Abstract

Complete verification of unmodified code is a challenging task, well-motivated by the costs of software debugging. In this work we rise to the challenge by proposing a model checking method that operates on unmodified parallel programs, specifically accepting LLVM bitcode as input. Apart from being complete, our method proves correctness of a program w.r.t. a temporal specification and is sound w.r.t. arithmetic overflows of integer variables. To overcome the limitations of classical model checking: state space explosion, state matching, etc. we further propose reduced to model checking problem to a specific instance of the non-termination checking and lift the recently proposed property directed reachability to compute approximations of recurrent sets.

1 Present State

Our tool accepts LLVM bitcode as the program and an LTL formula as the specification where the atomic propositions are quantifier free bit-vector formulae over global variables. Since variable evaluations are also represented as bit-vector formulae, comparing two states while searching for a fair cycle within the system can be reduced to satisfiability modulo theories query. The major limitation of the present state arises from this query being quantified, thus increasing the complexity of individual state comparisons.

1.1 Work Flow

The figure below illustrate how control explicit—data symbolic model checking transforms the input pair (program P, specification A) to generate the set-reduced transition system A*.

The set-based reduction using bit-vector formulae is not the only output of our tool SymDiVine [1]. We can also generate the state space encoded with BDDs or generate the control flow graph (supporting other tools with access to parallel programs).

1.2 Experiments

We have evaluated SymDiVine on examples translated from C programs. Apart from Erik Koskinen’s tool (reducing model checking to termination analysis) we also compare with NUxMV (using SMT-based bounded model checking), two state-of-the-art model checkers.

References


2 Avoid Loop Unrolling: Idea

Apart from expensive state comparison the present approach is also limited by the fact that program loops may be exhaustively unrolled during verification. A recently proposed property guided reachability (PDR) [2] avoids unrolling loops while also allowing extensions for LTL and CTL model checking by proving unreachability among fair states. We propose an alternative extension via non-termination analysis [3], specifically to lift PDR to compute approximations of recurrent sets of evaluations pertaining to fair states.

Formalisation

Definition: By Reach(l) we denote the set of evaluations at location l reachable from the initial location, i.e. \( \forall x. x \in \text{Reach}(l) \) there is a path from \( l \) to \( l \) that performs an identity on \( x \), i.e. \( (l, x) \mapsto (l, x) \).

Definition: By Recur(l) we denote a set of evaluations, such that \( \forall x \in \text{Recur}(l) \) there is a path from \( l \) to \( l \) that performs an identity on \( x \), i.e. \( (l, x) \mapsto (l, x) \).

Definition: Let \( X \) be a set of evaluations characterised by \( \Phi \), i.e. \( \forall x. \Phi(x) \Rightarrow x \in X \). Then by \( \downarrow X \) we denote any over-approximation of \( X \), i.e. \( \forall x. \Phi(x) \Rightarrow x \in \downarrow X \). And by \( \downarrow X \) we denote any under-approximation of \( X \), i.e. \( \forall x. \Phi(x) \Rightarrow x \notin \downarrow X \).

Theorem: For a program \( P \) and a specification \( \varphi \) let \( F \) be the set of fair locations. Then \( \forall l \in \downarrow F, \downarrow \text{Reach}(l) \cap \downarrow \text{Recur}(l) \neq \emptyset \Rightarrow P \not\models \varphi \) and dually \( \forall l \in \downarrow F, \downarrow \text{Reach}(l) \cap \downarrow \text{Recur}(l) \neq \emptyset \Rightarrow P \models \varphi \).

The above theorem leads to a model checking algorithms that iteratively refines the approximations of Reach(l) and Recur(l) for a fair location \( l \) until one of the termination conditions becomes valid.

3 Avoid Loop Unrolling: Implementation

As the driving force behind such iterative refinement we propose using PDR, lifted according to the following points:

- refine \( \text{Recur}(l) \) sets using counter-examples to recurrence (CtR1) and \( \text{Reach}(l) \) sets using counter-examples to induction;
- localise the refinements to program locations;
- extend the method to the bit-vector theory (similarly as in [4]).

Under-approximation Search for under-approximations (w.r.t. a fair location \( l \)) has two cooperating stages:

1. standard PDR with \( P \equiv \set{x} \) describing the safe states \( \Rightarrow x \in \text{Reach}(l) \);
2. PDR with \( P \equiv \set{x} \), initial state set to \( (l, x) \), and limited to the strongly connected component containing \( l \).

Over-approximation The standard definition of the recurrent set requires quantifier alternation: \( L. X. X' \) be the sets of program variables for the current and the next state \( \Phi(x) \).\( \Rightarrow F \) and \( \Phi(x) \).\( \Rightarrow G(x) \).

It follows that a CtR is any evaluation without either a predecessor or a successor. The figure below illustrates the search for single-step CtRs.

References