Automated Program Verification

Azadeh Farzan[2], Matthias Heizmann[1(✉)], Jochen Hoenicke[1], Zachary Kincaid[2], and Andreas Podelski[1]

[1] University of Freiburg, Freiburg im Breisgau, Germany
heizmann@informatik.uni-freiburg.de
[2] University of Toronto, Toronto, Canada

Abstract. A new approach to program verification is based on automata. The notion of automaton depends on the verification problem at hand (nested word automata for recursion, Büchi automata for termination, a form of data automata for parametrized programs, etc.). The approach is to first construct an automaton for the candidate proof and then check its validity via automata inclusion. The originality of the approach lies in the construction of an automaton from a correctness proof of a given sequence of statements. A sequence of statements is at the same time a word over a finite alphabet and it is (a very simple case of) a program. Just as we ask whether a word has an accepting run, we can ask whether a sequence of statements has a correctness proof (of a certain form). The automaton accepts exactly the sequences that do.

1 Introduction

The verification of a program can often be divided into two steps: 1) the construction of a candidate proof and 2) the check of the validity of the candidate proof for the given program. An example is the construction of a Floyd-Hoare style annotation and the check of its inductiveness. In a new approach to program verification, the candidate proof in Step 1 comes in the form of an automaton and Step 2 is reduced to an automata inclusion test. If the inclusion test succeeds, the program is proven correct. The approach lends itself to a verification algorithm in the form of a loop: the automaton for the candidate proof is constructed incrementally until the inclusion holds (see also Figure 1 in Section 2).

The approach introduces a novel separation between

- the symbolic reasoning about data and
- the automata-theoretic reasoning about control.

By data we mean the values of program variables (e.g., integers) which are read and written by program statements. Examples of statements are tests of conditions and updates. We apply symbolic reasoning to mechanize the analysis of the data and produce a correctness proof for the sequence of statements. For example, we can first translate the sequence of statements into a logical formula (in the logical theory corresponding to the data domain) and then apply a dedicated decision procedure (as implemented by an SMT solver).

A.-H. Dediu et al. (Eds.): LATA 2015, LNCS 8977, pp. 25–46, 2015.
DOI: 10.1007/978-3-319-15579-1_2

The originality of the approach lies in the construction of the automaton from a correctness proof of a given sequence of statements. The construction relies on the following observation: one can first decompose the correctness proof into its base components and then *rearrange* the base components to obtain a correctness proof for a new sequence built up from the same set of statements. In the automaton that we construct, the non-determinism reflects the combinatorial choice of ways to rearrange the base components. A sequence of statements is at the same time a word over a finite alphabet and it is (a very simple case of) a program. Just as we ask whether a word has an accepting run, we can ask whether a sequence of statements has a correctness proof (one which can be obtained by rearranging the base components). The automaton accepts exactly the sequences that do.

The control of the program can be expressed through a graph, the so-called control flow graph of the program. The paths in the graph define the set of sequences of statements that are possible according to the control flow alone (i.e., ignoring the data and ignoring in particular the outcome of tests of conditions). It is this set of sequences which is the language recognized by the *program automaton* (we here use the finite set of the statements in the program as the alphabet and sequences of statements as words).

The control of the program comes in only in Step 2. We test the inclusion between the program automaton and the automaton for the candidate proof. The inclusion means that each sequence of statements that is possible according to the control flow of the program has a correctness proof.

Roadmap. In the remainder of this paper, we will instantiate the approach for six different verification problems. Each verification problem requires a specific class of automata. In the table below, *unbounded parallelism* refers to programs with an unbounded number of threads, and *predicate automata* are a new version of data automata that we introduce. The term *proofs that count* refers to programs whose verification involves the task to synthesize ghost variables that count. Each verification problem poses a new challenge in finding an appropriate notion of automata for the program and for the candidate proof, a way of representing and constructing the automata, and finally an algorithm for checking automata inclusion. We will explain each challenge and our approach to the solution informally, by way of examples. For technical details, we refer to the corresponding paper.

	verification problem	inclusion problem	reference
Section 2	sequential programs	nondeterministic finite automata	[10,12]
Section 3	termination	Büchi automata	[13]
Section 4	recursion	nested word automata	[11]
Section 5	concurrency	alternating finite automata	[4]
Section 6	unbounded parallelism	predicate automata	[6]
Section 7	proofs that count	Petri net $\subseteq$ counting automaton	[5]

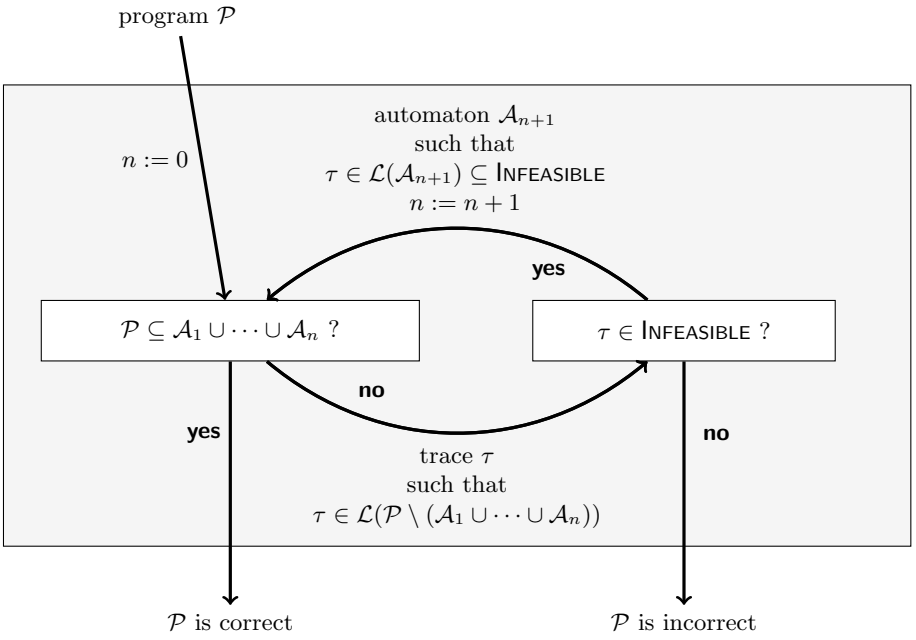


Fig. 1. Automated program verification

2 Sequential Programs: Nondeterministic Finite Automata

In this section we instantiate the approach for verifying sequential programs. We present two ways to construct an automaton from the correctness proof of a sequence of statements. We often refer to a sequence of statements as a *trace*.

Automata from unsatisfiable cores. The program $\mathcal{P}_{\text{ex1}}$ in Figure 2 is the adaptation of an example in [14]. The original program in [14] allocates a pointer `p` and then enters a while loop which *uses* `p` and conditionally *frees* `p`. The original correctness property in [14] is "the pointer `p` is not *used* after it has been *freed*."

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*).

Informally, 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,

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

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

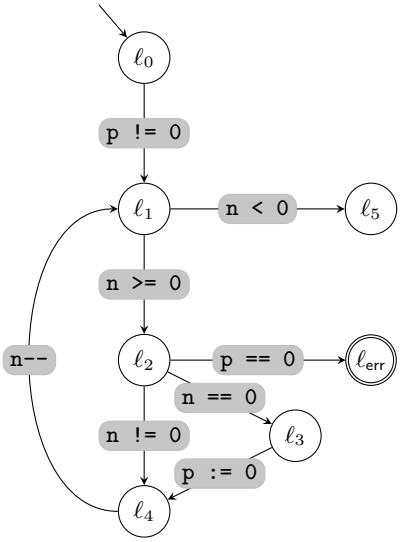


Fig. 2. Example program $\mathcal{P}_{\text{ex1}}$

and the while loop will exit directly, without executing the `assert` statement. It turns out that our verification algorithm will automatically reproduce this case split.

The algorithm starts with the sequence of statements on some path from ℓ_0 to ℓ_{err} in the *control flow graph* of $\mathcal{P}_{\text{ex1}}$ (see Figure 2). We take the shortest path which goes from ℓ_0 to ℓ_{err} via ℓ_1 and ℓ_2. The sequence of statements on this path is *infeasible* because it is not possible to execute the `assume` statements `p!=0` and `p==0` without an update of `p` in between. Formally, the formula obtained by translating the sequence of statements is unsatisfiable, and the conjuncts $p \neq 0$ and $p = 0$ form an *unsatisfiable core* of the formula.

We construct the automaton $\mathcal{A}_1$ in Figure 3 by first constructing an automaton that accepts only the sequence of the `assume` statements `p!=0` and `p==0` and then adding a number of self-loops. The idea behind the construction is that the sequence of statements remains infeasible if we add any statement before or after and any statement other than an update of `p` in-between.

The automaton $\mathcal{A}_1$ does not accept a sequence of statements *with* an update of `p` in between the statements `p!=0` and `p==0`. 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 again infeasible: it is not possible to execute the `assume` statement `n==0`, the update statement `n--`, and then the `assume` statement `n>=0` (without an update of `n` between `n==0` and `n--` and between `n--` and `n>=0`).

We construct the automaton $\mathcal{A}_2$ depicted in Figure 3 in the analogous way. Now, the unsatisfiable core corresponds to the sequence of the statements `n==0`, `n--`, and `n>=0`. Thus, we first construct an automaton that accepts only this

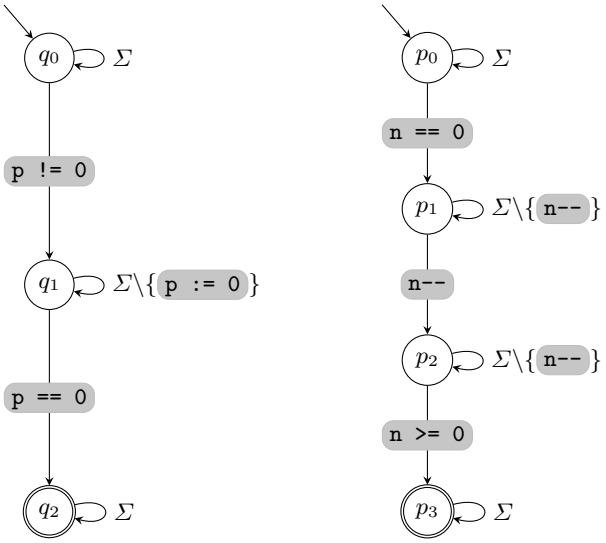


Fig. 3. Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ whose union forms a proof of correctness for $\mathcal{P}_{\text{ex1}}$ (an edge labeled with Σ means a transition reading any letter, an edge labeled with $\Sigma\backslash\{$ `p := 0` $\}$) means a transition reading any letter except for `p := 0`)

sequence and then add a number of self-loops. Now we are careful to not add a self-loop with an update of `n`.

To summarize, we have twice taken a path from ℓ_0 to ℓ_{err} and constructed an automaton from the unsatisfiable core of the proof of the infeasibility of the sequence of statements on the path.

The control flow graph $\mathcal{P}_{\text{ex1}}$ defines an automaton that recognizes the set of all sequences of statements on paths from ℓ_0 to ℓ_{err}. We can thus check that all such sequences are accepted by one of the two automata by testing the inclusion

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

Automata from sets of Hoare triples. It is "easy" to justify the construction of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in the example above: 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 4 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 can no longer base the construction solely on unsatisfiable cores.

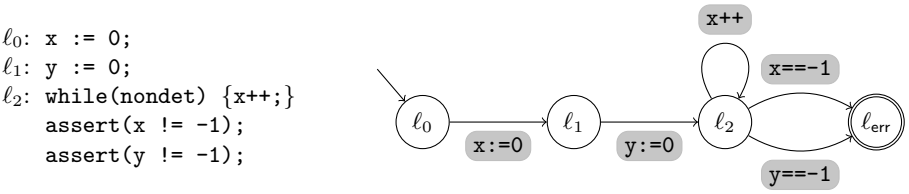


Fig. 4. Example program $\mathcal{P}_{\text{ex2}}$

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`.

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

The automaton $\mathcal{A}_1$ in Figure 5 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.

In our implementation [8], the set of Hoare triples comes from an interpolating SMT solver such as [2] 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`.

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

We use them in the same way as above in order to construct the automaton $\mathcal{A}_2$ in Figure 5. The two automata are sufficient to prove the correctness of the program; i.e., $\mathcal{P}_{\text{ex2}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$.

We could have based the construction of the automaton $\mathcal{A}_2$ in Figure 5 on the unsatisfiable core of the infeasibility proof, as in the example of $\mathcal{P}_{\text{ex2}}$. Intuitively, we do not need to know the precise form of the assertion $y \geq 0$ in order to know that it is invariant under `x++`. It is sufficient to know that the variabe x does not occur in the assertion (which we can assume because x does not appear in the unsatisfiable core).

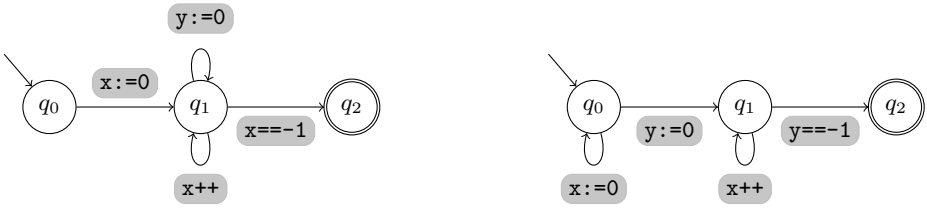


Fig. 5. Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for $\mathcal{P}_{\text{ex2}}$

To summarize, we have presented two ways to construct an automaton from the correctness proof of a sequence of statements. The first gets away without the synthesis of assertions, but the second is more general and leads to a complete verification method [10].

In the verification algorithm depicted in Figure 1, the union of the automata constructed from the correctness proofs for sequences of statements is constructed incrementally until the inclusion holds. In our implementation [8], we need not construct the union explicitly. Instead, we can incrementally construct the difference automaton $\mathcal{P} \setminus (\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n)$.

3 Termination: Büchi Automata

In this section we present how we use Büchi automata to construct a termination proof of a program.

In the presence of loops with branching or nesting, the termination proof has to account for all possible interleavings between the different paths through the loop. If the program is *lasso-shaped* (a stem followed by a single loop without branching), the control flow is trivial: there is only one path. Consequently, the termination proof can be very simple. Many procedures are specialized to lasso-shaped programs and derive a simple termination proof rather efficiently [9,16,18]. The relevance of lasso-shaped programs stems from their use as the representation of an *ultimately periodic* infinite trace through the control flow graph of a program with arbitrary nesting (the *period*, i.e., the cycle of the lasso, may itself go through a sequence of loops in the program).

We can explain our algorithm informally using the program $\mathcal{P}^{\texttt{sort}}$ depicted in Figure 6 which is an implementation of bubblesort. We begin by picking some ω-trace of $\mathcal{P}^{\texttt{sort}}$. We take the trace that first enters the outer while loop and then takes the inner while loop infinitely often. We denote this trace using the ω-regular expression $\textsc{Outer}.\textsc{Inner}^{\omega}$. We see that this trace is terminating: its termination can be shown using the linear ranking function $f(\texttt{i},\texttt{j}) = \texttt{i} - \texttt{j}$. Moreover, we see that this ranking function is applicable not only to this trace, but to all traces that eventually always take the inner loop. Such traces can be represented by the ω-regular expression

$$(\textsc{Inner} + \textsc{Outer})^*.\textsc{Inner}^{\omega} \ . \tag{1}$$

```
program sort(int i)
ℓ1: while (i>0)
ℓ2:    int j:=1
ℓ3:    while(j<i)
       //   if (a[j]>a[i])
       //      swap(a[j],a[i])
ℓ4:       j++
ℓ5:    i--
```

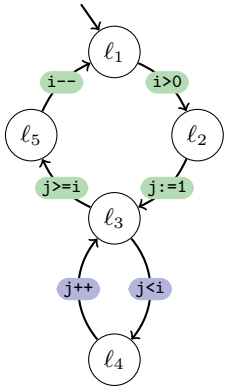


Fig. 6. Program $\mathcal{P}^{\texttt{sort}}$ which is an implementation of bubblesort

Now, let us pick another ω-trace from $\mathcal{P}^{\texttt{sort}}$. This time we take the trace that always takes the outer while loop. We see that this trace is terminating. Its termination can be shown using the linear ranking function $f(\texttt{i},\texttt{j}) = \texttt{i}$. Moreover, we see that this ranking function is applicable not only to this trace, but to all traces that take the outer while loop infinitely often, as represented by the ω-regular expression

$$(\textsc{Inner}^*.\textsc{Outer})^\omega \ . \tag{2}$$

Finally, we consider the set of all ω-trace of the program $\mathcal{P}^{\texttt{sort}}$

$$(\textsc{Outer} + \textsc{Inner})^\omega,$$

check that each trace has the form (1) or has the form (2), and conclude that $\mathcal{P}^{\texttt{sort}}$ is terminating.

The approach is based on the notion of an *ω-trace*, which is an infinite sequence of program statements $\pi = st_1st_2\ldots$. Like in the section before, we assume that the statements are taken from a given finite set of program statements Σ. If we consider Σ as an alphabet and each statement as a letter, then an *ω-trace* is an infinite word over this alphabet. For example, we can write the alphabet of our running example $\mathcal{P}^{\texttt{sort}}$ as $\Sigma_{sort} = \{$`i>0`, `j:=1`, `j<i`, `j++`, `j>=i`, `i--`$\}$ and $\pi =$ `j<i` `j:=1` $.($`j:=1` `j++` `j:=1`$)^\omega$ is an ω-trace.

We call an ω-trace *terminating* if it does not correspond to any possible execution (i.e., if there is no starting state such that all statements in the trace can be executed). The ω-traces (`x<0` `x:=1`)$^\omega$ and (`x>=0` `x--`)$^\omega$ are terminating. In the first one, already the finite prefix `x<0` `x:=1` `x<0` does not correspond to any possible execution. In the second, every finite prefix has a possible execution (for a prefix of length $2n$, take a starting state where x is greater than $n-1$).

As before, a program is represented as a control flow graph whose edges are labeled with statements (one node is singled out as the initial nodes; here,

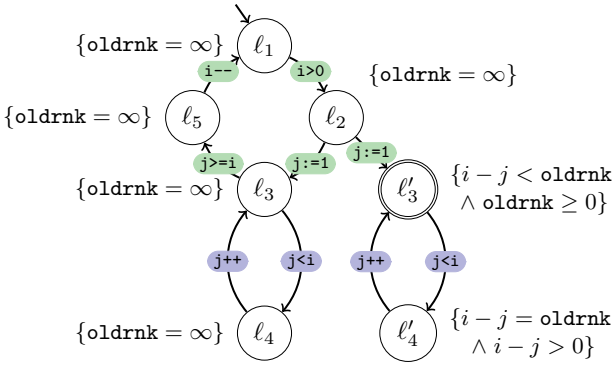
there are no error nodes). We may view a program $\mathcal{P} = \langle \mathsf{Loc}, \delta, \ell_{\mathsf{init}} \rangle$ as a *Büchi automaton* where every state is a final state. We call the program $\mathcal{P}$ *terminating* if each of its ω-traces is terminating.

We define a *module* to be a restricted form of Büchi automaton which has exactly one final state. A Büchi automaton of this form recognizes an ω-regular language of the form $U.V^\omega$, where U and V are regular languages over the alphabet of statements $U, V \subseteq \Sigma^*$.

A *fair ω-trace* of a module $\mathcal{P}$ is an ω-trace that labels a fair path in the graph of $\mathcal{P}$, i.e., a path that visits the distinguished location ℓ_{fin} infinitely often. We call the module $\mathcal{P}$ *terminating* if each of its fair ω-traces is terminating. A *non-fair* ω-trace of a terminating module (i.e., an ω-trace that labels a path in its control flow graph without satisfying the fairness constraint) can be non-terminating.

We define a *certified module* to be a module that is equipped with a termination argument. The termination argument consists of two parts: a ranking function and an annotation of the module's location with assertions that certify that the ranking function decreases every time the final location ℓ_{fin} is visited. The certificate ensures that the module is terminating (every fair ω-trace of the module terminates).

The figure on the right depicts a certified module $(\mathcal{P}_1^{\mathtt{sort}}, f, \mathcal{I})$ where f is the ranking function $f(i,j) = i - j$ and $\mathcal{I}$ is the mapping of locations to predicates indicated by writing the predicate beneath the location.



The variables `oldrnk` is an auxiliary variable whose value is the value of the ranking function at the previous visit of the final location. We note that for each transition (ℓ, st, ℓ') the corresponding triple $\{\mathcal{I}(\ell)\}\ st\ \{\mathcal{I}(\ell')\}$ is a valid Hoare triple (with the understanding that outgoing transitions of final states implicitly assign $\mathtt{oldrnk} := f(\mathtt{i}, \mathtt{j})$).

4 Recursion: Nested Word Automata

A new verification method for recursive programs is based on the theory of nested words [1]. The verification method constructs a nested word automaton from an inductive sequence of "nested interpolants", i.e., an inductive annotation for the "nested trace" of the recursive program with assertions. Such an annotation may come from an interpolating SMT solver such as [2].

The theory of *nested word automata* offers an interesting potential as an alternative to the low-level view of a recursive program as a stack-based device

that defines a set of traces. A nested word expresses not only the linear order of a trace but also the nesting of calls and returns. Regular languages of nested words enjoy the standard properties of regular language theory, of which we will use the closure under intersection and complement, and the decidability of emptiness [1].

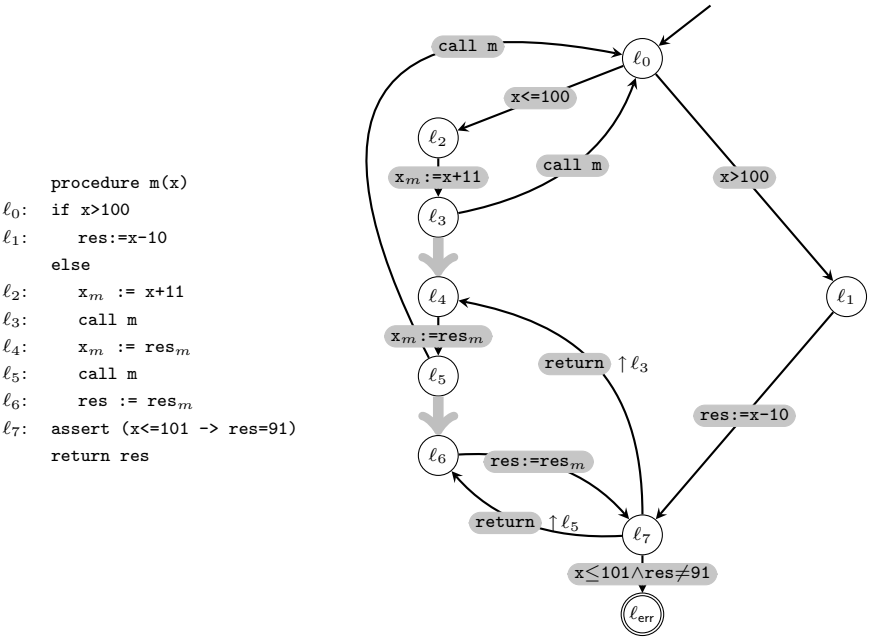


Fig. 7. McCarthy's 91 function with correctness specification given as pseudocode and recursive control flow graph $\mathcal{P}^{91}$. The program is correct if the assert statement never fails resp. if there is no feasible trace from the initial location ℓ_0 to the error location ℓ_{err}. In a different reading, the graph presents a nested word automaton, the *control automaton* $\mathcal{P}^{91}$.

Figure 7 shows an implementation of McCarthy's 91 function,

$$m(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ m(m(x + 11)) & \text{if } x \leq 100 \end{cases}$$

together with the correctness specification (if the argument x is not greater than 101, the function returns 91).

Following [19], we present a recursive program formally as a *recursive control flow graph*; see Figure 7 for an example. Each node is a program location ℓ. Each edge is labeled with a statement st, which is either an assignment `y:=t`, an assume φ, a call `call p`, or a return `return p`. We note that transitions

labeled with `return p` have two predecessors. First the exit location of the called procedure, second the location of the corresponding procedure call. In Figure 7 we label edges additionally with $\uparrow\ell$ (reminiscent to pop transitions in a pushdown automaton) to denote the location of the corresponding call transition.

Following [1], a *nested word* over an alphabet Σ is a pair $(w, \rightsquigarrow)$ consisting of a word $w = a_0 \dots a_{n-1}$ over the alphabet Σ and the *nesting relation* $\rightsquigarrow$ (a binary relation between the n positions of w). We can use the nesting relation $i \rightsquigarrow j$ to express that i is the position of a call and j the position of the matching return.

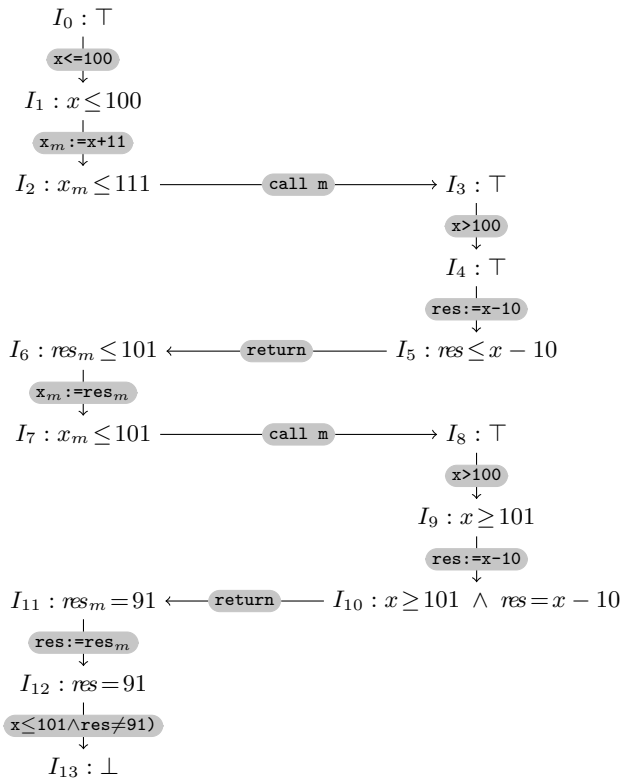


Fig. 8. Error trace of $\mathcal{P}^{91}$ in Figure 7 annotated with an inductive sequence of state assertions that prove infeasibility of this trace

In Figure 8, we present an error trace (a nested word accepted by $\mathcal{P}^{91}$) that is infeasible. The trace is interleaved with an inductive sequence of state assertions that proves infeasibility of this trace. The sequence of state assertions is modular in the sense that each state assertion describes only local states of the current calling context.

The global state of the program can be obtained using the nesting relation of the nested word. We call such a sequence of state assertions a sequence of nested interpolants. A method that uses Craig interpolation to compute a sequence of nested interpolants is presented in [11]. We note that we cannot apply Craig interpolation directly to a nested trace, because then variables of parent contexts may occur in the current context. Using nested interpolants and nested word automata we can use the scheme presented in Section 2 to analyze programs with procedures in a modular way.

5 Concurrent Programs: Alternating Finite Automata

In principle, one could apply the method developed in Section 2 to verify concurrent programs with shared memory. If each thread of a program is represented as an NFA, then their Cartesian product gives an NFA which recognizes the set of interleaved traces of the program. The challenge posed by concurrency is that the size of this Cartesian product is exponential in the number of threads, and the number of interleaved traces is greater still.

In [4], we propose a method for overcoming the exponential explosion problem using a novel proof system which is based on the notion of an *inductive data flow graph (iDFG)*. An iDFG is a data flow graph with incorporated inductive assertions. It accounts for a set of dependencies between data operations in interleaved traces. It stands as a representation for the set of traces which give rise to these dependencies, and acts as certificate that each of these traces is infeasible. This set of traces can be recognized by an *alternating finite automaton (AFA)*, enabling the reduction of the iDFG proof checking problem to a language inclusion problem for AFA. This problem suffers from high worst-case complexity (PSPACE-complete), but this is vastly superior to the exponential space complexity (not just in the worst case) of constructing the Cartesian product.

Thread i

$\ell_{i,1}$: `m`$_i$ `:= t++`
$\ell_{i,2}$: `[`$\mathtt{m}_i \leq s$`]`
`// critical section`
$\ell_{i,3}$: `s := s + 1`

We will use the Ticket mutual exclusion protocol as an example to illustrate iDFGs. The program has two global variables, `t` and `s`, representing a ticket counter and service counter, respectively. We suppose that the protocol is executed by three threads (Threads 1, 2, and 3), where each Thread i is executing the sequence of three instructions shown to the right. The program begins in a state where `s` and `t` are both zero. To execute the protocol, a thread first acquires its (unique ticket) and stores it in the local variable $\mathtt{m}_i$ ($\ell_{i,1}$), then waits until the service counter reaches its ticket to enter its critical section ($\ell_{i,2}$), and then finally leaves its critical section by incrementing the service counter ($\ell_{i,3}$). The property we wish to prove is mutual exclusion: no two threads may be in the critical section at the same time. We accomplish this by proving that every trace which *violates* mutual exclusion is infeasible.

One trace of the program which violates mutual exclusion (Thread 2 and Thread 3 both end in their critical sections) is pictured (on the left hand side) in Figure 9, along with a Hoare-style proof of its infeasibility. To its right is an

iDFG, which represents the *essence* of this proof: the trace is infeasible because Thread 3 enters its critical section when it is Thread 2's turn to do so. The iDFG abstracts away the details of the Hoare-style proof which are irrelevant to this essential argument, such as the relative order between the events `s++` and $\texttt{m}_3$ `:= t++`, or whether the events $[\texttt{m}_1 \leq \texttt{s}]$ or $[\texttt{m}_2 \leq \texttt{s}]$ occur at all. The graph is labeled with program assertions on each edge, where each incoming edge represents a pre-condition, and each outgoing edge a post-condition. Bifurcation in the graph represents pre-conditions which can potentially be achieved in parallel. For example, consider the two incoming edges to $[\texttt{m}_3 \leq \texttt{s}]$: it does not matter in which order the pre-conditions $\{\texttt{s} = 1\}$ and $\{\texttt{m}_3 > 1\}$ are achieved; as long as both hold when $[\texttt{m}_3 \leq \texttt{s}]$ is executed, then the resulting state will satisfy the post-condition $\{\mathit{false}\}$. The assertions are *inductive* in the sense that each node corresponds to a valid Hoare triple, where the pre-condition is the conjunction of the labels of all incoming edges, and the post-condition is the conjunction of the labels of all outgoing edges.

Each edge in the iDFG represents a constraint on the traces which are recognized by the iDFG. For example, the edge $\texttt{s++} \xrightarrow{\{s=1\}} [\texttt{m}_3 \leq \texttt{s}]$ indicates that `s++` must appear before $[\texttt{m}_3 \leq \texttt{s}]$ in the trace, and every instruction which appears in between must leave the assertion $\{\texttt{s} = 1\}$ invariant. A trace is recognized by the iDFG when it satisfies all of these constraints. The inductiveness condition for the assertion labels ensures that every trace which is recognized by the iDFG is infeasible.

A more operational view of the language of traces recognized by an iDFG can be given by translation into an AFA. AFAs may be understood as a generalization of nondeterministic finite automata. We may think of NFAs as having a transition function which maps each state and letter to a disjunction of states, with the interpretation that at least one of them must lead to an accepting state for the input word to be accepted. AFAs generalize this by also allowing *conjunctions* of states, with the interpretation that *all* states must lead to an accepting state for the input word to be accepted (i.e., the transition function maps each state and letter to a (positive) propositional formula where the propositions are states).

For any iDFG we may construct an AFA that recognizes the set of all traces τ such that the *reversal* of τ is recognized by the iDFG. Each assertion in the iDFG corresponds to a state of the AFA and each iDFG node corresponds to an AFA transition. Since the AFA accepts the reversed language, each node in the iDFG should be read as a *backwards* transition. This allows the bifurcation in the iDFG to be interpreted using conjunction. For example, iDFG node labeled $[\texttt{m}_3 \leq \texttt{s}]$ indicates that starting in the state $\{\mathit{false}\}$, we may read the letter $[\texttt{m}_3 \leq \texttt{s}]$ and transition to *both* $\{\texttt{s} = 1\}$ and $\{\texttt{m}_3 > 1\}$, and must accept along each path. More explicitly, the transition rule corresponding to this vertex is as follows:

$$\delta(\{\mathit{false}\}, [\texttt{m}_3 \leq \texttt{s}]) = \{\texttt{s} = 1\} \wedge \{\texttt{m}_3 > 1\} \ .$$

A complete iDFG proof for the 3-thread Ticket protocol is given in Figure 10. This iDFG illustrates the need for disjunction as well as conjunction. Consider that there are two nodes of the iDFG which are labeled $[\texttt{m}_3 \leq \texttt{s}]$ and which

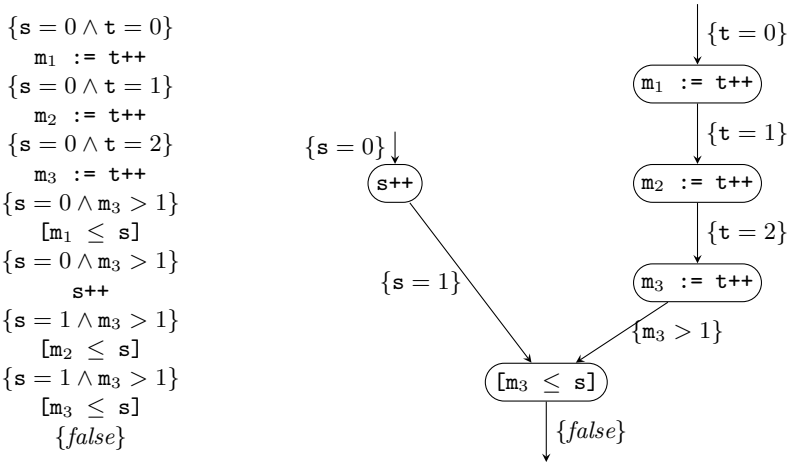


Fig. 9. Example trace with a Hoare-style proof and iDFG proof

have $\{false\}$ as a post-condition. This means that there are *two* transition rules corresponding to reading the letter $[\mathtt{m}_3 \leq \mathtt{s}]$ at the state $\{false\}$. The transition rules can be combined by disjunction, yielding the following transition function:

$$\delta(\{false\}, [\mathtt{m}_3 \leq \mathtt{s}]) = (\{\mathtt{s} = 0\} \wedge \{\mathtt{m}_3 > 0\}) \vee (\{\mathtt{s} = 1\} \wedge \{\mathtt{m}_3 > 1\}) \ .$$

The main appeal of iDFGs is that they are succinct proof objects for concurrent programs. Generalizing the Ticket example to N threads, the iDFG proof has $O(N^2)$ vertices, while the product control flow graph has $O(3^N)$. In [4], we make the claim of succinctness more general and formal by defining a measure of *data complexity* and showing that iDFG proofs are polynomial in this measure. Intuitively, this succinctness is possible because iDFGs represent only the *data flow* of the program, and abstract away control features that are irrelevant to the proof. This approach shifts the burden of the exponential explosion incurred by concurrency towards the check whether all program traces are represented, which is an automata-theoretic problem.

6 Unbounded Parallelism: Predicate Automata

The preceding section discusses a method for attacking the problem that the size of the automaton for a concurrent program is exponential in the number of threads. For many programs (filesystems, device drivers, web servers, ...), the number of threads is not statically known, or may increase without bound during the course of the program's execution. For such a program, the Cartesian product is *infinite* (as is the alphabet of program instructions), and the set of program traces is not a regular language. Thus, the problem of unbounded parallelism

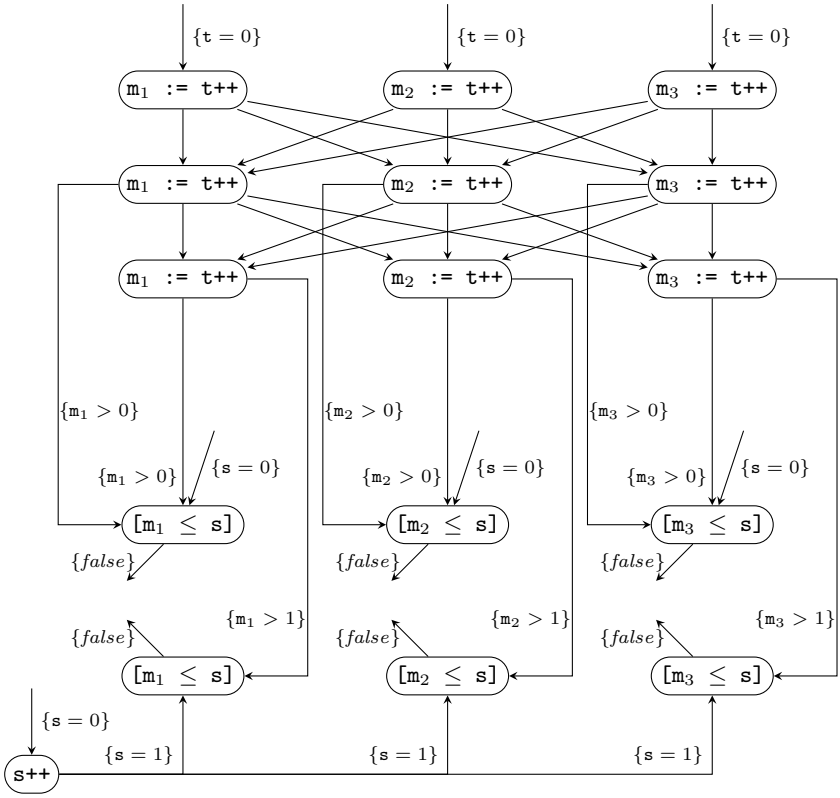


Fig. 10. Complete iDFG proof for the 3-thread Ticket protocol

is not merely one of high complexity, and we must develop new technology to address it.

In [6], we present *proof spaces*, a proof system which generalizes iDFGs to allow unboundedly many threads. The proof checking problem for proof spaces is carried out using *predicate automata*, which are an infinite-state (and infinite-alphabet) generalization of alternating finite automata.

We will start by demonstrating proof spaces on a simple example. Consider a program in which an arbitrary number of threads concurrently execute the code below. The goal is to verify that, if $\mathtt{g} \geq 1$ holds initially, then it will always hold (regardless of how many threads are executing).

```
global int g
local  int x
1: x := g;
2: g := g+x;
```

Consider the set of the Hoare triples (A) - (D) given below.

$$
\begin{array}{ll}
(\text{A}) & \{\mathtt{g} \geq 1\}\ \langle \mathtt{x}\ \mathtt{:=}\ \mathtt{g} : 1\rangle\ \{\mathtt{x}(1) \geq 1\} \\
(\text{B}) & \{\mathtt{g} \geq 1 \wedge \mathtt{x}(1) \geq 1\}\ \langle \mathtt{g} := \mathtt{g} + \mathtt{x} : 1\rangle\ \{\mathtt{g} \geq 1\} \\
(\text{C}) & \{\mathtt{g} \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 1\rangle\ \{\mathtt{g} \geq 1\} \\
(\text{D}) & \{\mathtt{x}(1) \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 2\rangle\ \{\mathtt{x}(1) \geq 1\}
\end{array}
$$

Here we use $\mathtt{x}(1)$ to refer to Thread 1's copy of the local variable $\mathtt{x}$, and $\langle \mathtt{x}\ \mathtt{:=}\ \mathtt{g} : 1\rangle$ to indicate the instruction $\mathtt{x}\ \mathtt{:=}\ \mathtt{g}$ executed by Thread 1.

The question of how such Hoare triples can be generated automatically is discussed in more detail in [6]; for our present purposes, we suppose that they are received from an oracle. We pose the question: given a set of ordinary Hoare triples (of the type one might expect to generate using sequential verification techniques), *what can we do with them?* We consider a deductive system in which these triples are taken as axioms, and the only rules of inference are *sequencing*, *symmetry*, and *conjunction*. These rules are easily illustrated with concrete examples:

– *Sequencing* composes two Hoare triples sequentially. For example, sequencing (A) and (D) yields

$$(\text{A} \circ \text{D}) \quad \{\mathtt{g} \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 1\rangle\langle \mathtt{x} := \mathtt{g} : 2\rangle\ \{\mathtt{x}(1) \geq 1\}$$

– *Symmetry* permutes thread identifiers. For example, renaming (A) and (C) (mapping $1 \mapsto 2$) yields

$$
\begin{array}{l}
(\text{A'})\ \{\mathtt{g} \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 2\rangle\ \{\mathtt{x}(2) \geq 1\} \\
(\text{C'})\ \{\mathtt{g} \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 2\rangle\ \{\mathtt{g} \geq 1\}
\end{array}
$$

and, renaming (D) (mapping $1 \mapsto 2$ and $2 \mapsto 1$) yields

$$(\text{D'})\ \{\mathtt{x}(2) \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 1\rangle\ \{\mathtt{x}(2) \geq 1\}$$

– *Conjunction* composes two Hoare triples by conjoining pre- and postconditions. For example, conjoining (A') and (C') yields

$$(\text{A'} \wedge \text{C'})\ \{\mathtt{g} \geq 1\}\ \langle \mathtt{x} := \mathtt{g} : 2\rangle\ \{\mathtt{g} \geq 1 \wedge \mathtt{x}(2) \geq 1\}$$

and conjoining (A) and (D') yields (A ∧ D')

$$\{\ \mathtt{g} \geq 1 \wedge \mathtt{x}(2) \geq 1\ \}\ \langle \mathtt{x} := \mathtt{g} : 1\rangle\ \{\ \mathtt{x}(1) \geq 1 \wedge \mathtt{x}(2) \geq 1\ \}$$

Naturally, the deductive system may apply inference rules to deduced Hoare triples as well: for example, by sequencing (A' ∧ C') and (A ∧ D'), we get the Hoare triple

$$\{\ \mathtt{g} \geq 1\ \}\ \langle \mathtt{x} := \mathtt{g} : 2\rangle\langle \mathtt{x} := \mathtt{g} : 1\rangle\ \{\ \mathtt{x}(1) \geq 1 \wedge \mathtt{x}(2) \geq 1\ \}$$

A *proof space* is a set of valid Hoare triples which is closed under sequencing, symmetry, and conjunction (that is, it is a *theory* of this deductive system). Any

finite set of valid Hoare triples generates an infinite proof space by considering those triples to be axioms and taking their closure under deduction; we call such a finite set of Hoare triples a *basis* for the generated proof space. Fixing a pre-condition φ_{pre} and a post-condition φ_{post} (for instance, taking both to be $\mathtt{g} \geq 1$ for our example), a proof space can be said to recognize all of those traces τ such that $\{\ \varphi_{\mathsf{pre}}\ \}\ \tau\ \{\ \varphi_{\mathsf{post}}\ \}$ belongs to the space.

As with iDFGs, we can give a more operational view of the traces recognized by a proof space using automata. For this purpose, we developed the notion of *predicate automata* (PA), an infinite-state, infinite-alphabet generalization of alternating finite automata (closely related to *alternating register automata* [3,7,15,17]). If one conceives of alternating finite automata as the automata of propositional logic, then predicate automata may be thought of as the automata for first-order logic. A PA A is equipped with a finite vocabulary of *predicates*, and its states are propositions over this vocabulary (i.e., if p is a binary predicate symbol of A, then $p(1,2)$ is a state of A). The transition function of a PA maps each predicate symbol and letter to a positive Boolean formula over its vocabulary. For example, the transition

$$\delta(p(i,j), a:k) = (p(i,j) \wedge i \neq k) \vee (q(i) \wedge q(j) \wedge i = k)$$

indicates that, if the PA is at state $p(1,2)$ and reads $a:2$, then it transitions to $p(1,2)$; if it then reads $a:1$, then it transitions to *both* the state $q(1)$ and $q(2)$..

From a finite basis B of Hoare triples, we may construct a predicate automaton which recognizes the same traces as the proof space generated by B. Each n-thread assertion which appears in the basis corresponds to an n-ary predicate, and each Hoare triple in the basis corresponds to a transition. For example, the Hoare triple (B) corresponds to the PA transition

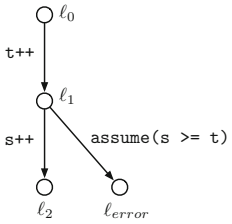
$$\delta(\{\mathtt{g} \geq 1\}, \mathtt{g\ :=\ g\ +\ x} : k) = \{\mathtt{g} \geq 1\} \wedge \{\mathtt{x}(k) \geq 1\}$$

(where $\{\mathtt{g} \geq 1\}$ is a nullary predicate and $\{\mathtt{x}(k) \geq 1\}$ is a unary predicate).

The proof checking problem for proof spaces reduces to the inclusion problem for PA. Although this problem is undecidable in general, [6] gives a semi-algorithm which is a decision procedure for the special case of PAs where each predicate symbol in its vocabulary has arity at most one.

7 Proofs that Count: Petri Nets

Consider the program that consists of an arbitrary number of threads whose control flow graph is pictured below. The (global) integer variables s and t are initially 0. The task is to automatically construct a proof that the error location ℓ_{error} is unreachable (i.e., the program satisfies the specification $s = t = 0/\mathit{false}$). This deceptively simple property is surprisingly difficult to prove correct using automated techniques.



We illustrate the difficulty of proving this example by informally applying the technique from Section 2. We begin by sampling an error trace from the program, say (a trace that involves two threads)

$$\tau = \texttt{t++; t++; s++; assume(s >= t)}$$

A correctness proof for τ is a sequence of intermediate assertions, shown below in Figure 11(a). We may generalize the proof to apply to a language of traces, as shown in the NFA in Figure 11(b).

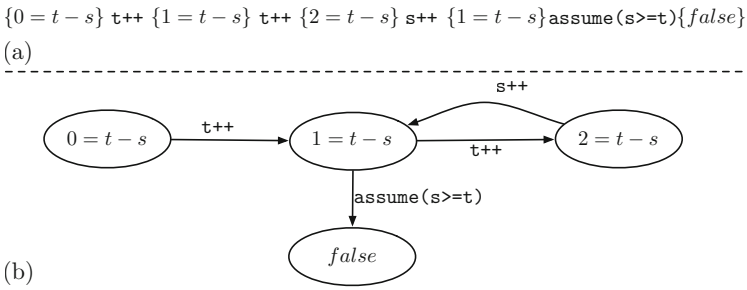


Fig. 11. Proof for the sample trace τ

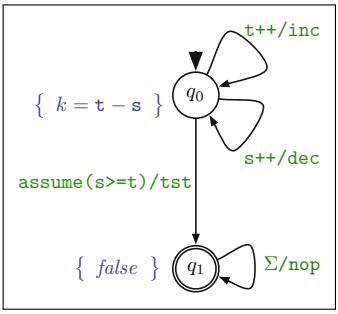
This automaton does not yet accept every trace of the program. We could continue by sampling a new trace, for instance

$$\tau' = \texttt{t++; t++; t++; s++; assume(s >= t)},$$

but it is already clear that this strategy is doomed to fail. There is no regular language which contains all the program traces and which does not contain incorrect traces. Similarly, there is no finitely-generated proof space which proves the correctness of every trace.

A *counting argument* (in the context of formal methods) is a program proof that makes use of one or more *counters*, which are not part of the program itself, but which are useful for abstracting program behavior. One informal argument for correctness is as follows: a global, inductive invariant for this program is that the number of threads at line ℓ_1 (i.e., after executing `t++` but before executing

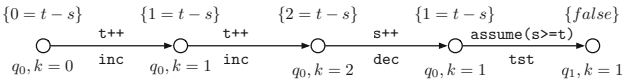
`s++`), let us call this k, is equal to the difference $t - s$. Since the number of threads at line ℓ_1 is non-negative, we must always have $s \leq t$, and ℓ_{error} must be unreachable. This counting argument is clear and simple to our human intuition, but *how can we take this intuition and formalize it into a mechanically constructed proof*? This question was investigated and answered in [5].



Our solution to this problem is pictured to the right. This *counting proof* consists of a counting automaton A (a kind of restricted counter machine) paired with an annotation φ mapping the states of A to assertions. The counting automaton A is a finite automaton equipped with a $\mathbb{N}$-valued counter denoted k (initially 0). Each transition of the automaton is equipped with an action for k, which may be `inc` (increment the counter), `dec` (decrement, but block unless the counter is ≥ 1), `tst` (block unless the counter is ≥ 1), or `nop` (do nothing). The annotation φ associates with each state of this automaton a formula over the program variables and the counter variable k. This annotation is *inductive* in the sense that each transition is associated with a valid Hoare triple: for example,

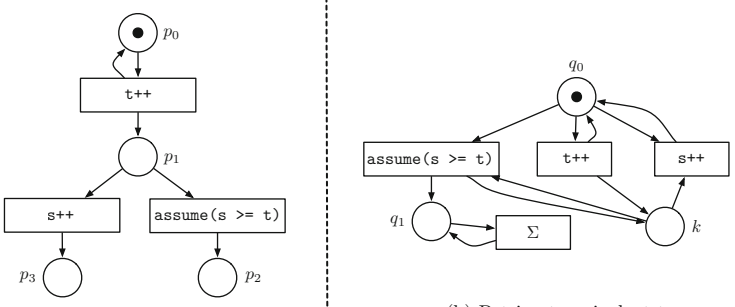
$$\begin{array}{lll} \{k = t - s\} & \texttt{t++; k++} & \{k = t - s\} \\ \{k = t - s\} & \texttt{s++; k--} & \{k = t - s\} \\ \{k = t - s\} & \texttt{assume(s>=t); assume(k}\geq\texttt{1)} & \{\mathit{false}\} \end{array}$$

A trace is accepted by A if it labels a path from q_0 (the initial state) to q_1 (the final state), and none of the counter actions block. Every trace which is accepted by A is associated with a sequence of assertions (thus proving its correctness). This sequence is obtained from the accepting run of A by taking, for each position in the run, the assertion at the current state with k replaced by its current value. For example, the proof for the trace τ above is as follows:



This counting proof works not only for the trace τ, but for *every* trace of the program (that is, the proof is enough to show that ℓ_{error} is unreachable). The key to this proof is the use of the counter variable k, which counts the number of `t++` statements in excess of `s++` statements along a trace. Using this auxiliary counter allows us to make a simple, succinct argument for the correctness of this program.

The essential idea for constructing counting proofs is to encode the problem as an SMT query. Our encoding requires us to specify the "size" of the candidate proof to find (e.g., the number of states that may be used), and will always succeed if a proof of that size exists. The main insight behind our proof construction procedure is that by looking for *small* proofs, we can force an SMT solver to synthesize nontrivial counting arguments. For example, we can force an SMT



(a) Control flow net of the program

(b) Petri net equivalent to counting automaton proof

Fig. 12. The language of Petri net (a) is included in the language of the deterministic Petri net (b)

solver to "discover" the need to count the number of `t++` statements in excess of `s++` statements in the proof above completely automatically, simply by asking for a proof with 2 states.

The idea behind proof checking is based on the observation that counting automata can be converted into deterministic labeled Petri nets (Figure 12(b)). Similarly, the language of program traces can be represented by a Petri net (Figure 12(a)). The final step of the correctness argument is performed by showing that the language of the Petri net for the program is included in the language of the deterministic Petri net for the counting automaton, a problem which is known to be decidable.

8 Conclusion

We have described several instances of a new approach to program verification which constructs and checks automata. We have shown that, in order to instantiate the approach for a specific verification problem, one has to come up with the appropriate notion of automaton, one has to define the construction of an automaton from the proof of a sequence of statements (i.e., a trace), one has to define the *program automaton* which recognizes the set of *error traces*, and one has to present an algorithm for solving the corresponding automata inclusion problem.

There are several interesting verification problems (timed systems, hybrid systems, game-theoretic properties, termination for unbounded parallelism, ...) where the question whether an appropriate notion of automaton exists, is still open and we do not know whether the approach can be instantiated.

Conversely, given a notion of automaton, one may ask whether there exists a verification problem for which this notion may be useful. For example, in some restricted cases in Section 5 it may be useful to define the denotation of an iDFG as a set of trees and replace alternating finite automata by tree automata.

An obvious topic for future research is the scalability of the automata operations used in the approach: the check of automata inclusion, minimization for the incremental construction of the difference automaton, etc.

Finally, the construction of an automaton from the proof of a sequence of statements may be interesting in settings other than verification. For example, given a failed test for a program with a bug, we can again construct an automaton and use the automaton for the *diagnosis* of the bug [20].

Acknowledgements. We would like to thank Jürgen Christ for comments and discussions.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM **56**(3) (2009)
2. Christ, J., Hoenicke, J., Nutz, A.: Proof tree preserving interpolation. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 124–138. Springer, Heidelberg (2013)
3. Demri, S., Lazić, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Logic **10**(3), 16:1–16:30 (2009)
4. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: POPL, pp. 129–142. ACM (2013)
5. Farzan, A., Kincaid, Z., Podelski, A.: Proofs that count. In: POPL, pp. 151–164. ACM (2014)
6. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: POPL. ACM (2015)
7. Figueira, D.: Alternating register automata on finite words and trees. Logical Methods in Computer Science **8**(1) (2012)
8. Heizmann, M., et al.: Ultimate automizer with unsatisfiable cores (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 418–420. Springer, Heidelberg (2014)
9. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 365–380. Springer, Heidelberg (2013)
10. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
11. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, pp. 471–482. ACM (2010)
12. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013)
13. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Heidelberg (2014)
14. Junker, M., Huuck, R., Fehnker, A., Knapp, A.: SMT-based false positive elimination in static program analysis. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 316–331. Springer, Heidelberg (2012)

15. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994)
16. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 172–186. Springer, Heidelberg (2014)
17. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Logic **5**(3), 403–435 (2004)
18. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
19. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61. ACM (1995)
20. Schäf, M., Schwartz-Narbonne, D., Wies, T.: Explaining inconsistent code. In: ESEC/FSE, pp. 521–531. ACM (2013)

Springer

http://www.springer.com/978-3-319-15578-4

Language and Automata Theory and Applications
9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings
Dediu, A.-H.; Formenti, E.; Martín-Vide, C.; Truthe, B. (Eds.)
2015, XV, 754 p. 140 illus., Softcover
ISBN: 978-3-319-15578-4