Towards Smaller Invariants for Proving Coverability *

Lenka Turoňová and Lukáš Holík

Faculty of Information Technology, Brno University of Technology
{turonova,holik}@fit.vutbr.cz

Abstract. In this paper, we explore a possibility of improving existing methods for verification of parallel systems. We particularly concentrate on safety properties of *well-structured transition systems*. Our work has relevance mainly to recent methods that are based on finding an *inductive invariant* by a sequence of refinements learned from counterexamples. Our goal is to improve the overall efficiency of this approach by concentrating on choosing refinements that lead to a more succinct invariants. For this, we propose to analyze so called *minimal counterexample runs*. They are digests of counterexamples concise enough to allow for a more detailed analysis. We experimented with a simple refinement algorithm based on analysing minimal runs and succeeded in generating significantly more succinct invariants than the state-of-the-art methods.

1 Introduction

Verification of parallel programs is challenging since they can generate a huge number of interleavings. This is called a *state space explosion* (the size of the state space grows exponentially to the number of processes). On top of that, the number of parallel processes may be unbounded or processes might be dynamically created which render the state space infinite. However, a large class of the parallel programs can be understood as *well structured transition systems* (WSTS) where many properties can be effectively verified. The class of WSTS include for instance Petri nets, lossy channel systems, or various broadcast and mutual exclusion protocols. We are interested especially in safety properties of WSTS, particularly in those that can be formulated as *coverability* of a set of incorrect configurations.

Our work is in the direction originating from or conceptually similar to backward reachability analysis [1], especially close to recent works [8, 7, 5] that are based on learning a safe inductive invariant of the similar form. A safe inductive invariant is a set of configurations of the system with the three following properties: (a) it contains its initial configurations; (b) it does not intersect with the target configurations; and (c) it is closed under the transition relation. The properties together are an inductive proof of safety of the system.

* This work was supported by the Czech Science Foundation project 16-24707Y.

The methods [8, 7, 5] build-up the invariant by iterative steps, refining the current invariant approximation in order to satisfy inductiveness. A crucial component of this process is always a use of abstraction used to accelerate the process and regulated by a variation on counterexample-guided refinement (CEGAR).

The basic variant of CEGAR [4] runs the program within the abstract domain and in the case of reaching the target, the path to the target, so called *counterexample*, is analyzed. If the path is feasible in the real system then the target is reachable. Otherwise, the counterexample is spurious due to a too coarse abstraction and the path is used to refine the abstraction.

The counterexample analysis of [8, 7, 5] is based on forms of backward exploration of the state space starting in the target, using the operation *pre*. Although the methods differ in many aspects, it can be said that all of them perform essentially an eager backward exploration of the state space—they do not take into account any counterexample path in particular. Almost all found configurations (modulo some local and indeed powerful optimizations such as the "generalization" of algorithms [8] and [7]) are being used to refine the representation of the invariant approximation. Due to the eagerness, much of the added precision is unnecessary and makes the inferred invariants much more verbose than needed.

We conjecture that heuristics for finding more succinct invariants would improve efficiency of the discussed methods [8, 7, 5] significantly, for the following reasons: the invariant approximations are the most costly data that the methods work with, and so the price of the analysis depends directly on the size of the invariant representation. Moreover, more succinct invariants can be usually found by less invariant building steps.

Our approach. Our work comes out from the work [5], referred to as GBR. Its backward counterexample analysis is based on an exhaustive backward search through the state space implemented using the precise precondition operator.

We propose a modification of GBR which instead of the exhaustive depth-bounded backward search concentrates on a single entire counterexample path called *minimal counterexample run*. Since it is usually very concise we can analyze it thoroughly. From each such a run, we try to derive a "reason" for why the run could not be executed in the real system. Intuitively, it consists of parts of preconditions of transitions in the run that cannot occur (similarly to interpolants [9]). The minimal reason has a potential to be a part of the minimal inductive invariant since 1) it is necessary to refute the examined spurious counterexample run and 2) it is a "minimal" such "reason". In contrast to the eager strategy, many useless candidates for parts of the invariant and unsafe overapproximations can be avoided in this way.

2 Well-quasi-ordered Transition Systems

The well-quasi-ordered transition systems, *WSTS* for short, are systems with infinitely many configurations with a well-quasi-ordering (*wqo*), and whose transitions satisfy the monotonicity property. It has been shown that the coverability problem (reachability of an UCS) is decidable for *WSTS* [2].

Formally, according to [8], *WSTS* is a tuple $(\Sigma, I, \rightarrow, \succeq)$ where Σ stands for a set of configurations, a finite set $I \subset \Sigma$ is a set of initial configurations, $\rightarrow \subset \Sigma \times \Sigma$ is a transition on Σ, and a $\succeq \subset \Sigma \times \Sigma$ is a *wqo*.

The transition relation $\rightarrow$ between configurations is *monotonic* wrt. to the relation $\succeq$ if for each two configurations s_0 and s_1 such that $s_0 \succeq s_1$ and a relation $s_0 \rightarrow s_0'$ there is a configuration s_1' such that $s_1 \rightarrow s_1'$.

The pre-order $\succeq$ is defined over a set S such that for any infinite sequence $s_0, s_1, s_1, \ldots$, there are i, j with $i < j$ and $s_i \succeq s_j$. If $\succeq$ is an equivalence relation, then the condition of $\succeq$ being a *wqo* amounts to the equivalence relation having a finite index [1].

Given a pre-order $\succeq$ defined over a set S set of configurations S, the upward-closure $T\uparrow$ of a subset $T \subseteq S$ and the downward closure $T\downarrow$ are defined as

$$T\uparrow \stackrel{\text{def}}{=} \{s \in S \mid \exists t \in T : t \succeq s\} \quad \text{and} \quad T\downarrow \stackrel{\text{def}}{=} \{s \in S \mid \exists t \in T : s \succeq t\}.$$

We define a set T to be an UCS, respectively a downward closed set (DCS), iff $T\downarrow = T$, respectively $T\uparrow = T$. If T is an UCS, its complement $S \backslash T$ is a DCS, and, conversely, if T is a DCS, its complement is an UCS [3]. Based on the monotocity of $\rightarrow$, for any UCS, the set of its predecessors is an UCS.

Let x, $y \in \Sigma$. If $x \rightarrow y$ we call x a *predecessor* of y and y a *successor* of x, and define

$$pre(x) := \{y \mid y \rightarrow x\} \quad \text{and} \quad post(x) := \{y \mid x \rightarrow y\}.$$

For $X \subset \Sigma$, $pre(X)$ and $post(X)$ are defined as usual, i.e.

$$pre(X) = \bigcup_{x \in X} pre(x) \quad \text{and} \quad post(X) = \bigcup_{x \in X} post(x).$$

For sets X and Y of configurations, we use $X \rightarrow Y$ to denote that there are $x \in X$ and $y \in Y$ such that $x \rightarrow y$. If there are configurations $x_0, \ldots, x_k \in \Sigma$ such that $x_0 = x, x_k = y$ and $x_i \rightarrow x_{k+1}$ for $0 \leq i < k$, then we write $x \xrightarrow{k} y$. Furthermore, $\xrightarrow{*}$ represents the reflexive transitive closure of $\rightarrow$. A set X of configurations is said to be reachable if $X_{init} \xrightarrow{*} X$. The set of *k-reachable* configurations, reachable in at most k steps, is defined as:

$$Reach_k := \{y \in \Sigma \mid \exists k' \leq k, \exists x \in I, x \xrightarrow{k'} y\}.$$

The set of all reachable configurations is formally defined as:

$$Reach := \bigcup_{k \geq 0} Reach_k = \{y \in \Sigma \mid \exists x \in I, x \xrightarrow{k} y\}.$$

Given a $WSTS$ $S = (\Sigma, I, \rightarrow, \succeq)$, we denote by $Cover$ the covering set of S, consisting of all configurations covered by some of the reachable configurations:

$$Cover(S) \stackrel{\text{def}}{=} post^*(I)\downarrow.$$

3 Testing Coverability Using CEGAR

Let us have given a WSTS $S_0 = (\Sigma, I, \rightarrow, \succeq)$ and a target configuration **bad**. The coverability problem means to decide whether a configuration from the set **bad**$\uparrow$ is reachable from the set of initial configurations I or not. A system where **bad**$\uparrow$ is not reachable is called safe. Formally, we say that the set **bad** is coverable if **bad** $\in Cover(S_0)$.

The method of [5] on which we build uses abstract interpretation with the abstraction defined by a set of configurations D, called domain. Every configuration $x \in D$ gives the abstraction means to distinguish configurations in $x\uparrow$ from the rest. We say that D is "expressive enough" when it can express a safe and inductive invariant, that is, if it has a subset V such that the complement of $V\uparrow$ is a safe inductive invariant. All runs of the system then stay within the complement of $V\uparrow$.

The domain is refined using CEGAR until unreachability or reachability of **bad**$\uparrow$ is proven. One iteration of the CEGAR loop takes the current value of D and generates an abstract forward run until it reaches a fixpoint, or until **bad** is reached. If the fixpoint is reached without reaching **bad**, then **bad** is not coverable. If it generates **bad**, then it is a counterexample run starting from the complement of $D\uparrow$ and ending to the target **bad**.

A counterexample run either signifies that **bad** is indeed reachable, or it is a spurious counterexample. A spurious counterexample is generated due to the abstraction not being able to distinguish certain dangerous configurations that can reach **bad** from some configurations that are reachable. The abstraction has then to be refined. This is done by adding the configurations from the counterexample run to D in order to give the abstraction the means of distinguishing the dangerous configurations from the reachable ones. CEGAR then continues by the next iteration with the refined abstraction. Otherwise, if the counterexample was not spurious, then it gives a proof of coverability of **bad**.

In the following sections, we present our modification of forward and backward analysis of the method [5].

3.1 Forward Abstract Interpretation

In this section, we define recall the forward abstract interpretation of WSTS of [5] that uses an abstract domain parameterised by a finite set D of configurations. Initially, the input of the algorithm is given as a $WSTS$ $S = (\Sigma, I, \rightarrow, \succeq)$, a parameter D and a target **bad**. For simplicity, we assume that D contains **bad** and the set I contains a single initial configuration x_0. The abstract interpretation algorithm runs the system in the abstract domain and decides whether the target **bad** is reachable from the initial set I or not.

Intuitively, a set of configurations E is abstracted into the set of elements form D that are covered by it (in a sense of $\preceq$). Formally, the abstraction is defined as

$$\forall E \in 2^X : \alpha[D](E) \stackrel{\text{def}}{=} E\downarrow \cap D,$$

while the concretisation function is defined as:

$$\forall P \in 2^D : \gamma[D](P) \stackrel{\text{def}}{=} \{x \in X \mid x{\downarrow} \cap D \subseteq P\}.$$

The abstract post operator in the domain defined by D is defined in a standard way as:

$$post^\sharp[D] \stackrel{\text{def}}{=} \alpha[D] \circ post \circ \gamma[D].$$

To find all configurations reachable in the abstract system from a set P, the forward steps are carried out till no new elements within the abstract domain defined by D are found, computing the image under the transitive closure of the abstract post:

$$post^\sharp[D]^*(P) \stackrel{\text{def}}{=} \bigcup_{i \geq 0} post^\sharp[D]^i(P).$$

The concretisation function γ may return an infinitely large set of configurations. Therefore, it is necessary to compute $post^\sharp$ in a symbolic manner. In particular, the post image consists of those elements $x \in D$ which have their predecessors in the concretization of the elements from the set P. Formally defined:

$$x \in post^\sharp(P) \Leftrightarrow (x \in D \wedge \neg(pre(x{\uparrow}) \subseteq (D \backslash P){\uparrow})).$$

After the set of reachable configurations is computed, it is necessary to check whether it contains the target **bad**. The target **bad** is unreachable if

$$bad \notin post^\sharp[D]^*.$$

Otherwise, if $bad \in post^\sharp[D]^*$, then the system can reach the target **bad** or the precision of the abstract domain defined by D is not sufficient and has to be refined.

Algorithm 1: Forward reachability

input : a $WSTS$ $S = (\Sigma, I, \rightarrow, \succeq)$, a parameter D and a target **bad**
output: Is **bad** reachable?

```
1  P_0 = α[D](x_0)
2  i = 0
3  path = ϵ
4  do
5      for t ∈ T do
6          if t.isEnabled (P_i) then
7              path = path . t
8              P_{i+1} = P_i ∪ post^{t♯}[D](P_i)
9              i = i + 1
10 while P_i ≠ P_{i−1}
11 if bad ∈ P_i then
12     return UNREACHABLE
13 else if ∃x_0, ..., x_k ∈ D : x_0 → ··· → x_k and x_k = bad then
14     return REACHABLE
```

Algorithm 1 implements the fixpoint computation of $post^\sharp[D]^*(P)$ and decides whether **bad** was or was not reached. The function t.isEnabled() returns **true** if there is a configuration in $\gamma[D](P_i)$ from where the transition t can be fired, and **false** otherwise. The function $post^{t\sharp}[D](P_i)$ represents a set of abstract successors of P_i under the transition t. It is a restriction of the post operator $post^\sharp$ to a single transition t, namely:

$$x \in post^{t\sharp}(P) \Leftrightarrow (x \in D \wedge \neg(pre^t(x\uparrow) \subseteq (D \backslash P)\uparrow)),$$

where $pre^t(X) = \bigcup_{x \in X} pre^t(x)$ and $pre^t(x) = \{y \mid y \xrightarrow{t} x\}$. Algorithm 1 is analogous to the forward search presented in [5] up that it also records the sequence $P_0, \ldots, P_n$ of the fixpoint approximations that and the sequence $t_1, \ldots, t_n$ of transitions that were taken to compute them. They will be used in the counterexample analysis.

3.2 Counterexample Analysis and Abstraction Refinement

The analysis of the counterexample run is based on the construction of the so called *minimal (abstract) counterexample run*. A minimal counterexample run is considered to be a run within an abstract domain, leading from an abstraction of initial set I to the target **bad** which is executed using a shortest sequence of transitions. The minimal counterexample run records how exactly were the elements of P_i, $i < n$ that were necessary for reaching **bad** generated by the forward analysis.

Algorithm 2: Construction of the graph

input : $P_0, \ldots, P_n$; $t_1, \ldots, t_n$; a parameter D; a target **bad**; $G = (V, E)$ where $V =$ **bad** and $E = \emptyset$

output: a minimal counterexample run represented by a DAG (V, E)

1 $n = \texttt{length}(\texttt{path})$
2 $W = \emptyset$
3 $W' = \{\textbf{bad}\}$
4 **for** $i \leftarrow n$ **to** 1 **do**
5 $\quad W = W'$
6 $\quad W' = \emptyset$
7 $\quad$ **for** node $\in W$ **do**
8 $\quad\quad$ Choose t_i from path
9 $\quad\quad$ **if** node $\in post^{t_i\sharp}(P_{i-1})$ **then**
10 $\quad\quad\quad V = V \cup \{pre^{t_i}(\text{node})\} \cup \alpha[D](pre^{t_i}(\text{node}))$
11 $\quad\quad\quad E = E \cup \{(\text{node}, pre^{t_i}(\text{node}))\} \cup \{(pre^{t_i}(\text{node}), x) | x \in \alpha[D](pre^{t_i}(\text{node}))\}$
12 $\quad\quad\quad W = W \backslash \{\text{node}\}$
13 $\quad\quad\quad W' = W' \cup \alpha[D](pre^{t_i}(\text{node}))$
14 $\quad\quad\quad$ break

Algorithm 2 constructs the minimal counterexample run in the form of a graph G based on the records of intermediate states and transitions taken during

the forward analysis. Its nodes are the elements of $P_i's$ needed for reaching **bad** and also the concrete preconditions of these elements wrt. the transitions which generated them within the forward analysis.

We so far propose only a naive method for the analysis of minimal runs. It is sufficient to generate more succinct invariants, but it is not yet optimized for overall efficiency: From each minimal run, we randomly select a configuration from the DCS of preconditions and use it to extend D.

4 Experiments

We have implemented our method in a prototype tool *MINA* in Python and compared the size of invariants generated by our method with the invariants generated by the state-of-the-art methods BFC [7], IIC [8], and our implementation of the algorithm GBR on several verification tasks from the benchmark of MIST2 [6]. In BFC we have deactivated the coverability oracle which uses simple forward exploration to search for reachable configurations and excludes their downward closure from the candidates for invariant refinement.[1]

Benchmark name	MINA	GBR	BFC	IIC
basicME.spec	**6**	22	23	7
read-write.spec	**3**	Timeout	456	67
pingpong.spec	**14**	80	31	**14**
newrtp.spec	**45**	45	54	**45**
mesh2x2.spec	**10**	Timeout	16454	**10**
mesh3x2.spec	**10**	Timeout	Timeout	**10**
lamport.spec	**17**	68	70	33
newdekker.spec	**21**	Timeout	234	45
peterson.spec	**32**	135	191	67
multiME.spec	**7**	Timeout	64	**7**
manufacturing.spec	**6**	Timeout	39	**6**

Table 1: A comparison of the size of the invariants generated by our method (Random Search), our implementation of the algorithm GBR [5], BFC [7] and IIC [8]. The size of the smallest invariant are typeset bold.

Since our method chooses invariant candidates randomly, we used the most succinct invariant generated in 30 executions. The overall runtime was therefore was higher than that of the other tools. We have, however, succeeded in generating significantly more succinct invariants than the other tools, as reported in Table 1.

[1] Optimizations like this are rather orthogonal to the choice of the main algorithm.

This confirms that, with an efficient analysis of the minimal counterexample runs, our approach has a potential to be more efficient than the existing methods. Our future research will therefore focus on efficient analysis of minimal counterexample runs.

References

1. Parosh Aziz Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, (4), 2010.
2. Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS'96*, pages 313–321. IEEE Computer Society, 1996.
3. Aaron R. Bradley. Sat-based model checking without unrolling. In *VMCAI'11*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
4. Edmund M. Clarke. Sat-based counterexample guided abstraction refinement. In *SPIN'02*, volume 2318 of *LNCS*, page 1. Springer, 2002.
5. Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. A complete abstract interpretation framework for coverability properties of WSTS. In *VMCAI'06*, volume 3855 of *LNCS*, pages 49–64. Springer, 2006.
6. Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Bfc - a widening approach to multi-threaded program verification. http://www.cprover.org/bfc/.
7. Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Efficient coverability analysis by proof minimization. In *CONCUR'12*, volume 7454 of *LNCS*. Springer, 2012.
8. Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In *CAV'13*, volume 8044 of *LNCS*. Springer, 2013.
9. Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV'03*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.