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.

This is the author's version of the work published in *Static Analysis Symposium (SAS 2009)*, LNCS 5673, pages 69–85. Springer, 2009. The orginal publication is available at
`www.springerlink.com/index/10.1007/978-3-642-03237-0_7`

In this paper, we present a new counterexample-guided abstraction refinement scheme which overcomes the above shortcomings. The scheme refines an over-approximation of the set of possible traces (whereas existing schemes refine an over-approximation of the set of possible states). Each refinement adds another finite automaton that recognizes a set of infeasible traces. The alphabet of such a *trace automaton* is the set of statements; a trace is nothing but a word over this alphabet. A central idea enabling our approach is to use *interpolants* in order to automatically construct such an automaton (interpolants are assertions generated by the infeasibility proof for the error trace). The resulting *interpolant automaton* accepts not only the error trace but a whole set of 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]) are based on ad hoc criteria; while the resulting methods succeed on several interesting examples, they are not general or complete. We also note the relvance of the alphabet for the automaton. If the alphabet consists of labels of edges (of the control flow graph or, as in [14,21], a hybrid system), then the definition of infeasibility must refer to that labeling. In contrast, our notion of infeasibility depends solely on the semantics of statements (i.e., of the programming language).

One perspective opened up 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.

```
ℓ0: x=0
ℓ1: y=0
ℓ2: while(nondet) {x++}
    assert(x!=-1)
    assert(y!=-1)
```

Fig. 1. Annotated program $\mathcal{P}$. The program $\mathcal{P}$ is correct if the assertions are valid.

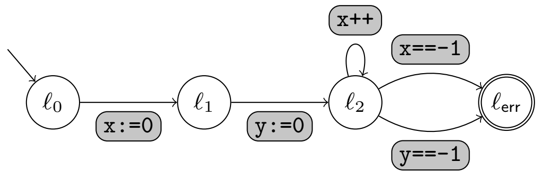

Fig. 2. Program automaton $\mathcal{A}_{\mathcal{P}}$ encoding the correctness of $\mathcal{P}$. The program $\mathcal{P}$ is correct iff every word accepted by $\mathcal{A}_{\mathcal{P}}$ is an infeasible trace.

First Iteration of Refinement Loop. The initial trace abstraction (for the first iteration of the refinement loop) is the empty tuple. The resulting restriction of the program automaton is the program automaton itself. In our example, the program automaton is not empty; it accepts, e.g., the trace π_1.

$$\pi_1 = \boxed{\texttt{x:=0}}.\boxed{\texttt{y:=0}}.\boxed{\texttt{x++}}.\boxed{\texttt{y==-1}}$$

The trace π_1 is returned as the counterexample of the first iteration of the refinement loop. A theorem prover is called to analyze the counterexample. The trace π_1 is infeasible. The unsatisfiability proof showing this, is used to construct the automaton $\mathcal{A}_1$ depicted in Fig. 3. This automaton accepts not only the trace π_1 but all traces that are infeasible for the same reason as π_1. In more detail: the unsatisfiability proof returns a sequence of *interpolants*. Each trace accepted by $\mathcal{A}_1$ has the same sequence of interpolants as π_1, up to repetition of subsequences of interpolants, and it is in this precise sense that it has the same "reason of infeasibility" as π_1. As explained later, the states q_i are in bijection with interpolants in the sequence, which is why we call $\mathcal{A}_1$ an *interpolant automaton*.

Second Iteration of Refinement Loop. The second abstraction (obtained from refining the initial abstraction) is the tuple $(\mathcal{A}_1)$ consisting of one component, the automaton derived in the previous refinement. The resulting restriction of the program automaton is the intersection of the program automaton with the complement of $\mathcal{A}_1$. In our example, the resulting automaton does not accept the trace π_1, the counterexample in the first refinement. Still, it is not empty; it accepts, e.g., the trace π_2.

$$\pi_2 = \boxed{\texttt{x:=0}}.\boxed{\texttt{y:=0}}.\boxed{\texttt{x++}}.\boxed{\texttt{x==-1}}$$

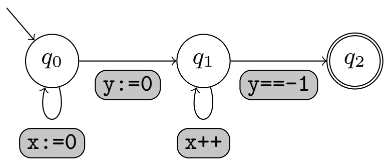

Fig. 3. Interpolant automaton $\mathcal{A}_1$, constructed from the unsatisfiability proof for the error trace $\pi_1 =$ `x:=0`.`y:=0`.`x++`.`y==-1`. It recognizes the set of traces that are infeasible for the same reason as π_1.

The trace π_2 is returned as the counterexample of the second iteration of the refinement loop. Again, a theorem prover is called to analyze the counterexample. The trace π_2 is infeasible as well. Again, the unsatisfiability proof showing this, is used to construct an automaton, $\mathcal{A}_2$ depicted in Fig. 4.

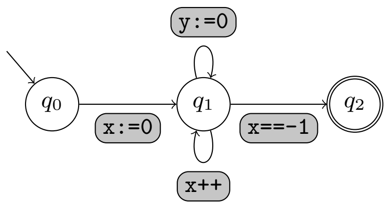

Fig. 4. Interpolant automaton $\mathcal{A}_2$ which is constructed from the unsatisfiability proof for the counterexample $\pi_2 =$ `x:=0`.`y:=0`.`x++`.`x==-1`.

Third and Final Iteration of the Refinement Loop. The third abstraction (obtained from refining the second abstraction) is the tuple $(\mathcal{A}_1, \mathcal{A}_2)$ constructed by extending the previous tuple with another component, the automaton derived in the previous refinement. The resulting restriction of the program automaton is the intersection of the program automaton with the complement of $\mathcal{A}_1$ and with the complement of $\mathcal{A}_2$. In our example, the resulting automaton $\mathcal{A}_{\mathcal{P}} \cap \overline{\mathcal{A}_1} \cap \overline{\mathcal{A}_2}$ does not accept the trace π_2, the counterexample in the first refinement. In fact, it does not accept any word; it is empty. The emptiness of $\mathcal{A}_{\mathcal{P}} \cap \overline{\mathcal{A}_1} \cap \overline{\mathcal{A}_2}$ proves the correctness of the annotated program $\mathcal{P}$.

In each iteration of the refinement loop, only the first step (the construction of the interpolant automaton from the unsatisfiability proof for the counterexample) involves theorem proving work. The second step is an operation on automata, i.e., on graphs; it does not involve formulas.

3 Traces

We assume a fixed set of statements Σ. We will consider Σ as an alphabet and statements as its letters. A *trace* $\pi = \mathit{st}_1 \ldots \mathit{st}_n$ is a word over this alphabet; i.e., $\pi \in \Sigma^*$.

It is important to realize that the notion of a trace is independent of a program (a trace may not correspond to a path in the program's control flow graph) and independent of the programming language semantics (a trace may not correspond to any possible execution). In order to stress the usage of statements as letters of an alphabet, we sometimes frame each statement/letter. For example, we can write the alphabet of the example program in Section 2 as

$$\Sigma_{ex} = \{\boxed{\texttt{x:=0}}, \boxed{\texttt{y:=0}}, \boxed{\texttt{x++}}, \boxed{\texttt{x==-1}}, \boxed{\texttt{y==-1}}\}$$

and

$$\pi = \boxed{\texttt{x++}}.\boxed{\texttt{x:=0}}.\boxed{\texttt{x:=0}}.\boxed{\texttt{y==-1}}.\boxed{\texttt{x==-1}}$$

is a possible trace. All automata that we consider in this work are automata over the given alphabet Σ; i.e., they recognize sets of traces.

Program Automaton $\mathcal{A}_\mathcal{P}$. We present an annotated program $\mathcal{P}$ directly as a trace automaton $\mathcal{A}_\mathcal{P}$ which we call the *program automaton*. We can obtain the program automaton in two ways. If we start with an annotated program as in Fig. 1 then we translate the assertions to edges to an error location in the control flow graph. In the resulting program automaton

$$\mathcal{A}_\mathcal{P} = \langle LOC, \delta_\mathcal{P}, \{\ell_{\mathsf{init}}\}, \{\ell_{\mathsf{err}}\}\rangle,$$

- the automaton states are program locations,
- the transition relation $\delta_\mathcal{P}$ contains exactly the edges $(\ell, \mathit{st}, \ell')$ of the control flow graph,
- the (unique) initial state is the (unique) initial location,
- the (unique) final state is the (unique) error location.

Alternatively, we may follow the automata-theoretic approach to program verification [22]. We use an LTL formula φ to specify a safety property (e.g., a second *lock* statement is not executed before an *unlock* statement). Let $\mathcal{A}_{\neg\varphi}$ be the automaton that accepts all *bad prefixes*, i.e., traces that witness the violation of the safety property specified by φ. Let $\mathcal{A}_{CFG}$ be the automaton whose states are the program locations of $\mathcal{P}$, whose transition graph is the control flow graph of $\mathcal{P}$, and each state is accepting. We construct the program automaton $\mathcal{A}_\mathcal{P}$ as the intersection of $\mathcal{A}_{CFG}$ and $\mathcal{A}_{\neg\varphi}$. It accepts all traces that violate the safety property φ and follow a path in the control flow graph of $\mathcal{P}$.

Error Trace. We call a trace accepted by the program automaton $\mathcal{A}_\mathcal{P}$ an error trace. An error trace corresponds to a path from the initial location to the error location. In order to determine if such a path corresponds to a possible execution, the semantics of the statements has to be taken into account.

Infeasibility. We assume that the semantics of statements is given by the (strongest) postcondition operator *post*. The predicate $\mathit{post}(\mathit{st}, \varphi)$ is the *(strongest) postcondition* of the predicate φ under the statement st. The extension of the postcondition from a statement to a trace is straightforward.

We define that the trace $\pi = \mathit{st}_1 \dots \mathit{st}_n$ is *infeasible* if

$$\mathit{post}(\pi, \top) \subseteq \bot$$

which expresses that the trace π has no possible execution.

Alternatively, infeasibility can be stated in terms of the (weakest) precondition operator. We use $\mathit{wp}(\mathit{st}, \varphi)$ to denote the *(weakest) precondition* of the predicate φ under the statement st. A trace π is infeasible if

$$\top \subseteq \mathit{wp}(\pi, \bot)$$

It is important to realize that the notion of feasibility is independent from the control flow graph (i.e., a feasible trace may not correspond to any path in the control flow graph).

Correctness. Having defined the notion of infeasibility, we can use the program automaton $\mathcal{A}_{\mathcal{P}}$ to define correctness. The annotated program $\mathcal{P}$ is correct if every trace accepted by $\mathcal{A}_{\mathcal{P}}$ is infeasible, formally

$$\mathcal{L}(\mathcal{A}_{\mathcal{P}}) \subseteq \textsc{Infeasible}.$$

4 Trace Abstraction

The program automaton $\mathcal{A}_{\mathcal{P}}$ encodes what trace is a path according to the control flow graph of the program. We will use a trace abstraction to encode a sufficient condition for when a trace is infeasible according to the semantics of the programming language.

Definition 1 (Trace Abstraction $(\mathcal{A}_1, \dots, \mathcal{A}_n)$). *A trace abstraction is given by a tuple of automata $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ such that each $\mathcal{A}_i$ recognizes a subset of infeasible traces, for $i = 1 \dots n$.*

Having separated the program-specific information and the programming language-specific information by the program automaton $\mathcal{A}_{\mathcal{P}}$ and a trace abstraction $(\mathcal{A}_1, \dots, \mathcal{A}_n)$, we need to combine the two in order to reason about correctness. This combination comes again in the form of an automaton; we define it as the intersection of the program automaton with the complements of $\mathcal{A}_1, \dots, \mathcal{A}_n$, which we write

$$\mathcal{A}_{\mathcal{P}} \cap \overline{\mathcal{A}_1} \cap \dots \cap \overline{\mathcal{A}_n}$$

where we use the symbol $\cap$ for the intersection of automata and $\overline{\mathcal{A}}$ for the complement of the automaton $\mathcal{A}$ (and assume that these two operations on automata implement the corresponding two operations on the recognized languages).

Proof Method Based on Trace Abstraction. Given the program automaton $\mathcal{A}_\mathcal{P}$, we say that *the trace abstraction* $(A_1, \dots, A_n)$ *does not admit an error trace* if the language recognized by the automaton $\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \dots \cap \overline{\mathcal{A}_n}$ is empty.

$$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \dots \cap \overline{\mathcal{A}_n}) = \emptyset$$

In this presentation we do not investigate how one can implement the emptiness test efficiently. Let us mention however, that the emptiness test can be done on the fly; only the reachable part of this abstraction has to be computed and not all components q_i in a state of the product $(q_1, \dots q_n)$ have to be made explicit.

The next two theorems state that a sound and complete proof method can be based on trace abstraction. By the algebraic properties of the intersection operation, the proof method based on trace abstraction is modular; i.e., the components $\mathcal{A}_i$ can be constructed independently one from another and their order does not matter.

Theorem 1 (Soundness). *If a trace abstraction* $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ *does not admit an error trace, i.e.,* $\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \dots \cap \overline{\mathcal{A}_n}) = \emptyset$*, then the program* $\mathcal{P}$ *is correct.*

Proof. The assumption $\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \dots \cap \overline{\mathcal{A}_n}) = \emptyset$ means that every trace accepted by $\mathcal{A}_\mathcal{P}$ is accepted by one of $\mathcal{A}_1, \dots, \mathcal{A}_n$.

$$\mathcal{L}(\mathcal{A}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \dots \cup \mathcal{L}(\mathcal{A}_n)$$

By definition of trace abstraction, each $\mathcal{A}_i$ recognizes a subset of infeasible traces.

$$\mathcal{L}(\mathcal{A}_1) \cup \dots \cup \mathcal{L}(\mathcal{A}_n) \subseteq \textsf{Infeasible}$$

Hence, every trace accepted by $\mathcal{A}_\mathcal{P}$ is infeasible, which is how the correctness of the program $\mathcal{P}$ is defined. □

According to folklore wisdom, if completeness holds, then it does for a trivial reason which does not provide any further insight. The proof method based on trace abstraction is no exception.

Theorem 2 (Completeness). *If the program* $\mathcal{P}$ *is correct, then there exists a trace abstraction* $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ *that does not admit an error trace, i.e.,* $\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \dots \cap \overline{\mathcal{A}_n}) = \emptyset$.

Proof. Assume $\mathcal{P}$ is correct. Then, by definition, $\mathcal{A}_\mathcal{P}$ does not accept an error trace, which is equivalent to $\mathcal{L}(\mathcal{A}_\mathcal{P}) \subseteq \textsf{Infeasible}$. We set $n = 1$ and choose the trace abstraction $(\mathcal{A}_1)$ where $\mathcal{A}_1 = \mathcal{A}_\mathcal{P}$. If we "implement" this abstraction we get the automaton $\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_\mathcal{P}}$ which recognizes the empty set. Hence $(\mathcal{A}_1)$ does not admit an error trace. □

5 CEGAR for Trace Abstraction

In the iterated refinement scheme depicted in Fig. 5, we transfer the classical check-analyze-refine loop to trace abstraction. The initial trace abstraction is the empty tuple of automata ($n = 0$). If the trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ admits an error trace, say π, we check whether π is infeasible. If this is the case, we extend the trace abstraction with an automaton $\mathcal{A}_{n+1}$ that accepts (at least) the infeasible trace π.

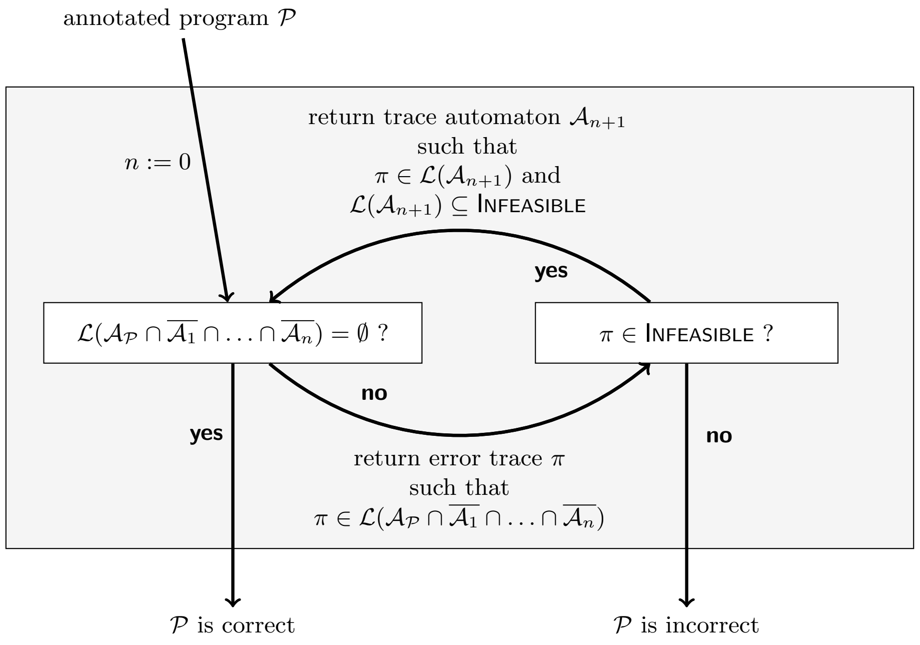
annotated program $\mathcal{P}$
$n := 0$
return trace automaton $\mathcal{A}_{n+1}$ such that $\pi \in \mathcal{L}(\mathcal{A}_{n+1})$ and $\mathcal{L}(\mathcal{A}_{n+1}) \subseteq$ INFEASIBLE
yes
$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}) = \emptyset$?
$\pi \in$ INFEASIBLE ?
no
yes
no
return error trace π such that $\pi \in \mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n})$
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect

Fig. 5. Counterexample-guided abstraction refinement scheme for trace abstraction. The program $\mathcal{P}$ is correct if $\mathcal{L}(\mathcal{A}_\mathcal{P}) \subseteq$ INFEASIBLE.

Incrementality. If the trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ does not admit an error trace then it still does not when we add any number of components to the tuple. As a consequence, in a series of successive refinements, one never has to withdraw a previously added component ("superfluous components in a tuple do not hurt").

Progress. The infeasible error trace π returned in the n-th iteration of the refinement loop gets eliminated by the refined abstraction; i.e., the trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{A}_{n+1})$ does not admit the error trace π.

6 Interpolant Automata

In the setting of iterated refinement for trace abstraction (Fig. 5) in the previous section, it is trivial to construct an automaton $\mathcal{A}_{n+1}$ that accepts exactly the infeasible error trace π. The corresponding trivial refinement excludes one and only one infeasible error trace. The question is how one can generalize the counterexample, i.e., construct an automaton $\mathcal{A}_{n+1}$ that recognizes a set consisting of π and more infeasible traces. Ideally, those traces share with π the 'reason of infeasibility'.

An immediate idea is to augment the trivial automaton (which accepts exactly the infeasible error trace π) with transitions that are labeled by "irrelevant statements" and that are self-loops (transitions from and to the same automaton state). A statement is irrelevant if it does not modify a variable whose value determines the infeasibility. In our example from Section 2, in the construction of the automaton $\mathcal{A}_1$ for the error trace π_1 (see Fig. 3), one would thus obtain the self-loop labeled $\boxed{\texttt{x++}}$. One would, however, fail to add the self-loop labeled $\boxed{\texttt{x++}}$ in the construction of the automaton $\mathcal{A}_2$. One would also fail to introduce general loops in the construction of the automaton.

The first step towards a generally applicable construction is to consider a *sequence of predicates* $I_0, I_1, \ldots, I_n$ (to which we refer as interpolants for reasons that will become apparent later). In many settings that we consider, this sequence is related to the error trace π; it may be generated, for example, by the proof of the infeasibility of π.

The general notion of an *interpolant automaton* that we introduce below, however, does not refer to an error trace. It refers to an arbitrary sequence of predicates $I_0, I_1, \ldots, I_n$. Given such a sequence, we will associate each predicate I_i with an automaton state q_i. The automaton states are not necessarily pairwise distinct; i.e., we may associate two different predicates I_i and I_j with the same automaton state, and we may associate the same predicate with two different states (i.e., we may have $I_i \neq I_j, q_i = q_j$ and we may have $I_i = I_j, q_i \neq q_j$). The non-constructive definition below accommodates a wide range of possible constructions. The definition of a *canonical interpolant automaton* further below is constructive.

Definition 2 (Interpolant Automaton $\mathcal{A}_\mathcal{I}$). *Given a sequence of predicates $\mathcal{I} = I_0, I_1, \ldots, I_n$ (to which we will refer as "interpolants"), we call a trace automaton*

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q_\mathcal{I}^{\mathsf{init}}, Q_\mathcal{I}^{\mathsf{fin}} \rangle$$

an interpolant automaton *if we can index its set of states $Q_\mathcal{I}$ with the set of indices of the sequence $\{0, \ldots, n\}$,*

$$Q_\mathcal{I} = \{q_0, \ldots, q_n\}$$

and thus associate each interpolant I_i with a state q_i, such that the following three conditions hold.

– *Each pair of interpolants associated with a state transition is inductive.*

$$(q_i, st, q_j) \in \delta_{\mathcal{I}} \quad \textit{implies} \quad post(st, I_i) \subseteq I_j$$

– *Each interpolant associated with an initial state is the* true *predicate.*

$$q_i \in Q_{\mathcal{I}}^{\mathsf{init}} \quad \textit{implies} \quad I_i = \top$$

– *Each interpolant associated with a final state is the* false *predicate.*

$$q_i \in Q_{\mathcal{I}}^{\mathsf{fin}} \quad \textit{implies} \quad I_i = \bot$$

Theorem 3. *An interpolant automaton $\mathcal{A}_{\mathcal{I}}$ recognizes a subset of infeasible traces.*

$$\mathcal{L}(\mathcal{A}_{\mathcal{I}}) \subseteq \textit{Infeasible}$$

Proof. We show (by induction on the length of a trace π) that if $q_j \in \delta(\pi, q_i)$ then the inclusion $post(\pi, I_i) \subseteq I_j$ holds. Thus, for every trace π accepted by $\mathcal{A}_{\mathcal{I}}$ the inclusion

$$post(\pi, \top) \subseteq \bot,$$

holds, which means that π is infeasible. □

Completeness. We may ask whether a proof method based on trace abstraction is still complete if the automata $\mathcal{A}_i$ of a trace abstraction $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ are restricted to be interpolant automata. Again, the completeness argument is disappointingly simple. If the program is correct, the program automaton $\mathcal{A}_{\mathcal{P}}$ is an interpolant automaton. To see this, define I_i to be the set of states at the location ℓ_i reachable from any state at location ℓ_0 (assuming the locations of the program automaton $\mathcal{A}_{\mathcal{P}}$ are exactly $\ell_0, \dots, \ell_n$). We associate I_i with the state ℓ_i of the program automaton. Since we assume that the program is correct, we know that the interpolant associated with the error location ℓ_{err} is the *false* predicate $\bot$.

Interpolant Automata and Floyd-Hoare style Proofs of Program Correctness. In the discussion of completeness above, we can more generally define I_i to be any invariant assertion associated with the location ℓ_i in a Floyd-Hoare style proof of the partial correctness of the program $\mathcal{P}$. This is because, when we transfer the partial-correctness statement to the control flow graph with the error location ℓ_{err}, we will label the error location ℓ_{err} with the *false* predicate $\bot$. The condition on each pair of interpolants associated with a transition (in Definition 2) is exactly the inductiveness of the invariant assertions in the Floyd-Hoare style proof.

The proof of partial-correctness may refer to the full program or just a program fragment, constituted, e.g., by the *slice* of the control flow graph which is executed by the error trace π. In a concrete setting, there are many ways in which one may obtain such a proof: manually, or by a constraint solving method as, e.g., in [3], or by one of the methods based on (counterexample-guided abstraction refinement of) state abstraction, e.g., [1,3,4,11,13,15,18].

Once the interpolant automaton is formed, it no longer carries any reference to program states (or invariant assertions and such). A trace automaton is a graph; it is detached from both the original program and the semantics of statements (as predicate transformers). As a consequence, refinement of trace abstraction does not involve logical conjunction and theorem prover calls; it is a graph operation.

Determinism. The general setting of non-deterministic trace automata is potentially useful for a compact representation of infeasibility. If the trace automaton is deterministic then its complement can have the same transition graph (up to *sink states* which are introduced to obtain a total transition relation). This is the case, e.g., when the trace automaton is the program automaton $\mathcal{A}_\mathcal{P}$ or, more generally, when the transition graph of the trace automaton is a subgraph of the (possibly partially unfolded) control flow graph (since a statement cannot lead to different locations).

Canonical Interpolant Automaton. Next, we will introduce the notion of a *sequence of interpolants* for the error trace π and use it to give a constructive definition of a special case of an interpolant automaton.

Sequence of Interpolants for an Error Trace. Given an infeasible error trace $\pi = st_1, \dots, st_n$, we call a sequence of predicates $\mathcal{I} = I_0, I_1, \dots, I_n$ a *corresponding sequence of interpolants* (corresponding to π) if the following conditions hold (where $\top$ is the *true* predicate and $\bot$ is the *false* predicate).

- $I_0 = \top$
- $post(st_{i+1}, I_i) \subseteq I_{i+1}$, for $i = 0 \dots n-1$
- $I_n = \bot$

If we split the trace π at any position i into a prefix $st_1 \dots st_i$ and suffix $st_{i+1} \dots st_n$ then every state reached under a possible execution of the prefix $st_1 \dots st_i$ satisfies I_i and no state satisfying I_i has a possible execution under the suffix $st_{i+1} \dots st_n$. In other words, the interpolant I_i is an overapproximation of the postcondition of *true* under the prefix and an underapproximation of the weakest precondition *false* under the suffix, formally

$$post(st_1 \dots st_i, \top) \subseteq I_i \subseteq wp(st_{i+1} \dots st_n, \bot).$$

A sequence of interpolants may be, but is not necessarily a sequence of Craig interpolants generated from the proof of infeasibility of a counterexample (in the spirit of [13,15,17,18]). A sequence of interpolants may also arise as the sequence of invariant assertions along the sequence of program locations in a Hoare-style proof (for the correctness of the program fragment corresponding to the spurious counterexample).

In order to motivate the definition of the canonical interpolant automaton, we will give a schematic example of its construction.

Example. In the schematic setting depicted in Fig. 6, we assume that

- $\pi = \mathit{st}_1 \dots \mathit{st}_n$ is an infeasible error trace along the locations $\ell_0, \dots, \ell_n$,
- $\mathcal{I} = I_0, \dots, I_n$ is a corresponding sequence of interpolants,
- i and j are two positions such that $j < i$, $\ell_i = \ell_j$, and $\mathit{post}(\mathit{st}_{j+1}, I_i) \subseteq I_{j+1}$.

To make the example simple, let us assume that ℓ_j is the only repeated location in π. The assumption that ℓ_i is the same location as ℓ_j implies the existence of a loop in the control flow graph that goes from ℓ_j via $\ell_{j+1} \dots \ell_{i-1}$ back to ℓ_j. The error trace π executes this loop exactly once.

We now construct the automaton $\mathcal{A}^{\pi}_{\mathcal{I}}$ depicted in Fig. 6 by taking the trivial automaton (which accepts only the one infeasible error trace π) and add exactly one 'back edge', namely the transition $(q_i, \mathit{st}_{j+1}, q_j)$. The automaton $\mathcal{A}^{\pi}_{\mathcal{I}}$ accepts all traces that follow the same path in the control flow graph as π (and that execute the loop through ℓ_j at least once).

$$\mathcal{L}(\mathcal{A}^{\pi}_{\mathcal{I}}) = \mathit{st}_1 \dots \mathit{st}_j \mathit{st}_{j+1} \dots \mathit{st}_i \big(\mathit{st}_{j+1} \dots \mathit{st}_i\big)^{\star} \mathit{st}_{i+1} \dots \mathit{st}_n$$

To see that the trace $\pi_k = \mathit{st}_1 \dots \mathit{st}_j \mathit{st}_{j+1} \dots \mathit{st}_i (\mathit{st}_{j+1} \dots \mathit{st}_i)^k \mathit{st}_{i+1} \dots \mathit{st}_n$ is infeasible for $k \geq 0$, we first observe that the inclusion $\mathit{post}(\mathit{st}_{j+2} \dots \mathit{st}_i, I_{j+1}) \subseteq I_i$ holds by the definition of a sequence of interpolants for π. This together with the assumption $\mathit{post}(\mathit{st}_{j+1}, I_i) \subseteq I_{j+1}$ implies $\mathit{post}(\mathit{st}_{j+1} \dots \mathit{st}_i, I_i) \subseteq I_i$. Thus, the inclusion

$$\mathit{post}(\mathit{st}_{j+1} \dots \mathit{st}_i (\mathit{st}_{j+1} \dots \mathit{st}_i)^k, I_j) \subseteq I_i$$

holds for $k \geq 0$. This implies $\mathit{post}(\pi_k, \top) \subseteq \bot$, the infeasibility of π_k. $\square$

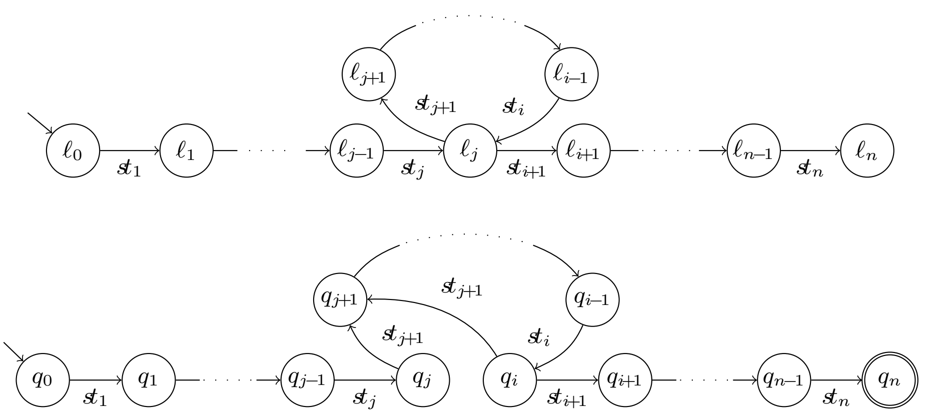

Fig. 6. The infeasible error trace $\pi = \mathit{st}_1 \dots \mathit{st}_n$ follows the path with the locations $\ell_0, \dots, \ell_n$ in the control flow with a loop through ℓ_j ($\ell_i = \ell_j$). We assume that $\mathcal{I} = I_0, \dots, I_n$ is a corresponding sequence of interpolants with $\mathit{post}(\mathit{st}_{j+1}, I_i) \subseteq I_{j+1}$. Adding the transition $(q_i, \mathit{st}_{j+1}, q_j)$ to the trivial automaton (which accepts only π) results in the automaton $\mathcal{A}^{\pi}_{\mathcal{I}}$.

Definition 3 (Canonical Interpolant Automaton $\mathcal{A}^{\pi}_{\mathcal{I}}$). *Given a sequence of interpolants $\mathcal{I} = I_0, I_1, \dots, I_n$ corresponding to the infeasible error trace $\pi = \mathit{st}_1, \dots, \mathit{st}_n$ along the sequence of locations $\ell_0, \dots, \ell_n$, we introduce pairwise different states $q_0, \dots, q_n$ and define the* canonical interpolant automaton *$\mathcal{A}^{\pi}_{\mathcal{I}}$ for π and $\mathcal{I}$ as follows.*

$$\mathcal{A}^{\pi}_{\mathcal{I}} = \langle Q_{\mathcal{I}}, \delta_{\mathcal{I}}, Q^{\mathsf{init}}_{\mathcal{I}}, Q^{\mathsf{fin}}_{\mathcal{I}} \rangle$$

- $Q_{\mathcal{I}} = \{q_0, \dots, q_n\}$
- $\delta_{\mathcal{I}} = \{(q_i, \mathit{st}_{j+1}, q_{j+1}) \mid 0 \leq j \leq i \leq n-1,\ \ell_i = \ell_j,\ \mathit{post}(\mathit{st}_{j+1}, I_i) \subseteq I_{j+1}\}$
- $Q^{\mathsf{init}}_{\mathcal{I}} = \{q_0\}$
- $Q^{\mathsf{fin}}_{\mathcal{I}} = \{q_n\}$

The canonical interpolant automaton $\mathcal{A}^{\pi}_{\mathcal{I}}$ accepts the error trace π. This follows from the definition of the sequence of interpolants. In general $\mathcal{A}^{\pi}_{\mathcal{I}}$ accepts an infinite set of traces. In a sense, $\mathcal{A}^{\pi}_{\mathcal{I}}$ accepts exactly the traces that are infeasible for the same reason as π. More precisely, in order to prove the infeasibility of a trace accepted by $\mathcal{A}^{\pi}_{\mathcal{I}}$, we can use the same sequence of interpolants (up to repetition of subsequences) as in the proof of infeasibility of π.

The inclusions $\mathit{post}(\mathit{st}_{i+1}, I_i) \subseteq I_{i+1}$ hold by the definition of sequence of interpolants. Thus, after having generated the sequence of interpolants $\mathcal{I}$ (for the proof of the infeasibility of the trace π), one needs additional theorem prover calls only for each inclusion $\mathit{post}(\mathit{st}_{j+1}, I_i) \subseteq I_{j+1}$ where $j < i$ and $\ell_i = \ell_j$. Thus, the number of additional theorem prover calls is bounded by the number of repeated locations in the sequence of locations along the error trace π.

Speculative Computation of Infeasibility. The general definition of interpolant automata accommodates optimizations where one invests theorem proving work speculatively. That is, one checks the validity of inclusion other than the ones required in the construction of the canonical interpolant automaton. The goal is to add more edges to the transition graph, and thus obtain an interpolant automaton that recognizes a larger set of infeasible traces. One possibility is to remove the side-condition for i, j from the definition of $\delta_{\mathcal{I}}$, i.e., one checks the inclusions $\mathit{post}(\mathit{st}_{j+1}, I_i) \subseteq I_{j+1}$ for all pairs of locations. If the interpolant I_i is subsumed by the interpolant I_j we may add a transition (q, st, q_j) to the state q_j if the corresponding transition (q, st, q_i) to the state q_i exists. Yet another possibility is to check the validity of inclusions $\mathit{post}(\mathit{st}, I_i) \subseteq I_j$ where st is not necessarily a statement in the error trace π. This leads to, e.g., exploring both branches of a conditional statement and thus adding a branching structure to the interpolant automaton.

Caching Infeasibility. When verifying many programs or program parts, similar patterns of infeasible error traces may occur several times. Our notion of infeasibility is independent of a particular program. It allows the reuse of interpolant automata for the verification of other programs. One can imagine a refinement scheme based on a database of interpolant automata. If the trace abstraction $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ admits an error trace π then the database can be queried for an automaton that accepts π (modulo variable renaming). If such an automaton exists, then π is infeasible and the abstraction is refined to $(\mathcal{A}_1, \dots, \mathcal{A}_n, \mathcal{A}_{n+1})$ by adding the new automaton to the tuple.

Example. Reconsider the example of Section 2. The automata $\mathcal{A}_1$ (depicted in Fig. 3) and $\mathcal{A}_2$ (depicted in Fig. 4) are interpolant automata resulting from a construction similar to the canonical interpolant automaton where states q_i and q_j are merged if the interpolants I_i and I_j are equal. (The test of equality of predicates requires in general a call to the theorem prover.)

The automaton $\mathcal{A}_1$ is obtained from the corresponding sequence of interpolants

$$\top, \quad \top, \quad y = 0, \quad y = 0, \quad \bot$$

given the error trace

$\pi_1 =$ `x:=0`.`y:=0`.`x++`.`y==-1`.

The automaton $\mathcal{A}_2$ is obtained from the corresponding sequence of interpolants

$$\top, \quad x \geq 0, \quad x \geq 0, \quad x \geq 0, \quad \bot$$

given the error trace

$\pi_2 =$ `x:=0`.`y:=0`.`x++`.`x==-1`.

□

7 Predicate Abstraction

In this section we compare predicate abstraction as in [1,3,4,5,13,12,15,16]) with trace abstraction. We start by formalizing predicate abstraction. Given a finite set of predicates, say

$$\mathsf{Pred} = \{p_1, \dots, p_m\}$$

we call an m-tuple $\langle b_1, ..., b_m \rangle$ of possibly negated predicates a *bitvector* (we assume a fixed order on the predicates).

$$\langle b_1, ..., b_m \rangle \;\text{ where } b_j \text{ is either } p_j \text{ or } \neg p_j, \text{ for } j = 1, \dots, m$$

Given a program $\mathcal{P}$ with the post operator *post*, we construct the relation $\delta_\#$ between bitvectors (in principle by calling a theorem prover for each pair of bitvectors and each statement).

$$\delta_\# = \{(\langle b_1, ..., b_m \rangle, st, \langle b'_1, ..., b'_m \rangle \mid \mathit{post}(st, b_1 \wedge ... \wedge b_m) \cap b'_1 \wedge ... \wedge b'_m \neq \bot\}$$

The predicate abstraction of the program $\mathcal{P}$ wrt. Pred can be defined as the finite-state abstract program $\mathcal{P}^\#_{\mathsf{Pred}}$, whose states are pairs of a program location and a bitvector, and whose transitions are induced by the relation $\delta_\#$ between bitvectors.

Theorem 4. *Predicate abstraction is a special case of trace abstraction: the abstraction defined by a tuple of predicates can be expressed by a tuple consisting of one single trace automaton.*

Proof. We define the trace automaton $\mathcal{A}_{\#}$ whose states are the bitvectors, the transition relation is $\delta_{\#}$, and each state is an initial and a final state.

$$\mathcal{A}_{\#} = \langle Q_{\#}, \delta_{\#}, Q_{\#}, Q_{\#} \rangle$$

This automaton recognizes a superset of *all* feasible traces (and *not* just feasible traces of $\mathcal{A}_{\mathcal{P}}$; note that the transition relation $\delta_{\#}$ is total). We define the automaton $\mathcal{A}_{\mathsf{Pred}}$ as the complement of $\mathcal{A}_{\#}$. Since $\mathcal{A}_{\mathsf{Pred}}$ recognizes a subset of infeasible traces, the 1-tuple $(\mathcal{A}_{\mathsf{Pred}})$ is a trace abstraction.

$$\mathcal{A}_{\mathsf{Pred}} = \overline{\mathcal{A}_{\#}}$$

The product of the program automaton $\mathcal{A}_{\mathcal{P}}$ with the complement of $\mathcal{A}_{\mathsf{Pred}}$, i.e., with $\mathcal{A}_{\#}$, is exactly the abstract program $\mathcal{P}^{\#}_{\mathsf{Pred}}$, the predicate abstraction of the concrete annotated program $\mathcal{P}$ wrt. the set of predicates Pred.

$$\mathcal{P}^{\#}_{\mathsf{Pred}} = \mathcal{A}_{\mathcal{P}} \cap \overline{\mathcal{A}_{\mathsf{Pred}}}$$

Thus, the trace abstraction $(\mathcal{A}_{\mathsf{Pred}})$ expresses the predicate abstraction of the program $\mathcal{P}$ wrt. Pred. □

Trace abstraction is strictly more expressive than predicate abstraction, since it is not possible to derive predicates from trace automata (as explained above, a trace automaton is detached from the original program and in particular it does not convey the semantics of its statements as predicate transformers).

Refinement, Combination of Abstractions. Trace abstraction allows one to combine abstractions with a minimal investment of theorem proving work. In order to explicate this point, we will build on the fact (established above) that one can use predicate abstraction to construct one or more of the component automata $\mathcal{A}_i$ in a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$.

We now consider the combination of the two predicate abstractions defined by the sets of predicates Pred_1 and Pred_2, once as the trace abstraction defined by the 2-tuple of the two predicate abstractions, i.e.,

$$(\mathcal{A}_{\mathsf{Pred}_1}, \mathcal{A}_{\mathsf{Pred}_2}),$$

and once as the predicate abstraction for the set of predicates Pred defined by the union of the two sets of predicates, $\mathsf{Pred} = \mathsf{Pred}_1 \cup \mathsf{Pred}_2$. As seen above, this predicate abstraction can be expressed equivalently as the trace abstraction defined by the 1-tuple with the automaton $\mathcal{A}_{\mathsf{Pred}}$, i.e.,

$$(\mathcal{A}_{\mathsf{Pred}_1 \cup \mathsf{Pred}_2}).$$

The combination as a trace abstraction $(\mathcal{A}_{\mathsf{Pred}_1}, \mathcal{A}_{\mathsf{Pred}_2})$ is coarser (in general strictly coarser) than the combination as a predicate abstraction $(\mathcal{A}_{\mathsf{Pred}_1 \cup \mathsf{Pred}_2})$, but it can be computed without additional theorem proving work. It is possible to formally account for this phenomenon in terms of *products of abstract domains* [6,7].

8 Conclusion

We have presented a refinement scheme whose novelty lies in the following points.

– Trace abstraction instead of state abstraction. The goal of the iterated refinement is the successive restriction of the approximation of the set of execution traces (and not of the set of reachable states).
– Compositionality of refinement. The refinement is decomposed into independent steps (the construction of one automaton does not build on another automaton). The results of the individual steps are composed by a graph operation (the intersection of automata).
– Interpolant automata. We use interpolants in order to construct an automaton (which recognizes a set of infeasible traces). Interpolants can be Craig interpolants, or any other inductive assertions in a Floyd-Hoare style proof.
– Coarse-grained caching. Each trace automaton represents the *macro* result of a coherent set of theorem prover calls.
– Reuse from one program to another. The notion of an infeasible trace refers to the programming language semantics. The refinement through an automaton is applicable beyond one specific program.

The scope of this paper is to introduce the principles of the refinement scheme for trace abstraction. Hence, we have aimed at the most general formulation possible. The question of the most practical instantiation of the refinement scheme remains a topic of future work. In particular, the realization of a data base of interpolant automata which accounts for common programming patterns raises a number of interesting research issues.

Acknowledgements. We thank Ahmed Bouajjani, Ken McMillan, Natarajan Shankar, and Ashish Tiwari for fruitful discussions. This work was partly supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by NSF grant CNS-0749931 while the third author visited SRI International.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS '02*, pages 158–172. Springer, 2002.
2. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02*, pages 1–3. ACM, 2002.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI '07*, pages 300–309. ACM, 2007.
4. I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In *FSEN '07*, pages 17–32. Springer, 2007.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV '00*, pages 154–169. Springer, 2000.

6. P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French).* Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.
7. P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM, 1977.
9. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.
10. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *SAS '01*, pages 356–373. Springer, 2001.
11. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS '08*, pages 443–458. Springer, 2008.
12. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL '04*, pages 232–244. ACM, 2004.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, 2002.
14. S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *HSCC '07*, pages 287–300. Springer, 2007.
15. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS '06*, pages 459–473. Springer, 2006.
16. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS '01*, pages 98–112. Springer, 2001.
17. K. L. McMillan. Interpolation and sat-based model checking. In *CAV '03*, pages 1–13. Springer, 2003.
18. K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136. Springer, 2006.
19. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS '08*, pages 413–427. Springer, 2008.
20. A. Podelski and T. Wies. Boolean heaps. In *SAS '05*, pages 268–283. Springer, 2005.
21. M. Segelken. Abstraction and counterexample-guided construction of *omega* -automata for model checking of step-discrete linear hybrid models. In *CAV '07*, pages 433–448. Springer, 2007.
22. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS '86*, pages 332–344. IEEE Computer Society, 1986.
23. T. Wies. *Symbolic Shape Analysis.* PhD Thesis, Albert-Ludwigs-Universität, Freiburg, Germany, 2009.