Excel@FIT 2015
http://excel.fit.vutbr.cz

Coverability for Parallel Programs

Lenka Turoňová*

Abstract
We improve existing method for the automatic verification of systems with parallel running processes. The technique is based on an effort to find an inductive invariant. We use a variant of counterexample-guided refinement (CEGAR). The effectiveness of the method depends on the size of the invariant. In this paper, we explore the possibility of improving the method by focusing on finding the smallest invariant.

Keywords: Parallel systems — Formal verification — Petri nets — Coverability — Abstraction — Well-quasi-ordered transition systems

Supplementary Material: *N/A*

*xturon02@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The pace of change in the entire world is speeding up. World population growth and thereby a hunger for new and faster machines, connections and technologies in general that would bring people comfort, spare their money and especially time. Time is actually the most valuable commodity. You cannot buy it and therefore you should use every second. Software engineers spend 50% of their programming time finding and fixing bugs. Hence, they are trying their best to reduce its amount.

On one hand, the systems are constantly getting more complex, powerful and faster. On the other hand, the software engineers have to deal with higher possibility of error occurrence associated with simultaneous data access and modification requests which are extremely difficult tasks, however, important steps in the software development. In some cases, a small error can lead to massive problems. A bright example in history could be considered the Therac-25 medical radiation therapy device where a side effect of the buggy software powering the device was overdose of radiation that several patients received.

One of the approaches proving that systems work correctly is formal verification. The formal verification denotes verification methods based on formal, mathematical roots and (at least potentially) capable of proving error freeness of systems with respect to some correctness specification by systematically searching the state space of the systems. For example, it is checked whether a given concurrent program is deadlock-free, how many elements can appear in some buffer. Formal verification is expected to be fully automated (no human help needed), be sound (if claims that a system is correct wrt. a given specification, it is indeed correct), be complete (if announces that a system is not correct, there is indeed an error in the system i.e., no false alarms (false positives) and always terminate. The main disadvantage of formal verification is that it is not suitable for large-scale systems unless a high level of abstraction is used. There are the following four main approaches to verification.

Theorem proving is usually a semi-automated approach using some inference system for deducing various theorems about the examined system from the facts known about the system and from general theorems of various logical theories.

Model checking is an approach of automated checking whether a system (or a model of a system) satisfies a certain correctness specification based on a systematic exploration of the state space of the system. Model checking is automatic and particularly good at analyzing (concurrent) reactive systems. State space of some systems may infinitely expand. However, model checking typically requires a closed system, i.e., a system together with its environment, and a bound on input

sizes.

Static analysis is usually characterised as analysis that collects some information about the behaviour of a system without actually executing it under its original semantics.

There is also testing which is widely used for checking correctness of systems, but its disadvantages are that cannot be used to prove system safe, and is not suitable for covering corner cases.

The state space explosion problem is a demanding technical challenge that can occur if the system has components that make transitions in parallel. A number of state reduction approaches have been proposed to reduce the number of states in the model. Among these techniques, abstraction is considered the most general and flexible for handling the state explosion problem.

Intuitively, abstraction amounts to removing or simplifying details as well as removing entire components of the original design that are irrelevant to the property under consideration. The evident rationale is that verifying the simplified ("abstract") model is more efficient than verifying the original model. The information loss incurred by simplifying the model however has a price: verifying an abstract model potentially leads to wrong results, at least if performed naively.

The technique called *counter-example-guided abstraction refinement* (CEGAR) is a technique that iteratively refines an abstract model using counter-examples. A counter-example is a witness of a property violation. In software verification, the counter-examples are error paths, i.e., paths through the program that violate the property.

We focus on parallel systems with infinite many threads that can lead to state exploration. This area has attracted the attention of a large number of people from various research communities. Due to unbounded number of processes, the systems have infinite number of configuration. There is a relation on configurations which is monotonic wrt. the transition relation and is well quasi ordered (*wqo*). Therefore, they can be modelled as well quasi ordered systems (*WSTS*). Examples of these systems include programs manipulating integer data, concurrency protocols involving arbitrary numbers of processes, and systems containing buffers where the sizes are described parametrically.

Petri nets offer a mathematical concept for modelling such systems. They are used as one of natural and simple tools for verifying the correctness of programs. They help practitioners to make their models more methodical, and theoreticians to make their models more realistic. Using Petri net we can simulate the behaviour of the system and check analytically their properties concerning safeness, coverability or liveness. Moreover, they offer a possibility to describe systems in a graphical way. On the other hand, Petri net use abstractions - the time consumption of any action and the data dependencies among conflict decisions. They usually report huge amount of potential behaviours for large and complex systems and hence, the developers need to decide which are relevant to the examined property.

Safety property of parallel programs can be checked via reduction to the coverability problem for well-quasi-ordered transition systems. The well-quasi-ordered transition systems have helpful properties that can be used for description of infinitely many states. The states of the systems have a well-quasi-ordering and transitions among the states satisfy a monotonicity property. The well-quasi-ordered systems include Petri nets, broadcast protocols, and lossy channel systems. We discuss the existing research approaches in the field of verification parallel programs, especially we focus on the coverability property of Petri nets which plays a key role in providing correctness of systems since Petri nets can be used as a modelling tool for such systems.

The existing algorithms inspect state space in order to find a safe inductive invariant that would be used for proving correctness of systems. The intention of our research was to find the proof of existence of smaller invariants than that have been already found by the existing methods since the smaller invariant can speed up the overall verification process.

2. Previous Works

Parallel programs are an active area of interest of many research teams that try their best to improve approaches how to automatically prove the correctness of systems. Model checkers are often based on Petri net coverability and generally use properties of *WSTS* such as the monoticity property of *WSTS*, the symbolic representation of reachable configurations as downward-closed sets or well-quasi order (*wqo*) of configurations that provides the finiteness of sequence of these sets.

The article *Incremental, Inductive Coverability* [1] describes a procedure that generalizes the IC3 procedure [2] for checking coverability of well-structed transition systems. The procedure is sound and complete and is based on computing an inductive invariant by maintaining a sequence of over-approximations of forward-reachable states and finding counter-examples to inductiveness using a backward exploration from a

set of incorrect states. It uses the properties of *WSTS* that the the reachability procedure must terminates. The general algorithm was also adapted to Petri nets that were used encoding each state to a Petri net place. Nevertheless, it has been proved that there are tools that outperform the tool presented in this article since they use more appropriate encoding.

The core of the algorithm represented in [3] is tracking uncoverability only for minimal uncoverable elements, even if these elements are not part of the original coverability query. The elements from the downward closure that are backward-reachable from a target state for smallest uncoverable members are inspected if they are not a part of coverable set from a set of initial configuration that is countering using a forward search engine that simultaneously generates it. It affects the speed of the overall algorithm, not its result.

The approach (*GRB*) presented in [4] is based on an abstract interpretation and used to solve the coverability problem of *WSTS*. First, the forward run is carried out from a set of initial states. All reachable states are represented by downward-closed sets. The generic representation of downward-closed set is formalized as a generic abstract domain. The abstract domain is refined using CEGAR to obtain sufficiently precise overapproximation to decide the coverability problem.

Nevertheless, still many configurations are generated to decide the coverability problem. Hence, it is necessary to optimalize the size of the parameters of the abstract domain to decrease the number of the inspected configurations and invariants that are added into the finite domain parametrizing the abstract domain.

The aim of the research is to optimalize the backward run and implement a modified version of the algorithm presented in [4].

3. Background

The most recent and most efficient methods are based on the mathematical concepts of the backward search (closed sets and monotonicity), but combine them with forward search and an effort to find an inductive invariant with a small representation (which usually coincides with the most general inductive invariant). A safe inductive invariant is a set of states of the system that (1) contains its initial states, (2) does not intersect with the undesired states, (3) and is closed under the transition relation.

One can observe that human-designed systems have simple inductive invariants sufficient to verify most safety properties. More than anywhere else, reasoning about parallel systems is hard and not intuitive, and hence their designers construct them so that arguments for their correctness—inductive invariants—are simple. Therefore, with a candidate for a simple invariant in hand, proving the system's safety by checking the inductivity conditions is usually easy. As a simple example, consider Peterson's mutual exclusion algorithm [5]. A simple invariant that proves that the algorithm guarantees mutual exclusion can be expressed by a conjunction of 9 clauses:

$\neg((q=3\wedge x=0)\vee(q=4\wedge x=0)\vee(q=2\wedge x=0)\vee(r=4\wedge y=0)\vee(r=3\wedge y=0)\vee(r=2\wedge y=0)\vee(q=4\wedge r=3\wedge T=1)\vee(q=3\wedge r=4\wedge T=0)\vee(q=4\wedge r=4))$ where $q=i$ or $r=i$ means that the control of the left or of the right process, respectively, is at the i-th line.

It is considerably less than the 135 disjunct inferred by the method of [4]. The other state-of-the-art methods [1, 3] would obtain similar results. We are convinced that by focusing on fast and accurate inference of simple invariants, we will get much better results than the current state of the art.

The works [1, 3, 4] learn invariants by CEGAR. Counterexample runs are analyzed in a kind of "local" way, the refinement method never sees a spurious counterexample run as a whole, but decides refinements based on narrow views of the run. We observe that to find one of the simplest invariants, it is usually necessary to capture certain subtle arguments for inductivity that will reveal themselves only when taking a global view, and analyzing every counterexample run as a whole, as, e.g., in Craig interpolation used in predicate abstraction. By focusing counterexample analysis this way, we should be able to get much simpler invariants much faster.

3.1 Methodology

As it was mentioned in Section 2, our method is built upon the algorithm represented in the article [4] which is based on inspecting counterexample runs within an abstract domain. From each counterexample run, the method infers a part of an inductive invariant. This part of invariant is then used to refine the abstract domain.

We represent the invariant expressed by a conjunction of clauses as a set of "words". The words contained in a set D parameterize the abstract domain. In case of Peterson's algorithm, a word $q3x0$ is used to denote a clause $q=3\wedge x=0$. Each word represents an infinite set of system configurations containing the word.

Initially, the set D consists only a set of incorrect configurations. In case of Peterson's algorithm, the

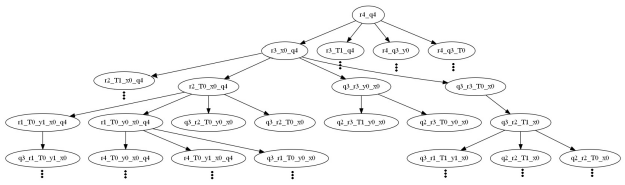

Figure 1. Backward run of *GRB*

configuration $q4r4$ ($q = 4 \wedge r = 4$) is included in the domain since it represents the state of the system when two processes are in a critical section.

To increase the precision of the abstraction counterexample runs are performed. In each iteration of CEGAR, the algorithm performs the forward search followed by the backward analysis trying to find configurations leading to the set of incorrect configurations. Subsequently, these configurations are included in the parameter D. The same iteration process is carried out till the set of the incorrect configurations is not reachable during the forward run or the set of the initial configurations is reached during the backward analysis. When the domain parameter D contains a whole inductive invariant, the abstraction becomes precise enough so that the system can be verified (there will be no more counterexample run).

The approach from the article [4] uses the technique breath-first search (Figure 1) for the state space exploration. This method causes an exponential increase in the number of inspected configurations and their length.

Our approach is based on the technique backtracking. It searches for the configurations by traversing the systems configurations in a depth first manner. Thus not all configurations are inspected and only small trees are constructed.

Starting from the incorrect configurations we explore their predecessors. They are obtained by applying the rules in a reverse way. First, each predecessor is abstracted. We expressed it with a set of minimal configurations from the parameter D. The words from the set are subwords of the predecessor.

We obtain the relevant predecessors of the configuration that represent preconditions necessary to reach the set of the incorrect configurations. Since the backtracking starts from the abstract set of the configuration we obtain smaller configurations than with the previous method.

A set of all predecessors and their subwords is created and one of the configurations is selected. It is added to the parameter D of the abstract domain as a part of the inductive invariant. The forward run is performed. If the set of incorrect configurations is reached then the configurations from the set D form the

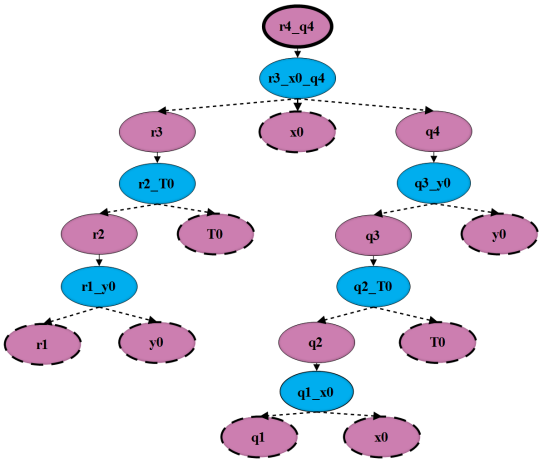

Figure 2. Backward counterexample run

whole invariant since it excludes all counterexample paths.

The disadvantage compared to the previous method is a large number of needed iterations of CEGAR since every iteration adds only one word to the parameter D. The previous method adds many words at each iterations so that less iterations were necessary to performed, however, many of them are useless and not optimal.

4. Experiments

Based on the method, we have implemented a prototype in Python to find invariants for checking safety properties for parameterized programs communicating via shared variables and mutexes. Our tool uses the input format of mist2[1]. We have implemented also a version of the algorithm from the article [4] (*GRB*) and compare the performance of the algorithm with our implementation in experiments on a set of Petri net benchmarks.

Name	GRB		Our approach		Reduction
	Size	Iter.	Size	Iter.	
basicME	45	5	22	19	44%
rw	331	8	36	35	90%
Peterson	135	8	107	21	21%
newrtp	45	9	54	53	-20%
pingpong	80	10	28	27	65%

Table 1 presents results for both algorithms. Column on the left shows the benchmark name while columns on the right show a size of invariant and the number of CEGAR iterations. The last column shows the percentage reduction in the size of the invariant.

[1] http://www.cprover.org/bfc/

The results demonstrate that our approach discovered the possibility of finding the smaller invariant than which was determined with the method *GRB*. The difference can be seen at example of Peterson's mutual exclusion algorithm. While the method *GRB* found invariant of size 135 our approach discovered that there can be even smaller one of size 107. However, in Section 3, we presented even smaller invariant of size 9. However, the price for smaller invariant is a greater number of iterations required to find it. Moreover, the method is still not yet sufficient enough to find the smaller invariant for more benchmarks.

5. Conclusions

In this paper, we presented an approach which is built upon the method represented in the article [4]. It runs a program in an abstract domain to find the invariant for verification. The domain is refined using CEGAR to obtain overapproximation which is precise enough so that the system can be verified.

While the method represented in the article [4] uses the technique of *BFS* our method determines the invariant based on backtracking. We obtain configurations representing preconditions necessary for reaching the set of incorrect configurations. In each iteration, randomly one configuration is added to the parameter D of the abstract domain.

Based on the method, the prototype in Python was implemented and tested on the set of Petri net benchmarks (see Section 4). The results show that our approach in some cases reduces the size of the invariant.

In this paper, we prove that there is a possibility to find the smaller invariant than which was discovered using method *GRB*. The price for smaller invariant is an increase in the number of iterations required to find it. Since our implemented method is unable to find the invariant for all the benchmarks future research will focus on developing a method which would find always the smaller invariant and in the best case, find the smallest one.

References

[1] KLOOS, J., MAJUMDAR, R., NIKSIC, F., PISKAC, R. Incremental, inductive coverability. *Lecture Notes in Computer Science*, 8044, 2013.

[2] BRADLEY, A. Sat-based model checking without unrolling. *Verification, Model Checking, and Abstract Interpretation*, 6538, 2011.

[3] KAISER, A., KROENING, D., WAHL, T. Efficient coverability analysis by proof minimization. *CONCUR 2012 - Concurrency Theory*, 7454, 2012.

[4] GANTY, P., RASKIN, J.F., VAN BEGIN, R. A complete abstract interpretation framework for coverability properties of wsts. *Verification, Model Checking, and Abstract Interpretation*, 3855, 2006.

[5] PETERSON,G.I. Myths about the mutual exclusion problem. *IPL*, 12(3), 1981.