Refinement of Trace Abstraction

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski

University of Freiburg, Germany

Abstract. We present a new counterexample-guided abstraction refinement scheme. The scheme refines an over-approximation of the set of possible traces. Each refinement step introduces a *finite automaton* that recognizes a set of infeasible traces. A central idea enabling our approach is to use *interpolants* (assertions generated, e.g., by the infeasibility proof for an error trace) in order to automatically construct such an automaton. A data base of *interpolant automata* has an interesting potential for reuse of theorem proving work (from one program to another).

1 Introduction

The automatic refinement of abstraction is an active research topic in static analysis [1,3,4,5,6,7,8,9,10,11,13,12,15,16,18]. It is widely agreed that the calls to a theorem prover, as used in existing methods for the construction of a sequence of increasingly precise abstractions, represent an obstacle to scalability. The problem is accentuated when costly decision procedures are employed to deal with arrays and heaps [19,20,23]. One way to address this obstacle is to increase the reuse of theorem work [11,13,12,18]. The question is in what form one should combine the results of theorem prover calls, and in what form they should be presented and stored.

Let us informally investigate the shortcomings inherent to the usage of theorem provers in the classical counterexample-guided abstraction refinement scheme (as, e.g., in [1,2,5,12,13,15]).

- In a first step, the theorem prover is called to prove the infeasibility of an error trace (in case it is a spurious counterexample). The corresponding unsatisfiability proof is then used for nothing but *guessing* the constituents of the new abstraction. If, as in [12,15], the unsatisfiability proof is used to generate interpolants which contain valuable information about the reason of infeasibility, then these are *cannibalized* for their atomic conjuncts.
- In a second step, the theorem prover is called to construct the transformer for the new abstraction; this step does not exploit the theorem proving work invested in the first step; in fact, the subsequent analysis of the new abstraction realizes a second proof of the infeasibility of the previous error trace.
- The theorem prover constructs the transformer for each new abstraction from scratch (at least on the part of the transformer's domain that has changed).
- The theorem proving work starts for each new program from scratch. This means all theorem proving work is done on-line, whereas ideally, most if not all of it should be done off-line, i.e., in a pre-processing step.

J. Palsberg and Z. Su (Eds.): SAS 2009, LNCS 5673, pp. 69–85, 2009.

In this paper, we present a new counterexample-guided abstraction refinement scheme. The scheme refines an over-approximation of the set of possible traces (in contrast to existing schemes which refine an over-approximation of the set of possible states). Each refinement step introduces a *finite automaton* that recognizes a set of infeasible traces. Such a *trace automaton* uses the *alphabet of statements*; each word over this alphabet is a trace. A central idea enabling our approach is to use *interpolants* (assertions generated, e.g., by the infeasibility proof for an error trace) in order to automatically construct such an automaton. The resulting *interpolant automaton* accepts not only the given error trace but many other (in general infinitely many) infeasible traces of varying shape and length.

The idea of using interpolants for the construction of an automaton overcomes a major difficulty in the construction of automata for the approximation of possible traces. Existing constructions (e.g., in [14,21] for hybrid systems) are based on ad hoc criteria; while the resulting methods succeed on several interesting examples, they are not general or complete. We also note a difference in the kind of alphabets used. In [14,21], the alphabet consists of action labels (or edge labels) defined by the input program (hybrid system), and the infeasibility property is specific to that program. In contrast, our notion of infeasibility depends solely on the programming language semantics.

One perspective opened by our work is a refinement loop that queries a database of interpolant automata; if there exists one that accepts the submitted error trace (which means that the error trace is not feasible), then the interpolant automaton gets added as another component to the trace abstraction. In this scenario, the interpolant automata can be constructed off-line (automatically, or manually using interactive verification methods).

2 Example

The correctness of the annotated program $\mathcal{P}$ in Fig. 1 is defined by the validity of its assertions. The correctness can be stated equivalently with the help of the automaton $\mathcal{A}_\mathcal{P}$ depicted in Fig. 2, the so-called program automaton. The transition graph of $\mathcal{A}_\mathcal{P}$ is the control flow graph of $\mathcal{P}$ where assertions are translated to edges to an error state.

The program automaton recognizes a set of words over the alphabet of statements (statements are framed in order to stress that they are used as letters of an alphabet). Each accepted word is a trace along a path in the control flow graph. The correctness of the annotated program $\mathcal{P}$ is expressed by the fact that all such traces are infeasible (which means that there is no valid execution leading from the initial location to the error location).

We next describe how our refinement scheme will generate a sequence of *trace abstractions* and, finally, prove the correctness of $\mathcal{P}$. Generally, each trace abstraction is a tuple of automata $(\mathcal{A}_1 \dots \mathcal{A}_n)$ over the alphabet of statements. An automaton in the tuple recognizes a subset of infeasible traces. This subset is used to restrict the set of traces recognized by the program automaton.