Resourceful Reachability as HORN-LA

Josh Berdine, Nikolaj Bjørner, Samin Ishtiaq, Jael E. Kriener, and Christoph M. Wintersteiger

Microsoft Research; University of Kent

Abstract. The program verification tool SLAyer uses abstractions during analysis and relies on a solver for reachability to refine spurious counterexamples. In this context, we extract a reachability benchmark suite and evaluate methods for encoding reachability properties with heaps using Horn clauses over linear arithmetic. The benchmarks are particularly challenging and we describe and evaluate pre-processing transformations that are shown to have significant effect.

1 Introduction

When a proof attempt by a static analyzer or model checker fails, an abstract counterexample is commonly produced. The counterexample does not necessarily correspond to a real bug because the analyzer's abstraction could be too coarse. Here we describe and evaluate new techniques to check concrete feasibility of abstract counterexamples produced as failed memory safety proofs by SLAyer, a separation logic–based shape analyzer [3]. The problem addressed in this paper is a particular instance of the more general one of state reachability in "resourceful" abstract transition systems, where the state space is theoretically unbounded, and changes over time, due to behavior such as dynamic allocation of memory.

This poses challenges to reachability tools along two dimensions: scaling search for long counter-examples, and encoding state transformations for heaps in a scalable way. Previous work [2] developed an encoding using bit-vectors and quantifiers and used bounded model checking (BMC). Fig. 1 illustrates that on a representative instance, unfortunately, it exhibits exponential slowdowns as the length of the explored path is increased. To further evaluate tradeoffs we extracted around 100 benchmarks from SLAyer that come from failed proof attempts in analysis of real-world C programs.

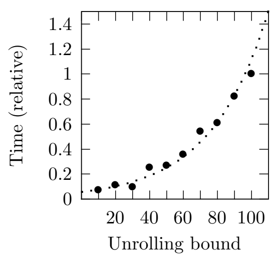


Fig. 1. Average solving time (Sat+Unsat), relative to the solving time of 100 unrollings

We encode reachability into logic as satisfiability of Horn clauses and use two backends of the Z3 SMT-solver for solving such clauses. The *PDR solver* handles Horn clauses over linear arithmetic (HORN-LA). It is compared with a solver based on BMC. (For details on the underlying semantics of the Horn-clause fragment and on the internals of the PDR solver, please see [7]).

Our main methodological contribution is encodings for resourceful reachability into Horn-LA. A basic encoding (§2.2) is refined (§2.3) to use a family of transition relations indexed by "worlds". These limit the state space based on the amount of memory allocated. This refinement allows the solver to explore smaller state spaces over fewer variables, constrained by smaller formulae. Our evaluation shows that this encoding helps the PDR solver on the hard instances, while causing visible overhead on the easy cases. The results (Fig. 6(b)) with the BMC solver are similar, but the overhead is significantly more detrimental. Additionally, we propose and evaluate (Fig. 5) two alternative approaches to shrinking heaps of error states. The evaluation of our methodology establishes that our PDR solver benefits critically from pre-processing transformations that we identify and evaluate (Fig. 7(a),7(b)). This paper is our first thorough experimental evaluation of [7] over arithmetical benchmarks.

2 Resourceful Transition System Reachability

Consider the faulty routine on the right. It contains a memory access error. SLAyer [3] can be used to find such errors. When SLAyer fails in its attempt to establish memory safety, it produces an abstract counterexample in the form of a *Resourceful Transition System* (RTS) – a transition system over states of some resource (memory, in our case). Fig. 2 is an RTS extracted from this example. An RTS is given by $\langle \mathbb{V}, \mathbb{E}, v_0, v_{err}, \nu, \rho \rangle$, where $\mathbb{V}$ and $\mathbb{E}$ are sets of vertices and edges respectively, v_0 identifies the root and v_{err} the error vertex, ν labels vertices with states, and ρ labels edges with transition relations. The abstract counterexample is feasible if there is a path $v_0, e_0, v_1, e_1 \ldots, v_n, e_n, v_{err}$ such that the conjunction $\nu(v_0)(\mathbf{s}_0) \wedge \bigwedge \rho(e_i)(\mathbf{s}_i, \mathbf{s}_{i+1})$ is feasible.

```
void access_error()
{
  int* x0, *x1;

  x1 = malloc(sizeof(int));
  if (nondet()) {
    x0 = malloc(sizeof(int));
  }
  while (nondet()) {
    *x0 = 3;
  }
}
```

2.1 Encoding Resources

To encode resourceful reachability into SMT we adapt a model where states are summarized as a predicate over a store and a heap. The store tracks the values of program variables, $\mathbb{X}$, and the heap tracks memory objects. This representation is given by a triple $(f, \boldsymbol{x}, \mathbf{a})$, where

- $f \in \mathbb{N}$ is a frontier counter, indicating the 'next' free address,
- $\boldsymbol{x} \in \mathbb{D}^{|\mathbb{X}|}$ is the store over values in $\mathbb{D}$ ($\mathbb{D}$ includes $\mathbb{N}$), and
- $\mathbf{a} \in \mathit{Array}\langle \mathbb{N}, \mathbb{B} \times \mathbb{N} \times \mathbb{D} \rangle$ is an array mapping addresses to triples encoding if the address is allocated, the size of the allocated object, and its value.

The state labeling ν is a relation over $(f, \boldsymbol{x}, \mathbf{a})$, where initially f is 0 and $\mathbf{a}$ maps all addresses to an un-allocated state. Similarly, ρ is a relation $\rho(e_i)(f, \boldsymbol{x}, \mathbf{a}, f', \boldsymbol{x}', \mathbf{a}')$.

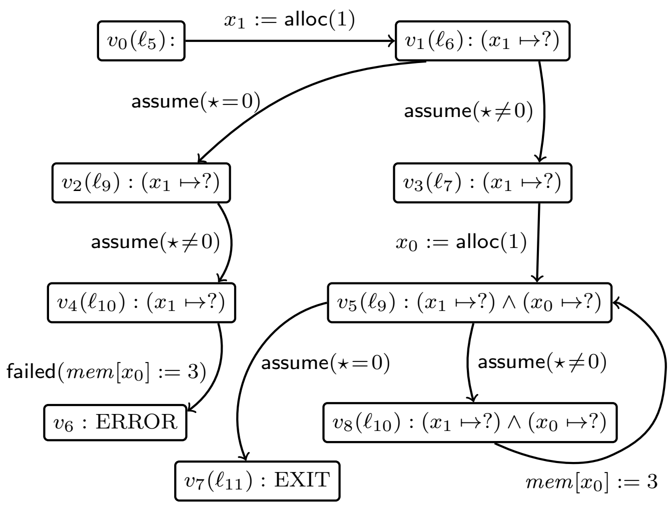

$x_1 :=$ alloc(1)
$v_0(\ell_5):$
$v_1(\ell_6): (x_1 \mapsto ?)$
assume($\star = 0$)
assume($\star \neq 0$)
$v_2(\ell_9) : (x_1 \mapsto ?)$
$v_3(\ell_7) : (x_1 \mapsto ?)$
assume($\star \neq 0$)
$x_0 :=$ alloc(1)
$v_4(\ell_{10}) : (x_1 \mapsto ?)$
$v_5(\ell_9) : (x_1 \mapsto ?) \wedge (x_0 \mapsto ?)$
failed($mem[x_0] := 3$)
assume($\star = 0$)
assume($\star \neq 0$)
v_6 : ERROR
$v_8(\ell_{10}) : (x_1 \mapsto ?) \wedge (x_0 \mapsto ?)$
$v_7(\ell_{11})$: EXIT
$mem[x_0] := 3$

Fig. 2. failed attempt at proving safety for `access_error`

The unsuccessful transition corresponds to an update into an unallocated memory location, and is marked as failed. An encoding into Horn-LA requires eliminating arrays, so we flatten **a** into a finite tuple of triples. The (pre-set) size n of this tuple, which is the number of available memory locations and therefore the bound on the resource, plays a crucial role in deciding reachability.

2.2 Resourceful Reachability as SMT - Basic Encoding

Given an RTS $\langle \mathbb{V}, \mathbb{E}, v_0, v_{err}, \nu, \rho \rangle$, and a bound n, reachibility can be modeled as a predicate $R(v, f, \boldsymbol{x}, \mathbf{a})$, where $v \in \mathbb{V}$ and $(f, \boldsymbol{x}, \mathbf{a})$ is a state as described above. R is defined by the following set of Horn clauses, where the schema in the second line is repeated for each $e_{ij} \in \mathbb{E}$ s.t. $e_{ij} = (v_i, v_j)$:

$$\begin{aligned} R(v_0, f, \boldsymbol{x}, \mathbf{a}) &\leftarrow \nu(v_0)(f, \boldsymbol{x}, \mathbf{a}) \\ R(v_j, f', \boldsymbol{x}', \mathbf{a}') &\leftarrow R(v_i, f, \boldsymbol{x}, \mathbf{a}) \wedge \rho(e_{ij})(f, \boldsymbol{x}, \mathbf{a}, f', \boldsymbol{x}', \mathbf{a}') \wedge f' \leq n \\ \mathit{false} &\leftarrow R(v_{err}, f, \mathbf{x}, \mathbf{a}) \end{aligned} \tag{1}$$

The free variables $f, f', \boldsymbol{x}, \mathbf{a}, \boldsymbol{x}', \mathbf{a}'$ are implicitly universally quantified. Reachability corresponds to unsatisfiabilty of these Horn clauses:

Proposition 1 (Bounded Resource Reachability). *For an RTS, there is a feasible path from v_0 to v_{err} allocating at most n resources if and only if the clauses defined by (1) are unsatisfiable.*

Conversely, if the clauses are satisfiable, the error state is unreachable *within a resource of size n*. At this point we see why the bound is crucial: unreachability, i.e. safety, can be proven only relative to a given bound.

Example 1 (RTS encoding). Continuing our running example, below we show the clauses encoding the edges between v_0 and $v_{err} = v_6$ of the ATS in Fig. 2 over a space of four addresses, i.e. $n = 4$.

$$R(0, f, \boldsymbol{x}, \mathbf{a}) \leftarrow f = 0 \wedge \mathbf{a} = \underbrace{(\mathsf{false}, \_, \_), \ldots, (\mathsf{false}, \_, \_)}_{4} \qquad \text{// initial state}$$

$$
\begin{aligned}
R(1,f',\boldsymbol{y},\mathbf{b}) &\leftarrow R(0,f,\boldsymbol{x},\mathbf{a}) \wedge \boldsymbol{y} = \langle x_0, f\rangle && //\mathsf{v_0} \to \mathsf{v_1} : \mathsf{alloc}(\mathsf{x_1}, 1)\\
&\quad f' = f+1 \wedge f' \leq 4\\
&\quad \wedge \textstyle\bigwedge_{h=0}^{h<4} \mathbf{b}[h] = \big(\mathsf{if}\ h = f\ \mathsf{then}\ (\mathsf{true}, 1, \_)\ \mathsf{else}\ \mathbf{a}[i]\big)\\
R(2,f,\boldsymbol{x},\mathbf{a}) &\leftarrow R(1,f,\boldsymbol{x},\mathbf{a}) && //\mathsf{v_1} \to \mathsf{v_2} : \mathsf{assume}(\mathsf{nondet}() = 0)\\
R(4,f,\boldsymbol{x},\mathbf{a}) &\leftarrow R(2,f,\boldsymbol{x},\mathbf{a}) && //\mathsf{v_2} \to \mathsf{v_4} : \mathsf{assume}(\mathsf{nondet}() \neq 0)\\
R(6,f,\boldsymbol{x},\mathbf{a}) &\leftarrow R(4,f,\boldsymbol{x},\mathbf{a}) \wedge \mathbf{a}[x_0] = (\mathsf{false}, \_, \_) && //\mathsf{v_4} \to \mathsf{v_6} : \mathsf{unsuccessfull\ store}\\
\mathit{false} &\leftarrow R(6,f,\boldsymbol{x},\mathbf{a})
\end{aligned}
$$

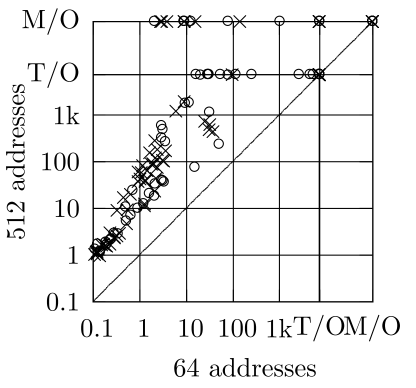


Fig. 3. Slowdown when n increases: BMC $\times$, PDR $\circ$

Observe that the value of n, which is fixed prior to encoding, has two effects on the relation R: First, it affects satisfiability of the Horn clauses. Second, it affects, and in fact dominates, the number of parameters of R, which is given by $1+|\mathbb{X}|+3n$. The problems this can cause are demonstrated by the example in Fig. 2. First, suppose the edge (v_0, v_1) was labeled with '$\mathsf{alloc}(\mathsf{x_1}, 5)$', allocating 5 words rather than one word. In that case, the corresponding rule would contain a constraint equivalent to $f+5 \leq 4$, and as a result, v_{err} would become unreachable. However, given an $n \geq 5$ it would still be reachable. That is to say, underestimating the size of n required to reach v_{err} may result in a loss of completeness. Second, notice that none of $\mathbf{a}[1]$, $\mathbf{a}[2]$ nor $\mathbf{a}[3]$ are required in a traversal from v_0 to v_{err} here. However, they all contribute to the resulting constraints and become part of the search space during solving. Fig. 3 shows that as n is increased, the runtime and memory-outs increase as well.

2.3 Encoding Reachability - Kripke Style

To reduce the cost of propagation over large predicates, we propose and evaluate a method that introduces a sequence of predicates of increasing arity. The aim is to search over low arity predicates first, and resort to larger arity predicates only when search in the lower arity predicates has been exhausted. If a counterexample trace exists that requires only a part of the available resource, finding it does not require assigning (irrelevant) values to the entire resource. This encoding is inspired by possible-world semantics of programming languages. The idea is that of Kripke models for intuitionistic logics where sets of possible worlds or facts grow monotonically.

The idea is to choose a chain $\mathcal{W} := w_0(= 0) < w_1 < \cdots < w_m = n$, of increasing sizes for the available resource $\mathbf{a}$. From $\mathcal{W}$ we encode a set of predicates $\{R_0, \ldots, R_m\}$, such that R_k has arity $1 + |\mathbb{X}| + 3w_k$. Analogously to the monotonically growing worlds in a Kripke-style program semantics, the predicates R_k capture the set of reachable states with increasing precision. Each R_k is defined by clauses instantiating the second clause of (1), for each $e \in \mathbb{E}$, s.t. $e_{ij} = (v_i, v_j)$. The first clause of (1) is replaced by the single non-recursive clause, reflecting that initially no resource is allocated:

$$R_0(v_0, \boldsymbol{x}) \leftarrow \nu(v_0)$$

In addition, to allow the solver to access R_l from R_k for all $l > k$, 'world-changing clauses' are introduced at each step instantiating the following schema for all $k < l \leq m$, where $|\mathbf{a}| = w_k$ and $|\mathbf{b}| = w_l$:

$$\begin{aligned} R_l(v_j, f', \boldsymbol{x}', \mathbf{b}') \leftarrow\ & R_k(v_i, f, \boldsymbol{x}, \mathbf{a}) \\ & \wedge \rho(e_{ij})(f, \boldsymbol{x}, \mathbf{b}, f', \boldsymbol{x}', \mathbf{b}') \wedge w_{l-1} < f' \leq w_l \\ & \wedge \textstyle\bigwedge_{h=0}^{h<w_l} \mathbf{b}[h] = \big(\mathsf{if}\ h < w_k\ \mathsf{then}\ \mathbf{a}[h]\ \mathsf{else}\ (\mathsf{false}, \_, \_)\big) \end{aligned} \tag{2}$$

When changing worlds, first the current state of the smaller world $\mathbf{a}$ is 'copied' over to a 'fresh' larger one $\mathbf{b}$, in which the new elements are initialised to be free. In practice, this step is only required for transitions that may actually consume resource (alloc in SLAyer).

$$\begin{tikzcd}
 & \dots\ R_k(v_{err}, \boldsymbol{x}, \_)\ \dots \arrow[dd] & \\
R_1(v_{err}, \boldsymbol{x}, \_) \arrow[dr] & & R_m(v_{err}, \boldsymbol{x}, \_) \arrow[dl] \\
 & R_0(v_{err}, \_) &
\end{tikzcd}$$

Fan

$$\begin{tikzcd}
 & R_n(v_{err}, \boldsymbol{x}, \_) \arrow[d] \\
 & \vdots \arrow[d] \\
 & R_1(v_{err}, \boldsymbol{x}, \_) \arrow[dl] \\
R_0(v_{err}, \_) &
\end{tikzcd}$$

Chain

Fig. 4. Schemata for ways of connecting R_0 to R_k in a set of Kripke predicates

Since there are several predicates R_k in this encoding, v_{err} may be reachable in each of them. The purpose of this encoding is to explore smaller R_k before larger ones, hence we will pose the query in R_0 – $R_0(v_{err}, \boldsymbol{x})$ – and let the solver choose a larger one if neccessary. There are several ways of setting up that choice: we could either add m clauses of the form $R_0(v_{err}, \boldsymbol{x}) \leftarrow R_k(v_{err}, \boldsymbol{x}, \mathbf{a})$ (creating a *fan* into R_0, so to speak); or add a *chain* of clauses $R_k(v_{err}, \boldsymbol{x}, \mathbf{a}) \leftarrow R_{k+1}(v_{err}, \boldsymbol{x}, \mathbf{a}')$ for $0 \leq k < m$ – see Fig. 4.

2.4 Evaluation

There are now several parameters to an encoding: the size of the maximum available resource n, the set $\mathcal{W}$ of increasing sizes of parts of the resource, and the choice between a fan and a chain of clauses for reaching $R_0(v_{err}, \boldsymbol{x})$.

We considered growing w_k with a linear increase and a log-step increase, i.e. each w_{k+1} is double the size of w_k and settled for the latter encoding for our evaluation. Fig. 5 also suggests that the fanning approach on large heap sizes is better overall, though remarkably chaining handles some hard instances not handled by fanning.

Reducing the sensitivity to the bound n on heap size exhibited by the previous encoding was the motivation for the encoding using Kripke transition relations. This aim is largely achieved when using the PDR backend, as shown in Fig. 6(a). The results show that solving problems encoded using Kripke relations with the PDR backend times out much less frequently, and solves many problems that time out with the single relation encoding.

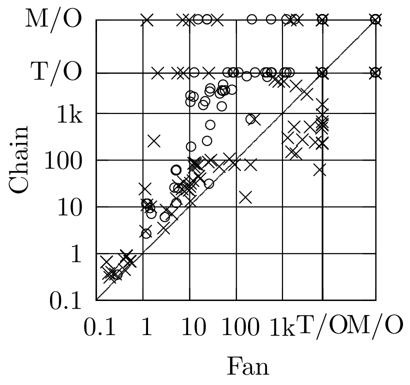
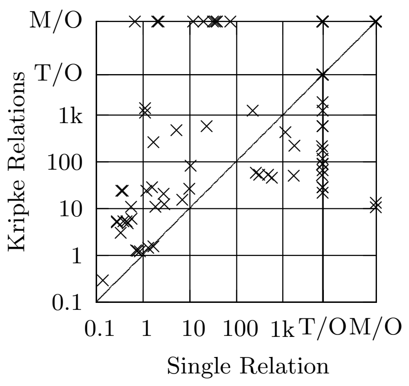


Fig. 5. Fan vs. Chain; PDR ×: 64, ∘: 512 addresses



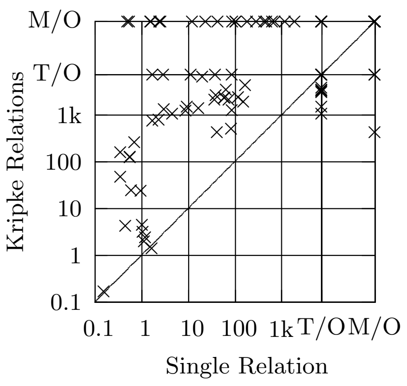


Fig. 6. Single vs Kripke Relations

(a) PDR: 512 addresses - chain

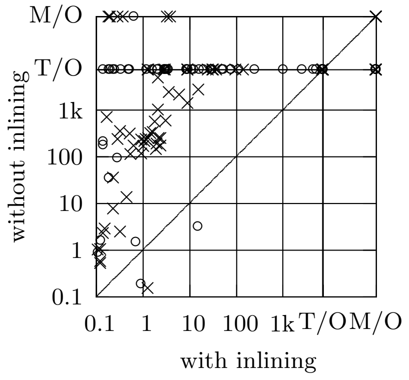
(b) BMC: 512 addresses

The improvements can be quite dramatic, but the results also indicate that the Kripke predicates encoding imposes a considerable overhead, and a number of problems exhaust memory only with it. Overall, there is a clear improving trend from above to below the break-even line as problems get hard enough that less sensitivity to heap size dominates the overhead. We have observed this effect in both the chaining and the fanning approach.

As shown in Fig. 6(b), the Kripke predicates encoding is, on the other hand, detrimental to the BMC backend. This effect is independent of the chaining or fanning approach. The basic encoding solves many more problems than the Kripke encoding, only a few problems are solved only using the latter. This should not be surprising because BMC effectively causes the largest arity predicate to always be present in the constraints sent to the SMT solving backend.

3 Pre-processing simplifications

We here evaluate pre-processing simplifications that are key to the performance of our Horn clause solvers. We summarize two transformations on a set of Horn clauses $\mathcal{C}$. The **Inline** transformation replaces two clauses by a single clause in a transformation of the form $\mathcal{C},\ p(u) \leftarrow B_1 \wedge q(t),\ q(s) \leftarrow B_2 \Longrightarrow \mathcal{C},\ (p(u) \leftarrow B_1 \wedge B_2)\theta$, where the head predicate $q(s)$ unifies with $q(t)$ with the substitution θ and there are no other occurences of q in $\mathcal{C}$ that unifies with $q(s)$. The **Unfold** transformation generalizes inlining by replacing m clauses with q in the body and n clauses with q in the head and creating up to $m \times n$ new clauses. It corresponds to an iterative squarring transformation or a Davis Putnam resolution step. Figures 7(a),7(b) demonstrate the significant effect of the exhaustive application of these transformations on PDR and neutral effect on BMC.



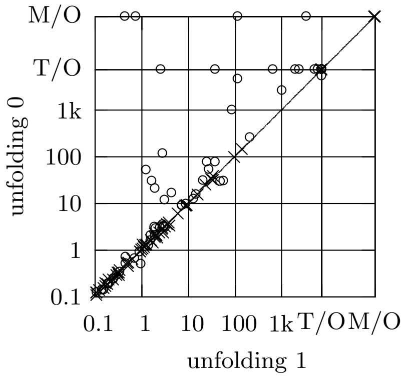


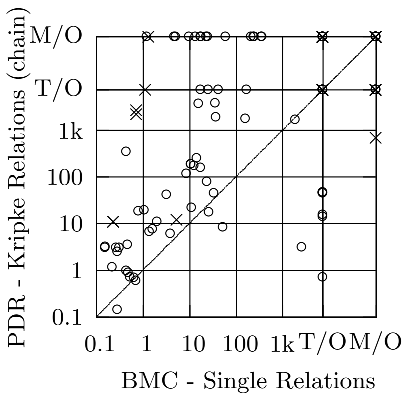
Fig. 7. Effects of Inline and Unfold transforms

(a) Inlining improves performance for BMC and PDR: BMC: $\times$, PDR: $\circ$

(b) Unfolding is good for PDR and doesn't break BMC: BMC: $\times$, PDR: $\circ$

4 PDR and BMC backends

We have compared the two backend solvers on our benchmark suite. The results, shown in Fig. 8(a), show that there are instances where each solver succeeds while the other does not. On instances where both succeed, the BMC solver tends to spend less time than the PDR solver. For unsatisfiable instances, there is also a tendency for the BMC solver to succeed only on the very easy ones, while the PDR solver has more success. This effect is more clearly seen when considering smaller heap sizes, see Fig. 8(b). In this configuration we see that the PDR solver dominates for unsatisfiable instances, where the BMC solver usually exhausts resources; while for satisfiable instances, the BMC solver is faster.



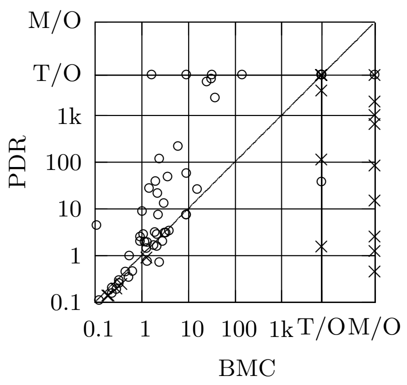


Fig. 8. BMC vs PDR. Reachable ○, Unreachable ×

(a) BMC in simple encoding vs PDR in Kripke-style encoding - 256 addresses

(b) BMC vs PDR, both in the simple encoding - 64 addresses

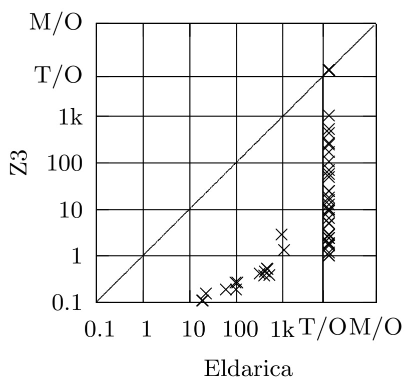
5 Comparison with other solvers

Horn clause solvers Z3 is not the only SMT solver that can check satisfiability of Horn clauses over linear arithmetic. The HSF/QARMC tools [5] check satisfiability of Horn clauses over linear real arithmetic and the Eldarica tool [10] accepts Horn clauses over linear real and integer arithmetic. Furthermore, constraint logic programming systems, such as MAP [1], TRACER [8], and CHiAO [6] support different aspects of Horn clauses over arithmetic. To our knowledge, they don't yet work in a way compatible with the SMT-LIB benchmark suite.

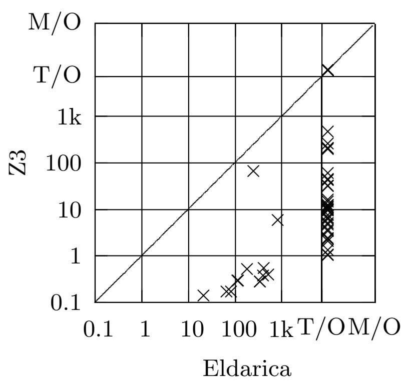
HSF/QARMC solves Horn clauses where constraints are given as a conjunction of literals. It therefore relies on converting constraints in the bodies of Horn clauses into disjunctive normal form (DNF). Given the way memory is encoded in our benchmarks, the DNF transformation is infeasible and therefore the tool times out on all the problems we have produced.

Eldarica uses the SMT solver Princess [9] for handling arithmetical constraints and generating interpolants. It is able to handle Horn clauses with nested constraints and we include a comparison in Figure 9. First of all, we interpret the results to establish these benchmarks as highly challenging for current state-of-the-art Horn clause solvers. We also see the results as a testament to the significance of pre-processing that we described in Section 3.

Bounded model checking tools We have compared our implementation of BMC on Horn clauses against both the implementation reported in [2] and the well-established, well-tested model checking tool CBMC [4]. The first comparison, shown in Fig. 10(a), indicates that while our BMC backend sometimes outperforms that of [2], when the a priori unrolling depth is chosen well the latter



(a) Chaining Encoding



(b) Fanning Encoding

Fig. 9. Eldarica vs. Z3

performs very well. At present, there is a meaningful price our method pays for the robustness with respect to choice of unrolling bound.

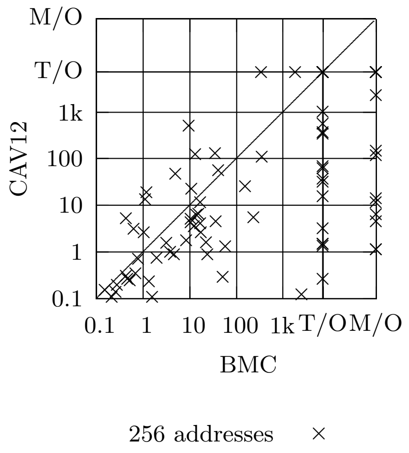
The results of the second comparison, shown in Fig. 10(b), demonstrate that our backend can compete with CBMC on instances of these problems, returning relatively quickly for numerous instances for which CBMC runs out of memory.

There is a significant caveat regarding these results though: our backends operate over the theory of linear real arithmetic, while the other two are over bit-vectors. The benchmarks themselves do not exercise the difference between these theories, so the same high-level problem is being solved here. So the currently best-performing solving method to establish unsafety/reachability is BMC and is based on quantified bit-vectors, while the main available method establishing safety is based on linear arithmetic.

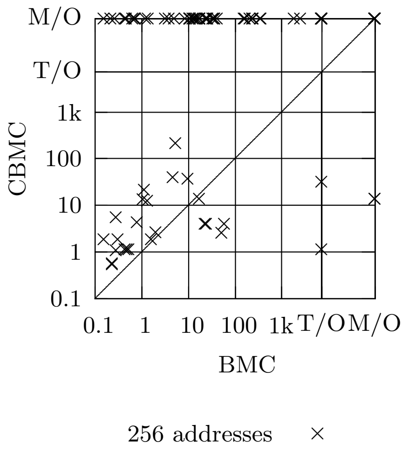
6 Summary

We have presented encodings of resourceful reachability problems into HORN-LA and evaluated these using two engines BMC and PDR. PDR can solve for both reachability and unreachability, wheareas BMC can only determine reachability. We found that BMC is generally faster on the reachable cases, but given significant attention to encoding and pre-processing, our implementation of PDR performs adequately. The raw data from our experiments is available at:
http://www.cs.kent.ac.uk/people/rpg/jek26/cex-data.zip.
The experimental data forms the basis of publicly available benchmarks:
https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/SLayerCF.

Acknowledgements Matthew Parkinson and Nathan Chong for invaluable feedback on drafts.



(a) BMC vs CAV12



(b) BMC vs CBMC

Fig. 10. BMC vs. CAV12 and CBMC

References

1. Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verifying programs via iterated specialization. In *PEPM*, 2013.
2. J. Berdine, A. Cox, S. Ishtiaq, and Christoph M. Wintersteiger. Diagnosing abstraction failure for separation logic-based analyses. In *CAV*, 2012.
3. Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, 2011.
4. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.
5. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
6. Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An overview of Ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.
7. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
8. Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV*, 2012.
9. Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, 2008.
10. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.