A-local if we see the subtree of v as the A partition of a binary interpolation problem. Likewise the condition $\ell \downharpoonright v' = \top$ for $v' \in \mathit{st}(v)$ means that the literal is B-local. If neither is the case, the literal is shared.

Lemma 2. *The rule above is correct, i. e., if I_1 is a partial tree interpolant of $\ell \vee C_1$ and I_2 a partial tree interpolant of $\neg\ell \vee C_2$, then I_3 is a partial tree interpolant of $C_1 \vee C_2$.*

5.2 Occurrences of Symbols and Scope of Mixed Literals

An SMT proof may involve literals that are not in the original input formulae. These literals may be mixed, i. e., they contain symbols from different nodes. Let ℓ be a literal with $\mathit{symb}(\ell) = \{a_1, \ldots, a_n\}$. If for a node v, some symbols occur only inside the subtree rooted at v and some symbols occur only outside the subtree, we say that the literal ℓ *is mixed in* v. We denote with $\mathsf{mixed}(\ell)$ the set of all nodes v such that ℓ is mixed in v.

For a symbol a we overload *lca* and denote with $\mathit{lca}(a)$ the least common ancestor of all nodes $v \in V$ with $a \in \mathit{symb}(v)$. By the definition of $\mathit{symb}(v)$ for $v \in V$ we have $a \in \mathit{symb}(\mathit{lca}(a))$ and $a \notin \mathit{symb}(v)$ for $v \notin \mathit{st}(\mathit{lca}(a))$. For every symbol a, $\mathit{lca}(a)$ is the unique node such that all occurrences of a are in the subtree of this node and a occurs in the node itself. Having $a \in \mathit{symb}(\mathit{lca}(a))$ is the main reason why we defined $\mathit{symb}(v)$ in this way.

If a literal ℓ is mixed in some nodes we denote by $\mathsf{mixedparent}(\ell)$ the least common ancestor of nodes that have a child where ℓ is mixed. Then we can exactly characterise the set $\mathsf{mixed}(\ell)$ as follows:

Lemma 3. *Let ℓ be a literal that is mixed in some nodes and contains the symbols $a_1, \ldots, a_n$. Then*

$$\mathsf{mixed}(\ell) = \{v \in V \mid \exists i.\ 1 \leq i \leq n.\, lca(a_i) \in st(v) \text{ and } \mathsf{mixedparent}(\ell) \text{ is a proper ancestor of } v\}$$

5.3 Extending Projection

We now extend the projection functions to cope with mixed literals. For every literal ℓ and every node $v_j \in \mathsf{mixed}(\ell)$, an auxiliary variable x_j is introduced. The projection $\ell \lfloor v$ is chosen such that it is correct with respect to the following definition.

Definition 2. Let $\lfloor$ be a projection function. The projection function is *correct*, iff for all literals ℓ:

$$\ell \iff \exists \vec{x}.\ \bigwedge_{v \in V} \ell \lfloor v$$

where $\vec{x} = \{x_j \mid v_j \in \mathsf{mixed}(\ell)\}$ is the set of all auxiliary variables introduced for the literal ℓ.

5.3.1 Mixed Equalities

We start by giving the projection function for an equality literal $\ell :\equiv a_1 = a_2$. By Lemma 3, every node $v_p \in \mathsf{mixed}(\ell)$ lies on a path between $lca(a_i)$ and $\mathsf{mixedparent}(\ell)$ (for some $i \in \{1,2\}$). The a_i is unique, since v_p is not mixed if $lca(a_i) \in st(v_p)$ for both $i = 1,2$. For each node $v_p \in \mathsf{mixed}(\ell)$ we introduce an auxiliary variable x_p that captures the value of this a_i. The projection of ℓ achieves this by fixing the value x_p of a mixed node v_p to the value x_c of the (uniquely defined) child v_c that lies on the path to the unique $lca(a_i)$, or to the value of a_i if $v_p = lca(a_i)$. The projection of the node $\mathsf{mixedparent}(\ell)$ ensures that $a_1 = a_2$ by making the auxiliary variables of the corresponding children equal.

$$a_1 = a_2 \lfloor v_p = \begin{cases} x_{c_1} = x_{c_2} & \text{if } (v_{c_1}, v_p), (v_{c_2}, v_p) \in E \text{ and } v_{c_1}, v_{c_2} \in \mathsf{mixed}(\ell) \\ a_i = x_c & \text{if } (v_c, v_p) \in E,\ v_c \in \mathsf{mixed}(\ell), \text{ and } lca(a_i) = v_p \\ x_c = x_p & \text{if } (v_c, v_p) \in E,\ v_c, v_p \in \mathsf{mixed}(\ell) \\ a_i = x_p & \text{if } lca(a_i) = v_p,\ v_p \in \mathsf{mixed}(\ell) \text{ for some } i \in \{1,2\} \\ \top & \text{otherwise} \end{cases}$$

In the first two cases, we observe that a_1, a_2 occur inside the subtree of v_p. Hence, v_p is not mixed but has at least one mixed child. By Lemma 3, $v_p = \mathsf{mixedparent}(\ell)$. Usually, this means that there are exactly two child nodes v_{c_1} and v_{c_2} in which ℓ is mixed, one an ancestor of $lca(a_1)$ and one an ancestor of $lca(a_2)$ (first case). However, it is also possible that $v_p = lca(a_i)$ for one of the two symbols a_1, a_2 (second case). In both cases, the corresponding projection ensures that $a_1 = a_2$.

When ℓ is mixed in v_p, the third or the fourth case applies. Then, for exactly one $i \in \{1,2\}$, $lca(a_i)$ occurs in the subtree of v_p. If already $v_p = lca(a_i)$, we are in the forth case. Otherwise, the third case applies and v_c is the child containing $lca(a_i)$. Both projections ensure that $x_p = a_i$ for the value a_i that occurs in the subtree of v_p. The last case only applies if ℓ is not mixed in v_p and $v_p \neq \mathsf{mixedparent}(\ell)$. The correctness of this projection is proved in the appendix.

The projection of a disequality $a_1 \neq a_2$ is tricky. Instead of a plain auxiliary variable x_p we introduce a set-valued auxiliary variable X_p for every node v_p where the literal is mixed. For such a node v_p one a_i ($i = 1,2$) occurs only in the subtree of v_p and the other only outside the subtree. The projections of the literal enforce that X_p contains the value a_i that occurs in the subtree of node v_i and does not contain the other value. It may contain other values different from a_1 and a_2 when the value of a_i cannot be

expressed using only symbols shared between the subtree of v_i and its complement. The projections of $a_1 \neq a_2$ are defined as follows.

$$a_1 \neq a_2 \downarrow v_p = \begin{cases} X_{c_1} \cap X_{c_2} = \emptyset & \text{if } (v_{c_1}, v_p), (v_{c_2}, v_p) \in E \text{ and } v_{c_1}, v_{c_2} \in \mathsf{mixed}(\ell) \\ a_i \notin X_c & \text{if } (v_c, v_p) \in E,\ v_c \in \mathsf{mixed}(\ell), \text{ and } lca(a_i) = v_p \\ X_c \subseteq X_p & \text{if } (v_c, v_p) \in E,\ v_c, v_p \in \mathsf{mixed}(\ell) \\ a_i \in X_p & \text{if } lca(a_i) = v_p,\ v_p \in \mathsf{mixed}(\ell) \text{ for some } i \in \{1,2\} \\ \top & \text{otherwise} \end{cases}$$

Although the formulae are different, the cases are exactly the same as for equality. The fourth and third formulae ensure that X_p contains the value a_i that occurs in the subtree of v_p. With this property, each of the first two formulae ensures that $a_1 \neq a_2$.

Despite the definition of the projection function, we do not need set-theoretic reasoning in our solver. The projections are only used to prove the correctness of the resolution rule and the theory specific interpolation rules. The theory specific interpolation algorithm is specialised to conflicts arising from the Congruence Closure algorithm. Such conflicts may only contain a single disequality $a_1 \neq a_2$ and a chain of equalities that force the value of a_1 and a_2 to be equal. For each node v_p where this literal is mixed, we use the usual algorithm [9] to summarise equality chains originating from the subtree of v_p, which gives us a formula of the form $a_i = s_1 \wedge s_2 = s_3 \wedge \cdots \wedge s_{n-1} = s_n$ where a_i is the symbol that occurs in the subtree of v_p. The projection of the mixed literal to the subtree of v_p has the form $a_i \in X_{c_1} \wedge \cdots \wedge X_{c_k} \subseteq X_p$, which can be summarised by $a_i \in X_p$. The interpolant returned by our algorithm is $s_1 \in X_p \wedge s_2 = s_3 \cdots \wedge s_{n-1} = s_n$. Note, that if the conflict also contains mixed *equalities*, the plain auxiliary variable x_p introduced by that equality may occur in a shared terms s_i.

The syntactic restriction we pose on the partial invariants is that X_p occurs only in a literal $s \in X_p$, where s is an arbitrary term (not containing a set-valued variable). In particular, $s \in X_p$ may occur only positively. To get a similar notation as in our previous paper [3], we define $EQ(X_p, s) :\equiv s \in X_p$. On the other hand, a variable x_p introduced by a mixed equality may occur anywhere in the partial interpolant, even under a function application or in the s-part of an $EQ(X_p, s)$ term.

5.3.2 Mixed Inequalities

A linear inequality $\ell :\equiv c_1 a_1 + \ldots + c_n a_n \leq c$ is a comparison between a sum of n constant symbols a_i multiplied by a constant $c_i \in \mathbb{Z}$ and a constant $c \in \mathbb{Q}_\varepsilon$ for linear real arithmetic or $c \in \mathbb{Z}$ for linear integer arithmetic. We introduce an auxiliary variable x_j for every $v_j \in \mathsf{mixed}(\ell)$. We define a helper projection for the sum.

$$c_1 a_1 + \cdots + c_n a_n \downarrow v_p = \sum_{c \mid (v_c, v_p) \in E \wedge v_c \in \mathsf{mixed}(\ell)} x_c + \sum_{i \mid lca(a_i) = v_p} c_i a_i$$

Thus, the projection of the sum to v_p is the sum of all terms that occur in v_p for the last time and the sum of all auxiliary variables for all mixed child nodes. The projection of ℓ to the nodes v is defined as follows.

$$\ell \downarrow v_p = \begin{cases} (c_1 a_1 + \cdots + c_n a_n \downarrow v_p) \leq c & \text{if } v_p = \mathsf{mixedparent}(\ell) \\ (c_1 a_1 + \cdots + c_n a_n \downarrow v_p) \leq x_p & \text{if } v_p \in \mathsf{mixed}(\ell) \\ \top & \text{otherwise} \end{cases}$$

Again, the introduced auxiliary variable is shared between the node where it was introduced and its parent node. It is allowed to occur in the partial interpolant of its node but only in the special pattern $LA(s, k, F) :\equiv F$ which must occur positively in the interpolant. Here s is an affine sum of shared terms and auxiliary variables. Every variable x occurring in F must also appear in s with a positive coefficient

and F must be monotone in x, i. e., $x \geq x' \implies F(x) \to F(x')$. Finally we require that $F = \bot$ for $s > 0$ and $F = \top$ for $s < -k$.

The algorithm we present here is a slight improvement of the algorithm in [3]. We could use the same algorithm but the new one will compute slightly smaller interpolants. The basic difference is that $LA(s,k,F)$ was defined as $s \leq 0 \wedge (s \geq -k \to F)$, while in our new definition we assume that F is already false for $s > 0$ and true for $s < -k$. Also the monotonicity condition of F simplifies the correctness proof by avoiding the weak and strong interpolant that was needed in our previous paper.

5.4 Interpolation of Theory Lemmas

Our algorithm uses the Congruence Closure algorithm to produce conflicts in the theory of equality. We can compute the partial tree interpolant separately for every node of the tree. However, we must carefully assign every literal to a unique node of the tree. Then for every node the A part of the interpolation problem consists of all literals assigned to a node in the subtree and the B part consists of all other literals.

Usually an interpolant is computed as the summary of the A paths built from the conflict. In the presence of function congruence a more elaborated algorithm is needed [9]. This algorithm also works for mixed equality literals if they are split into their projections. The auxiliary variables can only occur in the interpolants of the nodes for which the variables were introduced.

However, for a mixed *disequality* it is not feasible to replace it by the projections of the literals as it involves new predicates and universally quantified formulae. Instead, they need to be treated separately. A conflict always involves exactly one disequality that contradicts an equality path. If this disequality is mixed, there is always an A path of equalities that start at the A part of the mixed disequality and ends at a shared symbol s. Instead of adding a summary equality, we add the literal $X(s)$, where X is the mixed predicate that was added for the mixed disequality in the current partition.

For inequalities we apply the algorithm of [3] for each partition to compute the tree interpolant. Mixed inequalities are replaced by their projections on the nodes of the tree. This sums up all inequalities that occur in the A part of the interpolation problem, i. e., the subtree of the node v for which we compute the interpolant. Again we need to ensure that every literal is assigned to a unique node.

Conjecture 1. *The computed partial interpolant will fulfil the invariants of partial tree interpolants.*

5.5 Extended Interpolation Rules

The interpolation rules for tree interpolants are a straight-forward extension of the interpolation rules for binary interpolation presented in Section 3. For every resolution step in the resolution proof of unsatisfiability, we compute for every node $v \in V$ an interpolant. If $\ell \vee C_1$ has partial tree interpolant I_1 and $\neg\ell \vee C_2$ has partial tree interpolant I_2, we can compute a tree interpolant I_3 for $C_1 \vee C_2$ by node-wise computing partial interpolants. For $v \in V$, $I_3(v)$ is a combination of the pivot literal ℓ and the partial interpolants $I_1(v)$ and $I_2(v)$ of the antecedents of the resolution step.

The computation of $I_3(v)$ from $I_1(v)$ and $I_2(v)$ is done by the same algorithm as for binary interpolation [3]. If the literal ℓ is not mixed in v we use the definition from Section 5.1. If the literal is mixed in v, we use a special interpolation rule $\mathsf{mixcomb}(\ell, I_1(v), I_2(v))$ that takes two partial interpolants (i. e., the labels of the corresponding partial tree interpolants at the node v) and computes a new partial interpolant for the resolvent. The partial interpolants $I_1(v)$ and $I_2(v)$ may contain the auxiliary variables introduced by ℓ in v, which may not occur in the resulting partial interpolant.

$$\frac{\ell \vee C_1 : I_1 \qquad \neg\ell \vee C_2 : I_2}{C_1 \vee C_2 : I_3}, \text{ where } I_3(v) = \begin{cases} I_1(v) \vee I_2(v) & \text{if } \ell \lfloor v' = \top \text{ for all } v' \notin st(v) \\ I_1(v) \wedge I_2(v) & \text{if } \ell \lfloor v' = \top \text{ for all } v' \in st(v) \\ \mathsf{mixcomb}(\ell, I_1(v), I_2(v)) & \text{if } v \in \mathsf{mixed}(\ell) \\ (I_1(v) \vee \ell) \wedge & \\ (I_2(v) \vee \neg\ell) & \text{otherwise} \end{cases}$$

For a mixed equality, our syntactic restriction of interpolants guarantees that the first interpolant is of the form $I_1[EQ(X,s_1)]\dots[EQ(X,s_n)]$, i. e., all occurrences of the auxiliary variable X are in a literal $EQ(X,s_i)$ occurring positively in the formula. The second interpolant $I_2(x)$ has no syntactic restrictions. The combined interpolant is obtained by replacing each literal $EQ(X,s_i)$ in I_1 by $I_2(s_i)$, which is expressed as

$$\mathsf{mixcomb}(a = b, I_1[EQ(X,s_1)]\dots[EQ(X,s_n)], I_2(x)) :\equiv I_1[I_2(s_1)]\dots[I_2(s_n)].$$

For an inequality, a partial interpolant of $\ell \vee C_1$ has the shape

$$I_1[LA(c_{11}x_1 + s_{11}, k_{11}, F_{11})]\dots[LA(c_{1n}x_1 + s_{1n}, k_{1n}, F_{1n})].$$

We use $LA_{1i}(x_1)$ as short-hand for $LA(c_{1i}x_1 + s_{1i}, k_{1i}, F_{1i})$ and write the formula above as $I_1[LA_{1i}(x_1)]$. Similarly, we write $I_2[LA_{2j}(x_2)]$ for a partial interpolant $I_2[LA_{21}(x_2)]\dots[LA_{2m}(x_2)]$ of $\neg\ell \vee C_2$. For each pair LA_{1i} and LA_{2j} we compute a formula $LA_{3ij} := LA(s_{3ij}, k_{3ij}, F_{3ij})$ such that $LA_{3ij} \iff \exists x.\, LA_{1i}(x) \wedge LA_{2j}(-x)$. This is possible since the value of an $LA(s,k,F)$ is only unknown for $-k \le s(x) \le 0$. In the integer case we can enumerate all possible values of x in this interval:

$$\begin{aligned} s_{3ij} &:= c_{2j}s_{1i} + c_{1i}s_{2j} \\ k_{3ij} &:= c_{2j}k_{1i} + c_{1i}k_{2j} + c_{1i}c_{2j} \\ F_{3ij} &:\equiv \bigvee_{i=0}^{\left\lceil \frac{k_{1i}+1}{c_{1i}} \right\rceil} F_{1i}\left(\left\lfloor \frac{-s_{1i}}{c_{1i}} \right\rfloor - i\right) \wedge F_{2j}\left(i - \left\lfloor \frac{-s_{1i}}{c_{1i}} \right\rfloor\right) \end{aligned}$$

In the real case the constant k is guaranteed to be either $-\varepsilon$ or 0. Thus, there is at most one interesting value. We use the following definitions.

$$\begin{aligned} s_{3ij} &:= c_{2j}s_{1i} + c_{1i}s_{2j} \\ k_{3ij} &:= \begin{cases} k_{2j} & \text{if } k_{1i} = -\varepsilon \\ 0 & \text{if } k_{1i} = 0 \end{cases} \\ F_{3ij} &:\equiv \begin{cases} F_{2j}\left(\frac{s_{1i}}{c_{1i}}\right) & \text{if } k_{1i} = -\varepsilon \\ s_{3ij} < 0 \vee \left(F_{1i}\left(-\frac{s_{1i}}{c_{1i}}\right) \wedge F_{2j}\left(\frac{s_{1i}}{c_{1i}}\right)\right) & \text{if } k_{1i} = 0 \end{cases} \end{aligned}$$

The partial interpolant of the resolvent $C_1 \vee C_2$ in the mixed case can be expressed as

$$\mathsf{mixcomb}(t \le c, I_1[LA_{1i}], I_2[LA_{2j}]) :\equiv I_1[I_2[LA_{311}]\dots[LA_{31m}]]\dots[I_2[LA_{3n1}]\dots[LA_{3nm}]].$$

The interpolation rules are exactly the same as for the binary case. The only differences between the definition above and the one in [3] is a small simplification of the rule for linear arithmetic that would also be applicable to the binary interpolation case and the exact definition of $EQ(X,s)$, which is only needed for the correctness proof and not used in the interpolation algorithm.

Conjecture 2. *The extended interpolation rules with the definition of* mixcomb *for the equality and inequality literals given above is correct. Thus, if I_1 and I_2 are partial tree interpolants for the clauses $\ell \vee C_1$ and $\neg\ell \vee C_2$ respectively, then I_3 is a partial tree interpolant for the clause $C_1 \vee C_2$.*

6 Conclusion

We presented our ongoing work to extend proof tree preserving interpolation to tree interpolation, a generalisation of sequence interpolation. The key ingredients are the extension of the projection functions to mixed literals by introducing auxiliary variables and predicates, a syntactic restriction of the occurrence of these auxiliary symbols in the partial tree interpolants, and a set of rules to compute partial tree interpolants for resolution steps on mixed literals. The major difficulty with this technique lies in the correctness proofs that are still part of ongoing work. To our knowledge, this is the first paper to focus on the problem of extracting tree interpolants from resolution proofs produced by state-of-the-art SMT solvers. The interpolation technique is implemented in the interpolating SMT solver SMTInterpol [2].

References

[1] iZ3 documentation. `http://research.microsoft.com/en-us/um/redmond/projects/z3/old/iz3documentation.html`. Accessed: 2012-10-05.

[2] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN'12*, pages 248–254. Springer, 2012.

[3] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Proof tree preserving interpolation. In *TACAS'13*, pages 124–138. Springer, 2013.

[4] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Proof tree preserving interpolation. Reports of SFB/TR 14 AVACS 89, SFB/TR 14 AVACS, February 2013. ISSN: 1860-9821, `http://www.avacs.org`.

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

[6] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS'10*, pages 271–274. Springer, 2010.

[7] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV'06*, pages 81–94. Springer, 2006.

[8] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *POPL'13*, pages 129–142. ACM, 2013.

[9] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstic, and Cesare Tinelli. Ground interpolation for the theory of equality. In *TACAS'09*, pages 413–427. Springer, 2009.

[10] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS'11*, pages 188–203. Springer, 2011.

[11] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *POPL'10*, pages 471–482. ACM, 2010.

[12] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL'04*, pages 232–244. Springer, 2004.

[13] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV'03*, pages 1–13. Springer, 2003.

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

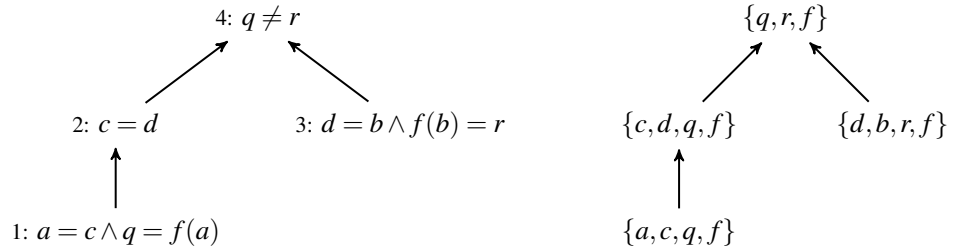
[15] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, pages 123–136. Springer, 2006.

[16] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997.

[17] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *FMCAD'12*, pages 114–121. IEEE, 2012.

A Example

We present our algorithm on the tree interpolation problem from Figure 1(a) consisting of four nodes. In the figure we draw the arrows from children to parent. Each node contains the labelling depicted by the corresponding figure, i. e., either the input labelling, or a (partial) tree interpolant. The numbers in Figure 1(a) will be used to identify the individual nodes. The (extended) symbol set $symb(v)$ for each node is given in Figure 1(b).



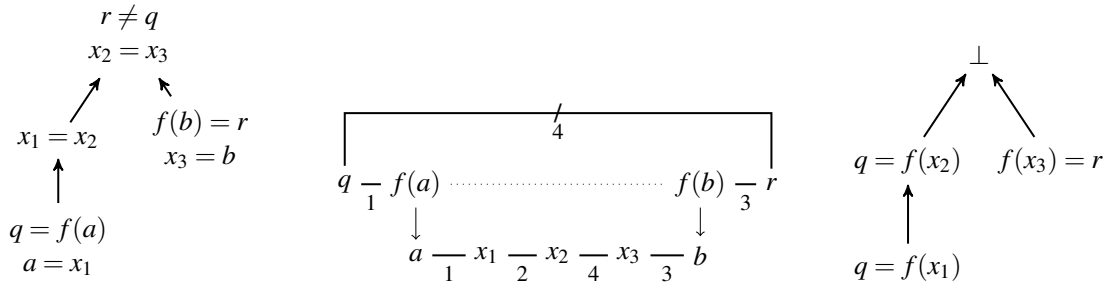
(a) Interpolation Problem (b) Symbol set $symb(v)$ for each node

Figure 1: Tree interpolation problem and symbol set used throughout this section.

A.1 Leaf Interpolation for Equality Theory

We assume the solver for the theory of equalities produces the literal $a = b$ and detects the conflicts $q = f(a) \wedge a = b \wedge f(b) = r \wedge q \neq r$ and $a = c \wedge c = d \wedge d = b \wedge a \neq b$. We show how to derive partial tree interpolants for these conflicts from the corresponding congruence graphs.

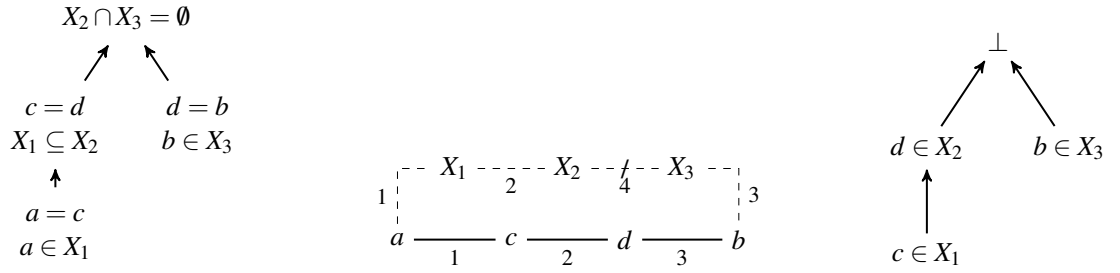
Figure 2(a) gives the projection of the conflict onto the individual nodes. For the literal $a = b$, which is mixed in nodes 1, 2, and 3, we introduce the auxiliary variables x_1, x_2, x_3. In Figure 2(b) we show the corresponding Congruence Closure graph which we already extended by the auxiliary variables. The horizontal edges denote equalities and are labelled by the node that contains the equality literal. The vertical arrows denote function application and the dotted edge is a derived congruence. The inequality is depicted by the top edge. The interpolation algorithm summarises for each node the equalities occurring in the corresponding subtree. If the equality chain crosses a function application, e. g., $q = f(a)$ and $a = x_1$, we need to lift the end-point yielding $q = f(x_1)$. For Node 4, the (dis-)equality chain spans the whole cycle and is summarised by $\bot$. The resulting partial tree interpolant is given in Figure 2(c).



(a) Interpolation Problem (b) Corresponding Congruence Graph (c) Partial Tree Interpolant

Figure 2: Interpolating the conflict $q = f(a) \wedge a = b \wedge f(b) = r \wedge q \neq r$.

The second conflict $a = c \wedge c = d \wedge d = b \wedge a \neq b$ contains the negated literal $a \neq b$, for which we introduce set-valued auxiliary variables X_1, X_2, X_3. Figure 3(a) gives the projection of the conflict onto the individual nodes. The corresponding Congruence Closure graph is given in Figure 3(b). Here, we split the disequality into the literals $a \in X_1$, $X_1 \subseteq X_2$, $X_2 \cap X_3 = \emptyset$ and $b \in X_3$. These literals are sketched in the figure by dashed edges. The interpolants are computed as usual by summarising the edges belonging to one subtree. Here, $d = c$, $c = a$, $a \in X_1$ and $X_1 \subseteq X_2$ is summarised by $d \in X_2$. The literal $X_2 \cap X_3 = \emptyset$ occurs only in the node $\mathsf{mixedparent}(a \neq b)$ and the literal is not mixed in that node. Thus, we never need to build a summary including this edge. The resulting partial tree interpolant is given in Figure 3(c).



(a) Interpolation Problem (b) Corresponding Congruence Graph (c) Partial Tree Interpolant

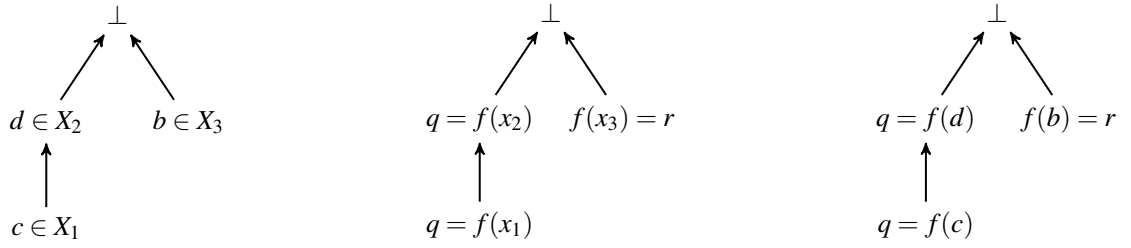
Figure 3: Interpolating the conflict $a = c \wedge c = d \wedge d = b \wedge a \neq b$.

A.2 Interpolation Rule for Resolution Proof

The theory lemma clauses corresponding to the conflicts in the previous section are combined by the resolution rule to a new clause.

$$\frac{a = b \vee a \neq c \vee c \neq d \vee d \neq b \qquad a \neq b \vee q \neq f(a) \vee f(b) \neq r \vee q = r}{a \neq c \vee c \neq d \vee d \neq b \vee q \neq f(a) \vee f(b) \neq r \vee q = r}$$

Since the pivot is mixed in nodes 1, 2, and 3, we need to apply $\mathsf{mixcomb}$ to combine the partial interpolants of these nodes. For equality literals the interpolants have the shape $I_1[s \in X]$ and $I_2(x)$ (in our case $I_1[F] \equiv F$). The resulting interpolant for each node is computed as $I_1[I_2(s)]$, which basically means that we just have to replace in the second interpolant x_i by the term s, where $s \in X_i$ is the first interpolant. The result is shown in Figure 4.



(a) Partial Tree Interpolant of $a = b \vee \ldots$ (b) Partial Tree Interpolant of $a \neq b \vee \ldots$ (c) Partial Tree Interpolant of resolvent

Figure 4: Applying the interpolation rule for resolution.

B Proofs, Proof Sketches, and Proof Ideas

B.1 Proof of Lemma 1

Proof. Assume we added a to $symb(w)$ for some $w \in st(v)$. For w there are $v_1, v_2 \in V$ with $a \in symb(L(v_i))$. At least one of them is a descendent of w, hence there is another node in $v_i \in st(v)$ with $a \in symb(L(v_i))$. On the other hand, if we added a to $symb(w')$ for some $w' \notin st(v)$. Then again there are $v_1, v_2 \in V$ with $a \in symb(L(v_i))$. If both of these nodes would lie in $st(v)$, then v would be a common ancestor of v_1 and v_2 which contradicts $w \notin st(v)$. Hence there is a node $v_i \notin st(v)$ with $a \in symb(L(v_i))$. □

B.2 Proof for Theorem 1

Proof. Given a sequence interpolation problem $F_1, \ldots, F_n$, construct a tree as follows. Let $T = (V, E)$ with $V = \{v_0, v_1, \ldots, v_n\}$, $E = \{(v_{i-1}, v_i) \mid 1 \le i \le n\}$, $L(v_i) = F_i$ for $1 \le i \le n$, and $L(v_0) = \top$. Given a solution I to the tree interpolation problem $T = (V, E)$ and L, a solution to the sequence interpolation problem is $I_i := I(v_i)$. □

B.3 Proof Sketch of Lemma 2

Proof Sketch. Fixing a node v_p we have to show $\bigwedge_{(v_c, v_p) \in E} I_3(v_c) \wedge L(v_p) \wedge (\neg C_1 \wedge \neg C_2) \lfloor v_p \models I_3(v_p)$ and can assume that a similar condition holds for I_1 and I_2. We distinguish three cases.

Case 1. There is an edge $(v_c, v_p) \in E$ with $\ell \lfloor v = \top$ for all $v \notin st(v_c)$. Then $I_3(v_c) = I_1(v_c) \vee I_2(v_c)$ and $I_3(v_p) = I_1(v_p) \vee I_2(v_p)$. Also for all other edges $(v_c', v_p) \in E$, $\ell \lfloor v = \top$ for all $v \in st(v_c')$, hence $I_3(v_c') = I_1(v_c') \wedge I_2(v_c')$.

Assume $\bigwedge_{(v_c, v_p) \in E} I_3(v_c)$ holds, then $\bigwedge_{(v_c, v_p) \in E} I_1(v_c)$ or $\bigwedge_{(v_c, v_p) \in E} I_2(v_c)$ hold. Using the induction hypothesis for I_1 and I_2 we derive that $I_1(v_p)$ or $I_2(v_p)$ hold (note that $\ell \lfloor v_p = \top$ since $v_p \notin st(v_c)$). Then $I_3(v_p)$ holds.

Case 2. Assume $\ell \lfloor v = \top$ for all $v \in st(v_p)$. Then $I_3(v_p) = I_1(v_p) \wedge I_2(v_p)$ and $I_3(v_c) = I_1(v_c) \wedge I_2(v_c)$ for all $(v_c, v_p) \in E$. From $\bigwedge I_3(v_c)$ we conclude that $\bigwedge_{(v_c, v_p) \in E} I_1(v_c)$ and $\bigwedge_{(v_c, v_p) \in E} I_2(v_c)$ hold. Using the induction hypothesis (again $\ell \lfloor v_p = \top$) we derive $I_1(v_p)$ and $I_2(v_p)$ thus $I_3(v_p)$.

Case 3. Otherwise $I_3(v_c) \implies (I_1(v_c) \vee \ell) \wedge (I_2(v_c) \vee \neg \ell)$ for all $(v_c, v_p) \in E$ since we are not in Case 1. With the induction hypothesis and $\ell \implies \ell \lfloor v_p$ we derive from $\bigwedge I_3(v_c)$ that $(I_1(v_p) \vee \ell) \wedge (I_2(v_p) \vee \neg \ell)$ holds. Since we are not in Case 2, this implies $I_3(v_p)$. □

B.4 Proof of Lemma 3

Proof. We first show

$$\mathsf{mixed}(\ell) \subseteq \{v \in V \mid \exists i.\ 1 \le i \le n.\, lca(a_i) \in st(v) \text{ and } \mathsf{mixedparent}(\ell) \text{ is a proper ancestor of } v\}.$$

Let $v \in \mathsf{mixed}(\ell)$. Since ℓ is mixed in v, there is at least one symbol a_i that occurs only inside the subtree of v. Hence, $lca(a_i) \in st(v)$ for some i. Moreover, the $\mathsf{mixedparent}(\ell)$ is an ancestor of the parent of v, hence it is a proper ancestor of v.

For the other direction

$$\mathsf{mixed}(\ell) \supseteq \{v \in V \mid \exists i.\ 1 \le i \le n.\, lca(a_i) \in st(v) \text{ and } \mathsf{mixedparent}(\ell) \text{ is a proper ancestor of } v\}$$

take a node v, from the set on the right-hand side. Then there is an i such that $lca(a_i) \in st(v)$, i. e., a_i occurs only inside the subtree of v. It remains to show that there is another symbol that occurs only outside the subtree of v. There must be a node $w \in \mathsf{mixed}(\ell)$ such that v is not a proper ancestor of w (otherwise v would be an ancestor of $\mathsf{mixedparent}(\ell)$).

Case 1: w is an ancestor of v. There is a symbol a_j that only occurs outside of the subtree of w. Thus, this symbol occurs only outside of the subtree of v, so ℓ is mixed in v.

Case 2: w and v have disjoint subtrees. There is a symbol a_j that only occurs inside of w. Thus, this symbol occurs only outside of the subtree of v, so ℓ is mixed in v. □

B.5 Projection Function is Correct

Lemma 4 (Correctness of the Projection Function)**.** *The projection function defined in Section 5.3 is correct (in the sense of Definition 2.*

Proof Sketch. For $\ell \equiv a_1 = a_2$, show by induction on v_p that

$$\exists \{x_j | v_j \in (st(v_p) \setminus \{v_p\}) \cap \mathsf{mixed}(\ell)\}. \bigwedge_{v_j \in st(v_p)} \ell \lfloor v_j$$
$$\iff \begin{cases} \top & \text{if } v_j \notin \mathsf{mixed}(\ell) \text{ for all } v_j \in st(v_p), \\ a_i = x_p & \text{if } v_p \in \mathsf{mixed}(\ell),\ lca(a_i) \in st(v_p), \\ a_1 = a_2 & \text{if } \mathsf{mixedparent}(\ell) \in st(v_p). \end{cases}$$

For $\ell \equiv a_1 \neq a_2$, show by induction on v_p that

$$\exists \{X_j | v_j \in (st(v_p) \setminus \{v_p\}) \cap \mathsf{mixed}(\ell)\}. \bigwedge_{v_j \in st(v_p)} \ell \lfloor v_j$$
$$\iff \begin{cases} \top & \text{if } v_j \notin \mathsf{mixed}(\ell) \text{ for all } v_j \in st(v_p), \\ a_i \in X_p & \text{if } v_p \in \mathsf{mixed}(\ell),\ lca(a_i) \in st(v_p), \\ a_1 \neq a_2 & \text{if } \mathsf{mixedparent}(\ell) \in st(v_p). \end{cases}$$

For $\ell \equiv \sum c_i a_i \leq c$, show by induction on v_p that

$$\exists \{x_j | v_j \in (st(v_p) \setminus \{v_p\}) \cap \mathsf{mixed}(\ell)\}. \bigwedge_{v_j \in st(v_p)} \ell \lfloor v_j$$
$$\iff \begin{cases} \top & \text{if } v_j \notin \mathsf{mixed}(\ell) \text{ for all } v_j \in st(v_p), \\ \sum_{lca(a_i) \in st(v_p)} c_i a_i \leq x_p & \text{if } v_p \in \mathsf{mixed}(\ell), \\ \sum_{lca(a_i) \in st(v_p)} c_i a_i \leq c & \text{if } \mathsf{mixedparent}(\ell) \in st(v_p). \end{cases}$$

□

B.6 Proof Idea for Leaf Interpolation (Conjecture 1)

Proof Idea. For equality conflicts that does not contain a mixed *disequality* the A paths in the interpolant of the parent node are the summary of all literals occurring in the subtree. It is thus the summary of the literals occurring in the parent node and the equalities in the interpolants of the interpolants of the child nodes. Thus it follows from them.

For mixed disequality we also summarise the literals $s = a$, $a \in X_1$, and $X_1 \subseteq X_2$ to $s \in X_2$. When moving to a mixed parent of a mixed child it is not too difficult to see that when the equality chain is extended at the front by $s' = s$ (e. g. in another child node) and the subset chain is extended by $X_2 \subseteq X_3$, that the summary $s' \in X_3$ of the parent node can be derived from these formulae. Finally, when moving to the node $\mathsf{mixedparent}(a_1 \neq a_2)$ we summarise the child interpolants $a_1 \in X_{c_1}$ and $a_2 \in X_{c_2}$ together with $X_{c_1} \cap X_{c_2} = \emptyset$ to $a_1 \neq a_2$.

For inequality conflicts, the interpolant of the parent node is the sum of all inequalities occurring in the subtree (multiplied with their respective Farkas coefficient). This is just the sum of the interpolants of the child nodes and the inequalities occurring in the parent node. □

B.7 Proof Idea for Extended Interpolation Rule (Conjecture 2)

Proof Idea. If the pivot literal is not mixed, the correctness follows from Lemma 2.

For mixed equality literals we need to do a case split over the four cases of the projection function (two children are mixed, one child is mixed, both parent and one child is mixed, and only the parent is mixed). The proof uses the induction hypothesis for I_1 and I_2. In the induction hypothesis for I_1 the variables X_c and X_p may occur. The trick is now to instantiate the variable X_c by the set $\{x | I_2^c(x)\}$ and likewise for X_p. The remaining proof is tedious but straight-forward.

For a mixed inequality literal we need the fact that $LA_{3ij} \iff \exists x. LA_{1i}(x) \wedge LA_{2j}(-x)$ holds, which we proved in [4]. The technical difficulty of the proof lies in finding a common x that works for all terms $LA_{1i}(x)$ and $LA_{2j}(-x)$ that occur in the interpolants I_1 and I_2. This can be achieved by choosing the minimum x for i and the maximum x for j using the monotonicity of LA. Then one can show that if for a mixed child $I_1[I_2[LA_{3ij}]]$ holds, there is an x such that $I_1[LA_{1i}(x)]$ and $I_2[LA_{2j}(-x)]$ holds (except for one special case where $I_2 \equiv \bot$ that needs to be handled separately). Likewise for a mixed parent $I_1[LA_{1i}(x)]$ and $I_2[LA_{2j}(-x)]$ imply $I_1[I_2[LA_{3ij}]]$. Now the remaining proof is straight-forward using the induction hypothesis. □

Reducing the Complexity of Quantified Formulas via Variable Elimination

Aboubakr Achraf El Ghazi, Mattias Ulbrich, Mana Taghdiri and Mihai Herda

Karlsruhe Institute of Technology, Germany
{elghazi, ulbrich, mana.taghdiri}@kit.edu, mihai.herda@student.kit.edu

Abstract

We present a general simplification of quantified SMT formulas using variable elimination. The simplification is based on an analysis of the ground terms occurring as arguments in function applications. We use this information to generate a system of set constraints, which is then solved to compute a set of *sufficient ground terms* for each variable. Universally quantified variables with a finite set of sufficient ground terms can be eliminated by instantiating them with the computed ground terms. The resulting SMT formula contains potentially fewer quantifiers and thus is potentially easier to solve. We describe how a satisfying model of the resulting formula can be modified to satisfy the original formula. Our experiments show that in many cases, this simplification considerably improves the solving time, and our evaluations using Z3 [9] and CVC4 [1] indicate that the idea is not specific to a particular solver, but can be applied in general.

1 Introduction

Determining the satisfiability of first-order formulas with respect to theories is of central importance for system specification and verification. Current Satisfiability Modulo Theories (SMT) solvers have made significant progress in handling this problem efficiently. SMT solvers such as CVC4 [1], Yices1 [5], and Z3 [9] successfully address formulas containing quantifiers. They solve quantified formulas using heuristic quantifier instantiation based on the E-matching instantiation algorithm which was first introduced by Simplify [4]. Although E-matching, because of its heuristic nature, is not complete, not even refutationally, it is best suited for integration into the DPLL(T) framework. Some techniques (e.g. [11, 7]) have extended E-matching in order to make it complete for some fragments of first-order logic.

In spite of all the advances, the presence of quantifiers still poses a challenge to the solvers. In this paper, we propose a simplification of quantified SMT formulas that can be applied as a pre-process before calling an SMT solver. Given a (skolemized) SMT formula A, our simplification returns an equisatisfiable SMT formula A' with potentially fewer universally quantified variables. Our simplification approach is syntactic in the sense that it extracts a set of set-valued constraints from the structure of A whose solution is a set of *sufficient ground terms* for every variable. Those variables whose sets of sufficient ground terms are finite can be eliminated by instantiating them with the computed ground terms. If the resulting formula A' is unsatisfiable, A is guaranteed to be unsatisfiable too. However, if A' has a model, it is not necessarily a model of A. We describe how any model of A' can be modified into a model for A without any significant overhead. This requires a special treatment of the interpreted functions. Our simplification procedure can also be applied if the logic of the input formula is not decidable; it can still reduce the number of quantifiers, thus simplifying the proof obligation.

Although our elimination process reduces the number of quantifiers, it may increase the number of occurrences of the remaining quantified variables (if any) (Appendix A gives an example). Depending on the complexity of the involved terms, this may introduce additional

overhead for the solver. Therefore, in order to apply our simplification as a general preprocessing step, it is important to balance the number of eliminated variables and the number of newly introduced variable occurrences. We define a metric that aims for estimating the cost of variable elimination, and allow the user to provide a threshold for the estimated cost.

We have applied our simplification approach to 201 benchmarks from the SMT competition 2012 using CVC4 and Z3. The results indicate that in many cases, this simplification significantly improves the solving time, especially when a cost threshold is applied.

2 Background

This section provides a background on the first-order logic (FOL) (see [12] for more details). *Terms* are constructed from variables in Var, predicate symbols in P and function symbols in F[1]. Predicate and function symbols are given an arity by $\alpha : F \cup P \to \mathbb{N}$. Function symbols with arity 0 are called constants and are denoted as $Con \subseteq F$. The set $Term$ of terms and the set For of formulas are defined inductively as usual. Terms without variables are called *ground terms* and denoted as $Gr \subseteq Term$. The set $Gr(t)$ denotes all the ground terms occurring as subterms in a term t. We write $t[x_{1:n}]$ to denote that the variables $x_1, \ldots, x_n$ (for short $x_{1:n}$) occur in a term t. For an expression $t \in Term \cup For$, a variable x and a ground term gt, the expression $t[gt/x]$ substitutes gt for all the occurrences of x in t. We apply substitutions (aka. instantiations) also to finite sets S of ground terms as $t[S/x] := \{t[gt/x] \mid gt \in S\}$. The Herbrand universe $\mathcal{H}(A)$ of a formula A is the set of all ground terms built from A. That is, all constants occurring in A, are in $\mathcal{H}(A)$, and for each function f occurring in A and $gt_1, \ldots, gt_{\alpha(f)} \in \mathcal{H}(A)$, $f(gt_1, \ldots, gt_{\alpha(f)}) \in \mathcal{H}(A)$.

A literal is an atomic formula or a negated atomic formula. A clause is a disjunction of literals. A formula is in *clause normal form* (CNF) if it is a conjunction $(C_1 \wedge \ldots \wedge C_n)$ of clauses where all C_i are quantifier-free and all variables are implicitly universally quantified. We assume, unless stated otherwise, that all considered formulas are in CNF and all variables are unique. When required, we refer to clauses and CNFs as sets of literals and clauses, respectively.

A semantical *structure* (also called a *model*) $\mathcal{M}$ is a tuple $(|M|, M)$, with a non-empty universe $|M|$, and a mapping M that defines an *interpretation* for every symbol in $F \cup P$, i.e. for $f \in F$, $M(f) : |M|^{\alpha(f)} \to |M|$, and for $p \in P$, $M(p) \subseteq |M|^{\alpha(p)}$. Variables get their values from a variable assignment function $\beta : Var \to |M|$. The interpretation $(M, \beta)(t)$ of a term t is defined inductively, and the interpretation of a set of terms S is defined as $(M, \beta)(S) = \{(M, \beta)(s) \mid s \in S\}$. For a formula $A \in For$, we use $\mathcal{M} \models A$ if $\mathcal{M}$ is a satisfying model (or, for short, a model) of A, i.e. A is true in $\mathcal{M}$. We use $\models A$ if A is universally valid.

A *theory* $\mathcal{T}$ is a deductively closed set of formulas. A $\mathcal{T}$-model $\mathcal{M}$ is a model that satisfies all the formulas in $\mathcal{T}$. A formula $A \in For$ is satisfiable modulo theory $\mathcal{T}$ if there exists a $\mathcal{T}$-model with $\mathcal{M} \models A$, for short $\mathcal{M} \models_{\mathcal{T}} A$. The function symbols that have their semantics (partially) fixed by $\mathcal{T}$ are called *interpreted* and all others are *uninterpreted*. If a term contains an interpreted function which is applied to a variable, we call it an *interpreted term*, otherwise, an *uninterpreted term*. We denote variables by $x, y, \ldots$; constants by $a, b, \ldots$; ground terms by gt_i; uninterpreted functions by $f, g, \ldots$; interpreted functions by op_i; predicates by $p, q, \ldots$; terms by $s, t, \ldots$; formulas by $A, B, \ldots$; values by v_i; and the considered SMT theory by $\mathcal{T}$.

[1] We distinguish between functions and predicates only when needed.

(1) $c_1 \neq c_2$ (2) $\forall x \mid f(x) = f(c_1)$ (3) $\exists z \mid \forall y \mid \neg p(y,z) \vee f(y) = c_2$ (4) $\exists z \mid f(z) = c_1$	(1) $c_1 \neq c_2$ (2) $\forall x \mid f(x) = f(c_1)$ (3) $\forall y \mid \neg p(y,c_3) \vee f(y) = c_2$ (4) $f(c_4) = c_1$	(1) $c_1 \neq c_2$ (2) $f(c_1) = f(c_1)$ (2) $f(c_4) = f(c_1)$ (3) $\neg p(c_1,c_3) \vee f(c_1) = c_2$ (3) $\neg p(c_4,c_3) \vee f(c_4) = c_2$ (4) $f(c_4) = c_1$
(a)	(b)	(c)

$M(c_1) = 1, M(c_2) = 2, M(c_3) = 3, M(c_4) = 4$

$$M(f)(v) = \begin{cases} 1 & \textit{if } v = 1 \\ 1 & \textit{if } v = 4 \\ \textit{any value} & \textit{else} \end{cases} \qquad M(p)(v,3) = \begin{cases} \textbf{false} & \textit{if } v = 1 \\ \textbf{false} & \textit{if } v = 4 \\ \textit{any value} & \textit{else} \end{cases}$$

(d)

$M^\pi(c_1) = M(c_1) = 1, M^\pi(c_2) = M(c_2) = 2, M^\pi(c_3) = M(c_3) = 3, M^\pi(c_4) = M(c_4) = 4$

$$M^\pi(f)(v) = \begin{cases} M(f)(v) & \textit{if } v \in \{1,4\} \\ M(f)(M(c_1)) & \textit{else} \end{cases} = 1 \quad \textit{for all } v$$

$$M^\pi(g)(v,c_3) = \begin{cases} M(g)(v, M(c_3)) & \textit{if } v \in \{1,4\} \\ M(g)(M(c_1), M(c_3)) & \textit{else} \end{cases} = \textbf{false} \quad \textit{for all } v$$

(e)

Figure 1: Example. (a) original SMT formula, (b) CNF formula, (c) instantiated formula, (d) a model for the instantiated formula, and (e) a model for the original formula.

3 Example

Figure 1(a) shows an SMT formula (as a set of implicitly conjoined subformulas) in which c_1 and c_2 represent constants, f is a unary function, and p is a binary predicate. Figure 1(b) shows the same formula after conversion to CNF: constants c_3 and c_4 denote the skolems for the formulas (3) and (4), respectively. Instead of solving the original formula (denoted by A), we produce an *instantiated formula* A^{inst} in which the x and y variables are instantiated with certain ground terms. A^{inst} is given in Figure 1(c) where the numbers correspond to the lines in the CNF (and original) formula. Formula A^{inst} has fewer quantifiers than A (in fact, it has zero quantifiers), and thus is easier to solve. We use $vGT(x)$ to represent the set of ground terms that is used to instantiate a variable x. Variable x (in Formula 2) refers to the first argument of f, and thus we instantiate it with all the ground terms that occur in that position, namely $\{c_1, c_4\}$. We call this the set of ground terms of f for argument position 1, and denote it by $fGT(f,1)$. Variable y (in Formula 3), on the other hand, refers to both the first argument of p and the first argument of f. Therefore, $vGT(y) = fGT(p,1) \cup fGT(f,1)$. In order to guarantee equisatisfiability of A^{inst} and A, if two functions are applied to the same variable, they should be instantiated with the ground terms of both functions (see Section 4). Therefore, in this example, $fGT(p,1) = fGT(f,1) = \{c_1, c_4\}$ although p is not directly applied to any constants.

The instantiated formula is an implication of the original formula. Hence, if A^{inst} is unsatisfiable, A is also unsatisfiable. However, not every model of A^{inst} satisfies A. But the instantiation was chosen in such a way that we can modify the models of A^{inst} to satisfy A. Figure 1(d) gives a sample model $\mathcal{M}$ for A^{inst} which does not satisfy A. Since in A^{inst}, f is

only applied to c_1 and c_4, and p only to (c_1, c_3) and (c_4, c_3), $\mathcal{M}$ may assign arbitrary values to f and p applied to other arguments. Although these values do not affect satisfiability of A^{inst}, they affect satisfiability of A. Therefore, we modify $\mathcal{M}$ to a model $\mathcal{M}^\pi$ by defining acceptable values for the function applications that do not occur in A^{inst}. Figure 1(e) gives the modified model $\mathcal{M}^\pi$ that our algorithm constructs. It is easy to show that this model satisfies A.

The basic idea of modifying a model is to fix the values of the function applications that do not occur in A^{inst} to some arbitrary value of a function application that does occur in A^{inst}. This works well for this example as f and g are uninterpreted symbols and thus their interpretations are not restricted beyond the input formula. Were they interpreted symbols, this would be different. As an example, assume that p is the interpreted operator "$\leq$". In this case, the original formula $A_\leq$ becomes unsatisfiable[2], but its instantiation $A_\leq^{inst}$ stays satisfiable[3]. To guarantee the equisatisfiability in the presence of interpreted literals, we require the ground term sets to contain some terms that make the interpreted literals false. This makes the solver explore the cases where clauses become satisfiable regardless of the interpreted literals. In this example, the interpreted literal $\neg(y \leq c_3)$ becomes false if y is instantiated with the ground term $c_3 - 1$. Instantiating $A_\leq$ with the ground terms $\{c_1, c_4, c_3 - 1\}$ reveals the unsatisfiability.

4 Sufficient Ground Term Sets

Definition 1. *Given a variable x in an SMT formula A (in CNF), a set of ground terms $S \subseteq \mathcal{H}(A)$ is* sufficient *for x w.r.t a theory $\mathcal{T}$ if A and $A[S/x]$ are equisatisfiable modulo $\mathcal{T}$.*

A variable x in a formula A can have more than one sufficient set of ground terms. $\mathcal{H}(A)$ is always a sufficient set of ground terms as a result of the Gödel-Herbrand-Skolem theorem which states that a formula A in Skolem Normal Form (SNF) is satisfiable iff $A[\mathcal{H}(A)/x]$ is satisfiable [12]. But $\mathcal{H}(A)$ is usually infinite, and our goal is to determine whether a *finite* set of sufficient ground terms exists, and to compute it if one exists. This computation is done by generating and solving a system of set constraints over sets of ground terms.

Figure 2 presents our (syntactic) rules to generate the set constraints for a formula A in CNF. The notation $t \,\dot{\in}\, C$ denotes that a term t occurs as a subterm of a clause C. We use $\mathcal{S}_A$ to denote the set constraints system that results from applying these rules exhaustively to all the clauses of A. The constraints range over the sets $vGT(x) \subseteq Gr$ for all variables x in A. These sets denote the relevant instantiations for the respective variables. Auxiliary sets $fGT(f, i) \subseteq Gr$ are introduced to denote the set of relevant ground terms for an uninterpreted function $f \in F$ at an argument position $i \in \mathbb{N}$. We assume that the theory of integers is part of the considered $\mathcal{T}$, and that integers are included in the universe of every $\mathcal{T}$-model $\mathcal{M}$, i.e. $\mathbb{Z} \subseteq |M|$. The integer operators $<, \leq, +, -, \geq, >$ are fixed with their obvious meanings.

Rule R_0 of Figure 2 guarantees that the set of relevant ground terms is not empty for any variable in A. Rule R_1 establishes a relationship between sets of ground terms for variables and function arguments. Rule R_2 ensures that the ground terms that occur as arguments of a function f are added to the corresponding ground term set of f. Rule R_3 states that if a term $t[x_{1:n}]$ with variables $x_{1:n}$ occurs as the i-th argument of f, then all the instantiations of t with the respective sets $vGT(x_i)$ must be in $fGT(f, i)$. Rule R_4 states that our approach does not currently handle the case where a variable x occurs as an argument of an unsupported

[2] (2) and (4) imply $f(c_1) = c_1$. $y \leq z$ holds for some pair of integers, thus (3) implies $f(y) = c_2$ for some y. But $f(y) = f(c_1)$ by (2) and so $f(c_1) = c_2 = c_1$. This contradicts (1).

[3] A model is $M'(c_1) = 1, M'(c_2) = 2, M'(c_3) = 0, M'(c_4) = 4, M'(f) \equiv 1$

$$\mathrm{R}_0: \frac{x \,\dot{\in}\, C}{vGT(x) \neq \emptyset} \qquad \mathrm{R}_1: \frac{f(\cdots, \overbrace{x}^{\text{i-th}}, \cdots) \,\dot{\in}\, C}{vGT(x) = fGT(f,i)} \qquad \mathrm{R}_2: \frac{f(\cdots, \overbrace{gt}^{\text{i-th}}, \cdots) \,\dot{\in}\, C}{gt \in fGT(f,i)}$$

$$\mathrm{R}_3: \frac{f(\cdots, \overbrace{t[x_{1:n}]}^{\text{i-th}}, \cdots) \,\dot{\in}\, C}{t[vGT(x_1)/x_1, \cdots, vGT(x_n)/x_n] \subseteq fGT(f,i)}$$

$$\mathrm{R}_4: \frac{op(\cdots, x, \cdots) \in C,\ op \notin \{=, <, \leq, >, \geq\}}{vGT(x) = \infty} \qquad \mathrm{R}_5: \frac{op(x,y) \in C,\ op \in \{=, <, \leq, >, \geq\}}{vGT(x) = \infty \quad vGT(y) = \infty}$$

$$\mathrm{R}_6: \frac{(x \leq gt) \in C}{gt + 1 \in vGT(x)} \qquad \mathrm{R}_7: \frac{(x \geq gt) \in C}{gt - 1 \in vGT(x)} \qquad \mathrm{R}_8: \frac{\neg op(x, gt) \in C, where\ op \in \{\leq, \geq\}}{gt \in vGT(x)}$$

$$\mathrm{R}_9: \frac{\neg(x < gt) \in C}{gt - 1 \in vGT(x)} \qquad \mathrm{R}_{10}: \frac{\neg(x > gt) \in C}{gt + 1 \in vGT(x)} \qquad \mathrm{R}_{11}: \frac{op(x, gt) \in C, where\ op \in \{<, >\}}{gt \in vGT(x)}$$

$$\mathrm{R}_{12}: \frac{\neg(x = gt) \in C}{gt \in vGT(x)} \qquad \mathrm{R}_{13}: \frac{(x = gt) \in C, x \in \mathbb{Z}}{\{gt - 1, gt + 1\} \subseteq vGT(x)} \qquad \mathrm{R}_{14}: \frac{(x = gt) \in C, x \notin \mathbb{Z}}{vGT(x) = \infty}$$

Figure 2: The syntactic rules for generating the set constraints system ($\mathcal{S}_A$).

interpreted function (supported operators are $\{=, <, \leq, >, \geq\}$), thus sets $vGT(x)$ to infinity[4] in order to be propagated to other relevant ground term sets. Moreover, we do not handle the case where a supported interpreted operator has more than one variable argument (rule R_5). The remaining rules infer additional constraints for $vGT(x)$ where x occurs as an argument of a supported interpreted function. They constrain $vGT(x)$ to contain at least one ground term that falsifies the corresponding (interpreted) literal.

Let $vGT_{\mathcal{S}_A}$ denote a collection of finite sets of ground terms which satisfies the constraints $\mathcal{S}_A$. We show that, if finite, $vGT(x)_{\mathcal{S}_A}$ is a sufficient ground term set for x in A. The variable x can hence be eliminated by instantiating it with all the ground terms in $vGT(x)_{\mathcal{S}_A}$. The resulting formula $A[vGT(x)_{\mathcal{S}_A}/x]$ is equisatisfiable to A and does not contain x anymore.

Theorem 1 (Main Theorem)**.** *Let x be a variable in A with $vGT(x)_{\mathcal{S}_A} \neq \infty$, then A and $A[vGT(x)_{\mathcal{S}_A}/x]$ are equisatisfiable.*

Proof. If $A[vGT(x)_{\mathcal{S}_A}/x]$ is unsatisfiable, so is A since the former is an implication of the latter. If $A[vGT(x)_{\mathcal{S}_A}/x]$ is satisfiable with a model $\mathcal{M}$, then we construct a modified model $\mathcal{M}^{\pi_x}$ (as defined below) and show in lemma 3 that $\mathcal{M}^{\pi_x}$ satisfies A. □

Given a model $\mathcal{M}$ for the formula $A[vGT(x)_{\mathcal{S}_A}/x]$, we construct a modified model $\mathcal{M}^{\pi_x}$ as follows: $|M^{\pi_x}| := |M|$. For any constant $c \in Con$, $M^{\pi_x}(c) := M(c)$. For any interpreted operator op, $M^{\pi_x}(op) := M(op)$. For any uninterpreted function f, $M^{\pi_x}(f)(v_{1:n}) := M(f)(\pi_x(f,1)(v_1), \cdots, \pi_x(f,n)(v_n))$, where $\pi_x(f,i)$ is defined as in Eq. 1. Intuitively, if the ground term set of x does not *subsume* the ground term set of the i^{th} argument of f, or if v_i is a value that M assigns to a ground term for the i^{th} argument of f, then $M^{\pi_x}(f)(.., v_i, ..) := M(f)(.., v_i, ..)$ Otherwise, $\pi_x(f,i)$ maps v_i to a value that M assigns to some ground term for the i^{th} argument of f. Integers must be mapped to the closest such value (see the proof of Lemma 1). A ground term set S *subsumes* a ground term set R, denoted by

[4]In theory, this infinite set denotes $\mathcal{H}(A)$, but we use it as the "unsupported" label that gets propagated to other relevant sets.

$R \dot{\subseteq} S$, if for every ground term $gt_1 \in R$ there exists a ground term $gt_2 \in S$ such that gt_1 is a subterm of gt_2.

$$\pi_x(f,i)(v) = \begin{cases} v & \text{if } fGT(f,i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A} \\ v & \text{else if } v \in M(fGT(f,i)_{\mathcal{S}_A}) \\ v' \in M(fGT(f,i)_{\mathcal{S}_A}) & \text{else if } v \notin \mathbb{Z} \\ v' \in M(fGT(f,i)_{\mathcal{S}_A}), s.t.\ |v-v'| \text{ is minimal} & \text{otherwise} \end{cases} \quad (1)$$

$$\pi_x(v) = \begin{cases} v & \text{if } v \in M(vGT(x)_{\mathcal{S}_A}) \\ v' \in M(vGT(x)_{\mathcal{S}_A}) & \text{else if } v \notin \mathbb{Z} \\ v' \in M(vGT(x)_{\mathcal{S}_A}), s.t.\ |v-v'| \text{ is minimal} & \text{otherwise} \end{cases} \quad (2)$$

We also define π_x (as in Eq. 2) to denote the value projection with respect to a variable x. If $vGT(x)_{\mathcal{S}_A} = fGT(f,i)_{\mathcal{S}_A}$, for instance because x occurs as the i^{th} argument of f, then $\pi_x = \pi_x(f,i)$. Before showing the proof of lemma 3 used in our main theorem, we introduce some auxiliary corollaries and lemmas. The proofs of the lemmas can be found in Appendix B.

Corollary 1. *If $vGT(x)_{\mathcal{S}_A} \neq \infty$, then $\pi_x(v) \in M(vGT(x)_{\mathcal{S}_A})$, for all $v \in |M|$.*

The following lemmas show that if $\mathcal{M}^{\pi_x}$ does not satisfy a literal l in a CNF formula A, a modified variable assignment β' can be found such that $\mathcal{M}$ together with β' does not satisfy l. Lemma 1 formulates the claim for interpreted literals, and Lemma 2 gives a stronger variant (with value equality rather than implication) for uninterpreted literals.

Lemma 1. *Let x be a variable with $vGT(x)_{\mathcal{S}_A} \neq \infty$, $\mathcal{M}$ a model, β a variable assignment, and $\beta' = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta(y))$ else $\beta(y)$. Then $(M,\beta') \models l$ implies $(M^{\pi_x},\beta) \models l$ for all interpreted literals l in A.*

Lemma 2. *Let x be a variable with $vGT(x)_{\mathcal{S}_A} \neq \infty$, $\mathcal{M}$ a model, β a variable assignment, and $\beta' = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta(y))$ else $\beta(y)$. Then $(M,\beta')(l) = (M^{\pi_x},\beta)(l)$ for all uninterpreted literals l in A.*

Lemma 3. *Let x be a variable in A with $vGT(x)_{\mathcal{S}_A} \neq \infty$ and $\mathcal{M}$ a model of $A[vGT(x)_{\mathcal{S}_A}/x]$, then $\mathcal{M}^{\pi_x}$ is a model of A.*

Proof. Let A' denote $A[vGT(x)_{\mathcal{S}_A}/x]$. Since $\mathcal{M}$ is a model of A', for every variable assignment $\beta : Var \to |M|$, we have $(M,\beta) \models A'$. Let β_0 be an arbitrary variable assignment. By corollary 1, we know that $\pi_x(\beta_0(x)) = M(gt_0)$ for some ground term $gt_0 \in vGT(x)_{\mathcal{S}_A}$. The instantiation $A[gt_0/x]$ is included in A' and thus $(M,\beta) \models A[gt_0/x]$ for any β. Let $\beta'_0 = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta_0(y))$ else $\beta_0(y)$. Assignment β'_0 maps x to $\pi_x(\beta_0(x)) = M(gt_0)$ and $(M,\beta'_0) \models A[gt_0/x]$, therefore $(M,\beta'_0) \models A$.
Assuming that A is in CNF, there must be for every clause C in A a literal l^C in C with $(M,\beta'_0) \models l^C$. Using lemma 1 for interpreted and lemma 2 for uninterpreted literals, we know that also $(M^{\pi_x},\beta_0) \models l^C$. Hence, M^{π_x} is a model for l^C, C and finally for A.

□

Algorithm 1: Heuristic detection of expensive variables with respect to a threshold

Data: $A : For, C_{max} : \mathbb{N}$
Result: $NoElim : Set\langle Var\rangle$

1 **begin**
2 $\quad NoElim \leftarrow \{x \in vars(A) \mid vGT(x)_{\mathcal{S}_A} = \infty\}$
3 $\quad$ **repeat**
4 $\quad\quad$ **for** $x \in vars(A) \setminus NoElim$ **do**
5 $\quad\quad\quad repFactor \leftarrow |scopevars(x) \cap NoElim| = \emptyset \ ?\ 0 : 1$
6 $\quad\quad\quad cost_x \leftarrow (\prod_{y \in scopevars(x) \setminus NoElim} |vGT(y)_{\mathcal{S}_A}|) * repFactor$
7 $\quad\quad\quad$ **if** $cost_x > C_{max}$ **then**
8 $\quad\quad\quad\quad$ *select* $m \in scopevars(x) \setminus NoElim$ *s.t.* $|vGT(m)_{\mathcal{S}_A}|$ *is maximum*
9 $\quad\quad\quad\quad NoElim \leftarrow NoElim \cup \{m\}$
10 $\quad$ **until** *NoElim is unchanged*;
11 $\quad$ **return** *NoElim*

5 Practical Optimizations

5.1 Simulating NNF

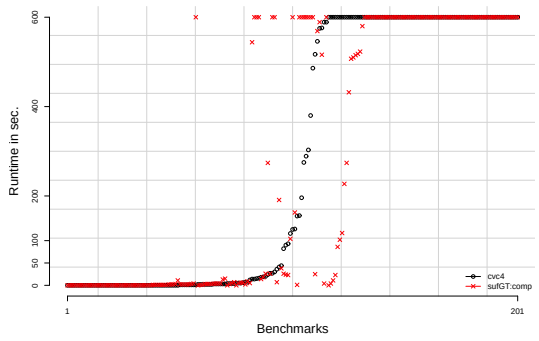
Previous section established that if the input formula is in CNF, we can instantiate variables with their computed sets of sufficient ground terms. Computing such sets, however, does not require the formula to be in CNF. That is, the constraint system of Figure 2 needs only the CNF polarity of the literals of the input formula (see rules R_6 to R_{13}). Therefore, instead of actually converting the original formula to CNF, we (1) *simulate* the NNF (negation normal form) conversion (without actually changing the formula) to compute polarity, and (2) skolemize all existential quantifiers[5]. This computation does not introduce any considerable overhead. It should be noted that conversion to CNF using distribution (as opposed to Tseitin encoding [13]) has the additional advantage that it minimizes the scope of each variable. This can significantly improve our simplification approach. Distribution, however, is very costly in practice. Computing minimal variable scopes without performing distribution is left for future work.

5.2 Limiting Instantiations

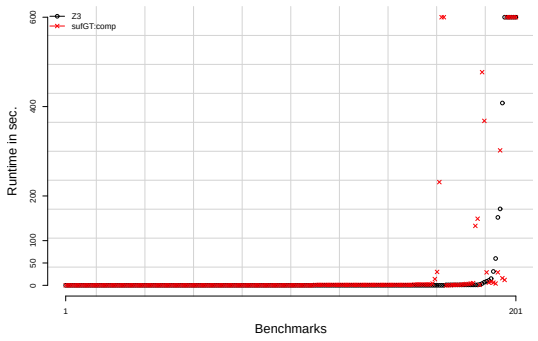
Our simplification approach eliminates those variables that have finite sets of sufficient ground terms by instantiating them with the computed ground terms. In practice, such instantiation may increase the occurrences of non-eliminable variables (see the example of Appendix A). Our experiments with Z3 and CVC4 show that this increase in the number of variable occurrences can considerably increase the solving time, specially for nested quantifiers.

We use Algorithm 1 to estimate and limit the cost of variable elimination based on the number of variable occurrences that it introduces. The algorithm tries to maximize the number of eliminated variables while keeping the cost low. Given a formula A and a threshold cost C_{max}, this algorithm returns a set of variables *NoElim* whose elimination causes the cost to exceed C_{max}. Line 2 initializes the *NoElim* set to the set of all variables whose sets of sufficient

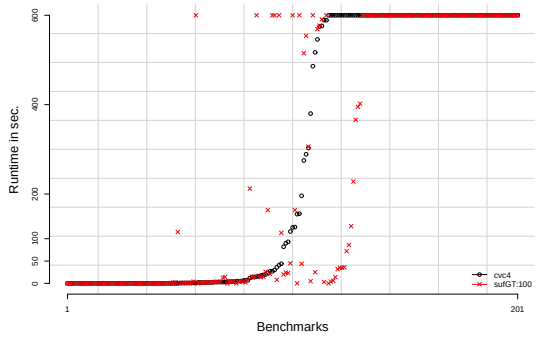
[5] If a formula A is not in CNF, the instantiation of a variable x with a set S of ground terms should be adjusted as $A[S/x] := A[\bigwedge_{gt \in S} B_x[gt/x]/B_x]$, where B_x is the smallest subformula containing x.



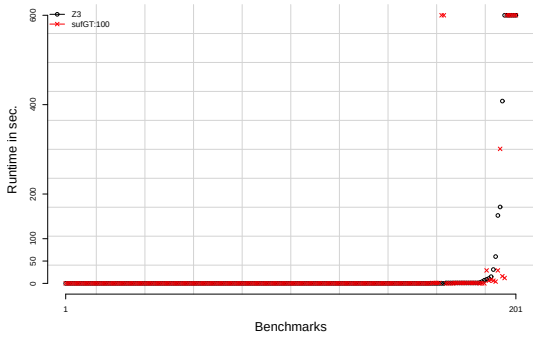
(a) CVC4, original vs. simplified (complete)



(b) Z3, original vs. simplified (complete)



(c) CVC4, original vs. simplified ($C_{max} = 100$)



(d) Z3, original vs. simplified ($C_{max} = 100$)

Figure 3: Experimental results on the benchmarks of the SMT-COMP/AUFLIA-p

ground terms are infinite, and thus will not be eliminated by our approach. Lines 4-9 evaluate the cost of eliminating a variable x that does not belong to *NoElim*. Instantiating x with its sufficient ground terms, in the worst case, replicates all non-eliminable variables (either free or bound) that appear in the scope of x (denoted by $scopevars(x)$), where the scope of x is the body of the quantified formula that binds x. We estimate the cost of eliminating all eliminable variables in the scope of x by $cost_x$. If this number exceeds the given threshold, then a variable m with the maximum number of instantiations will be marked as non-eliminable. The process then starts over.

6 Evaluation

We have implemented our approach in a prototype tool and performed experiments on the SMT-COMP benchmarks of 2012 in the AUFLIA-p/2012 division, using CVC4 (version 1.0) and Z3 (version 4.1) solvers. We ran both solvers on all benchmarks on an AMD DualCore Opteron Quad, 2.6GHz with 32GB memory.

For each benchmark, we compare the original runtime of each solver (with no simplification) against (1) a complete variable elimination, (2) a limited variable elimination where $C_{max} = 100$. Figures 3a and 3c give the comparison results for CVC4, and Figures 3b and 3d give the results for Z3. The x-axis of each plot shows the benchmarks, sorted according to the original runtime of the solvers, and the y-axis gives the runtime in seconds. Time-outs and 'unknown' outputs are represented identically. The time-out limit is 600 seconds.

For CVC4, the complete variable elimination improves the solving time of 37 cases (18%)–average speedup[6] 49x–out of which 16 were originally unsolvable, and worsens 55 cases (27%)–average speedup 0.45. The limited variable elimination, on the other hand, improves 39 cases (19%)–average speedup 57x–out of which 15 were originally unsolvable, and worsens 32 cases (15%)–average speedup 0.48. Z3 is known to be highly efficient in the AUFILA division (winner since 2008); its original runtime on many benchmarks is zero. The complete variable elimination, however, worsens 70 of these benchmarks (34%)–average speedup 0.38–and improves 11 cases (5%)–average speedup 10x–out of which one was originally unsolvable. The limited variable elimination, on the other hand, worsens only 8 cases (4%)–average speedup 0.35–and improves 14 cases (7%)–average speedup 9.4x–out of which one was originally unsolvable.

The main reason for slow down is the introduction of too many variable occurrences when not all variables are eliminable. Thus, as shown by these plots, for both solvers, the limited variable elimination produces stronger results[7]. However, even when *all* variables are eliminated, it is still possible that the solving time worsens as the number of instantiations that we produce can be higher than the number of instantiations that the solver would generate while solving the quantified formula. Although feasible in theory, this case was never observed in our experiments.

Although variable elimination with a limited cost can result in significant improvements of solving time, the experiments show that in some cases such as the two new time-outs of Figure 3d, a finer-grained limitation decision is needed. Investigating such cases is left as future work.

7 Related Work

Quantifier elimination in its traditional sense (aka. QE) refers to the property that an FOL theory $\mathcal{T}$ admits QE if for each formula ϕ, there exists a quantifier-free formula ϕ' so that for all models $\mathcal{M}$, $\mathcal{M} \models_{\mathcal{T}} \phi \Leftrightarrow \phi'$. Most applications of QE either provide decision procedures for fragments of FOL, or only prove their decidability. For example, the decidability proof of the Presburger arithmetic theory shows that the augmented theory with divisibility predicates admits QE [6]. Another example is the Fourier-Motzkin QE procedure for linear rational arithmetic (see [10]). QE is applicable to formulas that are purely in one of the known arithmetic theories, and eliminates those variables whose enclosing formulas are in a theory that admits QE. Consequently, it is not suitable as a general, stand-alone simplification for SMT formulas.

Another approach to eliminate quantifiers was proposed in [8] where partial FOL models are represented as programs. A program generation technique tries to heuristically generate a program P_i for a quantified formula ϕ_i in $F := \phi_1 \wedge \ldots \wedge \phi_n$ such that the proof obligation $[P_i](\phi_1, \ldots, \phi_n \Rightarrow \phi_i)$ can be discharged using a theorem prover. If such a program is found, F is modified to $\phi'_1 \wedge \ldots \wedge \phi'_n$ (without ϕ_i) where $\phi'_j \equiv [P_i]\phi_j$. The program generation and verification loop can be repeated until all quantified formulas are eliminated. Such an approach is very different from ours and is sound only for satisfiable formulas.

Our work was motivated by [3] and [7] in which quantifiers are eliminated via instantiation. In [3], a decision procedure is proposed for the *Array Property* fragment of FOL which supports a

[6] Speedup = old solving time / new solving time, where 0 second is changed to 0.5 second.

[7] Detailed information of the benchmarks are available at http://i12www.ira.uka.de/~elghazi/sufGT_smt13_expData/

combination of Presburger arithmetic for index terms, and equality with uninterpreted functions and sorts (EUF) for array terms. Similar to ours, this work instantiates universally quantified variables with a finite set of ground terms to generate an equisatisfiable formula. They prove the existence of such sets for their target fragment. Our approach, however, targets general FOL and leaves a variable uninstantiated if its set of ground terms is infinite. We believe that we can successfully handle the Array Property fragment. Experiments are left for future work.

In [7], Model-based Quantifier Instantiation (MBQI) is proposed for Z3. Similar to ours, this work constructs a system of set constraints Δ_F to compute sets of ground terms for instantiating quantified variables. Unlike us, however, they do not calculate a solution upfront, but instead, propose a fair enumeration of the (least) solution of Δ_F with certain properties. Assuming such enumeration, one can incrementally construct and check the quantifier-free formulas as needed[8]. If Δ_F is *stratified*, F is in a decidable fragment, and termination of the procedure is guaranteed. Otherwise the procedure can fall back on the quantifier engine of Z3 and provide helpful instantiation ground terms. Consequently, this technique can only act as an internal engine of an SMT solver and cannot provide a stand-alone formula simplification as ours does.

Variable expansion has also been proposed for quantified boolean formulas (QBF). In [2], a reduction of QBF to propositional conjunctive normal form (CNF) is presented where universally quantified variables are eliminated via expansion. Similar to our approach, they introduce cost functions, but with the goal of keeping the size of the generated CNF small.

8 Conclusion

We described a general simplification approach for quantified SMT formulas. Based on an analysis of the ground term occurrences at function applications, we compute *sufficient ground term sets* for each universally quantified variable. We proved that instantiating (thus eliminating) any variable whose computed set is finite, results in an equisatisfiable formula. Elimination of each variable is independent of the others. Thus we improve the performance of our technique by restricting the set of eliminable variables: we defined a prioritization algorithm that tries to maximize the number of eliminable variables while keeping the estimated elimination cost below a threshold. We evaluated our approach using two configurations and two solvers on a large subset of the SMT-COMP benchmarks. Our results show that (1) SMT benchmarks contain many variables that can be eliminated by our technique, (2) our complete variable instantiation may introduce significant overhead and thus slow down the solvers, (3) instantiation along with prioritization shows improvement of the solving time and score.

We believe that our technique can provide an easy framework for extending arbitrary SMT solvers with quantifier support. If we ignore termination and performance related rules when generating the set constraint system, we will have an incremental and fair procedure for building ground term sets. Using a finite model checker, like in [7], can then provide a framework for extending SMT solvers with quantifier support. Investigating this idea is left for future work.

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[8] In practice, they guide the quantifier instantiation using model checking which, in turn, uses an SMT solver.

[2] Armin Biere. Resolve and expand. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing*, SAT'04, page 59–70, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[5] Bruno Dutertre and Leonardo de Moura. The yices SMT solver. 2006.

[6] Herbert Enderton and Herbert B. Enderton. *A Mathematical Introduction to Logic, Second Edition.* Academic Press, 2 edition, January 2001.

[7] Yeting Ge and Leonardo Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320, 2009.

[8] Christoph D Gladisch. Satisfiability solving and model generation for quantified first-order logic formulas. In *FoVeOOS*, pages 76–91, 2011.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[10] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[11] Philipp Rümmer. E-matching with free variables. In *LPAR*, pages 359–374, 2012.

[12] Uwe Schöning. *Logic for Computer Scientists.* Birkhäuser, January 2008.

[13] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer, 1983.

A Expansion Example

The following example illustrates a case where eliminating one variable can result in increasing the occurrences of the other variables. This can introduce an overhead for the solver if the involved terms are complex.

Example 1. *Let $\forall x \mid (\psi(x) \vee \forall y, z \mid \varphi(x, y, z))$ be the input formula, and $S_y = \{gt_1, \ldots, gt_n\}$ be a set of sufficient ground terms for the variable y. Suppose that the sets of sufficient ground terms of x and z are infinite. In this case, instantiating and eliminating y will result in the formula*

$$\forall x \mid (\psi(x) \vee \forall z \mid (\varphi(x, gt_1, z) \wedge \ldots \wedge \varphi(x, gt_n, z)))$$

which has a higher number of occurrences of the variables x and z.

B Proofs

Corollary 1. *If $vGT(x)_{\mathcal{S}_A} \neq \infty$, then $\pi_x(v) \in M(vGT(x)_{\mathcal{S}_A})$, for all $v \in |M|$.*

Proof. The claim follows directly from the definition of π_x □

Corollary 2. *For all $gt \in Gr(A)$, $M^{\pi_x}(gt) = M(gt)$.*

Proof. By induction over the structure of gt. If $gt \in Const$, the claim follows directly from the definition of M^{π_x}. If, without loss of generality, $gt := f(t)$, where $f \in Fun$ and $t \in Gr$, we get by the induction hypothesis, $M^{\pi_x}(f(t)) = M^{\pi_x}(f)(M^{\pi_x}(t)) \overset{\text{i.h.}}{=} M^{\pi_x}(f)(M(t))$. Now we

have to distinguish between interpreted and uninterpreted functions. If f is interpreted, the claim follows directly from the definition of M^{π_x}. If f is uninterpreted, we get $M^{\pi_x}(f)(M(t)) = M(f)(\pi_x(f,1)(M(t)))$. Furthermore, we know, because of rule R_2 and $gt \in Gr(A)$, that $t \in fGT(f,1)_{\mathcal{S}_A}$. Now we can use the definition of $\pi_x(f,1)$ and we get $\pi_x(f,1)(M(t)) = M(t)$. □

For a variable assignment β, a value $v \in |M|$ and a variable $x \in Var$, we use the notation β_x^v to denote the modification of β where x is mapped to v.

Lemma 1. *Let x be a variable with $vGT(x)_{\mathcal{S}_A} \neq \infty$, $\mathcal{M}$ a model, β a variable assignment, and $\beta' = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta(y))$ else $\beta(y)$. Then $(M, \beta') \models l$ implies $(M^{\pi_x}, \beta) \models l$ for all interpreted literals l in A.*

Proof. Because of the rules R_4 and R_6, without loss of generality, we can restrict l to $l := op(x, gt_0)$ where $op \in \{=, <, \leq, >, \geq\}$ and β' to $\beta' = \lambda y.\ \beta_x^{\pi_x(\beta(x))}(y)$. Let us now assume that $(M, \beta_x^{\pi_x(\beta(x))}) \models l$ and $(M^{\pi_x}, \beta) \not\models l$. For $op := \text{" } < \text{"}$, we get from rule R_5, $gt_0 \in vGT(x)_{\mathcal{S}_A}$ and from the assumptions the inequality system $(\beta(x) \geq gt_0) \wedge (\pi_x(\beta(x)) < gt_0)$, which implies that $|\beta(x) - \pi_x(\beta(x))|$ is not minimal, since $|\beta(x) - gt_0|$ is strictly smaller. For $op \in \{\leq, >, \geq\}$, the proof is similar to the previous case. For $op := \text{" } = \text{"}$, we get from rule R_{13}, $\{gt_0 - 1, gt_0 + 1\} \subseteq vGT(x)_{\mathcal{S}_A}$ and from the assumptions, the inequality system $(\beta(x) \neq gt_0) \wedge (\pi_x(\beta(x)) = gt_0)$, which is equivalent to $(\beta(x) \leq gt_0 - 1) \vee (gt_0 + 1 \leq \beta(x))) \wedge (\pi_x(\beta(x)) = gt_0)$ and implies that $|\beta(x) - gt_0|$ is not minimal, since in the case $(\beta(x) \leq gt_0 - 1)$, $|\beta(x) - (gt_0 - 1)|$ is strictly smaller and in the case $(gt_0 + 1 \leq \beta(x))$, $|\beta(x) - (gt_0 + 1)|$ is strictly smaller. □

Proposition 1 provides a stronger result compared to lemma 1. It better reflects the intuition behind the rules R_5, R_7 to R_{13}. They guarantee that if a variable x occurs as an argument of an interpreted operator, then there is at least one $gt_l \in vGT(x)_{\mathcal{S}_A}$ with $\not\models_{\mathcal{T}} l[gt_l/x]$. That is, $C[gt_l/x]$ is either valid or its satisfiability is determined by literals other than l. We proved lemma 1 because it is sufficient for our main theorem, and it has a shorter proof.

Proposition 1. *Let C be a clause in A, x a variable in C with $vGT(x)_{\mathcal{S}_A} \neq \infty$, and M a model of $C[vGT(x)_{\mathcal{S}_A}/x]$, then either there exists an uninterpreted literal $l \in C$, where $M \models l[gt/x]$ for some $gt \in vGT(x)_{\mathcal{S}_A}$, or there exists a (tautology) subclause C' of C whose literals are interpreted and $\models_{\mathcal{T}} C'$.*

In the following, we use *expressions* to refer to both terms and formulas. That is, $Expr = Term \cup For$.

Lemma 2. *Let x be a variable with $vGT(x)_{\mathcal{S}_A} \neq \infty$, $\mathcal{M}$ a model, β a variable assignment, and $\beta' = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta(y))$ else $\beta(y)$. Then $(M, \beta')(l) = (M^{\pi_x}, \beta)(l)$ for all uninterpreted literals l in A.*

Proof. To prove the claim, we show the statement $(M, \beta')(l) = (M^{\pi_x}, \beta)(l)$ for all expressions but variables $l \in Expr \setminus Var$ occurring in A using structural induction.

If l is a ground term in A, then the claim follows directly from corollary 2.

Let $l = f(t_{1:n})$ be a function application in A with f an uninterpreted function. The evaluations of l are

$$\begin{aligned}(M^{\pi_x}, \beta)(f(t_{1:n})) &= M^{\pi_x}(f)((M^{\pi_x}, \beta)(t_1), \ldots, (M^{\pi_x}, \beta)(t_n)) \\ &= M(f)(\pi_x(f,1)((M^{\pi_x}, \beta)(t_1)), \ldots, \pi_x(f,n)(t_n)) \\ (M, \beta')(f(t_{1:n}) &= M(f)((M, \beta')(t_1), \ldots, (M, \beta')(t_n))\end{aligned}$$

It suffices to show that $\pi_x(f,i)((M^{\pi_x},\beta)(t_i)) = (M,\beta')(t_i)$ for $1 \leq i \leq n$. We do this by a case distinction over the type of the terms t_i.

If $t_i = y$ is a variable with $vGT(y)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$, then $\beta'(y) = \beta(y)$. Because of rule R_1 we additionally get $fGT(f,i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$, which implies that $\pi_x(f,i)$ is the identity.

If $t_i = y$ is a variable with $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, then $\beta'(y) = \pi_y(\beta(y))$. Because of rule R_1 we get $vGT(y)_{\mathcal{S}_A} = fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, which implies that $\pi_x(f,i) = \pi_y$.

If t_i is a function application, we assume $t_i = s[x_{1:m}]$ for some term s. By induction hypothesis, $\pi_x(f,i)((M^{\pi_x},\beta)(s[x_{1:m}])) \overset{\text{i.h.}}{=} \pi_x(f,i)((M,\beta')(s[x_{1:m}]))$. W.r.t. $fGT(f,i)_{\mathcal{S}_A}$, there is two possible cases to consider:
1) $fGT(f,i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$, then $\pi_x(f,i)$ is the identity and the claim follows directly.
2) $fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, then because of rule R_3 $vGT(x_i)_{\mathcal{S}_A} \dot{\subseteq} fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, for all $1 \leq i \leq m$. This implies that $\beta'(x_i) = \pi_{x_i}(\beta(x_i))$ for all $1 \leq i \leq m$. Using this fact together with corollary 1, there exists for each x_i a ground term gt_i, with $\pi_{x_i}(\beta(x_i)) = M(gt_i)$ and $gt_i \in vGT(x_i)_{\mathcal{S}_A}$. So we can write, $\pi_x(f,i)((M,\beta')(s[x_{1:m}])) = \pi_x(f,i)(M(s[gt_{1:m}]))$. Because of rule R_3 we know that $s[gt_{1:m}] \in fGT(f,i)_{\mathcal{S}_A}$, and so $M(s[gt_{1:m}]) \in M(fGT(f,i)_{\mathcal{S}_A})$. Finally the claim follows from the definition of $\pi_x(f,i)$ for values in $M(fGT(f,i)_{\mathcal{S}_A})$ and the assumption that $fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$.

Let $l := f(t_{1:n})$ be an expression with f an interpreted function. Using the definition of M^{π_x} for interpreted functions, $(M^{\pi_x},\beta)(f(t_{1:n})) = M(f)((M^{\pi_x},\beta)(t_1),\ldots,(M^{\pi_x},\beta)(t_n))$. Since l is uninterpreted, all t_is are non-variables and we can use the induction hypotheses on them and get, $M(f)((M^{\pi_x},\beta)(t_1),\ldots,(M^{\pi_x},\beta)(t_n)) = M(f)((M,\beta')(t_1),\ldots,(M,\beta')(t_n))$. Now the claim follows directly from the definition of M.

□