Software Model Checking for People who Love Automata

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski

University of Freiburg, Germany

Abstract. In this expository paper, we use automata for software model checking in a new way. The starting point is to fix the alphabet: the set of statements of the given program. We show how automata over the alphabet of statements can help to decompose the main problem in software model checking, which is to find the right abstraction of a program for a given correctness property.

1 Introduction

Automata provide the algorithmic basis in many applications. In particular, we can use automata-based algorithms to implement a data structure for operations over *sets of sequences*. An automaton defines a set of sequences over some alphabet. Or, in the terminology of formal language theory: an automaton *recognizes a language* (the elements in the sequence are *letters*, a sequence of letters is a *word*, a set of words is a *language*).

Formally, an automaton is a finite graph; its edges are labeled by letters of the alphabet; an initial node and a set of final nodes are distinguished among its nodes. The labeling of a path is a word. The automaton defines the set of all words that label a path from the initial node to one of the final nodes.

In this expository paper, we use automata for software model checking in a new way. The starting point is to fix the alphabet: the set of statements of the given program. The idea that a statement is a letter may take some time to get used to. As a letter, a statement is deprived of its meaning; the only purpose of a letter is to be used in a word. We are not used to freely compose statements to words, regardless of whether the word makes any sense as a sequence of statements or not.

In software model checking, a (if not *the*) central problem is to automatically find the right abstraction of a program for a given correctness property. In the remainder of this section, we will use three examples to illustrate how automata over the alphabet of statements can help to automatically decompose this problem. Then, in Section 2, we will fix the formal setting that allows us to relate the correctness of programs with automata over the alphabet of statements. In Section 3, we will define the notion of *Floyd-Hoare automata* for a given correctness property, and we will present different ways to construct such automata. We will see that, for a given program, the construction of Floyd-Hoare automata can be used to automatically decompose the task of finding the right abstraction of the program for the given correctness property.

This is the author's version of the work published in M. Heizmann, J. Hoenicke, and A. Podelski. Software Model Checking for People who Love Automata. In *Computer Aided Verification (CAV 2013)*, number 8044 in LNCS, pages 36-52. Springer, 2013. The orginal publication is available at
`www.springerlink.com/index/10.1007/978-3-642-39799-8_2`

```
ℓ0: assume p != 0;
ℓ1: while(n >= 0)
    {
ℓ2:     assert p != 0;

        if(n == 0)
        {
ℓ3:         p := 0;
        }
ℓ4:     n--;
    }
```

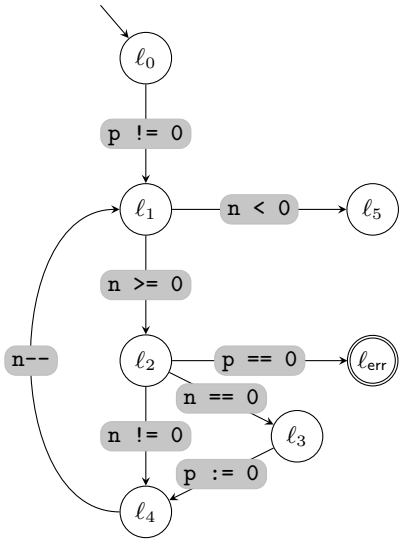


Fig. 1: Example program $\mathcal{P}_{\text{ex1}}$

Example 1: automata from infeasibility proofs

The program $\mathcal{P}_{\text{ex1}}$ in Figure 1 is the adaptation of an example in [15] to our setting. In our setting we use `assert` statements to define the correctness of the program executions. In the example of $\mathcal{P}_{\text{ex1}}$, an *incorrect* execution would start with a non-zero value for the variable `p` and, at some point, enter the body of the while loop when the value of `p` is 0 (and the execution of the `assert` statement *fails*).

We can argue the correctness of $\mathcal{P}_{\text{ex1}}$ rather directly if we split the executions into two cases, namely according to whether the `then` branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of `p` is never changed and remains non-zero (and the assert statement cannot fail). If the `then` branch of the conditional is executed, then the value of `n` is 0, the statement `n--` decrements the value of `n` from 0 to -1, and the while loop will exit directly, without executing the `assert` statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of $\mathcal{P}_{\text{ex1}}$.

We will describe an execution of $\mathcal{P}_{\text{ex1}}$ through the sequence of statements on the corresponding path in the *control flow graph* of $\mathcal{P}_{\text{ex1}}$; see Figure 1. The shortest path from ℓ_0 to ℓ_{err} goes via ℓ_1 and ℓ_2. The sequence of statements on this path is *infeasible* (it does not have a possible execution) because it is not

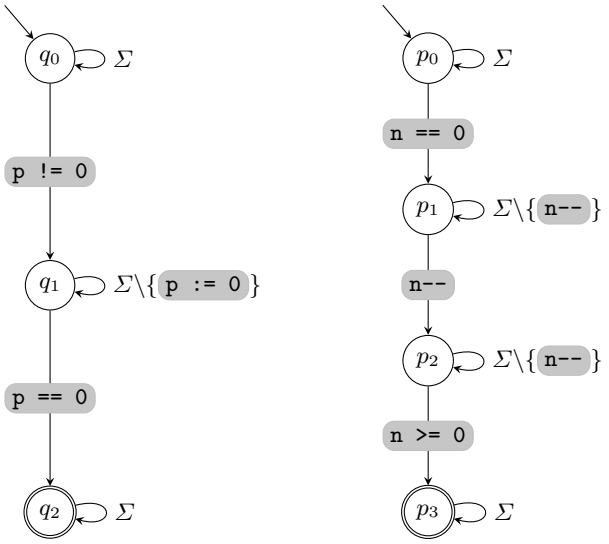


Fig. 2: Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ which are a proof of correctness for $\mathcal{P}_{\text{ex1}}$ (an edge labelled with Σ means a transition reading any letter, an edge labeled with $\Sigma\backslash\{$ `p := 0` $\}$) means a transition reading any letter except for `p := 0`, etc.)

possible to execute the `assume` statements `p!=0` and `p==0` without an update of `p` in between.

We construct the automaton $\mathcal{A}_1$ in Figure 2 which recognizes the set of all sequences of statements that contain `p!=0` and `p==0` without an update of `p` in between (and with any statements before or after). I.e., $\mathcal{A}_1$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of $p \neq 0$ and $p = 0$).

A sequence of statements is *not* accepted by $\mathcal{A}_1$ if it contains `p!=0` and `p==0` *with* an update of `p` in between. The shortest path from ℓ_0 to ℓ_{err} with such a sequence of statements goes from ℓ_2 to ℓ_{err} after it has gone from ℓ_2 to ℓ_3 once before. The sequence of statements on this path is infeasible for a new reason: it is not possible to execute the `assume` statement `n==0`, the update statement `n--`, and the `assume` statement `n>=0` unless there is an (other) update of `n` between `n==0` and `n--` or between `n--` and `n>=0`.

We construct the automaton $\mathcal{A}_2$ depicted in Figure 2 which recognizes the set of all sequences of statements that contain the statements `n==0`, `n--`, and `n>=0` without an update of `n` in between (and with any statements before or after). I.e., $\mathcal{A}_2$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts $n = 0$, $n' = n - 1$, and $n' \geq 0$).

To summarize, we have twice taken a path from ℓ_0 to ℓ_{err}, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set

of sequences of statements that are infeasible for the specific reason. The two automata thus characterize a case of executions in the sense discussed above.

Can one automatically check that every possible execution of $\mathcal{P}_{\text{ex1}}$ falls into one of the two cases? – The corresponding decision problem is undecidable. We can, however, check a condition which is stronger, namely that all sequences of statements on paths from ℓ_0 to ℓ_{err} in the control flow graph of $\mathcal{P}_{\text{ex1}}$ fall into one of the two cases (the condition is stronger because not every such path corresponds to a possible execution). The set of such sequences is the language recognized by an automaton which we also call $\mathcal{P}_{\text{ex1}}$ (recall that an automaton accepts a word exactly if the word labels a path from the initial state to a final state). Thus, the check amounts to checking the inclusion between automata, namely

$$\mathcal{P}_{\text{ex1}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2.$$

To rephrase our summary in the terminology of automata, we have twice taken a word accepted by the automaton $\mathcal{P}_{\text{ex1}}$, we have analyzed the reason of the infeasibility of the word (i.e., the corresponding sequence of statements), and we have constructed an automaton which recognizes the set of all words for which the same reason applies.

The view of a program as an automaton over the alphabet of statements may take some time to get used to because the view ignores the operational meaning of the program.

Example 2: automata from sets of Hoare triples

It is "easy" to justify the construction of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in Example 1: the infeasibility of a sequence of statements (such as the sequence `p!=0 p==0`) is preserved if one adds statements that do not modify any of the variables of the statements in the sequence (here, the variable `p`).

The example of the program $\mathcal{P}_{\text{ex2}}$ in Figure 3 shows that sometimes a more involved justification is required. The sequence of the two statements `x:=0` and `x==-1` (which labels a path from ℓ_0 to ℓ_{err}) is infeasible. However, the statement `x++` does modify the variable that appears in the two statements. So how can we account for the paths that loop in ℓ_2 taking the edge labeled `x++` one or more times? We need to construct an automaton that covers the case of those paths, but we cannot base the construction solely on infeasibility (as we did in Example 1).

We must base the construction of the automaton on a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion $x \geq 0$ holds after the update `x:=0`, that it is *invariant* under the updates `y:=0` and `x++`, and that is blocks the execution of the assume statement `x==-1`.

$$\begin{array}{l} \{\ \mathit{true}\ \}\ \texttt{x:=0}\ \{x \geq 0\} \\ \{x \geq 0\}\ \texttt{y:=0}\ \{x \geq 0\} \\ \{x \geq 0\}\ \texttt{x++}\ \{x \geq 0\} \\ \{x \geq 0\}\ \texttt{x==-1}\ \{\ \mathit{false}\ \} \end{array}$$

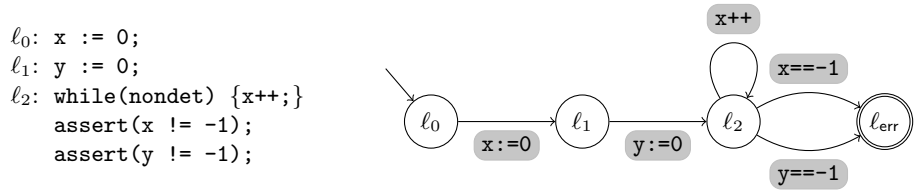


Fig. 3: Example program $\mathcal{P}_{\text{ex2}}$

The automaton $\mathcal{A}_1$ in Figure 4 has four transitions, one for each Hoare triple. It has three states, one for each assertion: the initial state q_0 for *true*, the state q_1 for $x \geq 0$, the (only) final state q_2 for *false*. The construction of such a *Floyd-Hoare automaton* generalizes to any set of Hoare triples. The resulting automaton can have arbitrary loops. In contrast, an automaton constructed as in the preceding example can only have self-loops.

Where does the set of Hoare triples come from? In this example, it may come from a static analysis [7] applied to the program fragment that corresponds to one path from ℓ_0 to ℓ_{err}; such a static analysis may assign an abstract value corresponding to $x \geq 0$ to the location ℓ_2 and determine that ℓ_{err} is not reachable.

In our implementation [11], the set of Hoare triples comes from an interpolating SMT solver [5] which generates the assertion $x \geq 0$ from the infeasibility proof.

The four Hoare triples below are sufficient to prove the infeasibility of all paths that reach the error location via the edge labeled with `y==-1`.

$$\begin{array}{lll} \{\ \mathit{true}\ \} & \texttt{x:=0} & \{\ \mathit{true}\ \} \\ \{\ \mathit{true}\ \} & \texttt{y:=0} & \{y = 0\} \\ \{y = 0\} & \texttt{x++} & \{y = 0\} \\ \{y = 0\} & \texttt{y==-1} & \{\ \mathit{false}\ \} \end{array}$$

We use them in the same way as above in order to construct the automaton $\mathcal{A}_2$ in Figure 4. The two automata are sufficient to prove the correctness of the program; i.e., $\mathcal{P}_{\text{ex2}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$.

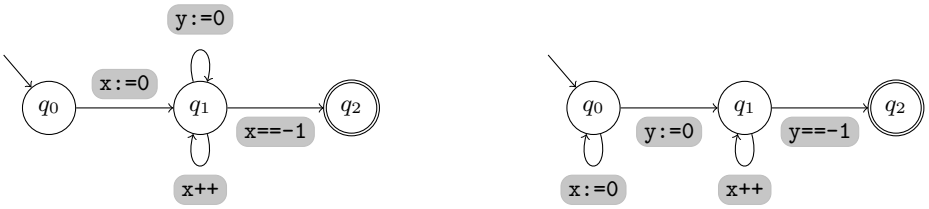


Fig. 4: Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for $\mathcal{P}_{\text{ex2}}$

The Hoare triple $\{y = 0\}$ `x++` $\{y = 0\}$ holds trivially. This is related to the fact that the statement `x++` does not modify the variable of the statements in the infeasible sequence `y:=0 y==-1`. I.e., we could have based the construction of the automaton $\mathcal{A}_2$ in Figure 4 on an infeasibility proof as for the automata in Example 1.

Example 3: automata for trace partitioning

The $\mathcal{P}_{\text{ex3}}$ in Figure 5 is a classical example used to motivate *trace partitioning* for static analysis (see, e.g., [18]). As shown in Figure 5, an interval analysis applied to the program will derive that the value of the variable `x` at location ℓ_3 lies in the interval $[-1, 1]$. This is not sufficient to prove that the error location is not reachable. One remedy is to *partition* the executions into two cases according to whether the execution takes the `then` or the `else` branch of the conditional. The static analysis applied to each of the two cases separately will derive that the value of the variable `x` at location ℓ_3 lies in the interval $[-1, -1]$ (in the `else` case) or in the interval $[1, 1]$ (in the `then` case). In either case, the static analysis derives the unreachability of the error location.

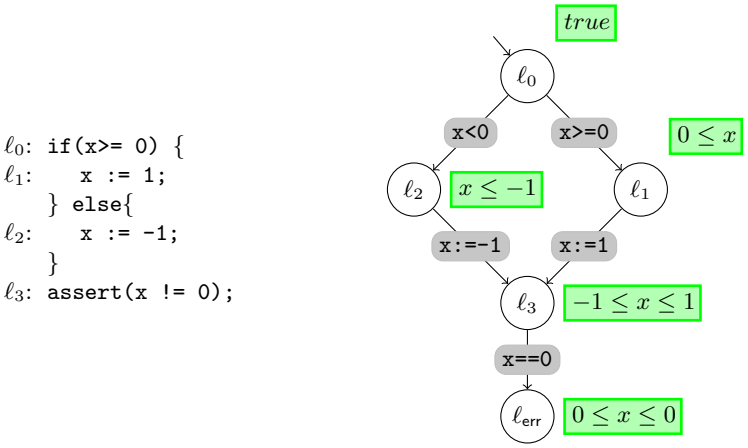
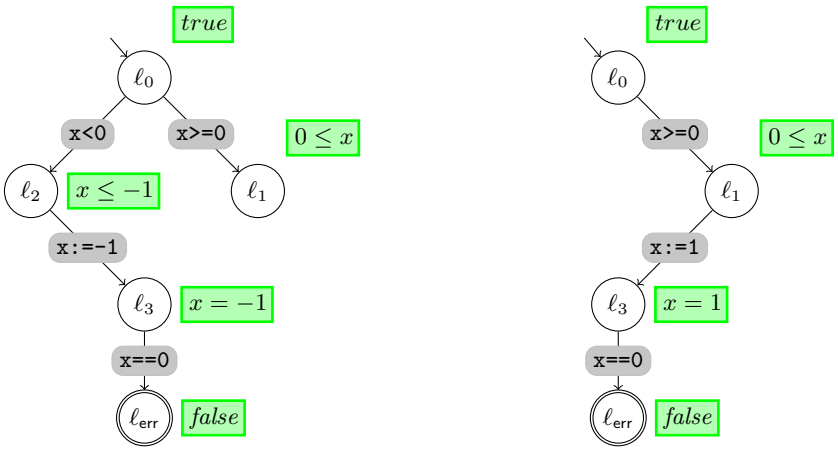


Fig. 5: Example program $\mathcal{P}_{\text{ex3}}$ (the labeling of program locations with assertions translates the result of an interval analysis; the labeling of the error location is not the assertion *false* which means that the interval analysis does not prove that the error location is unreachable; for each edge between two nodes, the two assertions and the statement form a Hoare triple)

We will use the example to illustrate how automata can be used to infer this kind of partitioning automatically for a given verification task.

Consider the partial annotation shown in Figure 6a. As in Figure 5, each edge corresponds to a Hoare triple, but there is no edge from ℓ_1 to ℓ_3. The



(a) $\mathcal{P}^1_{\text{ex3}}$ (b) $\mathcal{P}^2_{\text{ex3}}$

Fig. 6: Programs $\mathcal{P}^1_{\text{ex3}}$ and $\mathcal{P}^2_{\text{ex3}}$ obtained by automata-based trace partitioning (the labeling of program locations with assertions translates again the result of the interval analysis; the programs are defined by $\mathcal{P}^1_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \cap \mathcal{A}_1$ and $\mathcal{P}^2_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \setminus \mathcal{A}_1$ where the automaton $\mathcal{A}_1$ is constructed from an intermediate result of the interval analysis applied to $\mathcal{P}_{\text{ex3}}$; the intermediate result provides the assertions $x = -1$ for ℓ_3 and *false* for ℓ_{err} but does not provide a Hoare triple for the edge from ℓ_1 to ℓ_3; the executions of $\mathcal{P}^2_{\text{ex3}}$ are exactly those executions of $\mathcal{P}_{\text{ex3}}$ that have not yet been proven correct by the intermediate result of an interval analysis applied to $\mathcal{P}_{\text{ex3}}$)

partial annotation is established by taking the intermediate results of the static analysis. This includes in particular the Hoare triples $\{x \leq -1\}$ `x:=-1` $\{x = -1\}$ and $\{x \leq -1\}$ `x==0` $\{\mathit{false}\}$. We construct the automaton $\mathcal{A}_1$ from the set of Hoare triples used in the partial annotation. Since $\mathcal{A}_1$ has one state for each of the five assertions used in the partial annotation (namely *true*, $x \leq -1$, $x = -1$, *false* and $0 \leq x$), we can use the five program locations as automaton states and take the set of states $Q = \{\ell_0, \ldots, \ell_3, \ell_{\mathsf{err}}\}$. Since $\mathcal{A}_1$ has one transition for each of the four Hoare triples, the transitions are exactly the four edges in the graph in Figure 6a.

We now proceed to define the partition of the executions of $\mathcal{P}_{\text{ex3}}$. We compute the program $\mathcal{P}^1_{\text{ex3}}$ as the intersection of the program $\mathcal{P}_{\text{ex3}}$ with the automaton $\mathcal{A}_1$ and the program $\mathcal{P}^2_{\text{ex3}}$ as the difference between $\mathcal{P}_{\text{ex3}}$ and $\mathcal{A}_1$.

$$\mathcal{P}^1_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \cap \mathcal{A}_1$$
$$\mathcal{P}^2_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \setminus \mathcal{A}_1$$

We here exploit the fact that a program can be viewed as an automaton *and vice versa*. When we view a program as an automaton, we can apply set-theoretic operations (here intersection and set difference). When we view an automaton

as a program, we can consider its operational semantics, apply a static analysis, check its correctness, and so on.

The executions of $\mathcal{P}^1_{\text{ex3}}$ are exactly those executions of $\mathcal{P}_{\text{ex3}}$ that have been proven correct by the annotation in Figure 6a, and the executions of $\mathcal{P}^2_{\text{ex3}}$ are exactly those executions of $\mathcal{P}_{\text{ex3}}$ that have not yet been proven correct.

In our example, $\mathcal{P}^1_{\text{ex3}}$ happens to be equal to $\mathcal{A}_1$ (since $\mathcal{A}_1$ is a subset of $\mathcal{P}_{\text{ex3}}$). We depict the program $\mathcal{P}^2_{\text{ex3}}$ in Figure 6b. The two programs capture the above-mentioned two cases of executions. For each of them, the application of interval analysis proves the correctness, i.e., the unreachability of the error location.

2 Programs, Correctness, and Automata

We first present an abstract formal setting in which we define the notions of: trace, correctness of a trace, program, and correctness of a program, in terms of automata-theoretic concepts. To help intuition, we then discuss how the setting relates to some of the more concrete settings that are commonly used.

2.1 Formal setting

Trace τ. We assume a fixed set of statements Σ. A *trace* τ is a sequence of statements, i.e.,

$$\tau = \mathit{st}_1 \dots \mathit{st}_n$$

where $\mathit{st}_1, \dots, \mathit{st}_n \in \Sigma$ and $n \geq 0$ (the sequence is possibly empty). In order to connect our formal setting with automata theory, we view a statement st as a letter and a trace τ as a word over the alphabet Σ; i.e., $\tau \in \Sigma^\star$. Since one calls a set of words a *language*, the set of traces is the language of all words over the alphabet Σ.

$$\{\text{traces}\} = \Sigma^\star$$

Correctness of a trace τ. We assume a fixed set Φ of *assertions*. The set of assertions Φ contains the assertions *true* and *false* and comes with a binary relation, the *entailment* relation. We write $\varphi \models \psi$ if the assertion φ entails the assertion ψ.

We assume a fixed set of triples of the form $(\varphi, \mathit{st}, \psi)$ where φ and ψ are assertions in Φ and st is a statement in Σ. We say that every triple $(\varphi, \mathit{st}, \psi)$ in the set is a valid Hoare triple and we write

$$\{\varphi\}\ \mathit{st}\ \{\psi\} \quad \text{is valid.}$$

The Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$ is valid for the trace $\tau = \mathit{st}_1 \dots \mathit{st}_n$ if each of the Hoare triples below is valid, for some sequence of intermediate assertions $\varphi_1, \dots, \varphi_{n-1}$.

$$\{\varphi\}\ \mathit{st}_1\ \{\varphi_1\}, \dots, \{\varphi_{n-1}\}\ \mathit{st}_n\ \{\psi\}$$

If $n = 0$ and τ is the empty trace ($\tau = \varepsilon$) then φ must entail ψ.

In order to define correctness, we assume a fixed pair of assertions which we call the *pre/postcondition pair*,

$$(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}).$$

The trace τ is defined to be correct if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid.

$$\{\text{correct traces}\} = \{\tau \in \Sigma^\star \mid \{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\} \text{ is valid}\}$$

The notion of a trace and the correctness of a trace are independent of a given program. We will next introduce the notion of a program and define the set of its *control flow traces*. We can then define the correctness of the program: the program is correct if all its control flow traces are correct.

Program. We formalize a program $\mathcal{P}$ as a special kind of graph which we call a *control flow graph*. The vertices of the control flow graph are called *locations*. The (finite) set of locations Loc contains a distinguished *initial* location ℓ_0 and a subset F of distinguished *final* locations. The edges of the control flow graph are labeled with statements. We use δ for the labeled edge relation; i.e.,

$$\delta \subseteq \mathsf{Loc} \times \Sigma \times \mathsf{Loc}.$$

The edge between the two locations ℓ and ℓ' is labeled by the statement st if δ contains the triple (ℓ, st, ℓ').

Given a program $\mathcal{P}$, we say that the trace τ is a *control flow trace* if τ labels a path in the control flow graph between the initial location and a final location (the path need not be simple, i.e., it may repeat locations and edges).

Since a statement st is a letter of the alphabet Σ, the program

$$\mathcal{P} = (\mathsf{Loc}, \delta, \ell_0, F)$$

is an automaton over the alphabet Σ. Since a trace τ is a word (i.e., $\tau \in \Sigma^\star$), the automaton $\mathcal{P}$ recognizes a set of traces. We write $\mathcal{L}(\mathcal{P})$ for the language recognized by $\mathcal{P}$, which is a language of words over the alphabet Σ, i.e.,

$$\mathcal{L}(\mathcal{P}) \subseteq \Sigma^\star.$$

The condition that a trace τ is a control flow trace translates to the fact that the word τ is accepted by the automaton $\mathcal{P}$. Thus, the set of control flow traces is the language over the alphabet Σ which is recognized by $\mathcal{P}$, i.e.,

$$\{\text{control flow traces}\} = \mathcal{L}(\mathcal{P}).$$

Correctness of a program $\mathcal{P}$. We define that the program $\mathcal{P}$ is *correct* and write

$$\{\varphi_{\mathsf{pre}}\}\ \mathcal{P}\ \{\varphi_{\mathsf{post}}\}$$

if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid for every control flow trace τ of $\mathcal{P}$, which is equivalent to the inclusion

$$\{\text{control flow traces}\} \subseteq \{\text{correct traces}\}.$$

Thus, the correctness of a program is characterized by the inclusion between two languages over the alphabet of statements Σ. The language of correct traces is in general not recognizable by a finite automaton; if, for example, the one-letter alphabet consisting of the statement `x++`, the precondition "`x` is equal to 0" and the postcondition "`x` is a prime number", then the language of correct traces is the set of traces whose length is a prime number. However, for every correct program $\mathcal{P}$ there exists a finite automaton $\mathcal{A}$ that *interpolates* between the language of control flow traces and the language of correct traces (i.e., $\mathcal{A}$ accepts all control flow traces and $\mathcal{A}$ accepts only correct traces), formally

$$\{\text{control flow traces}\} \subseteq \mathcal{L}(\mathcal{A}) \subseteq \{\text{correct traces}\}.$$

The existence of such an automaton $\mathcal{A}$ follows for a trivial reason. If we assume that the program $\mathcal{P}$ is correct, then we can choose $\mathcal{A}$ to be the program $\mathcal{P}$ itself (by the definition of control flow traces, $\mathcal{P}$ accepts all control flow traces, and by the definition of program correctness, $\mathcal{P}$ accepts only correct traces). In fact, $\mathcal{P}$ is the *smallest* among all automata that one can use to prove the correctness of the program $\mathcal{P}$. Many existing methods for proving program correctness are restricted to this one example as the choice for the automaton $\mathcal{A}$. In Section 3 we will discuss other examples (examples of automata $\mathcal{A}$ that properly interpolate between the language of control flow traces and the language of correct traces).

2.2 Discussion

We next relate the abstract formal setting used above to some of the more concrete settings that are commonly used.

Assertions. Usually, an assertion φ is a first-order logic formula over a given vocabulary. Its variables are taken from a set Var of variables (the *program variables*). In our formal setting, we will not introduce states and related notions (state predicate, postcondition, ...). In a formal setting based on states, an assertion is used to define a state predicate (i.e., a set of valuations or states). There, the Hoare triple $\{\varphi\}\ st\ \{\psi\}$ signifies that the postcondition of φ under the statement st entails ψ, or: if the statement st is executed in a state that satisfies φ then the successor state satisfies ψ. In our formal setting, we abstract away from the procedure (first-order theorem prover, SMT solver, ...) used to establish entailment or the validity of a Hoare triple. We also abstract away from the specific procedure (static analysis, interpolant generation, ...) used to construct the sequence of intermediate assertions $\varphi_1, \ldots, \varphi_{n-1}$ needed to establish the validity of a Hoare triple for a trace of the length n.

Using `assume` statements. In order to accommodate control constructs like `if-then-else` and `while` of programming languages in a formal setting based on the control flow graph, we can use a form of statement that is often called `assume` statement. That is, for every assertion ψ we have an statement (also written ψ) such that the Hoare triple $\{\varphi\}\ \psi\ \{\varphi'\}$ is valid if the assertion φ' is entailed

by the conjunction $\psi \wedge \varphi$. The meaning of *assume* statements for the purpose of verification is clear. Their operational meaning is somewhat contrived: if an execution reaches the statement ψ in a state that satisfies the assertion ψ then the statement is ignored, and if an execution reaches the statement ψ in a state that violates the assertion ψ then the execution is blocked (and the successor location in the control flow graph is not reached).

Infeasibility $\implies$ *Correctness.* Formally, we define that a trace is infeasible if the Hoare triple

$$\{\mathit{true}\}\ \tau\ \{\mathit{false}\}$$

is valid. Intuitively, a trace τ is infeasible if there is no possible execution of the sequence of the statements in τ (in whatever valuation of the program variables the execution starts, one of the `assume` statements in the sequence cannot be executed). For example, the sequence of two `assume` statements `x==0` `x==1` is an infeasible trace; the sequence `x:=0` `x==1` is another example. An infeasible trace thus satisfies every possible pre/postcondition pair. In other words, an infeasible trace is correct (for whatever pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ defining the correctness).

The fact that infeasibility implies correctness is crucial. In general, the set of *feasible* correct control flow traces is not regular (the feasible control flow traces are exactly the sequences of statements along paths in the *transition system* of the program, i.e., in the—in general infinite—graph formed by transitions between program states). We obtain a regular set because we include the infeasible control flow traces (in addition to the feasible ones). As an aside, if the program $\mathcal{P}$ is not correct then the subset of correct control flow traces is in general not regular.

Non-reachability of error locations. In some settings, it is convenient to express the correctness of a program by the non-reachability of distinguished locations (often called *error locations*). In our setting, this corresponds to the special case where the set F consists of those locations and the postcondition φ_{post} is the assertion *false*. Moreover, if the precondition is *true*, the non-reachability of error locations is exactly the infeasibility of all control flow traces.

As mentioned above, an infeasible trace is correct (for any pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ that is used to define correctness of traces). In the special case of the pre/postcondition pair $(\mathit{true}, \mathit{false})$, the converse holds as well. I.e., in this case we have: a trace is correct exactly when it is infeasible.

Validity of `assert` *statements.* In other settings, it is convenient to express the correctness by the validity of *assert* statements. Informally, the statement `assert(e)` is valid if, whenever the statement is reached in an execution of the program, the Boolean expression e evaluates to *true*. This notion of correctness can be reduced to non-reachability (for each `assert` statement `assert(e)` for the expression `e`, one adds an edge to a new error location labeled with the assume statement `assume (not e)` for the negation of the expression `e`.)

Partial correctness. The partial correctness wrt. a general pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ can always be reduced to the partial correctness wrt. the special case of the precondition being *true* and the postcondition being *false* (by modifying the control flow graph of the program: one adds an edge from a new initial location to the old one labeled with the assume statement $[\varphi_{\mathsf{pre}}]$ for the precondition and an edge from each old final location to a new final location (an "error location") labeled with the assume statement $[\neg\varphi_{\mathsf{post}}]$ for the negated postcondition).

3 Floyd-Hoare Automata

Given an automaton $\mathcal{A} = (Q, \delta, q_0, Q_{\mathsf{final}})$ over the alphabet of statements Σ and given a pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, when do we know that $\mathcal{A}$ accepts only correct traces (i.e., traces τ such that the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid)? The definition below gives a general condition (and provides a general method to construct an automaton that accepts only correct traces).

Definition 1. *The automaton $\mathcal{A} = (Q, \delta, q_0, Q_{\mathit{final}})$ over the alphabet of statements Σ (with the finite set of states Q, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial state q_0 and the set of final states Q_{final}) is a* Floyd-Hoare automaton *if there exists a mapping*

$$q \in Q \ \mapsto\ \varphi_q \in \Phi$$

that assigns to each state q an assertion φ_q such that

- *for every transition $(q, \mathit{st}, q') \in \delta$ from state q to state q' reading the letter st, the Hoare triple $\{\varphi_q\}\ \mathit{st}\ \{\varphi_{q'}\}$ is valid for the assertions φ_q and $\varphi_{q'}$ assigned to q and q', respectively,*
- *the precondition φ_{pre} entails the assertion assigned to the initial state,*
- *the assertion assigned to a final state entails the postcondition φ_{post}.*

$$\begin{aligned} (q, \mathit{st}, q') \in \delta &\implies \{\varphi_q\}\ \mathit{st}\ \{\varphi_{q'}\} \quad \textit{is valid} \\ q = q_0 &\implies \varphi_{\mathit{pre}} \models \varphi_q \\ q \in Q_{\mathit{final}} &\implies \varphi_q \models \varphi_{\mathit{post}} \end{aligned}$$

The mapping $q \mapsto \varphi_q$ from states to assertions in the definition above is called an *annotation* of the automaton $\mathcal{A}$.

Theorem 1. *A Floyd-Hoare automaton $\mathcal{A}$ accepts only correct traces,*

$$\mathcal{L}(\mathcal{A}) \subseteq \{\text{correct traces}\}$$

i.e., if the trace τ is accepted by $\mathcal{A}$ then the Hoare triple $\{\varphi_{\mathit{pre}}\}\ \mathit{st}\ \{\varphi_{\mathit{post}}\}$ is valid.

Proof. We first prove, by induction over the length of the trace τ, the statement: for every pair of states q and q' and their corresponding assertions φ_q and $\varphi_{q'}$, if the trace τ labels some path from q to q' then the Hoare triple $\{\varphi_q\}\ \tau\ \{\varphi_{q'}\}$ is valid. The base case ($\tau = \varepsilon$ and $q = q'$) holds trivially. The induction step ($\tau' = \tau.\mathit{st}$ and $(q', \mathit{st}, q'') \in \delta$) uses the Hoare triple $\{\varphi_{q'}\}\ \mathit{st}\ \{\varphi_{q''}\}$ to show that $\{\varphi_q\}\ \tau.\mathit{st}\ \{\varphi_{q''}\}$ is valid. The theorem is the instance of the statement where we set q to the initial state and q' to a final state. □

A Floyd-Hoare automaton $\mathcal{A}$ is a correctness proof for the program $\mathcal{P}$ if it accepts all control flow traces, i.e., if $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A})$. Many (if not all) existing verification methods amount to constructing an annotation for the program $\mathcal{P}$ and thus to showing that $\mathcal{P}$ is a Floyd-Hoare automaton. Trivially, $\mathcal{P}$ can only be the smallest among all the Floyd-Hoare automata that prove the correctness of $\mathcal{P}$.

A well-known fact from the practice of automata is that the size of an automaton can be drastically reduced if the automaton is allowed to recognize a larger set. This is interesting in a setting where the size of the automaton we construct correlates with the number of Hoare triples we have to provide for an annotation. That is, instead of providing Hoare triples for an annotation of the control flow graph of the program $\mathcal{P}$ which recognizes exactly the set of control flow traces, it may be more efficient to provide Hoare triples for an annotation of the transition graph of an automaton $\mathcal{A}$ that recognizes a larger set (one can easily give examples of programs where an exponential reduction in proof size (when going from $\mathcal{P}$ to $\mathcal{A}$) can be obtained). If $\mathcal{A}$ is different from $\mathcal{P}$, one still needs to check that $\mathcal{A}$ is indeed a proof for $\mathcal{P}$, i.e., that the inclusion between the two automata, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A})$, does indeed hold. The efficiency of the proof check is thus an issue of efficient implementations of automata.

The next statement means that we can compose correctness proofs in the form of Floyd-Hoare automata to a correctness proof for the program $\mathcal{P}$.

Theorem 2. *If the Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ cover the set of control flow traces of the program $\mathcal{P}$ (i.e., $\mathcal{P} \subseteq \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$) then $\mathcal{P}$ is correct.*

Construction of a Floyd-Hoare automaton. The definition of Floyd-Hoare automata provides a general method to construct an automaton that accepts only correct traces, namely from a set of Hoare triples. We obtain different instances of the method by changing the approach to obtain the set of Hoare triples. For example, the set may stem from a *partial annotation* for the program, or the set of Hoare triples may be implicit from the *infeasibility proof* for a trace.

Let H be a finite set of Hoare triples, i.e., for each $(\varphi, \mathit{st}, \psi)$ in H, the Hoare triple $\{\varphi\}\ \mathit{st}\ \{\psi\}$ is valid (for now, we leave open how H is obtained). Let Φ_H be the finite set of assertions occurring in H. We assume that Φ_H includes the precondition φ_{pre} and the postcondition φ_{post}. We construct the Floyd-Hoare automaton $\mathcal{A}_H$ as follows.

$$\mathcal{A}_H = (Q_H, \delta, q_0, Q_{\mathsf{final}}) \quad \text{where} \quad \begin{aligned} Q_H &= \{q_\varphi \mid \varphi \in \Phi_H\} \\ \delta &= \{(q_\varphi, \mathit{st}, q_\psi) \mid (\varphi, \mathit{st}, \psi) \in H\} \\ q_0 &= q_{\varphi_{\mathsf{pre}}} \\ Q_{\mathsf{final}} &= \{q_{\varphi_{\mathsf{post}}}\} \end{aligned}$$

That is, we form the set of states Q_H by introducing a state q_φ for every assertion φ in H (i.e., Φ_H is bijective to Q_H). The transition relation δ defines a transition labeled by the letter st from the state q_φ to the state q_ψ for every Hoare triple $(\varphi, \mathit{st}, \psi)$ in H. The initial state is the state assigned to the precondition φ_{pre}. The (unique) final state is the state assigned to the postcondition φ_{post}.

Clearly, $\mathcal{A}_H$ is a Floyd-Hoare automaton: the inverse of the mapping $\varphi \mapsto q_\varphi$ is a mapping of states to assertions as required in Definition 1.

Construction of automaton from infeasibility proof. Assume, for example, that the trace $\tau = \mathit{st}_1 \ldots \mathit{st}_i \ldots \mathit{st}_j \ldots \mathit{st}_n$ is infeasible and that we have a proof of the form: the sequence of the two statements st_i and st_j is infeasible and the statements in between do not modify any variable used in st_i and st_j. We can construct an automaton with the set of states $Q = \{q_0, q_1, q_2\}$ as follows. The initial state q_0 has a transition to a state q_1 reading st_i. The state q_1 has a transition to the final state q_2 reading st_j. The initial state and the final state each have a self-loop standing for a transition reading any letter of the alphabet. The state q_1 has a self-loop standing for a transition reading any letter except for statements that modify a variable used in st_i or st_j. The construction generalizes to the case where the infeasibility involves more than two statements; see Example 1 in the introduction.

To see that this construction is a special case of the construction above, we take any assertion φ such that the two Hoare triples $\{\mathit{true}\}\ \mathit{st}_i\ \{\varphi\}$ and $\{\varphi\}\ \mathit{st}_j\ \{\mathit{false}\}$ are valid (for example, we can take the strongest postcondition of *true* under the statement st_i) and we form the set of Hoare triples H by those two Hoare triples, the Hoare triples of the form $\{\varphi\}\ \mathit{st}\ \{\varphi\}$ for any statement st in Σ that does not modify a variable used in st_i or st_j, and the trivial Hoare triples $\{\mathit{true}\}\ \mathit{st}\ \{\mathit{true}\}$ and $\{\mathit{false}\}\ \mathit{st}\ \{\mathit{false}\}$ for every statement st in Σ. Thus, the special case consists of leaving the assertion φ implicit. This is not always possible; see Example 2 in the introduction.

Construction of a correctness proof for the program $\mathcal{P}$. By Theorem 2, we can construct an automaton $\mathcal{A}$ for a correctness proof for the program $\mathcal{P}$, i.e.,

$$\{\text{control flow traces}\} \subseteq \mathcal{L}(\mathcal{A}) \subseteq \{\text{correct traces}\}. \tag{1}$$

as the union of Floyd-Hoare automata, i.e., $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$. The construction of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ can be done in parallel from the n correctness proofs (i.e., infeasibility proofs or sets of Hoare triples) for some choice of traces $\tau_1, \ldots, \tau_n$. The construction of $\mathcal{A}$ as the union $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$ can also be done incrementally (for $n = 0, 1, 2, \ldots$) until (1) holds. Namely, if the inclusion does not yet hold (which is the case initially, when $n = 0$), then there exists a control flow trace τ_{n+1} which is not in $\mathcal{A}$. We then construct the automaton $\mathcal{A}_{n+1}$ from the proof for the trace τ_{n+1} and add it to the union, i.e., we update $\mathcal{A}$ to $\mathcal{A} \cup \mathcal{A}_{n+1}$. This is how we proceeded for the examples in the introduction.

If (1) holds for $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$ then the programs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ where $\mathcal{P}_i = \mathcal{P} \cap \mathcal{A}_i$ (for $i = 1, \ldots, n$) define a decomposition of the program $\mathcal{P}$ (i.e., $\mathcal{P} = \mathcal{P}_1 \cup \ldots \cup \mathcal{P}_n$). This decomposition is constructed automatically from correctness proofs (in contrast with an approach where one constructs correctness proofs for the modules of a given decomposition).

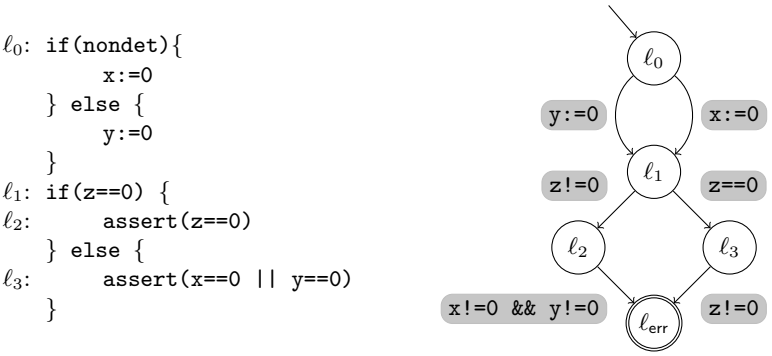


Fig. 7: Example program where it is useful to remove the restriction that the correctness argument must be based on (an unfolding of) the control flow graph

4 Conclusion and Future Work

We have presented a new angle of attack at the problem of finding the right abstraction of a program for a given correctness property. Existing approaches (see our list of references) differ mainly in their techniques to decompose the problem. Often the techniques (unfolding, splitting nodes, abstract states, ...) are based on the control flow graph. Phrased in the terminology of our setting, the techniques amount to constructing a cover (often a partitioning) of the set of control flow traces by automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The construction is restricted in that the automata must be merged into one automaton and, moreover, the states and transitions of the resulting automaton must be in direct correspondence with the nodes and edges of the control flow graph. This is needed to ensure that all control flow traces are indeed covered (in the absence of an inclusion check). Our approach allows one to remove this restriction. The example in Figure 7 may be used to illustrate the difference between the approaches.

It is a topic of future work to position existing approaches to software model checking in our setting. Since an *abstraction refinement* step eliminates in general not just one counterexample trace but a whole set, it may be interesting to characterize this set by an automaton and thus quantify the *progress* property.

The setting presented here can be extended to automata over *nested words* in order to account for programs with (possibly recursive) procedures [12], and to *alternating automata* in order to account for concurrent programs [9]. It is still open how one can extend the setting to *Büchi automata* in order to account for termination and cost analysis, although the use of omega-regular expressions to decompose the set of infinite traces in [10] may be a step in this direction.

The development of a practical method based on Floyd-Hoare automata must address a wide range of design choices. An initial implementation is part of ongoing work [11].

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, pages 158–172. Springer, 2002.
2. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309. ACM, 2007.
3. M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, and R. Huuck. High performance static analysis for industry. *ENTCS*, 289:3–14, 2012.
4. I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In *FSEN*, pages 17–32. Springer, 2007.
5. J. Christ, J. Hoenicke, and A. Nutz. Proof tree preserving interpolation. In *TACAS*, pages 124–138, 2013.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer, 2000.
7. 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.
8. P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, pages 333–348, 2007.
9. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142, 2013.
10. S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.
11. M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol (competition contribution). In *TACAS*, pages 641–643, 2013.
12. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, pages 471–482, 2010.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 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*, pages 287–300. Springer, 2007.
15. M. Junker, R. Huuck, A. Fehnker, and A. Knapp. SMT-based false positive elimination in static program analysis. In *ICFEM*, pages 316–331, 2012.
16. D. Kroening and G. Weissenbacher. Interpolation-based software verification with Wolverine. In *CAV*, pages 573–578, 2011.
17. Z. Long, G. Calin, R. Majumdar, and R. Meyer. Language-theoretic abstraction refinement. In *FASE*, pages 362–376, 2012.
18. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, pages 5–20, 2005.
19. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136. Springer, 2006.
20. M. Segelken. Abstraction and counterexample-guided construction of *omega* - automata for model checking of step-discrete linear hybrid models. In *CAV*, pages 433–448. Springer, 2007.
21. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986.