Implementing Abstraction Refinement for Model Checking in HOL

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. Abstracting infinite or large state spaces to ones feasible for model checking has met with much success. We have implemented an abstraction framework in HOL, on top of a deep-embedded model checker. We present the implementation, highlighting the role of HOL.

1 Introduction

Model checking and theorem proving are two complementary approaches to formal verification. Model checking models the system as a state machine and desired properties of the system are expressed as temporal logic formulae that are true in the desired states of the system. Verification is fully automatic and can provide counter-examples for debugging but suffers from the state explosion problem when dealing with complex systems. Theorem proving models the system as a collection of definitions and desired properties are proved by formal derivations based on these definitions. It can handle complex systems but requires skilled manual guidance for verification and human insight for debugging.

An increasing amount of attention has thus been focused on combining these two approaches (see [17] for a survey). In this paper we demonstrate an approach to embedding a model checker in a theorem prover. The expectation is that this will ease combination of state-based and definitional models and the respective property checking techniques. Model checkers are typically written in tightly optimised C with an emphasis on performance. Theorem provers typically are not. Preliminary benchmarking shows that the loss in performance using our approach is within acceptable bounds.

Since our emphasis is on security (in the sense of soundness not being compromised), we have chosen the HOL theorem prover [12] for our task. HOL is based on the HOL logic [11] which is an extension of Church's simple theory of types [4], and is written in Moscow ML. Terms (of ML type `term`) in the logic can be freely constructed. Theorems (of ML type `thm`) can be constructed using the core axioms and inference rules only, i.e. by proof. This reliance on a very small trusted core is often named the "fully-expansive" approach and gives a high assurance of security.

Symbolic model checking [13] is a popular model checking technique. Sets of states are represented by the BDDs [3] of their characteristic functions. This representation is compact and provides an efficient[1] way to test set equality and do

[1] The problem is NP-complete. So this efficiency is of heuristic value only.

image computations. This is useful because evaluating temporal logic formulae almost always requires a fixed point computation that relies on image computations to compute the next approximation to the fixed point and a set equality test to determine termination. Most of the work is done by the underlying BDD engine. Current model checking techniques are typically unable to verify real-world examples due to the large number of states involved. Abstraction [8, 15, **?**,6] is considered a promising approach in handling this problem.

By representing primitive BDD operations as inference rules added to the core of the HOL theorem prover, we can model the execution of a model checker for a given property as a formal derivation tree rooted at the required property [1]. These inference rules are hooked to a high performance BDD engine [**?**] external to the theorem prover. Thus the loss of performance is low, and the security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound. Since we do almost everything within HOL and use only the most primitive BDD operations, we expect a higher assurance of security than from an implementation that is entirely in C.

We now build upon this embedded model checker embedded in HOL by adding an abstraction framework. As before, the fully-expansive approach is used and all steps in the computation are justified by HOL proofs. This retains the high assurance of soundness, and also allows us to use the decision procedures and simplifiers of HOL without loss of compositionality.

2 Abstraction Refinement in HOL

Abstraction techniques reduce the number of states of a system so that it is more amenable to model checking. This is typically done using functional abstraction [6] or Galois connections [**?**]. Here we consider the functional abstraction approach supplemented with a refinement framework [5]. A functional abstraction typically computes an abstraction function h that is a surjection from the set of states of the system under consideration to the domain of abstract states.

We need to make the notion of "system" precise. For our purposes, the system is represented by a Kripke structure. If AP is the set of atomic propositions relevant to the system we wish to model, then a Kripke structure over AP is defined as follows:

Definition 1. *A Kripke structure M over AP is a tuple (S, S_0, T, L) where*

- *S is a finite set of states.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *T is the set of actions (or transitions or program letters) such that for any action $a \in T$, $a \subseteq S \times S$.*
- *$L : S \rightarrow 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

Each $p \in AP$ is a proposition constructed from the variables v of some finite domain D_v and the constants and operators over that domain. Let $V =$

$\bigcup_{p \in AP} freevars(p)$ and $n = |V|$. Thus the set $S = D_{v_0} \times D_{v_1} \times \ldots D_{v_{n-1}}$. A state $s \in S$ in M is thus a tuple over V. We write $s \vDash p$ if p is true when the variables of p are assigned the corresponding values from the state s.

If there is a transition a taking state s to state s', we write $R_a(s, s')$. The notation $R(s, s')$ indicates that there is some transition from s to s'. The value of a variable v in the next state is denoted by v'. For each transition a, the transition relation R_a is given by $\bigwedge_{i=0}^{n-1} v'_i = f_i(s)$ for synchronous and $\bigvee_{i=0}^{n-1} v'_i = f_i(s)$ for asynchronous systems, where the f_i are *next state formulae* over the variables of s that determine what the value of v_i should be in the next state. We write $M \vDash \phi$ if the temporal property ϕ constructed over the p_i holds in all states of M.

The abstraction function $h : S \rightarrow \hat{S}$ is a surjection to the set $\hat{S}$ of abstract states. In general we can use h to compute the abstract model $\hat{M} = (\hat{S}, \hat{S_0}, \hat{T}, \hat{L})$ as follows[2]:

1. $\hat{S}$ is the set of abstract states $\hat{s}$ where the $\hat{s}$ are partitions of S
2. $\hat{S_0}(\hat{s}) = \exists s.h(s) \iff \hat{s} \wedge s \in S_0$
3. $\hat{R}(\hat{s_1}, \hat{s_2}) \iff \exists s_1 s_2.h(s_1) = \hat{s_1} \wedge h(s_2) = \hat{s_2} \wedge R(s_1, s_2)$
4. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$

This is called *existential abstraction.*

Theorem 2. *For any temporal property ϕ, $\hat{M} \vDash \phi \Rightarrow M \vDash \phi$*

Proof Sketch The abstraction adds extra behaviours to $\hat{M}$ that were not present in M. However it does not take away any behaviour i.e. it computes an over approximation. Thus if a property holds in the abstract it will hold in the concrete. If a property fails in the abstract it may be because of the spurious behaviour. □

If we limit ourselves to universal properties, then if a property fails in the abstract system we can generate a counterexample trace in the abstract system and attempt to find a corresponding concrete trace. If one exists than the property is false in the concrete system and the verification fails. Otherwise the abstraction is too coarse and we refine it by splitting some abstract state into smaller sets of concrete states. We then recheck the property. This is continued until either the property is verified or a concrete counterexample found. We are guaranteed termination because each refinement is strict (i.e. the sets resulting from splitting an abstract state are non-empty) and eventually we will end up with the concrete system that cannot be further refined. Figure 2 shows the overall framework.

2.1 Generating the initial abstraction

The first step is to generate the initial abstraction. First, we partition AP (and hence V) into sets that do not have a variable in common. This induces a partitioning on S. Then for each partition we group together into one abstract state all concrete states that have transitions to the same concrete states.

[2] NOTE: Sets in HOL are identified with higher-order predicates i.e. $x \in P \iff Px$. We will use the predicate notation where it is convenient.

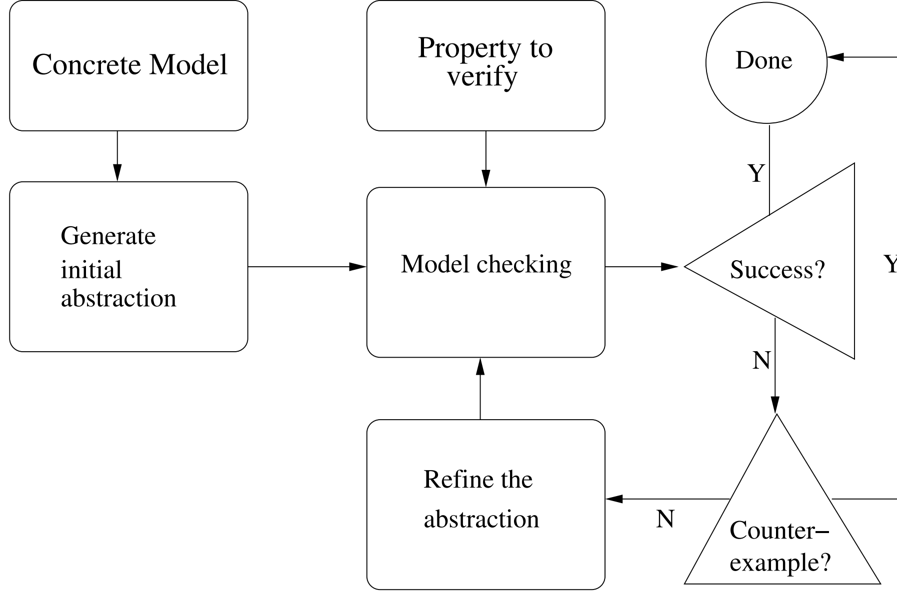

Fig. 1. Overview of Abstraction Refinement Framework

More precisely,

1. Let $(v_0, v_1, \ldots, v_n)$ be any state s_i.
2. Let I be the set of initial states and R be the transition relation given by $\bigwedge_i v'_i = f_i(s)$. Let F be the set of all next-state formulae f_i.
3. Let $f_i \equiv_f f_j \iff vars(f_i) \cap vars(f_j) \neq \emptyset$.
4. Let FC_i be the partitions $PART$ of F induced by $\equiv_f$.
5. Let $VC_i = \bigcup_{f \in FC_i} vars(f)$. Let $D_{VC_i} = \prod_{v_j \in VC_i} D_{v_j}$.
6. Let $h_i : D_{VC_i} \rightarrow \hat{D_{v_i}}$ where $\hat{D_{v_i}}$ is part of the abstract domain, be defined by $h_i(s_j) = h_i(s_k) \iff \forall f \in FC_i . s_j \models f \iff s_k \models f$
7. Then the abstraction function is $h = (h_0, \ldots, h_{|PART|-1})$.

The FC_i are called formula clusters, and the VC_i are called variable clusters. The domains D_{VC_i} represent the partitions of S. Note that the set of initial states S_0 is not used in the construction: $\hat{S_0}$ is simply the set $h(S_0)$ where h has been extended to sets in the obvious way. We can now use existential abstraction to construct $\hat{M}$.

2.2 Counterexample Detection

If the verification fails, the model checker implemented in [1] has the ability to generate a counterexample trace. A counterexample is a sequence of states starting with an initial state and following transitions to a state violating the property

being verified. Our technique for detecting whether a concrete counterexample exists is that presented in [7] and is best illustrated by an example.

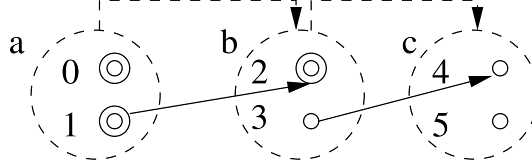

Fig. 2. Counterexample Detection Example

Figure 2.2 shows a system with $S = \{0, 1, 2, 3, 4, 5\}$. The abstract states are $\{a, b, c\}$ and the dashed circles indicate the concrete states they contain. Solid arrows represent transitions in the concrete system and dashed arrows represent transitions in the abstract system. The initial states are 0 and 1 and concentric-circles represent states reachable from the initial states. Note that the abstraction introduces extra behaviour by making states 4 and 5 reachable in the abstract system by making c reachable.

Suppose we check for a property P that holds in $\{0, 1, 2, 3\}$ but not in $\{4, 5\}$. We will get an abstract counterexample trace $\langle a, b, c \rangle$.

In general, given an abstract counterexample $\langle \hat{s_0}, \hat{s_1}, \ldots, \hat{s_k} \rangle$, we attempt to find a concrete counterexample $\langle s_0, s_1, \ldots, s_k \rangle$. To determine whether such a concrete trace exists, we try to find a satisfying assignment for the formula

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} h(s_i) = \hat{s_i}$$

using a SAT-solver.

If a concrete trace is found, the verification has failed and we are done. Otherwise we set $k = 0$ and attempt to find the longest prefix of $\langle \hat{s_0}, \ldots, \hat{s_k} \rangle$ for which there is a concrete trace by running the SAT tool on the formula above for increasing values of k.

In our example, there is no concrete counterexample and the longest prefix is $\langle a, b \rangle$. This information is then used to refine the state b into smaller abstract states.

2.3 Refining the Abstraction

We would like to find the coarsest possible refinement i.e. the fewest possible splits of the abstract state. This is an intractable problem. Our technique for refinement employs BDDs to get good results.

Consider Fig. 2.3 representing the same system as Fig. 2.2. We know that we need to refine abstract state b. Since the spurious behaviour is created by an

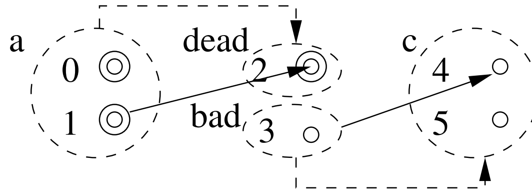

Fig. 3. Abstraction Refinement Example

unreachable state 3 in b having a transition to 4, we need to split all such states from the reachable states (in this case just $\{2\}$).

To do this, we compute the state sets

$$bad = \mathbf{EX}c = \{3\}$$

and

$$dead = b \backslash bad = \{2\}$$

where **EXf** computes all states such that there is a transition to a set in which the property represented by f is satisfied. In our case the property c yields precisely the set $\{4, 5\}$. This computation is done using standard BDD methods.

Intuitively the *dead* states are reachable "dead-ends" and the *bad* states are the ones causing the trouble by contributing to the creation of spurious behaviour. We can now replace the abstract state b in the abstraction by the abstract states *dead* and *bad* and repeat the procedure until the property is verified or a counterexample is found. In this case the abstract c is no longer reachable and this P is verified.

3 Implementation Issues

Figure 3 gives an overview of the implementation. It should be noted that the system is fully automatic and relies on fully-expansive proof at every step of the way.

The only exception to this are the dotted lines indicating the use of an external BDD engine. However, we use an LCF-style interface [10] to this engine which gives a higher assurance of soundness than integrating it as a one-shot proof rule as has been done in [14, 2].

The SAT engine we used is also external to HOL. However, checking the satisfiability of an assignment is in general much easier than finding such an assignment. Thus we use the SAT engine to obtain an assignment but check its validity by proof in HOL. Thus the use of the SAT engine does not risk introducing unsoundness in the system.

The ACTL to μ-calculus translation is not a requirement. However, recall that only universal properties can be used in this framework. The μ-calculus is a fairly non-intuitive logic and it is hard to manually check that a μ-calculus

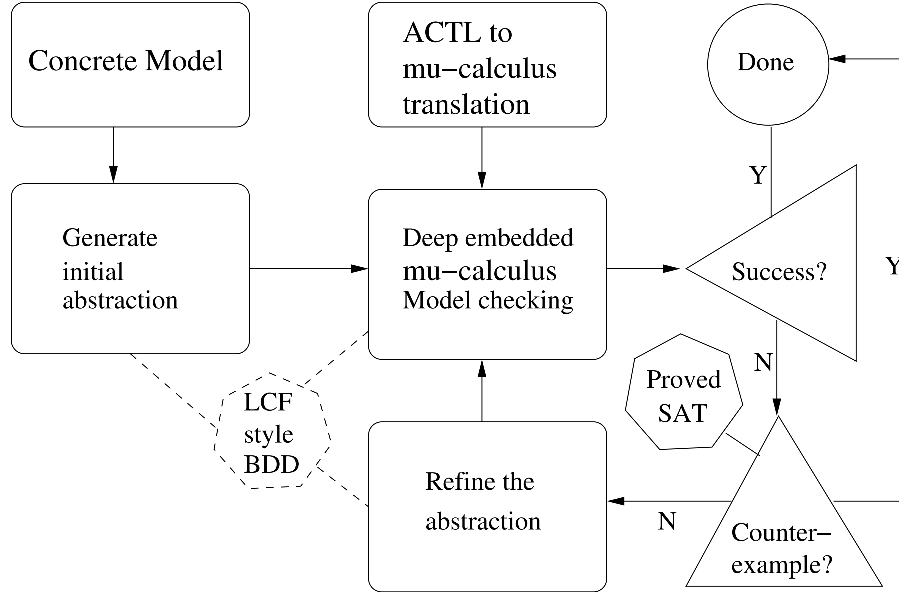

Fig. 4. Overview of Implementation in HOL

property is indeed universal. Thus we also accept properties in the more intuitive logic ACTL, which is a the universal fragment of CTL. Thus universality is enforced automatically because the HOL translation from CTL to the μ-calculus is done by proof based on the semantics of the two logics.

Note that step 6 of initial abstraction generation is in effect inducing equivalence classes of concrete states over each partition D_{VC_i} of S. When constructing a BDD representation of h to assist with the construction of $\hat{M}$ it is easier to compute these equivalence classes directly. However, the standard BDD methods for finding equivalence classed work by detecting strongly connected components (SCCs) in the graph representation of the model. SCC detection requires a total transition relation which is usual verifying properties in CTL. This is not guaranteed in our more general case with the μ-calculus. Thus we use the following algorithm for partitioning a given D_{VC_i} with respect to satisfiability.

The term $h_i(s_j) = h_i(s_k)$ can be considered as a relation $R'(s_j, s_k)$ on states. This relation induces the partitions we wish to compute. However R' can also be considered a transition relation on D_{VC_i}, with there being a transition between two states precisely when h_i agrees on them as defined by step 6 of section 2.1. Define the modality **EP** as the temporal inverse of **EX** i.e. $\mathbf{EP}f$ computes states such that there is a transition from states satisfying f to these states. Now we compute as follows:

1. Let $X = D_{VC_i}$. Let $P = []$.
2. If $X = \emptyset$ return P.

3. Let s be an arbitrary state in X (found using the BDD engines satisfiability finder).
4. Set $S = X, T = R'$ and use the model checker to compute $Y = (\mu Q.s \lor \mathbf{EX}q) \lor (\mu Q.s \lor \mathbf{EP}q)$. Then Y is the set of all states reachable from s plus all states from which s is reachable.
5. Let $P = Y :: P$
6. Let $X = X \backslash Y$ and repeat from step 2.

At the end of this the list P will contain the required partitions.

4 Related Work and Conclusion

Abstraction refinement for model checking in the context of theorem proving has also been done by [16, 2]. In [16] the abstraction framework is based on Galois connections and the refinement is done by adding the failed predicates from the previous proof attempt to get a richer abstract domain. Since the system is implemented as an atomic proof rule, access to the procedures and simplifiers in PVS itself for the purposes of the system cannot be done within the encompassing derivation tree. This inhibits a fully-expansive implementation of the system. It also restricts compositionality because for instance the system is unable to return a counterexample trace upon failure thus any counterexample guided predicate discovery system (e.g. [9]) cannot be used in this context.

In [2], a proof system for the μ-calculus relies on proof rules being executed by various decision procedures than can be plugged in according to need. This gives the framework great versatility in the choice of tools to be used to attack a given problem. Again, each proof rule is justified by an atomic call to a decision procedure and thus does not extend readily to a fully expansive approach.

Our approach is flexible in that any proof rule of HOL at any level of abstraction can be called upon at any time during the execution of the procedure. This enables us to provide the an entire run of the procedure as a derivation tree that can be plugged into any other proof. This high level of integration guarantees compositionality and makes the framework extensible. Thus we are able to use a BDD engine, a SAT engine and various HOL procedures in the same framework.

At the same time, the execution is fully expansive. All steps are accompanied by the application of a HOL proof rule. Thus we have a high assurance of soundness.

However, having to do fully expansive proof for the equivalent of fast BDD operations necessarily involves a performance penalty. To a great extent, we can ameliorate this by manipulating the higher-order equivalents of the propositional terms being manipulated by the BDD and SAT engines. Benchmarking for the current framework is pending some code optimisations. Benchmarking for the core model checker showed a performance penalty of about 20 percent as compared to the system with all proof machinery turned off [1]. This is acceptable in most situations, though the tradeoff with respect to increased assurance of soundness would need to be considered on a case-by-case basis.

In the current system the $p \in AP$ are restricted to propositional expressions. We hope to extend this to include Presburger formulae and – trading off some automation – full arithmetic and real numbers, leveraging the facilities in HOL for deciding these. Other future directions include support for other temporal logics and for other automatic abstraction and refinement techniques.

References

1. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In David A. Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.
2. S. Berezin. *Model Checking and Theorem Proving: a Unified Framework*. PhD thesis, Carnegie Mellon University School of Computer Science, 2002. Tool URL : http://www.cs.cmu.edu/~modelcheck/symp.html.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
5. E. M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification - (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
6. E. M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
7. E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. of Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.
9. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*. Springer, November 2002.
10. M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.
11. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
12. The HOL-4 Proof Tool. Tool URL http://hol.sf.net, 2003.
13. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
14. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking and automated proof checking. In Pierre Wolper, editor, *Proceedings of Computer Aided Verification*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

15. H. Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *LNCS*, pages 377–396. Springer, July 2000.
16. H. Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, jul 1999. Springer-Verlag.
17. T. E. Uribe. Combinations of model checking and theorem proving. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of the Third Intl. Workshop on Frontiers of Combining Systems*, volume 1794 of *LNCS*, pages 151–170. Springer-Verlag, March 2000.