On-the-fly Parameterized Boolean Program Exploration

Peizun Liu and Zhaoliang Liu
Northeastern University, USA, {lpzun, zliu}@ccs.neu.edu

***Abstract*—Reachability analysis for replicated Boolean programs run by an unbounded number of threads is decidable in principle via a reduction of the Boolean program families to *well-structured transition systems* (WSTS). The obtained transition systems would, however, in general be intractably large, due to local state explosion. Basler et al. give an *on-the-fly* algorithm that solves this problem for Boolean programs run by a fixed, finite number of threads [1]. In this paper, we extend this idea to families with unbounded thread counts, based on Abdulla's backward reachability analysis [2]. The challenges are to traverse the Boolean programs *backwards*, computing infinite-state *covering* pre-images, all while building the transition system being explored on the fly, to avoid local state explosion.**

I. Introduction

We consider the class of replicated finite-state programs executed by an unbounded number of threads. An important practical instance of this class is given by non-recursive concurrent Boolean programs with dynamic thread creation. Safety properties of such programs can in principle be checked via reduction to the *coverability problem for WSTS*.

An approach to solving the latter problem is the backward reachability algorithm proposed by Abdulla et al. [2]. To apply this algorithm to Boolean program families, these families need to be translated into infinite-state machines. This translation can cause a huge blow-up, triggered by the need to track a thread's local state changes in updates to local state counters. Basler et al. [1] proposed a context-ware strategy, which performs the translation *on the fly*, instead of constructing the entire transition model upfront. Their method is, however, restricted to finite thread families.

Contribution. In this work, we present a novel, sound and complete solution to this problem, by extending the on-the-fly idea from bounded to unbounded thread counts. State space exploration is performed using *covering pre-images* [2], which traverse the infinite state space backwards. We then propose two optimizations to limit the size of obtained covering pre-images. To our knowledge this is the first work that applies on-the-fly techniques to sound and complete infinite Boolean program safety checking problems.

II. Preliminaries

A. Basic Definitions

Let $\mathbb{B}$ be a non-recursive Boolean program. We adopt the Boolean program syntax from [3]; conditionals and loops are encoded implicitly using `assume` and `goto` statements. We sketch how $\mathbb{B}$ gives rise to a concurrent system $\mathbb{B}^n$ (a full formalization is given in [1]). Let pc be the program counter (PC) and V be set of program variables. V is partitioned into two subsets V_S and V_L of shared and local variables. Variables can evaluate to 0, 1 and $\star$, representing *true*, *false* and *non-deterministic*. Let $pc_{\max}$ be the maximal program counter, S and L be the finite sets of shared and local states, respectively. The elements of $T := S \times L$ are called *thread state*, the elements of $U := \cup_{n=1}^{\infty} S \times L^n$ are called *global state*. A global state of $\mathbb{B}^n$ is written as $\tau := (s, \ell_1, \ldots, \ell_n)$, where s is a shared state and ℓ_i is a local state. To distinguish the valuation of pc and local variables from ℓ_i, we denote a valuation for V_L as a *local configuration*.

B. Context-aware Counter Abstraction

In this paper, we borrow the idea of compact symbolic counter abstraction which is given in [1]. The merits of that are not only ignore the *zero* counters but also able to interpret the value $\star$ symbolically, as the set $\{0, 1\}$. The representation is shown as follows:

$$\tau := \langle s, (\ell_1, n_1), \ldots, (\ell_k, n_k) \rangle$$

In this notion, n_i is a natural number which record the number of threads reside in ℓ_i.

We sketch the idea of on-the-fly exploration as follows: suppose current to explore state is τ, a thread is picked to execute. Its thread state is translated into a Boolean program $\mathbb{B}$'s state: PC, the valuations for shared variables and local variables. Then $\mathbb{B}$ is executed 1 step forward on this thread state. The execution result is converted back into a thread state and merged into τ.

C. Well Quasi-ordering

BWRA operates on WSTS [2]. A WSTS is a transition system equipped with a well-quasi-ordering $\preceq$ on its states that satisfies the monotonicity property. To show $\mathbb{B}$ induces a WSTS, defining $\preceq$ as follows:

$$\langle s, (\ell_1, n_1), \ldots, (\ell_k, n_k) \rangle \preceq \langle s', (\ell_1, n_1'), \ldots, (\ell_k, n_k'), \ldots) \rangle$$

whenever $s = s'$ and $\forall 1 \leq i \leq k$: $n_i \leq n_i'$. We say τ' covers τ if $\tau \preceq \tau'$. Relation $\preceq$ is a quasi-order since it is neither symmetric nor anti-symmetric. In fact, relation $\preceq$ is a well quasi-order on U: any infinite sequence $\tau_1, \tau_2, \ldots$ of element from U contains an increasing pair $\tau_i \preceq \tau_j$ with $i < j$. It is easy to see that $\mathbb{B}$ induces a WSTS.

Research supported by the National Science Foundation, grant no. 1253331.

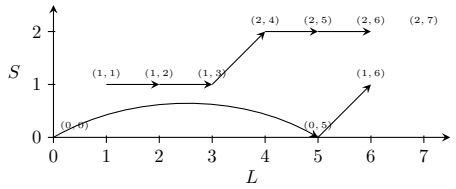

Fig. 1. An example of illustrating why restrict PCs

For any state τ, we define $\hat{\tau} := \{r \mid \tau \preceq r\}$. The *covering predecessors* of τ can be defined as follows: $\mathsf{CovPre}(\tau) := \{r \mid \exists r \longrightarrow \tau', \tau' \in \hat{\tau}\}$, and $\mathsf{C\text{-}Pre}(\tau) := \min\{r : r \in \mathsf{CovPre}(\tau)\}$.

III. On-the-fly Backward Exploration

A. BWRA in Boolean Program

BWRA can be used in Boolean programs using the definition of $\preceq$. However, how to compute C-Pre in a Boolean program is another impediment since we cannot do so by backward-executing the program as the forward search does.

1) Control Flow Graph and Weakest Precondition: In this project, we propose to compute C-Pre based on *control flow graph* (CFG) and *weakest precondition* (WP). CFG is a directed graph constructed from the program's execution flow. The nodes and edges corresponds to PCs and execution flows respectively. Given a statement $stmt_p$ with $pc = p$ and a thread state (s, ℓ), in this paper, WP of $stmt_p$ is a function mapping (s, ℓ) to potential predecessors, which is denoted as $\mathsf{WP}(stmt_p, (s, \ell))$. We use CFG to navigate the backward search, and weakest precondition to overapproximate C-Pre.

2) On-the-fly Exploration: Suppose current state is $\tau = \langle s, F\rangle$ with $F = \{(\ell_1, n_1), \ldots, (\ell_k, n_k)\}$. Let $\Gamma = \{\ell_1, \ldots, \ell_k\}$, CFG $G = (V, E)$, then on-the-fly BWRA algorithm executes as follows: for each $\ell \in L$, $\ell.pc = q$, if $\exists (p, q) \in G.E$, locate $stmt_p$, and then

(1) if $\ell \in \Gamma$ and $\mathsf{WP}(stmt_p, (s, \ell)) \neq \emptyset$, suppose $(s', \ell') \in \mathsf{WP}(stmt_p, (s, \ell))$, then $\langle s', F'\rangle \in \mathsf{C\text{-}Pre}(\tau)$, where $F' = F\backslash\{\ell, 1\} \cup \{\ell', 1\}$.
(2) if $\ell \in L\backslash\Gamma$ and $\mathsf{WP}(stmt_p, (s, \ell)) \neq \emptyset$, suppose $(s', \ell') \in \mathsf{WP}(stmt_p, (s, \ell))$, then $\langle s', F \cup \{(\ell', 1)\}\rangle \in \mathsf{C\text{-}Pre}(\tau)$.

The above algorithm does the translation of Boolean program to thread state on-demand. However, iterating over L to compute $\mathsf{C\text{-}Pre}(\tau)$ is inefficient due to (1) the local state explosion; and (2) almost all of the thread states have predecessors. Hence, the number of elements in $\mathsf{C\text{-}Pre}(\tau)$ would be exponential in $2^{|V_L|} \times |pc_{\max}|$.

B. Local States Reduction

Since a local state is comprised by two parts – the local configuration and PC – in order to restrict both or either of them if we want to improve the efficiency.

1) Restricting PCs: Scrutinizing the features of Boolean program, we can find that there are a lot of transitions happened between two thread states with same *share state*. These transitions are induced by `assume`, `goto` and `skip` statements (they do not change anything but PC) or `assignment` statements whose *left sides* only involve local variables. If doing BWRA goes along such a transition, then it is unhelpful to close to the termination. We can easily see the reason by the example shown in Figure 1: when doing BWRA from thread state $(2, 7)$ following algorithm presented in section III-A2, $\mathsf{C\text{-}Pre}((2,7))$=$\{(2|4, 7), (2|5, 7), (2|6, 7)\}$ due to all of the WP of $(2, 4)$, $(2, 5)$ and $(2, 6)$ are not empty. However, we can see that both $(2|5, 7)$ and $(2|6, 7)\}$ have direct predecessor $(2|4, 7)$ due to existing horizontal transitions. Their appearance are unnecessary. So we can safely ignore such PCs without influencing the final result.

We restrict PC to the ones whose predecessors point to the statements assigning values to share variables. For any $pc = q$ in a CFG $G = (V, E)$, we have:

- if $\exists\ (p, q) \in G.E$ and $stmt_p$ is an `assignment` statement changing shared variables, then consider q.
- otherwise, ignore q.

2) Local Configurations Reachability Analysis: We say a local configuration c is *reachable* if there exists a reachable local state contains it. If the unreachability of c we known a priori, then all of local states contain c can be safely removed from L. In general, detecting the reachability of c is a model checking problem. However, we do not need know the exact set of reachable cs; any over-approximation suffices.

To do this work, we partition the local variables into two types *independent* and *dependent*.

Definition 1. *An independent local variable $v \in V_L$ in Boolean program $\mathbb{B}$ is defined as follows:*

$$v := \top \mid \bot \mid R(v, \ldots, v)$$

where R is a propositional formula.

If v is not independent, then it is *dependent*. The set of reachable local configurations can be approximated in several ways. One solution is to execute $\mathbb{B}$ with only a single process. Guards are likewise replaced by true or false. Then,

- if v is independent, its value equals to execution result;
- if v is dependent, then v is non-deterministic.

Another technique to approximate reachable local configuration is borrowed from compilers, which sometimes optimize program behavior by confining the number of values that a local variable can have at some program point. Emerson and Wahl discussed this technique detailedly in [4].

References

[1] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Context-aware counter abstraction," *Form. Methods Syst. Des.*, vol. 36, no. 3, pp. 223–245, Sep. 2010.
[2] P. A. Abdulla, "Well (and better) quasi-ordered transition systems," *Bulletin of Symbolic Logic*, vol. 16, no. 4, pp. 457–515, 2010.
[3] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *Proc. 7th SPIN Workshop on SPIN Model Checking and Software Verification*, 2000, pp. 113–130.
[4] E. A. Emerson and T. Wahl, "Efficient reduction techniques for systems with many components," *Electron. Notes Theor. Comput. Sci.*, vol. 130, pp. 379–399, May 2005.