Appendix

We provide additional information and explanations that had to be left out of the paper for space reasons. Numbered (sub-)sections relate to those of the paper.

1 Introduction

Remark. A striking example for the difficulty of knowledge transfer in formal methods is the sparse interaction between the deductive verification and the abstract interpretation community, given the significant methodological overlap.

Remark. One can view *approximation* as a special case of *abstraction*. Since approximation can be expressed by a subset relation alone, it is unnatural to conflate them, however.

2 Programs, Logic, Traces and Abstractions

For completeness, we subsequently define a trace semantics for a simple deterministic while language. In Sect. 2, only the semantics for the more exotic cases of the **assert** and **assume** statements was defined. In the following definition, x represents program variables and e expressions. We write $[\![e]\!]_s$ for the semantics of expression e in a state s. For appending a trace τ_2 to a (finite) trace τ_1, we simply write $\tau_1\tau_2$. The empty trace ε is the neutral element of this operation. The predicate $\mathit{failed}(\tau)$ holds for a trace τ if it contains the failure state $\bot$.

$$
\begin{aligned}
\mathrm{Tr}_s(\texttt{x=e}) &\coloneqq \{s[\texttt{x} \mapsto [\![e]\!]_s]\} \\
\mathrm{Tr}_s(\texttt{if}(e)\ p_1\ \texttt{else}\ p_2) &\coloneqq \begin{cases} \mathrm{Tr}_s(p_1) & \text{if } [\![e]\!]_s = \mathit{true} \\ \mathrm{Tr}_s(p_2) & \text{otherwise} \end{cases} \\
\mathrm{Tr}_s(p_1\ ;\ p_2) &\coloneqq \{\tau \in \mathrm{Tr}_s(p_1) : \neg \mathit{finite}(\tau) \wedge \neg \mathit{failed}(\tau)\} \cup \\
&\quad \{\tau_1\tau_2 : \tau_1 \in \mathrm{Tr}_s(p_1) \wedge \mathit{finite}(\tau_1) \wedge \neg \mathit{failed}(\tau_1), \\
&\qquad \tau_2 \in \mathrm{Tr}_{\mathit{last}(\tau_1)}(p_2)\} \cup \\
&\quad \{\bot : \tau_1 \in \mathrm{Tr}_s(p_1) \wedge \mathit{failed}(\tau_1)\} \\
\mathrm{Tr}_s(\texttt{while}(e)\ p) &\coloneqq \begin{cases} \mathrm{Tr}_s(p;\ \texttt{while}(e)\ p) & \text{if } [\![e]\!]_s = \mathit{true} \\ \{\varepsilon\} & \text{otherwise} \end{cases} \\
\mathrm{Tr}_s(\texttt{assert}(\varphi)) &\coloneqq \begin{cases} \{s\} & \text{if } s \models \varphi \\ \{\bot\} & \text{otherwise} \end{cases} \\
\mathrm{Tr}_s(\texttt{assume}(\varphi)) &\coloneqq \begin{cases} \{s\} & \text{if } s \models \varphi \\ \emptyset & \text{otherwise} \end{cases} \\
\mathrm{Tr}_s(\texttt{havoc}) &\coloneqq \mathcal{S}
\end{aligned}
$$

Example 4 (Trace Semantics). The $\mathcal{L}$-Program $p = \texttt{x=-1;}\ \texttt{assume}(\texttt{x} \geq 0)$ evaluates to the empty trace set ($\mathrm{Tr}(p) = \emptyset$) because of the definition of the semantics for the sequencing operator. There is no $\tau_2 \in \mathrm{Tr}_s(\texttt{assume}(\texttt{x} \geq 0))$ since $\mathrm{Tr}_{s[\texttt{x} \mapsto -1]}(\texttt{assume}(\texttt{x} \geq 0))$ returns the empty set. Conversely, $p' = \mathrm{Tr}(\texttt{x=-1;}$

assume(x $\leq$ 0)) evaluates to the set $\{s[\mathrm{x} \mapsto -1] : s \in \mathcal{S}\}$, since now, the empty trace is produced for the **assume** statement, which is the neural element of trace concatenation. The trace modality formula $[p \Vdash \mathit{spec}]$ thus holds for any specification *spec*, since the empty set is a subset of any set; this is not the case for p', where satisfying the specification would be more meaningful. Consider now the following program: $p'' =$ x=-1; **assume**(y $\geq$ 0). There, we make an assumption about a variable y that is not mentioned before. It evaluates to

$$\mathrm{Tr}(p'') = \{s' : s \in \mathcal{S} \wedge s' = s[\mathrm{x} \mapsto -1] \wedge [\![\mathrm{y}]\!]_{s'} \geq 0\},$$

i.e. the set of all states that after setting x to -1 satisfy the property of y being positive. This is the actual use case of assumptions: To assume facts that *cannot* be otherwise established locally in the current program context.

The property of assumptions to trivially satisfy any specification if they are invalid motivated the introduction of the failure state $\bot$ for *assertions*. For the trace modality, a failed assertion in the implementation should only yield a provable result if there is also a failed assertion in the specification. The program $q =$ x=-1; **assert**(x $\geq$ 0) (similar to p, but with an assertion instead of an assumption) evaluates to $\{\bot\}$, while the program $q' =$ x=-1; **assert**(x $\leq$ 0) evaluates to the same trace set as p'. The program $q'' =$ x=-1; **assert**(y $\geq$ 0), on the other hand, evaluates to

$$\begin{aligned}\mathrm{Tr}(q'') = \{&s' : s \in \mathcal{S} \wedge s' = s[\mathrm{x} \mapsto -1] \wedge \quad [\![\mathrm{y}]\!]_{s'} \geq 0\} \cup \\ &\{\bot : s \in \mathcal{S} \wedge s' = s[\mathrm{x} \mapsto -1] \wedge \neg[\![\mathrm{y}]\!]_{s'} \geq 0\}\end{aligned}$$

It also contains a set of traces for all concrete traces that after the assignment satisfy the assertion, but also a trace with the failure state if there is *any* state which does not satisfy the assertion. The use case for assertions is, in contrast to assumptions, to verify a fact that is *assumed to be provable* locally in the current program context.

The **havoc** command is used in loop invariant reasoning (see Example 10). For instance, $r =$ x=-1; **assert**(*Inv*); **havoc**; **assume**(*Inv*) evaluates to

$$\begin{aligned}\mathrm{Tr}(r) = \{&s \in \mathcal{S} : [\![\mathit{Inv}]\!]_s\} \cup \\ &\{\bot : s \in \mathcal{S} \wedge s' = s[\mathrm{x} \mapsto -1] \wedge \neg[\![\mathit{Inv}]\!]_{s'}\}\end{aligned} \qquad \Diamond$$

3 The Trace Modality

Table 1 (page III) provides a quick summary of our formalizations for the considered verification tasks in Sect. 3.

Remark 1 (Axioms of Modal and Dynamic Logic). The necessitation rule (axiom **N**) and distribution axiom (axiom **K**) of modal logic are theorems of the trace modality: If φ is a theorem, then $[\mathrm{p} \Vdash_{\alpha_{big}} \varphi]$ holds for all p (that do not produce a failure $\bot$) since $\mathit{lift}_{\mathrm{Fml}}(s)(\varphi)$ contains *all* traces starting in s (axiom **N**). It is also straightforward to show that $[\mathrm{p} \Vdash_{\alpha_{big}} \varphi \to \psi]$ implies $[\mathrm{p} \Vdash_{\alpha_{big}} \varphi] \to [\mathrm{p} \Vdash_{\alpha_{big}} \psi]$

Task	Problem	D_l	D_r	Solution Techniques (excerpt)
Partial Correctness	$\models [\mathtt{p} \Vdash_{\alpha_{big}} \mathit{Post}]$	$D_{\mathcal{L}_0}$	D_{Fml}	Symbolic execution, weakest precondition reasoning, Hoare calculus
Total Correctness	$\models \langle \mathtt{p} \Vdash_{\alpha_{big}} \mathit{Post} \rangle$	$D_{\mathcal{L}_0}$	D_{Fml}	ditto; plus reasoning about variant / ranking function
Information Flow	$\models [\mathtt{p}(\mathtt{l},\mathtt{h}) \Vdash_{\alpha_{\{\mathtt{l}\}} \circ \alpha_{big}} \mathtt{p}(\mathtt{l},\mathtt{h}')]$	$D_{\mathcal{L}_0}$	$D_{\mathcal{L}_0}$	Security type systems, Hoare calculus, symbolic execution
Information Flow with Declassification	$\models \bigwedge_{i=1}^{n}(e_i(\mathtt{l},\mathtt{h}) \doteq e_i(\mathtt{l},\mathtt{h}')) \rightarrow [\mathtt{p}(\mathtt{l},\mathtt{h}) \Vdash_{\alpha_{\{\mathtt{l}\}} \circ \alpha_{big}} \mathtt{p}(\mathtt{l},\mathtt{h}')]$	$D_{\mathcal{L}_0}$	$D_{\mathcal{L}_0}$	ditto
Finite Space MC	$s \models [\mathtt{p} \Vdash \varphi]$	$D_{\mathcal{L}_0}$	D_{TL}	Automata constructions
Bounded MC	$\models [\mathtt{p} \Vdash \varphi]$	$D^k_{\mathcal{L}_0}$	D_{TL}	SMT solvers for checking encoded program paths
Abstraction-Based MC	$\models [\mathtt{p} \Vdash_{\alpha_d} \varphi]$	$D_{\mathcal{L}_0}$	D_{TL}	Overapproximation techniques, CEGAR loops
Symbolic Execution-Based MC	$\models [\mathtt{p} \Vdash \varphi]$	$D_{\mathcal{L}_0}$	D_{TL}	Invariant generation, k-induction
Bug Finding	$\models \langle \mathtt{p} \Vdash \neg\varphi \rangle$	$D_{\mathcal{L}_0}$	D_{TL}	All MC techniques; can be integrated with all abstractions
Program Synthesis	$\models [\mathtt{p}' \Vdash_{\alpha_{big}} \mathtt{p}]$	$D_{\mathcal{L}}$	$D_{\mathcal{L}}$	Proof-theoretic synthesis, proof mining
Correct compilation	$\models [\mathtt{p} \Vdash_{\alpha_{obs} \circ \alpha_{big}} \mathtt{c}]$	$D_{\mathcal{L}_0}$	$D_{\mathcal{L}}$	Simultaneous symbolic execution, compiler extraction from executable HOL specifications
Program evolution / Bug fixing	$\models [\mathtt{p}_{\mathit{buggy}} \Vdash_{\alpha_{patch} \circ \alpha_{big}} \mathtt{p}_{\mathit{fixed}}]$	$D_{\mathcal{L}_0}$	$D_{\mathcal{L}_0}$	Manual program refinement, automatic software repair

Table 1: Modeling Different Verification Tasks with the Trace Modality

(axiom **K**). As an example for an axiom of PDL [17], we consider the axiom for a PDL "test" $\psi?$, where $\psi?$ corresponds to our **`assume`** ψ: The assertion $[\texttt{assume}\ \psi \Vdash_{\alpha_{big}} \varphi]$ is equivalent to $\psi \to \varphi$. If ψ does not hold for a state s, the trace set for the **`assume`** statement is the empty set which is trivially contained in any set, and the premise ψ of the implication is false and the implication therefore holds. If, however, s *does* satisfy ψ, then the abstracted trace set for the **`assume`** contains the trace starting and ending in s which is only contained in the set for the specification if s also satisfies φ, and similar for the implication.

Remark 2 (Linearization). As a small exercise, we can prove a theorem corresponding to the "linearization" axiom $[\mathtt{p};\mathtt{q}]\varphi \leftrightarrow [\mathtt{p}][\mathtt{q}]\varphi$ also for the trace modality. The trace modality version of this axiom is

$$[\mathtt{p};\mathtt{q} \Vdash_{\alpha_{big}} \varphi] \leftrightarrow [\mathtt{p} \Vdash_{\alpha_{big}} [\mathtt{q} \Vdash_{\alpha_{big}} \varphi]] \tag{1}$$

To give this a meaning, we define a lifting function $\mathit{lift}_{D_{mod}}$ assigning trace sets to trace modality formulas as follows:

$$\begin{aligned}&\mathit{lift}_{D_{mod}}(s)([\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]) \coloneqq \\ &\quad \left\{s\tau_1 : \mathit{finite}(\tau_1) \wedge \exists \tau_2 \in \mathit{lift}_{D_l}(\mathit{last}(s\tau_1))(\mathcal{C}_l);\ \alpha(s\tau_1\tau_2) \subseteq \alpha(\mathit{lift}_{D_r}(s)(\mathcal{C}_r))\right\} \\ &\qquad\qquad \cup \left\{s\tau : \neg\mathit{finite}(\tau) \wedge \alpha(s\tau) \subseteq \alpha(\mathit{lift}_{D_r}(s)(\mathcal{C}_r))\right\}\end{aligned}$$

In other words, the set of (i) all finite prefixes τ_1 starting in s which can completed by the traces τ_2 for the implementation starting in the last state of τ_1 such that the result meets the specification after α-abstraction, and (ii) all infinite traces starting in s which meet the specification after α-abstraction. Part (ii) may seem strange since the implementation $\mathcal{C}_l$ does not occur there. The idea is that for a nonterminating program p in Eq. (1), q is never evaluated neither on the left nor on the right-hand side of the equality. If φ is a first-order post condition, it will in any case only be lifted to finite traces, which is why (ii) does not apply then. We prove the "$\to$" direction of Eq. (1). For simplicity, we assume that p, q terminate normally for all inputs. Our hypothesis is that for all states s, the set $\alpha_{big}(\{\tau_1\tau_2\})$, where τ_1 corresponds to an execution of p starting in s and τ_2 to an execution of q starting in the final state of τ_1, is contained in the set of all pairs (s, f) where f satisfies φ. We have to show that the $\alpha_{big}(\{\tau_1\})$, where τ_1 is as before, is contained in the set of all pairs (s, f'), where f' is the final state of a prefix trace that can be completed by execution of q to a trace the final state of which satisfies φ. Since from the hypothesis, we already know that when execution q after p, the final state satisfies φ, f' can be instantiated to a final state of τ_1. Direction "$\to$" is similar.

We point out that the trace lifting of trace modality formulas in Remark 2 is interesting in its own, since it closely resembles the definition of the semantics of PDL modality formulas in [17]. There, the semantics of $[\mathtt{p}]\varphi$ is the set of all states s for which, when executing p starting in s, the final states satisfy φ. Our definition is similar, only that we generalize from single initial states to

whole "prefix traces". Another consideration is that we could have regarded trace modality formulas (in the specification side of a trace modality formula) simply as an atom of first-order logic. Then, $\mathit{lift}_{D_{\mathrm{Fml}}}(s)([\mathtt{q} \Vdash_{\alpha_{big}} \varphi])$ is the set of all finite traces starting in s whose final states satisfy $[\mathtt{q} \Vdash_{\alpha_{big}} \varphi]$, which is equivalent to the first part of the union in Remark 2–only infinite traces are not considered.

3.1 Functional Verification

Remark 3. Termination only is expressed as $\langle \mathtt{p} \Vdash_{\alpha_{big}} \mathrm{true} \rangle$: There has to be a α_{big}-abstracted trace for p starting in s which is *not* contained in the set $\alpha_{big}(\overline{\mathit{lift}_{\mathrm{Fml}}(s)(\mathrm{true})})$ consisting of all *infinite* traces starting in s (and all traces not starting in s).

3.2 Information Flow Analysis

Remark 4. In Example 2, we could have omitted α_{big}; however, we cannot do so in general, if we want to allow *intermediate* violation of the policy. If we, for instance, added a statement `l=42;` to the program, it would be safe w.r.t. the big-step formalization, although in between, `l` attains a different value according to the value of `h`.

Declassification, such as *delimited information release* [28], can be easily encoded via preconditions. Assume e is an expression of $\mathcal{L}_0$ we want to declassify. We extend $\mathcal{L}_0$ by expressions **`declassify`**(e), as in [28], which evaluate to e while permitting flow of e to the *low* level. As for programs, write $e(\mathtt{l},\mathtt{h})$ to make the variables occurring in e explicit. Then non-interference with declassification is formalized as:

$$\models e(\mathtt{l},\mathtt{h}) \doteq e(\mathtt{l},\mathtt{h}') \rightarrow [\mathtt{p}(\mathtt{l},\mathtt{h}) \Vdash_{\alpha_{\{\mathtt{l}\}} \circ \alpha_{big}} \mathtt{p}(\mathtt{l},\mathtt{h}')] \ .$$

Example 5 (Declassification). We consider the classic PIN example, where a *low* variable `OK` is set to `true` depending on whether a *high* input `inp` equals a *high* variable `pin` containing a PIN. Let

```
p := if (declassify(pin==inp)) { OK=true } else { OK=false }
```

be this program. If we do not give special semantics to the **`declassify`** expression, there is a state s where $s(\mathtt{pin}) = s(\mathtt{inp})$, but $s(\mathtt{pin}') \neq s(\mathtt{inp}')$; for this state, the subset relation does not hold, which is why p would be classified as insecure. The additional precondition $\mathtt{pin} \doteq \mathtt{inp} \leftrightarrow \mathtt{pin}' \doteq \mathtt{inp}'$, however, rules this choice out, and we can classify the program as secure w.r.t. the delimited release semantics. ◊

3.3 Software Model Checking

In the following, we formalize four popular Software Model Checking approaches using the trace modality.

Finite Space MC Finite space model checkers (SPIN [20] is a prominent representative) exhaustively explore the state space of an abstract program model. This implies that the analysis starts from *a concrete input state s* and that no unbounded data structures are involved. We can formalize this problem as $s \models [\mathtt{p} \Vdash \varphi]$, where φ is an LTL formula.

Bounded MC (BMC) BMC [6,8] handles unbounded data structures, but restricts the search space according to a predefined upper bound on the number of loop executions. This problem can simply be expressed as $\models [\mathtt{p} \Vdash \varphi]$ when using a domain $D^k_{\mathcal{L}_0}$ with lifting function $\mathit{lift}^k_{\mathcal{L}_0}$ that only produces traces up to a fixed number k of loop executions (and recursive method calls).

Abstraction-Based MC This variant of SMC applies data abstraction to limit the search space. We can express it as $\models [\mathtt{p} \Vdash_{\alpha_d} \varphi]$, where α_d is an abstract interpretation of the data types of `p`.

Symbolic Execution-Based MC This variant of SMC is similar to functional verification (Sect. 3.1). They mainly differ in the used abstraction (identity vs. big-step) and that in MC less complex properties are proved: $\models [\mathtt{p} \Vdash \varphi]$.

3.4 Program Synthesis

Example 6 discusses our formalization of the program synthesis problem within the trace modality along an example from the literature computing integer square roots.

Example 6. We consider the square root example from [30]. Given a user-defined specification $\mathit{Pre} \coloneqq \mathtt{x} \geq 1$, $\mathit{Post} \coloneqq \mathtt{i}^2 \leq \mathtt{x} < (\mathtt{i}+1)^2$ and scaffold program $\bullet; *(\bullet); \bullet$, the synthesizer should generate a program `IntSqrt(int x)` satisfying the specification (i.e., computing the integer square root of a strictly positive variable `x`) and matching the structure of the scaffold. An additional user-defined constraint is that, apart from `x` and `i`, there must be at most one additional variable `v`, also of integer type. Listing 1 shows a concrete program matching the specification. To apply our formalization, we first translate the scaffold in a schematic $\mathcal{L}$-Program: `P;while(b){Q};R`. Let now $\mathit{Syn}_{\mathtt{P/Q/R}}$ be synthesis conditions for `P`, `Q` and `R` inferred by the synthesizer. Note that $\mathit{Syn}_{\mathtt{Q}}$ is an inductive invariant for `Q`. A concrete instantiation for $\mathit{Syn}_{\mathtt{Q}}$ is $\mathtt{v} \doteq \mathtt{i}^2 \wedge \mathtt{x} \geq (\mathtt{i}-1)^2 \wedge \mathtt{i} \geq 1$. The scaffold annotated by assert statements for the synthesis conditions is depicted in Listing 2 (we write $\mathtt{x}'$ for the value of `x` before the execution of a schematic statement). Suppose that now we refine the scaffold `sc` to a program `p` by replacing `Q` *and* the following **`assert`** statement by the following program `q`: `v=v+2i+1;i++;`. To prove this correct, we have to show $\models \mathtt{i} \geq 1 \rightarrow [\mathtt{p} \Vdash_{\alpha_{big}} \mathtt{sc}]$. Since the traces for `sc` include one trace for each possible instantiation of `Q` satisfying $\mathit{Syn}_{\mathtt{Q}}$ in the given context, and `q` also satisfies $\mathit{Syn}_{\mathtt{Q}}$ in this context, this is true. We point out that we cannot instead show $\mathit{Pre} \rightarrow [\mathtt{q} \Vdash_{\alpha_{big}} \mathit{Syn}_{\mathtt{Q}}]$, since the program before the insertion position, i.e. already substituted concrete programs as well as asserted synthesis conditions, also has to be considered. Here, in particular, it is important that `v` and `i` initially are 1 for the invariant to hold. ◊

```
v = 1; i = 1;

while (v<=x) {
  v = v+2i+1;
  i++;
}
i = i-1;
```

Listing 1: IntSqrt

```
P;
assert(v ≐ 1 ∧ i ≐ 1 ∧ x ≐ x′);
while (v<=x) {
  Q;
  assert(v ≐ i² ∧ x ≥ (i − 1)² ∧ i ≥ 1);
}
R;
assert(i² ≤ x < (i + 1)²);
```

Listing 2: Annotated scaffold for IntSqrt

3.5 Correct Compilation

Remark 5. A popular approach realizing correct compilation is the specification of the compiler within the executable fragment of an interactive proof assistant like Isabelle or Coq, as done in CompCert [23]. We proposed in earlier work a rule-based technique using *simultaneous* Symbolic Execution [31] with a *dual modality*, which can be seen as a specialization of the trace modality. The interesting property of this framework is that compilation rules can be proven automatically based on Symbolic Execution calculi for the source and target language. We think that a similar technique might be applicable to different verification tasks.

3.6 Program Evolution & Bug Fixing

There are (at least) four formalizations of the problem of program evolution / but fixing, of which two variants ("bug abstractions" and "patch abstractions") already have been presented in Sect. 3.6. Alternatives are:

- Like in declassification (Sect. 3.2), an added precondition Pre_{safe} excludes buggy traces: $\models Pre_{safe} \rightarrow [\mathtt{p}_{bug} \Vdash_{\alpha_{big}} \mathtt{p}_{fixed}]$.
- One could prove the buggy and fixed program *in isolation* (as in functional correctness, Sect. 3.1); then, though, one cannot use techniques of relational program verification to exploit similarities between the two program versions. Also, the existence of full separate functional specifications is required.

The following example demonstrates the application of the formalizations using additional preconditions, "bug abstractions" and "patch abstractions".

Example 7. We explain the mentioned techniques along a simple example. The program $\mathtt{p}_{buggy} \coloneqq$ `if (x<-1) {x=-x;}` should compute the absolute of a given integer x; i.e., after execution of the program, x should be positive. However, the program contains a bug: The programmer misspelled the "<" operator which should be a "<=" instead. For the input -1, a wrong result is thus produced. Let $\mathtt{p}_{fixed}$ be the corrected program. We choose $Pre_{safe} \coloneqq \mathtt{x} \neq -1$, which excludes the buggy path. Then, $\models Pre_{safe} \rightarrow [\mathtt{p}_{buggy} \Vdash_{\alpha_{big}} \mathtt{p}_{fixed}]$ can be proven, since apart from that path, the traces of the programs coincide. Note that Pre_{safe} is in fact the negation of the path condition for the buggy path. Choosing the

second formalization, we can define $\alpha_{bug}(\mathcal{T}) \coloneqq \{\tau \in \mathcal{T} | \mathit{first}(\tau) \models \mathit{Pre}_{safe}\}$ and show $\models [\mathtt{p}_{buggy} \Vdash_{\alpha_{bug} \circ \alpha_{big}} \mathtt{p}_{fixed}]$, which is in this case equivalent. Indeed, the latter formalization is more flexible than the former and allows for a more systematic approach which not simply excluding buggy paths, but rather encoding the correction as a "patch". Let $\alpha_{patch}(\mathcal{T}) \coloneqq \{\mathit{patch}(\tau) | \tau \in \mathcal{T}\}$, where

$$\mathit{patch}(\tau) \coloneqq \begin{cases} (\mathit{first}(\tau), \mathit{last}(\tau)[\mathtt{x} \mapsto -\mathit{first}(\tau)(\mathtt{x})]) & \text{if } \mathit{first}(\tau)(\mathtt{x}) = -1 \\ \tau & \text{otherwise} \end{cases} \qquad \lozenge$$

As demonstrated by the example, the abstraction approach makes program evolution more explicit by describing the applied patch. Also, just excluding the buggy path would be too easy, since apart from that path, the buggy program is likely to be equivalent to the original one—we could show the correctness of a "fixed" program where no fix was applied at all, or the fix introduced new wrong behavior for the buggy path. We therefore choose the "patch abstraction" as the canonical representation for program evolution within the trace modality.

4 Reasoning about the Trace Modality

Example 8. Let, as in Example 3, $s_1 = (\mathtt{x} := 17 \,||\, \mathtt{y} := 42 \,||\, \mathtt{z} := 2, \mathrm{true})$. It is subsumed by $s_3 = (\mathit{sto}_{\mathrm{P}} \circ \mathtt{x} := c, C_{\mathrm{P}} \wedge c \geq 0)$ containing an abstract store and path condition: We can prove

$$\models \{\mathtt{x} := 17\} P(\mathtt{x}) \rightarrow \mathit{subst}(\{\mathtt{x} := 17\}(C_{\mathrm{P}} \wedge 17 \geq 0) \wedge \{\mathit{sto}_{\mathrm{P}} \circ \mathtt{x} := 17\} P(\mathtt{x}))$$

for any *subst* replacing C_{P} with true and $\mathit{sto}_{\mathrm{P}}$ with any concrete store. Subsumption can*not* be shown, for instance, for $s_4 = (\mathtt{x} := 17 \,||\, \mathtt{w} := c', c' \geq 0)$, since (SUB1) is violated. The symbolic state $s_5 = (\mathtt{x} \leq 0)$ does not subsume s_1, since the first conjunct under *subst* in (SUB2), $\{\mathtt{x} := 17\}\mathtt{x} \leq 0$, does not hold. $\lozenge$

Our algorith for the creation of SFAs from symbolic traces is shown in Algo. 2. For an alphabet Σ, write Σ_ε for its extension by instantaneous ε-transitions. We use standard ε-elimination to convert an SFA on Σ_ε to Σ. During SFA construction, we maintain a map L for assertion labels, mapping states to assertions that should hold when arriving at them. Labels are, in the post processing step addAssertionEdges, transformed to assertion edges leading to a failure state for input states not satisfying them.

Algos. 3 and 4 show the auxiliary algorithms for Algo. 1 computing over-approximating initial simulation relations and checking two symbolic states for subsumption. The algorithms are explained in Sect. 4.

The approach of deriving SSRs from starting from the cross product by repeated filtering is polynomial in the size of the automata; we refer to [26] for more efficient approaches.

Symbolic Lifting of Loops Our symbolic trace language is a *regular* language, therefore it generally is not possible to encode loops with full precision. For this, it would be necessary to maintain a mutable state for tracking changes made

Algorithm 2 Creation of SFA from Symbolic Traces

function createSFA(τ : SymTr)
  $q_0 \leftarrow$ fresh state, $\Sigma \leftarrow \mathcal{S}$
  $(L, (Q, \Sigma_\varepsilon, \delta, q_0, F)) \leftarrow$ extendSFA$(\tau, q_0, \{q_0\}, \Sigma_\varepsilon, q_0)$
  $(L', (Q', \Sigma, \delta', q_0, F')) \leftarrow$ eliminateEpsilonTransitions$(L, (Q, \Sigma_\varepsilon, \delta, q_0, F))$
    ▷ Standard, but preserve labels
  return addAssertionEdges$(L', (Q', \Sigma, \delta', q_0, F'))$
end function

Ensure: Returns a pair $(L, (Q', \Sigma_\varepsilon, \delta, q_0, F))$ of a set of labels and an SFA for τ, where $L \subseteq Q' \times \mathrm{Fml}$, $Q' \supseteq Q$, $\delta \subseteq Q' \times \mathrm{SymState} \times Q'$, $F \subseteq Q'$. Σ_ε and q_0 are not changed.
function extendSFA(τ, q, Q, Σ_ε, q_0)
  if $\tau = s$ **then**
    $q' \leftarrow$ fresh state, $q' \notin Q$
    $Q' \leftarrow Q \cup \{q'\}$
    $\delta \leftarrow \{(q, s, q')\}$
    $F \leftarrow \{q'\}$
    return $(L, (Q', \Sigma_\varepsilon, \delta, q_0, F))$
  else if $\tau = \tau_1; \tau_2$ **then**
    $(L_1, (Q_1, \Sigma_\varepsilon, \delta_1, q_0, F_1)) \leftarrow$ extendSFA$(\tau_1, q, Q, \Sigma_\varepsilon, q_0)$
    $Q' \leftarrow Q_1$, $\delta \leftarrow \delta_1$, $F \leftarrow \emptyset$, $L \leftarrow L_1$
    for all $q' \in F_1$ **do**
      $(L_2, (Q_2, \Sigma_\varepsilon, \delta_2, q_0, F_2)) \leftarrow$ extendSFA$(\tau_2, q', Q', \Sigma_\varepsilon, q_0)$
      $Q' \leftarrow Q' \cup Q_2$, $\delta \leftarrow \delta \cup \delta_2$, $F \leftarrow F \cup F_2$, $L \leftarrow L \cup L_2$
    end for
    return $(L, (Q', \Sigma_\varepsilon, \delta, q_0, F))$
  else if $\tau = \tau_1 + \tau_2$ **then**
    $(L_i, (Q_i, \Sigma_\varepsilon, \delta_i, q_0, F_i)) \leftarrow$ extendSFA$(\tau_i, q, Q, \Sigma_\varepsilon, q_0)$, $i = 1, 2$
    return $(L_1 \cup L_2, (Q_1 \cup Q_2, \Sigma_\varepsilon, \delta_1 \cup \delta_2, q_0, F_1 \cup F_2))$
  else if $\tau = \varphi^!$ **then**
    return $(L \cup \{(q, \varphi)\}, (Q, \Sigma_\varepsilon, \emptyset, q_0, \emptyset))$
  else if $\tau = (\tau')^*$ **then**
    $(L, (Q', \Sigma_\varepsilon, \delta, q_0, F)) \leftarrow$ extendSFA$(\tau', q, Q, \Sigma_\varepsilon, q_0)$
    $\delta' \leftarrow \delta \cup \{(q', \varepsilon, q) : q' \in F\}$
    return $(L, (Q', \Sigma_\varepsilon, \delta', q_0, F \cup \{q\}))$
  end if
end function

function addAssertionEdges$(L, (Q, \Sigma, \delta, q_0, F))$
  $q_{fail} \leftarrow$ fresh state, $\delta' \leftarrow \delta$
  for all $(q, \varphi) \in L$ **do**
    for all $(q', (sto, \varphi'), q) \in \delta$ **do** ▷ similarly for φ'-only transitions
      $\delta' \leftarrow \delta' \setminus \{(q', (sto, \varphi'), q)\}$
      $s \leftarrow (sto, \varphi' \wedge \varphi)$
      $\delta' \leftarrow \delta' \cup \{(q', s, q), (q', \neg\varphi, q_{fail})\}$
    end for
  end for
  return $(Q \cup \{q_{fail}\}, \Sigma, \delta', q_0, F \cup \{q_{fail}\})$
end function

Algorithm 3 Construction of Initial Simulation Relation

function INITSIM(Q_l, Q_r, δ_l, δ_r)
  $R \leftarrow Q_l \times Q_r$, *changed* $\leftarrow$ *true*
  while *changed* = *true* **do**
    changed $\leftarrow$ *false*
    for all $(q_l, q_r) \in R$, $(q_l, s, q_l') \in \delta_l$ **do**
      if $\neg\exists (q_r, s', q_r') \in \delta_r$ **then** $R \leftarrow R \setminus (q_l, q_r)$, *changed* $\leftarrow$ *true* **end if**
    end for
  end while
  return R
end function

Algorithm 4 Subsumption Checking

function SUBSUMPTION(s, s', *substs*)
  substs$' \leftarrow \emptyset$
  for all *subst* $\in$ *substs* **do**
    substs$' \leftarrow$ *substs*$' \cup \{$*subst* $\circ$ *subst*$' \mid$ *subst*$'$ such that $s \sqsubseteq_{subst'}$ *subst*$(s')\}$
  end for
  return *substs*$'$
end function

inside the loop body, which then could be evaluated to decide whether to continue or leave the loop. Basically, we can within our framework apply the same solutions as known from *symbolic execution*: (Bounded) loop unwinding and invariant reasoning. The simplest solution is loop unwinding, by which symbolic lifting can be defined as follows:

$$\mathit{slift}_{\mathcal{L}_0}(\mathit{sto}, \varphi)(\texttt{while(b) } p) \coloneqq \mathit{slift}_{\mathcal{L}_0}(\mathit{sto}, \varphi)(\texttt{if(b) } p \texttt{ ; while(b) } p)$$

This, however, does generally not terminate for loops with symbolic guards. In the context of BMC, the bounded lifting function $\mathit{slift}^k_{\mathcal{L}_0}$ would unwind the loop as above exactly k times and then remove the loop statement. A standard approach in deductive program verification is to use *loop invariants*. Let $\mathit{Inv} \in$ Fml. Then, we can replace a loop `while(b)` p by the following program:

assert *Inv*; **havoc**; **assume** *Inv*; **if** (b) { p; **assert** *Inv* }

The **havoc** statement erases the state: $\mathit{slift}_{\mathcal{L}_0}(\mathit{sto}, \varphi)(\texttt{havoc}) \coloneqq \text{true}$. The replacement as above is sound for the left side of the trace modality: If *Inv* is not an invariant, the program evaluates to a failure trace, which can only be part of the traces for the right side if there also occurs a failed assertion. Otherwise, the new program evaluates to *at least* the same traces as the old one. It would also be sound *terminate* the trace after the assertion in the **if**, either by an **exit** statement or by wrapping the remaining program in an **else** block.

Reasoning with Symbolic Traces In the following, we provide two examples (Examples 9 and 10) to establish an intuition about how to solve the problem of symbolic trace subsumption. This is meant to support understanding the prin-

ciples behind Algo. 1, although the algorithm itself is not directly used there. Example 11 after that applies Algo. 1 to the problem of Example 9.

For the subsequent example, we assume that our term language has a conditional operator $\varphi\,?\,t_1 : t_2$, intuitively evaluating to the value of t_1 if φ holds and otherwise to that of t_2.

Example 9 (Functional Verification). Consider the following program p computing the difference of two integers a and b:

"`res=0; if (b < a) { tmp=a; a=b; b=tmp; } res=b-a`"

For this program, we want to show the post condition $\varphi \coloneqq \texttt{res} \geq 0$, i.e., $\models [\mathtt{p} \Vdash_{\alpha_{big}} \varphi]$. We first compute the symbolic traces by symbolic lifting, starting from an initial store $st_0 \coloneqq \mathtt{a} := a_0 \,||\, \mathtt{b} := b_0 \,||\, \mathtt{res} := res_0 \,||\, \mathtt{tmp} := tmp_0$:

$$
\begin{aligned}
&\mathit{slift}_{\mathcal{L}_0}(st_0, \text{true})(\mathtt{p}) = \\
&\quad (st_0 \circ (\,\mathtt{res} := 0\,), \text{true}); \\
&\quad \big(\quad ((st_0 \circ (\mathtt{res} := 0),\, b_0 < a_0\,); \\
&\qquad (st_0 \circ (\mathtt{res} := 0 \,||\, \mathtt{tmp} := \mathtt{a}\,), b_0 < a_0); \\
&\qquad (b := b_0 \,||\, \mathtt{res} := 0 \,||\, \mathtt{tmp} := a_0 \,||\, \mathtt{a} := b_0\,, b_0 < a_0); \\
&\qquad (\mathtt{res} := 0 \,||\, \mathtt{tmp} := a_0 \,||\, \mathtt{a} := b_0 \,||\, \mathtt{b} := a_0\,, b_0 < a_0); \\
&\qquad (\mathtt{tmp} := a_0 \,||\, \mathtt{a} := b_0 \,||\, \mathtt{b} := a_0 \,||\, \mathtt{res} := a_0 - b_0\,, b_0 < a_0)) \\
&\quad + ((st_0 \circ (\mathtt{res} := 0),\, b_0 \geq a_0\,); \\
&\qquad (\mathtt{a} := a_0 \,||\, \mathtt{b} := b_0 \,||\, \mathtt{tmp} := tmp_0 \,||\, \mathtt{res} := b_0 - a_0\,, b_0 \geq a_0))\big)
\end{aligned}
$$

During symbolic lifting, we simultaneously simplified the symbolic state, e.g., the update $st_0 \circ (\mathtt{res} := 0 \,||\, \mathtt{tmp} := \mathtt{a}) \circ (\mathtt{a} := \mathtt{b})$ is simplified to $\mathtt{b} := b_0 \,||\, \mathtt{res} := 0 \,||\, \mathtt{tmp} := a_0 \,||\, \mathtt{a} := b_0$. Note that alternatively, one could use state merging to create an equivalent trace with only one final state:

$$
\begin{aligned}
\mathit{slift}_{\mathcal{L}_0}(st_0, \text{true})(\mathtt{p}) = (st_0 \circ (\mathtt{res} := 0), \text{true}); \\
\big(\quad ((st_0 \circ (\mathtt{res} := 0), b_0 < a_0); \\
(st_0 \circ (\mathtt{res} := 0 \,||\, \mathtt{tmp} := a_0), b_0 < a_0); \\
(st_0 \circ (\mathtt{res} := 0 \,||\, \mathtt{tmp} := a_0 \,||\, \mathtt{a} := b_0), b_0 < a_0); \\
(\mathtt{res} := 0 \,||\, \mathtt{tmp} := a_0 \,||\, \mathtt{a} := b_0 \,||\, \mathtt{b} := a_0, b_0 < a_0)) \\
+ (st_0 \circ (\mathtt{res} := 0), b_0 \geq a_0)\big); \\
(\mathtt{tmp} := (b_0 < a_0)\,?\,a_0 : tmp_0 \,|| \\
\mathtt{a} := (b_0 < a_0)\,?\,b_0 : a_0 \,|| \\
\mathtt{b} := (b_0 < a_0)\,?\,a_0 : b_0 \,||
\end{aligned}
$$

$$\texttt{res} := (b_0 < a_0)\,?\,a_0 - b_0 : b_0 - a_0, \text{true})$$

Thus, it is always possible (for deterministic programs) to create symbolic traces with exactly one final state. Symbolically lifting the post condition leads to $\mathit{slift}_{\text{Fml}}(st_0, \text{true})(\varphi) = \text{true}^*; (\texttt{res} \geq 0)$. Applying big step abstraction, we have to show that the symbolic trace $\texttt{res} \geq 0$ subsumes both traces $(\texttt{tmp} := a_0 \,||\, \texttt{a} := b_0 \,||\, \texttt{b} := a_0 \,||\, \texttt{res} := a_0 - b_0, b_0 < a_0)$ and $(\texttt{res} := b_0 - a_0, b_0 \geq a_0)$. We can do so by evaluating the following formula in a theorem prover or SMT solver:

$$\begin{aligned}(b_0 < a_0 \rightarrow \{\texttt{tmp} := a_0 \,||\, \texttt{a} := b_0 \,||\, \texttt{b} := a_0 \,||\, \texttt{res} := a_0 - b_0\}\texttt{res} \geq 0) \,\wedge\\ (b_0 \geq a_0 \rightarrow \{\texttt{res} := b_0 - a_0\}\texttt{res} \geq 0)\end{aligned}$$

which is equivalent to $(b_0 < a_0 \rightarrow a_0 - b_0 \geq 0) \wedge (b_0 \geq a_0 \rightarrow b_0 - a_0 \geq 0)$ and therefore valid, which is why $\models [\texttt{p} \Vdash_{\alpha_{big}} \varphi]$ also holds. In this example, we also could have shown the assertion $\Box\varphi$, i.e. that the trace $\mathit{slift}_{TR}(st_0, \text{true})(\Box\varphi) = \varphi^*$ subsumes the symbolic trace of p, since φ holds for all intermediate states. ◊

The above example demonstrated how to match symbolic traces for a program and a post condition after big step abstraction. In the following, we investigate the situation for two *programs* in a program synthesis setting.

Example 10 (Synthesis). Consider `IntSqrt` and its scaffold for synthesis from Listings 1 and 2. We aim to show that the program p is a specialization of the scaffold sc, i.e., that $\models [\texttt{p} \Vdash \texttt{sc}]$. Since we already have an invariant at hand, we can use invariant reasoning to handle loops. Before, we mentioned that this is generally unsound for the right-hand side of the trace modality, since it constitutes an abstraction. However, we apply the same abstraction on the left and right-hand side, which is why this technique is admissible. Additionally, one has to check that the invariant is not unsatisfiable, since otherwise, both sides evaluate to the failure state and the property can be shown trivially. Let $st_0 = (\texttt{x} := x_0 \,||\, \texttt{v} := v_0 \,||\, \texttt{i} := i_0)$ be an initial store. The invariant is $\mathit{Inv}(\texttt{v}, \texttt{i}, \texttt{x}) := \texttt{v} = \texttt{i}^2 \wedge \texttt{x} \geq (\texttt{i} - 1)^2 \wedge \texttt{i} \geq 1$. We obtain the following symbolic traces (to ease the presentation, we omit abstract path conditions for schematic statements):

$$\begin{aligned}&\mathit{slift}_{\mathcal{L}}(st_0, x_0 \geq 1)(\texttt{p}) =\\
&\quad (st_0 \circ (\,\texttt{v} := 1\,), x_0 \geq 1);\\
&\quad (\texttt{x} := x_0 \,||\, \texttt{v} := 1 \,||\, \texttt{i} := 1, x_0 \geq 1);\\
&\quad (x_0 \geq 1 \rightarrow 1 = 1^2 \wedge x_0 \geq (1-1)^2 \wedge 1 \geq 1)^!;\\
&\quad (\texttt{x} := x_1 \,||\, \texttt{v} := v_1 \,||\, \texttt{i} := i_1, x_0 \geq 1);\\
&\quad (\texttt{x} := x_1 \,||\, \texttt{v} := v_1 \,||\, \texttt{i} := i_1, x_0 \geq 1 \wedge \mathit{Inv}(v_1, i_1, x_1));\\
&\quad \big(((\texttt{x} := x_1 \,||\, \texttt{v} := v_1 \,||\, \texttt{i} := i_1, \mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1);\\
&\qquad (\texttt{x} := x_1 \,||\, \texttt{i} := i_1 \,||\, \texttt{v} := v_1 + 2i_1 + 1, \mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1);\end{aligned}$$

$$
\begin{array}{l}
(\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 + 2i_1 + 1 \,||\, \mathtt{i} := i_1 + 1 \,, \mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1); \\
((\mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1) \rightarrow \mathit{Inv}(v_1 + 2i_1 + 1, i_1 + 1, x_1))^!) \; + \\
((\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1, \mathit{Inv}(v_1, i_1, x_1) \wedge \; v_1 > x_1 \;); \\
(\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1 - 1 \,, \mathit{Inv}(v_1, i_1, x_1) \wedge v_1 > x_1)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{slift}_{\mathcal{L}}(st_0, x_0 \geq 1)(\mathtt{sc}) = \\
\quad \text{true}^*; (st_0 \circ \mathit{sto}_{\mathtt{P}} \,, x_0 \geq 1); \\
\quad (x_0 \geq 1 \rightarrow \{st_0 \circ \mathit{sto}_{\mathtt{P}}\}(\mathtt{v} \doteq 1 \wedge \mathtt{i} \doteq 1 \wedge \mathtt{x} \doteq 1))^!; \\
\quad (x_0 \geq 1 \rightarrow \{st_0 \circ \mathit{sto}_{\mathtt{P}}\}\mathit{Inv}(\mathtt{v}, \mathtt{i}, \mathtt{x}))^!; \\
\quad (\mathtt{x} := \; x_1 \; || \, \mathtt{v} := \; v_1 \; || \, \mathtt{i} := \; i_1 \;, x_0 \geq 1); \\
\quad (\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1, x_0 \geq 1 \wedge \; \mathit{Inv}(v_1, i_1, x_1) \;); \\
\quad \big(((\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1, \mathit{Inv}(v_1, i_1, x_1) \wedge \; v_1 \leq x_1 \;); \\
\qquad \text{true}^*; ((\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1) \circ \; \mathit{sto}_{\mathtt{Q}} \,, \mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1); \\
\qquad ((\mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1) \rightarrow \\
\qquad\quad \{(\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1) \circ \mathit{sto}_{\mathtt{Q}}\}\mathit{Inv}(\mathtt{v}, \mathtt{i}, \mathtt{x}))^!; \\
\qquad ((\mathit{Inv}(v_1, i_1, x_1) \wedge v_1 \leq x_1) \rightarrow \\
\qquad\quad\; \{(\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1) \circ \mathit{sto}_{\mathtt{Q}}\}\mathit{Inv}(\mathtt{v}, \mathtt{i}, \mathtt{x}))^!) \; + \\
\quad ((\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1, \mathit{Inv}(v_1, i_1, x_1) \wedge \; v_1 > x_1 \;); \\
\qquad \text{true}^*; \\
\qquad ((\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1) \circ \; \mathit{sto}_{\mathtt{R}} \,, \mathit{Inv}(v_1, i_1, x_1) \wedge v_1 > x_1); \\
\qquad (((\mathit{Inv}(v_1, i_1, x_1) \wedge v_1 > x_1) \rightarrow \\
\qquad\quad \{(\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1) \circ \mathit{sto}_{\mathtt{R}}\}(\; \mathtt{i}^2 \leq \mathtt{x} < (\mathtt{i} + 1)^2 \;))^!))
\end{array}
$$

We point out that we performed the same anonymization $\mathtt{x} := x_1 \,||\, \mathtt{v} := v_1 \,||\, \mathtt{i} := i_1$ for the **`havoc`** statement in both traces; this corresponds to explicit loop coupling. Alternatively, we could have left the job of finding suitable substitutions to the subsumption checker. Now, we have to decide whether $(\mathit{slift}_{\mathcal{L}}(st_0, x_0 \geq 1)(\mathtt{p})$ is subsumed by $\mathit{slift}_{\mathcal{L}}(st_0, x_0 \geq 1)(\mathtt{sc}))$. We do not explicitly construct SFAs for this example, but follow the symbolic trace of `p`, mapping it to that of `sc` in a "lock-step" approach. This corresponds to an "on-the-fly" simulation construction process.

The state $(st_0 \circ (\mathtt{v} := 1), x_0 \geq 1)$, for instance, is subsumed by true^*, and the state $(\mathtt{x} := x_0 \,||\, \mathtt{v} := 1 \,||\, \mathtt{i} := 1, x_0 \geq 1)$ by $(st_0 \circ \mathit{sto}_{\mathtt{P}}, x_0 \geq 1)$, since $\mathit{sto}_{\mathtt{P}}$ is yet uninstantiated and can therefore be instantiated to $\mathtt{v} := 1 \,||\, \mathtt{i} := 1$. Next are the

assertions before the loop: Since based on the instantiation of $sto_{\mathtt{p}}$ all of those are satisfied, the failure trace is neither in the left, nor in the right trace set, which is why we can continue. This way, we process the whole symbolic trace and finally find a simulation relation. ◊

Example 11 (Applying Algo. 1). We consider the example from Example 9. Algo. 1 first lifts the implementation p and specification φ to symbolic traces; this is already done in Example 9 (we consider the version with state merging for p). Then, it creates SFAs for p and φ (Algo. 2). Those automata are shown in Figs. 2 and 3. Now, we have to find the simulation relation (Algo. 1). The initial simulation produced by INITSIM is $(\{q_0, q_1, q_2, q_3, q_4, q_5\} \times \{q_6\}) \cup \{(q_5, q_7)\}$. The pair (q_3, q_7), for instance, is not contained, since there is an outgoing edge from q_3, but not from q_7. If the pair (q_0, q_6) was not contained in the initial simulation, we could stop here since property (SR2) would not be satisfied. The subsequent subsumption checking steps performed by FINDSSR do not eliminate any pair from this relation, since all symbolic states in the implementation are subsumed by true in the specification, and the state on the transition from q_3 to q_5 satisfies the post condition $\mathtt{res} \geq 0$. The algorithm has found a relation also satisfying (SR2) and returns YES. If, however, the program would set res simply to, say, -1 in a last step, then the symbolic state in the last transition, $(\mathtt{res} := -1, \mathrm{true})$ would not be subsumed by $\mathtt{res} \geq 0$ and the algorithm would return UNKNOWN. ◊

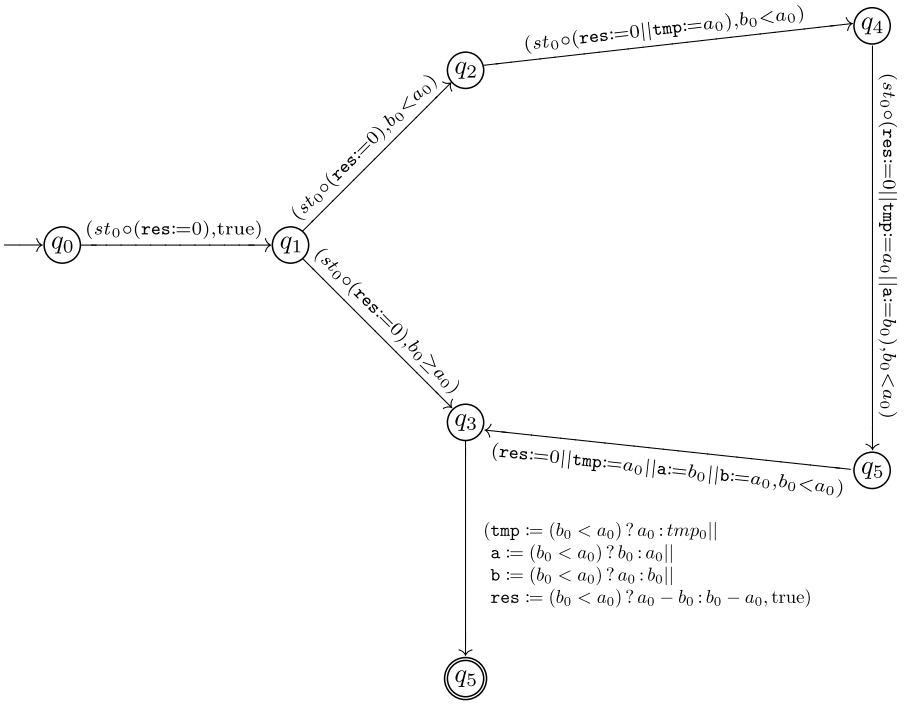

Fig. 2: Implementation SFA for Example 9

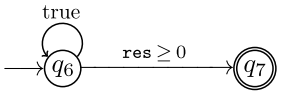

Fig. 3: Specification SFA for Example 9