Yasm: A Software Model-Checker for Verification and Refutation

Arie Gurfinkel, Ou Wei, and Marsha Chechik

Department of Computer Science, University of TorontoEmail:
{arie,owei,chechik}@cs.toronto.edu

1 Motivation

This paper presents YASM: a (yet another) software model-checker based on the Counter-Example Guided Abstraction Refinement (CEGAR) [6] framework. A number of well-engineered software model-checkers are available, e.g., SLAM [1] and BLAST [12]. Why build another one?

Traditional software model-checkers build over-approximating abstractions of the programs they analyze and typically bias their analysis towards proving that a (safety) property of interest holds (verification). On the other hand, since model-checkers are widely known for their bug-finding abilities, they are often used for refutation. In this case, the above approach seems unreasonable: why introduce spurious behaviour and make it more difficult to find a real bug? For such circumstances, one would just want to prove that the property is false (refutation). No witness for that is required.

A number of techniques for creating and combining over- and under-approximating abstractions have been proposed, e.g., [7, 9, 3, 15, 16]. In these approaches, model-checking yields either true or false, which are deemed to be conclusive, or maybe, in which case the abstraction needs to be refined. While all aspects of the CEGAR framework for such abstractions have been described theoretically [9, 3, 10, 15, 16], these ideas have not yet been implemented.

In this paper, we present YASM, which we believe to be the first symbolic software model-checker based on combining over- and under-approximating abstractions, which we refer to as *exact* [11]. It can prove and disprove properties with equal effectiveness. Our experiments [11] show that performance of the tool is comparable with standard over-approximating model-checkers. Moreover, we found that exact abstractions can become part of the standard CEGAR framework virtually without modifications and, more importantly, minor modifications of the framework enable an array of useful analyses, e.g., reasoning about the entire CTL, reusing previously computed abstractions, and many others.

The rest of the paper is organized as follows: Sec. 2 describes the design and the current state of the tool. Sec. 3 discusses the above observations. We conclude in Sec. 4.

2 Design and Implementation

YASM is based on the standard CEGAR loop.

Abstractions. Following SLAM, abstractions are represented by Boolean Programs. Unlike SLAM, the semantics of these programs is given via a variant of Mixed Transition Systems (MixTS) [7, 11]. Compared to Modal and 3-valued TSs, MixTSs allow for

(a)

```
1: int p1,p2,p3,x,y;
2:  p1=p2=p3=x=y=5;
3:  if(p3<=0) return;
4:  if(y<0)
5:    {if(x>2){
6:       if(y>10)
7:         {if(p2>0)
8:            ERROR;}}
9:     if(p1>0) ERROR;
10: } else {
11: if(p2>0)
12:   {if(p1>0) x = x+1;
13:    if(p2>0) x = x+2;
14:    if(p3>0) x = x+3;
15:    if(x>40)
16:      if(p1>0) ERROR;
17:    if(p1>0) ERROR;}}
```

(b)

```
1: bool b1,b2,b3;
2:  b1=b2=b3=1;
3:  if (b3) return;
4:  if(*)
5:    {if(*){
6:       if(*)
7:         {if(b2)
8:            ERROR;}}
9:     if(b1) ERROR;
10: } else {
11: if(b2)
12:   {if(b1) ;
13:    if(b2) ;
14:    if(b3) ;
15:    if(*)
16:      if(b1) ERROR;
17:    if(b1) ERROR;}}
```

Fig. 1. (a) A C program. (b) An abstraction of (a) using predicates `b1`:{`p1>0`}, `b2`:{`p2>0`}, `b3`:{`p3<=0`}.

a monotonic refinement of abstractions, yet are simpler to encode symbolically then disjunctive (or hyper-) TSs [16, 8]. We employ the standard technique for extracting an abstraction of a program by approximating weakest precondition of program statements. However, to allow for analysis of concurrent systems, we differentiate between program non-determinism and abstraction-induced non-determinism, both syntactically, in the Boolean Programs, and semantically, in MixTSs [11].

Model-Checking. YASM uses a specialized BDD-based symbolic model-checker, described in [4, 3]. The May and Must transition relations of a MixTS are encoded in a single BDD, and each 3-valued predicate is represented by two BDD variables.

Counter-examples. When a property is inconclusive, the model-checker generates a proof of this fact that is mined for new predicates. For safety properties, this proof, described in [10, 5], can be expanded into a standard counter-example – making all standard predicate discovery techniques applicable. However, unlike standard approaches, it does not need to be simulated in order to determine its feasibility.

Architecture and Implementation. The tool is written in Java (around 100K lines of code not including third-party components). It makes use of several tools including: CIL [13] for parsing and simplifying C code, CUDD BDD library [17] for decision diagrams, and CVCLite [2] for theorem proving, and we are currently working on the integration with Eclipse IDE.

YASM has been in operation for about a year and a half, and since that time has been used to check C programs up to 30K lines of code: network protocols, programs from the OpenSSH package, parts of Linux file system, etc. The tool is publicly available from `http://www.cs.toronto.edu/~arie/yasm`.

3 YASM with CEGAR Framework

The main advantage of exact (or even under-approximating) abstraction is its ability to refute properties. Consider abstracting a program shown in Fig. 1(a). Its abstraction using predicates `b1`, `b2`, `b3` (See Fig. 1(b)) is sufficient for YASM to conclude that `ERROR` is reachable. However, this abstraction is insufficient for an over-approximating model-checker: the shortest path to `ERROR` (line 9), the one typically found by a model-checker, is spurious.

YASM succeeds because it partitions the abstract states into: (a) states from which ERROR is unavoidable (A), (b) states from which ERROR is unreachable (B), and (c) states that have a (potentially spurious) path to ERROR (C). If the initial state belongs to either A or B, the result is conclusive; otherwise, a path to ERROR is available to guide the refinement process. Note that an over-approximating analysis combines A and C, and under-approximating combines B and C.

In the remainder of this section, we show how knowing the set A changes the dynamics of the CEGAR framework.

Aggressive Abstraction. Compared to an over-approximating model-checker, YASM's bug-detecting ability is preserved even in the face of a very aggressive abstraction. For example, when conditions of the if-statements at lines 12–14 are abstracted away, i.e., replaced by *, the resulting abstraction has more spurious paths, and yet YASM is still able to conclude that ERROR is reachable. This allows us to augment the CEGAR framework to prefer a more aggressive (and computationally cheaper) abstraction and employ heuristics during the refinement stage to decide between increasing precision of the abstraction and adding new predicates.

Shallow Counterexamples. If we restrict our abstraction to predicates `b1` and `b2`, YASM can show that ERROR is unavoidable from line 4. Yet the overall analysis is inconclusive due to a spurious counterexample: a path to ERROR on line 9. This path can be eliminated using new predicates `y<0` and `x>2`. Using the fact that error is unavoidable from line 4, we can instead: (a) only generate the counterexample up to that line, and (b) discover that we need the predicate `b3:`$\{$`p3<=0`$\}$ to finish the analysis.

Reusing Previous Results. The set A can also be reused between successive iterations of the CEGAR loop. Once an abstraction is refined, we can check for reachability of A, instead of ERROR. For example, after analyzing an abstraction restricted to the predicate `b1`, we know that ERROR is unavoidable frm $A = (pc \in \texttt{ERROR}) \vee (pc = 12 \wedge \texttt{p1>0})$, and can use the property $EF\ A$ instead of $EF(pc \in \texttt{ERROR})$ in all successive iterations.

Note that a combination of an aggressive abstraction and reuse of previous results achieves a similar effect to Lazy Abstraction employed by BLAST– only the parts of the program relevant to the analysis are actively refined. Furthermore, by changing the property at each refinement step, we can guide the refinement process to the *least spurious* execution, instead of the shortest one.

4 Conclusion

At each step of the abstraction/refinement loop, all abstractions get refined: either by removing possible behaviours for over-approximation, or by adding them to under-approximation. Clearly, combining both approaches allows substantial reuse of the analysis infrastructure and may lead to faster convergence of the analysis, since each step improves abstraction either towards truth, or towards falsity. This interplay also leads to many interesting analyses, some of which we've described in this paper.

The use of the exact abstraction further allows us to check arbitrary CTL properties. For example, we have successfully used YASM to prove non-termination (i.e., $EG\ \mathsf{true}$), and response (i.e., $AG(p \Rightarrow AFq)$) properties of C programs.

Finally, exact abstractions can precisely capture non-determinism present in concurrent programs. We have used YASM to check properties of the Bakery mutual exclusion protocol and error detection in RAX [14]. Our experiments look promising, yet more work is required to make YASM applicable to real-life concurrent programs written in fully-fledged programming languages such as C or Java.

Acknowledgments

We are grateful to Xin Ma, Kelvin Ku and Shiva Nejati for their help implementing, evaluating and improving YASM. Finantial support provided by NSERC and an IBM Ph.D. Fellowship are acknowledged.

References

1. T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". *STTT*, 5(1):49–58, 2003.
2. C. Barrett and S. Berezin. "CVC Lite: A New Implementation of the Cooperating Validity Checker". In *In CAV'04*, vol. 3114 of *LNCS*, pp. 515–518, 2004.
3. M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM TOSEM*, 12(4):1–38, 2003.
4. M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *CAV'02*, vol. 2404 of *LNCS*, pp. 505–509, 2002.
5. M. Chechik and A. Gurfinkel. "A Framework for Counterexample Generation and Exploration". In *FASE'05*, vol. 3442 of *LNCS*, pp. 217–233, 2005.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". *JACM*, 50(5):752–794, 2003.
7. D. Dams, R. Gerth, and O. Grumberg. "Abstract Interpretation of Reactive Systems". *ACM TOPLAS*, 2(19):253–291, 1997.
8. D. Dams and K.S. Namjoshi. "The Existence of Finite Abstractions for Branching Time Model Checking". In *LICS'04*, pp. 335–344, 2004.
9. P. Godefroid, M. Huth, and R. Jagadeesan. "Abstraction-based Model Checking using Modal Transition Systems". In *CONCUR'01*, vol. 2154 of *LNCS*, pp. 426–440, 2001.
10. A. Gurfinkel and M. Chechik. "Proof-like Counterexamples". In *TACAS'03*, vol. 2619 of *LNCS*, pp. 160–175, 2003.
11. A. Gurfinkel and M. Chechik. "Why Waste a Perfectly Good Abstraction?". In *TACAS'06*, 2006. To appear.
12. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy Abstraction". In *POPL'02*, pp. 58–70, 2002.
13. G. Necula, S. McPeak, S. Rahul, and W. Weimer. "CIL: Intermediate Language and Tools for Analsysis and Transformation of C Programs". In *In CC'02*, vol. 2304 of *LNCS*, pp. 213–228, 2002.
14. C. Pasareanu, R. Pelanek, and W. Visser. "Concrete Model Checking with Abstract Matching and Refinement". In *CAV'05*, vol. 3576 of *LNCS*, pp. 52–66, 2005.
15. S. Shoham and O. Grumberg. "A Game-Based Framework for CTL Counter-Examples and 3-Valued Abstraction-Refinement". In *CAV'03*, vol. 2725 of *LNCS*, pp. 275–287, 2003.
16. S. Shoham and O. Grumberg. "Monotonic Abstraction-Refinement for CTL". In *TACAS'04*, vol. 2988 of *LNCS*, pp. 546–560, 2004.
17. F. Somenzi. "CUDD: CU Decision Diagram Package Release", 2001.