The Trace Modality*

Dominic Steinhöfel[0000−0003−4439−7129] and Reiner Hähnle[0000−0001−8000−7613]

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
{steinhoefel, haehnle}@cs.tu-darmstadt.de

Abstract. We propose the *trace modality*, a concept to uniformly express a wide range of program verification problems. To demonstrate its usefulness, we formalize several program verification problems in it: Functional Verification, Information Flow Analysis, Temporal Model Checking, Program Synthesis, Correct Compilation, and Program Evolution. To reason about the trace modality, we translate programs *and* specifications to *regular symbolic traces* and construct simulation relations on first-order symbolic automata. The idea with this uniform representation is that it helps to identify synergy potential—theoretically and practically—between so far separate verification approaches.

1 Introduction

Since the foundational work on program verification during the 1960s [17], the program verification tasks that were studied have much broadened beyond mere functional (partial or total) correctness. Basic variations include termination [14], reachability [25], and program synthesis [16]. Starting in the early 2000s, verification of *relational* properties of programs, such as information flow [8], correct compilation [21], or correctness of program transformations (refactoring) [12] has been in the focus of interest. Relational properties compare two programs having similar behavior. It is even more challenging to reason about programs having related, but intentionally *differing* behavior, such as in program evolution [13].

For all these tasks *dedicated* verification approaches were developed: dynamic logic [15], Hoare quadruples [30], self composition [4,8], product programs [3], etc. Usually, the verification problem to be solved is stated informally, and then the problem is *directly* formalized in the approach to be used for its solution. Hence, the formalism that a problem is stated in and the formalism where it is solved, are *conflated*. We consider this problematic for two reasons:

(1) **Premature commitment to a specific solution approach.** If one has invested to master a specific methodology, the temptation to solve *any* problem by modifying or extending the familiar is considerable, even if a different approach would have been more efficient, flexible, or easily extensible: The well-known "for a hammer the world consists of nails" effect.

* This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. **The final authenticated version is available online at** https://doi.org/10.1007/978-3-030-38808-9.

(2) **Hard to detect commonalities and to transfer results.** One of the most powerful scientific stratagems is to detect structural similarity among different problem areas. This makes it possible to transfer insights and solutions from one problem space to another. In formal verification, this additionally opens the road to re-use of software tools for new tasks. To be able to spot commonalities, it is essential to know which aspects of a problem are genuinely new and hence require a novel approach. However, if a problem is *already formalized* in terms of a specific solution method, it is hard to identify commonality and analogy.

Experience with various software verification problems [14,8,1,28] let us realize that a small number of principles occur time and again in dedicated verification approaches: (1) *abstraction* of program runs in the sense of abstract interpretation [7]; (2) *approximation* of a set of program runs by a superset; (3) the capability to handle *schematic* programs, i.e. programs with unspecified parts. Abstraction makes it possible to compare programs written in different languages via a suitable abstraction of their traces. Approximation is needed to focus on a specific property and "forget" irrelevant information. Finally, to reason about program transformation (synthesis, compilation, refactoring, etc.) it is essential to be able to specify a program fragment in an unknown context. We propose a framework based on these principles that lets one express a wide variety of verification problems in a uniform, comparable manner.

We make only one assumption about the programs under verification: They must have a *trace semantics*, i.e. for an initial execution state s and a program p we can obtain the set of all traces ("program runs") that are possible when p is started in s. Our framework builds on the semantic notion of a *trace modality* $[\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]$, and a reasoning system based on *regular symbolic traces* and simulations on symbolic automata. The expression $[\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]$ is *valid* if the traces arising from the *implementation* $\mathcal{C}_l$ are *approximated* by the traces of the *specification* $\mathcal{C}_r$ after the *abstraction* step defined by α, where implementation and specification may be either programs (potentially containing *abstract contexts*) or formulas (e.g., in first-order or temporal logic). Symbolic traces approximate concrete traces. Our reasoning system translates implementation and specification to symbolic traces, transforms these to symbolic automata, and finally shows language inclusion by constructing a simulation relation "up to subsumption".

The paper is structured as follows. Sect. 2 defines elementary notions used in the paper. The semantics of the trace modality is described in Sect. 3, where we also formalize various verification tasks using the trace modality to demonstrate its expressiveness. In Sect. 4, we describe our reasoning system based on symbolic traces. Finally, Sect. 5 discusses related work, and Sect. 6 concludes the paper and describes future work opportunities. For space reasons, we moved some details of the paper (detailed examples, longer remarks) to an appendix, available at `www.key-project.org/papers/trace-modality/`.

2 Programs, Logic, Traces and Abstractions

We assume an imperative programming language $\mathcal{L}$ with the usual sequencing and assignment operators ";", "=". Programs may contain *schematic* statements, denoted with capital letters $\mathtt{P}$, $\mathtt{Q}$, etc. A program without schematic statements is called *concrete*. The set of concrete programs is $\mathcal{L}_0$. A program $\mathtt{p}$ with schematic statements represents the set $\mathrm{Concr}(\mathtt{p})$ of all well-formed $\mathcal{L}_0$-programs obtained by replacing each schematic statement in $\mathtt{p}$ with an arbitrary concrete statement.

At each point during the execution of a program $\mathtt{p} \in \mathcal{L}_0$ it is in a *state* $s \in \mathcal{S}$, mapping program variables to domain values. To model failed assertions, we distinguish a state $\bot$. We write $\mathcal{S}^\bot$ for $\mathcal{S} \cup \{\bot\}$. A *trace* τ is a possibly infinite sequence of states, denoted $s_0 s_1 \cdots s_n$ or $s_0 s_1 \cdots$ (the latter being infinite). For the empty trace we write ε and $\mathit{Traces} = (\mathcal{S}^\bot)^* \cup (\mathcal{S}^\bot)^\omega$ for the set of all traces. Predicate $\mathit{finite}(\tau)$ holds for finite traces and $\mathit{first}(\tau)$, $\mathit{last}(\tau)$ select a trace's first and final state (the latter is only defined for finite traces).

We assume a *trace semantics* $\mathrm{Tr}_s(\mathtt{p})$ that maps a *concrete* program $\mathtt{p} \in \mathcal{L}_0$ and initial state $s \in \mathcal{S}$ into the set of possible traces when $\mathtt{p}$ is started in s. When $\mathcal{L}_0$ is deterministic, this is a singleton. We define the set of all traces of $\mathtt{p} \in \mathcal{L}_0$ as $\mathrm{Tr}(\mathtt{p}) = \{\mathrm{Tr}_s(\mathtt{p}) \mid s \in \mathcal{S}\}$. If $\mathcal{P}$ is a set of concrete programs, then $\mathrm{Tr}(\mathcal{P}) = \{\mathrm{Tr}(\mathtt{p}) \mid \mathtt{p} \in \mathcal{P}\}$, similar for $\mathrm{Tr}_s(\mathcal{P})$. Now we define the semantics of a *schematic* program $\mathtt{p} \in \mathcal{L}$ as $\mathrm{Tr}(\mathrm{Concr}(\mathtt{p}))$ and $\mathrm{Tr}_s(\mathrm{Concr}(\mathtt{p}))$, respectively.

Let PVar denote the set of program variables and "$\circ$" the usual function composition operator. We define *abstraction operators* $\alpha : 2^{\mathit{Traces}} \to 2^{\mathit{Traces}}$ (in the sense of abstract interpretation [7]) on sets of traces $\mathcal{T}$:

Big-step abstraction: $\alpha_{big}(\mathcal{T}) = \{(s_0, s_n) \mid s_0 \cdots s_n \in \mathcal{T}\}$, i.e. the set of all pairs of the first and last state of any finite trace in $\mathcal{T}$. Observe that for infinite traces in $\mathcal{T}$, there is no corresponding pair in the abstracted set.

Observation abstraction: Let $\mathrm{obs} \subseteq \mathrm{PVar}$, $s \in \mathcal{S}$, then $s \downarrow \mathrm{obs}$ is the state s restricted to values from obs. We define the *observation abstraction* relative to obs as $\alpha_{\mathrm{obs}}(\mathcal{T}) = \{(s_0 \downarrow \mathrm{obs})(s_1 \downarrow \mathrm{obs}) \cdots \mid s_0 s_1 \cdots \in \mathcal{T}\}$. For a concrete set of variables, for instance $\{\mathtt{x}\}$, we write $\alpha_{\{\mathtt{x}\}}$.

Data abstraction: Let α_d be an abstract operator on data types in $\mathtt{p}$ [7]. We define the *data abstraction* of a set of traces as $\alpha_d(\mathcal{T}) = \{\alpha_d(s_0)\alpha_d(s_1) \cdots \mid s_0 s_1 \cdots \in \mathcal{T}\}$, where the state $\alpha_d(s)(\mathtt{x}) = \alpha_d(s(\mathtt{x}))$ is defined pointwise.

Combination: Combine two abstractions α_1, α_2 by composition $\alpha_1 \circ \alpha_2$.

We use a standard first-order language with equality. It contains the usual propositional connectives and first-order quantifiers. Terms and formulas are standard, but we permit trace modality formulas $[\mathcal{T}_l \Vdash_\alpha \mathcal{T}_r]$ as atomic formulas. With Trm and Fml we denote the sets of all terms and formulas. The semantics of a formula is provided by a first-order structure K and a state $s \in \mathcal{S}$ that define the *validity* relation $K, s \models \varphi$ for each $\varphi \in \mathrm{Fml}$. For example, $K, s \models \varphi \to \psi$ iff *either* $K, s \not\models \varphi$, *or* $K, s \models \varphi$ and $K, s \models \psi$. Given $s \in \mathcal{S}$, write $s \models \varphi$ and say φ is *valid* for s if $K, s \models \varphi$ for all K. Write $\models \varphi$ and say that φ is *valid* if $s \models \varphi$ for all $s \in \mathcal{S}$. While K interprets *rigid* first-order functions and predicates, a state s

assigns values to program variables that may *change* during execution. For the failure state $\bot$ we set $K, \bot \not\models \varphi$ for *any* K, φ.

We need $\texttt{assert}(\varphi)$ and $\texttt{assume}(\varphi)$ statements for asserting and assuming a first-order formula φ *in a program*. Our use cases are *program synthesis* for `assert` and *invariant reasoning* (appendix) for `assume` statements. For `assert`, we define $\mathrm{Tr}_s(\texttt{assert}(\varphi)) = \{s\}$ if $s \models \varphi$ and $\{\bot\}$ otherwise. The semantics of `assume` is defined as $\mathrm{Tr}_s(\texttt{assume}(\varphi)) = \{s\}$ if $s \models \varphi$ and $\emptyset$ otherwise. We define a full trace semantics for a simple $\mathcal{L}_0$-language in the appendix.

3 The Trace Modality

We define the *trace modality* $[\mathcal{T}_l \Vdash_\alpha \mathcal{T}_r]$, where the *implementation* $\mathcal{T}_l$ and *specification* $\mathcal{T}_r$ both are (possibly infinite) trace sets and α is a trace abstraction. It expresses that the specification is an approximation of the implementation relative to α. Its semantics is that the modality is *valid*, written $\models [\mathcal{T}_l \Vdash_\alpha \mathcal{T}_r]$, if $\alpha(\mathcal{T}_l) \subseteq \alpha(\mathcal{T}_r)$. We use *lifting functions* lift_l, lift_r that convert elements from the verification domain, such as programs or formulas, to trace sets. Formally:

Definition 1. *Let $\alpha : 2^{\mathit{Traces}} \to 2^{\mathit{Traces}}$ be a trace abstraction and $\mathcal{C}_l$, $\mathcal{C}_r$ elements of domains D_l, D_r with associated lifting functions $\mathit{lift}_{l/r} : \mathcal{S} \to D_{l/r} \to 2^{\mathit{Traces}}$. Then the trace modality $[\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]$ is valid in $s \in \mathcal{S}$, written $s \models [\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]$, iff $\alpha(\mathit{lift}_l(s)(\mathcal{C}_l)) \subseteq \alpha(\mathit{lift}_r(s)(\mathcal{C}_r))$.*

For readability, we omit lifting functions in the presentation and assume that verification domains have fixed lifting functions. We omit α if it is the identity.

Relation to Modal and Dynamic Logic Like in modal and dynamic logic (DL) we can define a dual modality as follows: $\langle \mathcal{C}_l \Vdash_\alpha \mathcal{C}_r \rangle \coloneqq \neg[\mathcal{C}_l \Vdash_\alpha \overline{\mathcal{C}_r}]$, where $\overline{\mathcal{C}_r}$ is the complement of $\mathcal{C}_r$. The semantics of this *diamond* trace modality can be phrased as: $s \models \langle \mathcal{C}_l \Vdash_\alpha \mathcal{C}_r \rangle$ iff there is a trace in $\alpha(\mathit{lift}_l(s)(\mathcal{C}_l))$ that is not in $\alpha(\overline{\mathit{lift}_r(s)(\mathcal{C}_r)})$. The axioms **N** (necessitation rule) and **K** (distribution axiom) of modal logic follow from Def. 1; axioms of Propositional Dynamic Logic (PDL) [15] also hold when defining suitable operators on trace sets (like sequencing and star). We refer to our discussion in the appendix (Remark 1) for these observations. Despite those similarities, the trace modality is strictly more general than DL. Specifications can originate from different verification domains; also a *program* can be a specification (we show several examples for this case later on) or a formula an implementation. Furthermore, we are not restricted to big-step reasoning. Because the notation $[p]p'$ would look strange for lengthy programs p' and to emphasize the mentioned differences between the trace modality and standard DL, we chose the notation also encapsulating the specification inside the box.

In the following, several verification tasks are described and formalized with the trace modality. We define suitable verification domains D, lifting functions lift_D, as well as abstractions. A summary table is in the appendix.

3.1 Functional Verification

In functional verification, one shows a given program $\mathtt{p} \in \mathcal{L}_0$ to satisfy a postcondition *Post* provided that a precondition *Pre* holds initially. The problem is frequently formalized with *Hoare triples* $\{Pre\}\mathtt{p}\{Post\}$ [17]. In DL [1,15], one writes $Pre \rightarrow [\mathtt{p}]Post$. We distinguish *partial correctness*, where *Post* is asserted to hold *if* p terminates, from *total correctness*, where it is also shown *that* p terminates. For the latter one can use the dual modality $\langle \mathtt{p} \rangle Post$.

Functional correctness is over the domain $D_{\mathcal{L}_0}$ for programs and D_{Fml} for first-order postconditions. We define lifting functions $lift_{\mathcal{L}_0}(s)(\mathtt{p}) \coloneqq \mathrm{Tr}_s(\mathtt{p})$ and $lift_{\mathrm{Fml}}(s)(\varphi) \coloneqq \{\tau \in Traces : finite(\tau) \wedge first(\tau) = s \wedge last(\tau) \models \varphi\}$ (all finite traces starting in s whose final state satisfies φ). Using the big-step abstraction, we can formalize (partial) functional correctness as $Pre \rightarrow [\mathtt{p} \Vdash_{\alpha_{big}} Post]$. Total correctness for *deterministic* programs is expressed as $Pre \rightarrow \langle \mathtt{p} \Vdash_{\alpha_{big}} Post \rangle$.

Example 1. Let $\mathtt{p} \coloneqq$`j=i*i; while (i<j) { i=i*2; }`. It diverges iff the initial value of `i` is negative. One can prove postcondition $even(\mathtt{i})$ (with the obvious meaning) for p *if it terminates* (partial correctness). Thus, $s \models [\mathtt{p} \Vdash_{\alpha_{big}} even(\mathtt{i})]$ must hold in all $s \in \mathcal{S}$, i.e. $\alpha_{big}(lift_{\mathcal{L}_0}(s)(\mathtt{p})) \subseteq \alpha_{big}(lift_{\mathrm{Fml}}(s)(even(\mathtt{i})))$. If $s(\mathtt{i}) < 0$ then the set $lift_{\mathcal{L}_0}(s)(\mathtt{p})$ contains a single infinite trace. Therefore, $\alpha_{big}(lift_{\mathcal{L}_0}(s)(\mathtt{p})) = \emptyset$ and the subset relation holds. If $s(\mathtt{i}) \geq 0$, p has a single finite trace whose final state assigns an even value to `i` (either because $s(\mathtt{i})$ is 0, or because it is greater than 0, and the initial value was multiplied by 2 a number of times in the loop's body). Hence, $\alpha_{big}(lift_{\mathcal{L}_0}(s)(\mathtt{p}))$ contains a single pair (s, s_f) where s_f satisfies $even(\mathtt{i})$. It is in $\alpha_{big}(lift_{\mathrm{Fml}}(s)(even(\mathtt{i})))$ by defn. of $lift_{\mathrm{Fml}}$. We cannot show $s \models \langle \mathtt{p} \Vdash_{\alpha_{big}} even(\mathtt{i}) \rangle$ for any s with $s(\mathtt{i}) < 0$, because then $\alpha_{big}(lift_{\mathcal{L}_0}(s)(\mathtt{p}))$ is empty and so cannot contain a trace not in $\alpha_{big}(\overline{lift_{\mathrm{Fml}}(s)(even(\mathtt{i}))})$. However, $\models \mathtt{i} \geq 0 \rightarrow \langle \mathtt{p} \Vdash_{\alpha_{big}} even(\mathtt{i}) \rangle$ is true. $\Diamond$

3.2 Information Flow Analysis

To prove that a given program treats secret inputs (for example, a password) confidentially, i.e. it does not inadvertently leak secret information, one can formally prove that it satisfies an *information flow policy*. In the simplest case such policies partition program variables into *low*-security variables that hold observable values and *high*-security ones whose values are secret. A policy imposes restrictions on the flow of values from *high* to *low* variables. A standard and very strong policy is *non-interference*: "Whenever two instances of the same program are run with equal *low* values and arbitrary *high* values, then the resulting *low* values are equal in the final state". This ensures that an attacker cannot learn anything about secret values by running the program with observable values. For simplicity, assume a program p contains exactly one low variable `l` and one high variable `h`, written $\mathtt{p}(\mathtt{l},\mathtt{h})$. Using *self composition* [4,8], this is formalized as a Hoare triple: If we can prove $\{\mathtt{l} \doteq \mathtt{l}'\}\mathtt{p}(\mathtt{l},\mathtt{h});\mathtt{p}(\mathtt{l}',\mathtt{h}')\{\mathtt{l} \doteq \mathtt{l}'\}$, p satisfies non-interference. It can also be directly expressed with the trace modality: $\models [\mathtt{p}(\mathtt{l},\mathtt{h}) \Vdash_{\alpha_{\{1\}} \circ \alpha_{big}} \mathtt{p}(\mathtt{l},\mathtt{h}')]$. Note that the renaming of `l` to $\mathtt{l}'$ is then not

necessary since programs are not composed, but evaluated separately. In the appendix, we discuss how *declassification* can be encoded with the trace modality.

Example 2. Let $\mathtt{p} \coloneqq \mathtt{l=42;}$ **if** $\mathtt{(h>20)}$ $\mathtt{\{l=17;\}}$. This program does not satisfy non-interference, because the final value of the observable variable $\mathtt{l}$ depends on the initial value of $\mathtt{h}$. We prove that indeed, $\models [\mathtt{p}(\mathtt{l},\mathtt{h}) \Vdash_{\alpha_{\{\mathtt{l}\}} \circ \alpha_{big}} \mathtt{p}(\mathtt{l},\mathtt{h}')]$ does *not* hold, by showing that there is a state $s \in \mathcal{S}$ for which

$$(\alpha_{\{\mathtt{l}\}} \circ \alpha_{big})(\mathit{lift}_{\mathcal{L}_0}(s)(\mathtt{p}(\mathtt{l},\mathtt{h}))) \subseteq (\alpha_{\{\mathtt{l}\}} \circ \alpha_{big})(\mathit{lift}_{\mathcal{L}_0}(s)(\mathtt{p}(\mathtt{l},\mathtt{h}')))$$

is not true. Let s be such that $s(\mathtt{l}) = 0$, $s(\mathtt{h}) = 0$ and $s(\mathtt{h}') = 30$. Then the trace set of the implementation is $\{(\{\mathtt{l} \mapsto 0\}, \{\mathtt{l} \mapsto 42\})\}$ which is not contained in the set for the specification $\{(\{\mathtt{l} \mapsto 0\}, \{\mathtt{l} \mapsto 17\})\}$. ◊

3.3 Software Model Checking

Software Model Checking (SMC) [19] describes a wide range of techniques for analyzing *safety* or *liveness* properties of programs. Those techniques have in common that they focus on *automation* at cost of expressivity. Frequently, the goal is not to prove correctness relative to a specification, but rather to quickly uncover bugs or to generate high-coverage test cases. Recently, there has been a *convergence* between *model checking* and *deductive verification* techniques [26], as more mechanisms traditionally known from the latter field, such as abstraction [29], symbolic execution [23], etc., are integrated to achieve greater expressivity. On the other side, Bounded Model Checking (BMC) approaches, which limit state space exploration by a user-defined upper bound on loop unwindings, are well-known and successful, and finite space checkers such as SPIN [18] continue being used, e.g. in protocol verification. Properties of interest to SMC (e.g., the absence of memory faults) can usually be formalized in Temporal Logic (TL).

We introduce the domain D_{TL} for Linear Temporal Logic (LTL) formulas, lift_{TL} is the standard trace semantics for temporal logic (e.g., $\mathit{lift}_{TL}(s)(\Box p)$ is the set of all traces starting in s where p always holds). We exemplarily instantiate the trace modality to *Finite Space* MC. Finite space model checkers like SPIN 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. In the appendix, we show how to instantiate the trace modality to Bounded MC, Abstraction-Based MC and Symbolic Execution-Based MC. Model Checking tools for bug finding can be formalized with the diamond trace modality: They eagerly try to show $\models \langle \mathtt{p} \Vdash \neg\varphi \rangle$, i.e. there is a trace of $\mathtt{p}$ violating φ. Such a trace constitutes a counterexample which can be used to fix the program, and/or to create a useful test case.

So far, we considered *concrete* programs $\mathtt{p} \in \mathcal{L}_0$. The two subsequently discussed verification tasks are over *schematic* programs in $\mathcal{L}$.

3.4 Program Synthesis

Automated program synthesis starts with a specification of programs at a higher level than executable code. The latter is created (semi-)automatically from the specification. In [27], for instance, the user supplies a *scaffold* consisting of a functional specification $(Pre, Post)$, domain constraints defining the domains of expressions and guards, and a *schematic program* (called "flowgraph template") of the form $\bullet|*(T)|T;T$. Here, $\bullet$ is an acyclic fragment, T again a schematic program and $*(T)$ a loop with body T. The synthesizer infers *synthesis conditions*. These are satisfiable whenever there exists a valid program for the scaffold.

We encode $\bullet$ of the flowgraph template by programs $\mathtt{p} \in \mathcal{L}$ with schematic statements $\mathtt{P}$, $\mathtt{Q}$, ..., and define a new verification domain $D_{\mathcal{L}}$ with $\mathit{lift}_{\mathcal{L}}(s)(\mathtt{p}) \coloneqq \mathrm{Tr}_s(\mathrm{Concr}(\mathtt{p}))$. Synthesis conditions are included in the intermediate program as suitable $\texttt{assert}(\varphi)$ statements. When refining an intermediate program $\mathtt{p}$ to a more concrete program $\mathtt{p}'$, the property to show is $\models \mathit{Pre} \to [\mathtt{p}' \Vdash_{\alpha_{big}} \mathtt{p}]$: that $\mathtt{p}'$ is indeed a refinement of $\mathtt{p}$. In the appendix, we provide an example for the synthesis of a program computing integer square roots.

3.5 Correct Compilation

A compiler translates a program $\mathtt{p}$ in a source language into a program $\mathtt{c}$ of a target language, preserving the behavior of $\mathtt{p}$. The translation can introduce new program variables. Then, preservation of behavior is typically restricted to a set of *observable* variables *obs*. In *modular* compilation, a program $\mathtt{p}$ is given within an unspecified context. In this case both $\mathtt{p}$ and $\mathtt{c}$ are abstract. Correctness of compilation can be expressed as $\models [\mathtt{c} \Vdash_{\alpha_{obs} \circ \alpha_{big}} \mathtt{p}]$. If we want to enforce inclusion of the traces of $\mathtt{c}$ in the traces of $\mathtt{p}$, we can—for deterministic languages—use the diamond modality instead. In particular for non-deterministic languages, we can *additionally* prove the reverse direction $\models [\mathtt{p} \Vdash_{\alpha_{obs} \circ \alpha_{big}} \mathtt{c}]$.

The formalization makes the similarity to program synthesis explicit. Indeed, one could create a scaffold by extracting synthesis conditions from $\mathtt{p}$, and then try to infer $\mathtt{c}$ automatically. For example, in [28], a symbolic execution tree of the source program is "mined" to extract the target program. It is related to proof mining techniques used in program synthesis.

3.6 Program Evolution & Bug Fixing

Sometimes, the behavior of the "specification" should intentionally be *not* preserved. This situation occurs in program evolution, e.g., after manual or automatic bug fixing [22]: the patched program is supposed to exhibit the bug no longer, but no new bug is to be introduced. Similarly, in fault propagation analysis, an injected fault typically *will* change the behaviour of a program, but not arbitrarily. This problem is most naturally expressed as $\models [\mathtt{p}_{\mathit{fixed}} \Vdash_{\alpha_{bug} \circ \alpha_{big}} \mathtt{p}_{\mathit{bug}}]$, where behavioral differences are conveyed for a suitable abstraction α_{bug} suppressing buggy traces or relating them to corrected ones. We can go a step further and not just exclude buggy traces, but encode the *fix* by an abstraction

α_{patch}. This is likely to produce a more reliable result asserting that apart from the fix, the programs behave equivalently, even for the formerly buggy paths. In the appendix, we discuss two alternative formalizations with an example.

4 Reasoning about the Trace Modality

We propose a reasoning algorithm based on *symbolic traces* for the trace modality. The idea is to lift all verification domains to a common language over symbolic traces that over-approximates the set of concrete traces produced by each lifting function. Abstractions are generalized to symbolic traces. Validity of the trace modality can then be established by *symbolic trace subsumption*. The symbolic traces we propose are a *regular* language. Hence, programs generally have to be over-approximated, for example, by loop invariant reasoning or bounded loop unwinding. Not all properties can be encoded in a regular symbolic trace language, such as complex Computation Tree Logic properties. Even so, the language is expressive enough to represent the problems formalized in Sect. 3, and problems encoded in it can be solved effectively (if the underlying first-order problems can be solved). We define symbolic stores, states and traces as follows:

Definition 2. *Let* $\mathtt{x} \in \mathrm{PVar}$, $t \in \mathrm{Trm}$, $\varphi \in \mathrm{Fml}$, *and* $\mathtt{P}$ *a schematic statement. The sets* SymSto *of* Symbolic Stores, SymState *of* Symbolic States, *and* SymTr *of* Symbolic Traces *are defined as follows in extended BNF:*

$$\begin{aligned} \mathrm{SymSto} &::= \mathtt{x}\ \text{“:=”}\ t \mid \mathit{sto}_{\mathtt{P}} \mid \mathrm{SymSto}\ \text{“||”}\ \mathrm{SymSto} \mid \text{“\{”}\ \mathrm{SymSto}\ \text{“\}”}\ \mathrm{SymSto} \\ \mathrm{SymState} &::= \varphi \mid \text{“(”}\ \mathrm{SymSto}\ \text{“,”}\ \varphi\ \text{“)”} \\ \mathrm{SymTr} &::= \mathrm{SymState} \mid \mathrm{SymTr}\,(\text{“;”} \mid \text{“+”})\,\mathrm{SymTr} \mid \varphi\ \text{“}^{!}\text{”} \mid \mathrm{SymTr}\ \text{“}^{*}\text{”} \end{aligned}$$

Here an abstract *store* $\mathit{sto}_{\mathtt{P}}$ *represents an unknown state transition induced by a schematic* $\mathcal{L}$*-statement* $\mathtt{P}$. *The sets* SymSto_0, $\mathrm{SymState}_0$ *and* SymTr_0 *are defined as above, but do not contain abstract stores.*

Symbolic stores record changes to program variables. Elementary stores $\mathtt{x} := t$ represent states where the variable $\mathtt{x}$ attains the valuation of the (symbolic) term t. Symbolic stores sto_1, sto_2 are combined to a parallel store $\mathit{sto}_1 \,||\, \mathit{sto}_2$. If both assign a value to the same variable, the later assignment (in sto_2) “wins”. A symbolic store sto_1 can be *applied* to a symbolic store sto_2, written $\{\mathit{sto}_1\}\mathit{sto}_2$. Left-hand sides in sto_2 are then evaluated in the states represented by sto_1. Combining two stores into one works by the *store concatenation operator* “$\circ$”, defined as $\mathit{sto}_1 \circ \mathit{sto}_2 = \mathit{sto}_1 \,||\, \{\mathit{sto}_1\}\mathit{sto}_2$. We permit the application of symbolic stores to terms and formulas, with similar semantics. We write $\overrightarrow{\mathtt{x}} := \overrightarrow{t}$ for the store $\mathtt{x}_1 := t_1 \,||\, \ldots \,||\, \mathtt{x}_n := t_n$, where $\mathtt{x}_i$, t_i are the i-th components of $\overrightarrow{\mathtt{x}}$, $\overrightarrow{t}$.

Symbolic states consist of an (optional) symbolic store *sto* and path condition φ representing concrete states satisfying both φ and, if present, the assignments in *sto*. A symbolic trace is in the simplest case a sequence of symbolic states. The choice operator + models nondeterministic choice as well as case distinctions for deterministic programs, depending on the path conditions of the argument traces. The trace $\varphi^{!}$, primarily used to model assertions, represents the empty

$$
\begin{aligned}
\mathsf{tval}_K :&\quad \mathrm{SymSto}_0 \to (\mathcal{S} \to \mathcal{S}) \\
\mathsf{tval}_K(\mathrm{x} := t)(s)(\mathrm{y}) &\coloneqq \mathsf{val}(K, s; t) \text{ if } \mathrm{y} = \mathrm{x},\ s(\mathrm{y}) \text{ otherwise} \\
\mathsf{tval}_K(sto_1 \,||\, sto_2)(s) &\coloneqq \mathsf{tval}_K(sto_2)(\mathsf{tval}_K(sto_1)(s)) \\
\\
\mathsf{tval}_K :&\quad \mathrm{SymState}_0 \to 2^{\mathcal{S}} \\
\mathsf{tval}_K(\varphi) &\coloneqq \{s \in \mathcal{S} \mid K, s \models \varphi\} \\
\mathsf{tval}_K(sto, \varphi) &\coloneqq \{\mathsf{tval}_K(sto)(s) \mid K, s \models \varphi,\ s \in \mathcal{S}\} \\
\\
\mathsf{tval}_K :&\quad \mathrm{SymTr}_0 \to 2^{Traces} \\
\mathsf{tval}_K(\tau_1; \tau_2) &\coloneqq \{\tau_1^0 \tau_2^0 \mid \tau_1^0 \in \mathsf{tval}_K(\tau_1) \setminus \{\cdots \bot\},\ \tau_2^0 \in \mathsf{tval}_K(\tau_2)\} \\
&\quad \cup \{\bot \mid \tau_1^0 \bot \in \mathsf{tval}_K(\tau_1)\} \\
\mathsf{tval}_K(\tau_1 + \tau_2) &\coloneqq \mathsf{tval}_K(\tau_1) \cup \mathsf{tval}_K(\tau_2) \\
\mathsf{tval}_K(\varphi^!) &\coloneqq \begin{cases} \varepsilon & \text{if } \forall s \in \mathcal{S} :\ K, s \models \varphi \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{tval}_K(\tau^*) &\coloneqq \{\tau_1^0 \tau_2^0 \cdots \tau_n^0 : \tau_i \in \mathsf{tval}_K(\tau),\ n \in \mathbb{N}\} \cup \{\varepsilon\}
\end{aligned}
$$

Fig. 1: The Valuation Function tval_K

trace if φ holds in the current state and the failure state otherwise. The traces τ^* represent all finite concrete traces in which all states satisfy τ. For instance, true^* represents the set of all finite concrete traces. We do not include an operator τ^ω for infinite traces which would significantly complicate validity checking: One would have to separate terminating from non-terminating traces—which is undecidable—or consider only non-terminating runs.

A formal semantics for symbolic traces is based on a first-order structure K with domain D and interpretation I, as well as $s \in \mathcal{S}$. The *valuation function* $\mathsf{val}(K, s; \cdot)$ assigns to terms a value in D, to formulas "true" or "false." We write equivalently $K, s \models \varphi$ or $\mathsf{val}(K, s; \varphi) = \mathrm{true}$, as well as $K, s \not\models \varphi$ or $\mathsf{val}(K, s; \varphi) = \mathrm{false}$. The function $\mathsf{val}(K, s; \cdot)$ is defined as usual, except for the application of symbolic stores and the valuation of program variables. For $\mathrm{x} \in \mathrm{PVar}$, we define $\mathsf{val}(K, s; \mathrm{x}) = s(\mathrm{x})$. If $t \in \mathrm{Trm}$ and $sto \in \mathrm{SymSto}$, we define $\mathsf{val}(K, s; \{sto\}t) \coloneqq \mathsf{val}(K, s'; t)$, where $s' = \mathsf{val}(K, s; sto)(s)$ (similarly for formulas). We define the trace valuation function tval_K first on *concrete* symbolic traces SymTr_0. It is parametric in a structure K that fixes the values of uninterpreted constant, function, and predicate symbols. The cumulative valuation function tval is canonically defined as $\mathsf{tval}(\tau) \coloneqq \bigcup_K \mathsf{tval}_K(\tau)$.

Definition 3. *We inductively define the valuation function* tval_K*, overloaded for symbolic stores, states and traces, as in Fig. 1.*

Symbolic traces SymTr_0 are created for concrete programs $\mathcal{L}_0$. The symbolic evaluation of schematic programs in $\mathcal{L}$ creates abstract stores sto_{P} and path conditions C_{P} (details below). Intuitively, they represent *all possible symbolic*

stores and path conditions that may arise from concrete program execution. We define their semantics by the union of the semantics of possible instantiations.

Definition 4. *Let* $\tau \in \mathrm{SymTr}$ *be a symbolic trace with occurrences of abstract stores* $sto_{\mathtt{P}_1}, \ldots, sto_{\mathtt{P}_n}$ *and path conditions* $C_{\mathtt{P}_1}, \ldots, C_{\mathtt{P}_n}$ *(with possibly multiple occurrences of each* $sto_{\mathtt{P}_i}$*,* $C_{\mathtt{P}_i}$*). We define* $\mathsf{tval}(\tau)$ *as the union* $\bigcup \mathsf{tval}(\tau_0)$ *of all* $\tau_0 \in \mathrm{SymTr}_0$ *that are obtained by instantiating all occurrences of* $sto_{\mathtt{P}_i}$*,* $C_{\mathtt{P}_i}$ *with concrete stores* $sto^0_{\mathtt{P}_i} \in \mathrm{SymSto}_0$ *and path conditions* $C^0_{\mathtt{P}_i} \in \mathrm{Fml}$*.*

Abstractions α are generalized to symbolic traces in the obvious manner, e.g., the big-step abstraction α_{big} takes the first and all final states of a trace. Symbolic representations of the lifting functions require more work. For a lifting function *lift*, we denote by *slift* its symbolic version. Like *lift*, *slift* takes a *symbolic* state and a verification domain construct and produces a *symbolic* trace.

Definition 5. *A symbolic lifting function slift is* correct *relative to lift if, for all* $s \in \mathrm{SymState}$ *and* $\sigma \in \mathsf{tval}(s)$*,* $lift(\sigma)(C) \subseteq \mathsf{tval}(slift(s)(C))$*.*

Symbolic lifting functions for first-order formulas are straightforward to define: $slift_{\mathrm{Fml}}(s)(\varphi) \coloneqq \mathrm{true}^*; \varphi$. For LTL formulas, $slift_{TL}(s)$ maps (1) φ to "φ", (2) $\Box\varphi$ to "φ^*", (3) $\Diamond\varphi$ to "$\mathrm{true}^*; \varphi; \mathrm{true}^*$", and (4) $\varphi \,\mathcal{U}\, \psi$ to "$\varphi^*; \psi; \mathrm{true}^*$".

Defining symbolic lifting for programs means encoding symbolic execution. E.g., one can extract symbolic traces from a symbolic execution tree. Symbolic *traces* are more flexible, though, since they can encode non tree-like structures. The lifting function $slift_{\mathcal{L}_0}$ is defined as follows for assignments, if-else, assume and assert, and sequential composition (for those, it coincides with $slift^k_{\mathcal{L}_0}$ for BMC). W.l.o.g., we assume symbolic states to be of the form (sto, φ).

$$
\begin{aligned}
slift_{\mathcal{L}_0}(sto, \varphi)(\mathtt{x=e}) &\coloneqq (sto \circ (\mathtt{x} := e), \varphi) \\
slift_{\mathcal{L}_0}(sto, \varphi)(\mathtt{if(g)}\ p_1\ \mathtt{else}\ p_2) &\coloneqq (slift_{\mathcal{L}_0}(sto, \varphi \wedge \{sto\}g))(p_1) \,+ \\
&\qquad\quad (slift_{\mathcal{L}_0}(sto, \varphi \wedge \neg\{sto\}g))(p_2) \\
slift_{\mathcal{L}_0}(sto, \varphi)(\mathtt{assume}(\psi)) &\coloneqq (sto, \varphi \wedge \psi) \\
slift_{\mathcal{L}_0}(sto, \varphi)(\mathtt{assert}(\psi)) &\coloneqq (\varphi \rightarrow \{sto\}\psi)^! \\
slift_{\mathcal{L}_0}(sto, \varphi)(p_1\,;\, p_2) &\coloneqq \{\tau_1; \tau_2 : \tau_1 \in slift_{\mathcal{L}_0}(sto, \varphi)(p_1), \\
&\qquad\quad \tau_2 \in slift_{\mathcal{L}_0}(last(\tau_1))(p_2)\}
\end{aligned}
$$

Symbolic lifting is more complex for loops, as usual in symbolic execution. Possible approaches are *loop unwinding* which generally does not terminate for loops with symbolic guards, *bounded unwinding* with a fixed upper bound on the number of unwinding steps, and *loop invariants*. In the appendix, we provide a more detailed discussion and define symbolic lifting for those cases.

To define $slift_{\mathcal{L}}$, we have to encode schematic statements $\mathtt{P}$. We choose to do this with *abstract stores* $sto_{\mathtt{P}}$ that model state changes caused by schematic statements. We also admit *abstract formulas* $C_{\mathtt{P}}$ to model (unknown) path condition constraints arising from an abstract program $\mathtt{P}$. We define:

$$slift_{\mathcal{L}}(sto, \varphi)(\mathtt{P}) \coloneqq \mathrm{true}^*; (sto \circ sto_{\mathtt{P}}, \varphi \wedge \{sto \circ sto_{\mathtt{P}}\}C_{\mathtt{P}}) \quad \text{for all } \mathtt{P}.$$

The Algorithm for Checking Validity of Trace Modalities. When presented with a problem $[\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]$ and a symbolic state s_0, the algorithm evaluates in three phases whether $\alpha(\mathit{lift}_l(\sigma_0)(\mathcal{C}_l)) \subseteq \alpha(\mathit{lift}_r(\sigma_0)(\mathcal{C}_r))$ holds for all $\sigma_0 \in \mathsf{tval}(s_0)$:

(1) Convert $\mathcal{C}_{l/r}$ to symbolic traces $\tau^s_{l/r}$ using symbolic lifting functions $\mathit{slift}_{l/r}$ (as described above for first-order and LTL formulas, as well as $\mathcal{L}$-programs).
(2) Construct Symbolic Finite Automata (SFAs) $\mathsf{SFA}_{l/r}$ accepting the languages $\mathsf{tval}(\tau^s_{l/r})$, i.e. concrete traces represented by symbolic ones.
(3) Check whether the language accepted by SFA_l is *included* in the language accepted by SFA_r through construction of a *simulation relation.*

Transitions in an SFA are labeled with symbolic states that may represent infinitely many concrete states. For example, a transition labeled with "true" models a transition for *any* concrete state. Formally, we define SFA as:

Definition 6. *A* Symbolic Finite Automaton *is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ of a finite set of states Q, an alphabet $\Sigma \subseteq \mathcal{S}$, a finite transition relation $\delta \subseteq Q \times \text{SymState} \times Q$, an initial state $q_0 \in Q$ and a set of accepting states $F \subseteq Q$. Automaton A* accepts *a concrete trace $\sigma_1\sigma_2\cdots\sigma_n$ if there is a path $q_1 \xrightarrow{s_1} q_2 \xrightarrow{s_2} \cdots q_n \xrightarrow{s_n} q_{n+1}$ in A such that $q_{n+1} \in F$ and for each $i = 1, \ldots, n$ it holds that $\sigma_i \in \mathsf{tval}(s_i)$. The* language $L(A)$ *of an SFA A is the set of all accepted traces.*

The construction of an SFA from symbolic traces (step (2)) is shown in Algo. 2 in the appendix. Lem. 1 states the soundness of the algorithm.

Lemma 1. *Function* createSFA *in Algo. 2 is correct: $L(\text{createSFA}(\tau)) = \mathsf{tval}(\tau)$ holds for all $\tau \in \text{SymTr}$.*

Simulation relations on automata for checking language inclusion [24] and the complexity of crating them [10] have been studied before. Our notion is non-standard, though, since we use symbolic automata with *first-order* transitions. It is not sufficient to relate edges with identical labels or to use existing *propositional* symbolic approaches. Instead, we try to *prove* that an edge in the specification automaton *subsumes* an edge in the implementation automaton. We define symbolic state subsumption as follows.

Definition 7. *Let $s_i = (\mathit{sto}_i, \varphi_i)$, $i = 1, 2$ be symbolic states. Let $\overrightarrow{\mathrm{x}_i}$ be the left-hand sides of sto_i, subst be a substitution of abstract symbols in s_2 not occurring in s_1 with concrete symbols; i.e. uninterpreted constants, function symbols, abstract stores, abstract path conditions are replaced with terms, stores, and formulas. Let P be a fresh predicate with arity $|\overrightarrow{\mathrm{x}_2}|$. Then s_2* subsumes *s_1 iff*

(SUB1) *all variables in $\overrightarrow{\mathrm{x}_2}$ are also contained in $\overrightarrow{\mathrm{x}_1}$ and*
(SUB2) *there is a substitution subst such that:*
$$\models \varphi_1 \wedge \{\mathit{sto}_1\}P(\overrightarrow{\mathrm{x}_2}) \rightarrow \mathit{subst}\big(\{\mathit{sto}_1\}\varphi_2 \wedge \{\mathit{sto}_2\}P(\overrightarrow{\mathrm{x}_2})\big) \ .$$

For states without stores omit the $\{\mathit{sto}_i\}$. In the following, we write $s_1 \sqsubseteq s_2$ if s_2 subsumes s_1, and $s_1 \sqsubseteq_{\mathit{subst}} s_2$ to make the substitution subst for (SUB2) explicit.

Example 3. Let $s_1 = (\mathrm{x} := 17 \,||\, \mathrm{y} := 42 \,||\, \mathrm{z} := 2, \mathrm{true})$. It is subsumed by $s_2 = (\mathrm{x} := c, c \geq 0)$, since (SUB2) holds for $subst := (c \mapsto 17)$:

$$\begin{aligned} & \models \{\mathrm{x} := 17\}P(\mathrm{x}) \rightarrow (c \mapsto 17)(\{\mathrm{x} := 17\}c \geq 0 \wedge \{\mathrm{x} := c\}P(\mathrm{x})) \\ \text{follows from } & \models \{\mathrm{x} := 17\}P(\mathrm{x}) \rightarrow (\{\mathrm{x} := 17\}17 \geq 0 \wedge \{\mathrm{x} := 17\}P(\mathrm{x})) \\ \text{follows from } & \models P(17) \rightarrow (17 \geq 0 \wedge P(17)) \end{aligned}$$

which is true (w.l.o.g. we omit parts of the store of s_1 that do not occur in the target formula). Two more small examples are in Example 8 (appendix). ◊

Lemma 2. *For s_1, $s_2 \in$ SymState, $s_1 \sqsubseteq s_2$ implies $\mathsf{tval}(s_1) \subseteq \mathsf{tval}(s_2)$.*

Subsumption can also be used to establish whether, for a concrete state σ and symbolic state s, it holds that $\sigma \in \mathsf{tval}(s)$ which is needed for the acceptance criterion of SFAs (Def. 6): for the symbolic state $s' = (\overrightarrow{\mathrm{x_s}} := \sigma(\overrightarrow{\mathrm{x_s}}), \mathrm{true})$, where $\overrightarrow{\mathrm{x_s}}$ are the left-hand sides of the store of s, it is sufficient to prove $s' \sqsubseteq s$.

Now we can define the notion of *a Subsumption Simulation Relation (SSR)*, a simulation relation on SFAs based on subsumption.

Definition 8. *A* Subsumption Simulation Relation *between SFAs $A_i = (Q_i, \Sigma, \delta_i, q_0^i, F_i)$, $i = 1, 2$, is any relation $R \subseteq Q_1 \times Q_2$ satisfying*

$$\begin{aligned} &\text{(SR1)} \quad \forall q_1 \in Q_1, q_2 \in Q_2, s, q_1' \in Q_1, \\ &\qquad \big((R(q_1, q_2) \wedge (q_1, s, q_1') \in \delta_1) \implies \\ &\qquad\quad \exists q_2' \in Q_2, s', (R(q_1', q_2') \wedge (q_2, s', q_2') \in \delta_2 \wedge \boxed{s \sqsubseteq s'}\big) \\ &\text{(SR2)} \quad (q_0^1, q_0^2) \in R \end{aligned}$$

Def. 8 equals the "safety simulation relation" of [10], except for the highlighted conjunct $s \sqsubseteq s'$ in (SR1). Constructing an SSR additionally requires to find a suitable substitution and to call a prover showing subsumption. Since SSRs are closed under union and (SR2) is monotone, one can compute R by repeatedly deleting pairs from $Q_1 \times Q_2$ that locally do not satisfy (SR1), and then check whether the result satisfies (SR2) [10]. For each local check, we might have to substitute abstract symbols in the specification automaton. The subsequent lemma, also stated in [10] for their similar notion, establishes a sufficient condition between simulation relations and language inclusion.

Lemma 3. *If there is an SSR between SFAs A_1 and A_2, then $L(A_1) \subseteq L(A_2)$.*

Our top-level algorithm EVALUATE is shown in Algo. 1. In the final step it tries to find an SSR. Only if this was successful, it returns YES. Function FIND-SSR (Algo. 1) starts with an "initial simulation" produced by function INITSIM (Algo. 3, appendix) instead of the cross product to save expensive subsumption checks. During the filtering to derive an SSR, it maintains a *set* of substitutions *substs*, since there might be multiple options. Function SUBSUMPTION$(s, s', substs)$ (Algo. 4, appendix) tries to find compatible *extensions* $subst'$ of the substitutions

Algorithm 1 Evaluation of a Trace Modality Formula using SSRs

function EVALUATE(s_0, $[\mathcal{C}_l \Vdash_\alpha \mathcal{C}_r]$)
  $\tau_l \leftarrow \alpha(slift_l(s_0)(\mathcal{C}_l))$, $\tau_r \leftarrow \alpha(slift_r(s_0)(\mathcal{C}_r))$ ▷ Step (1)
  $A_l \leftarrow$ CREATESFA(τ_l), $A_r \leftarrow$ CREATESFA(τ_r) ▷ Step (2)
  if $(q_0^l, q_0^r) \in$ FINDSSR(A_l, A_r) **then return** YES ▷ Step (3)
  else return `UNKNOWN` **end if**
end function

function FINDSSR($(Q_l, \Sigma, \delta_l, q_0^l, F_l)$, $(Q_r, \Sigma, \delta_r, q_0^r, F_r)$)
  $R \leftarrow$INITSIM(Q_l, Q_r, δ_l, δ_r), *substs* $\leftarrow \{\lambda x.x\}$, *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 $\exists (q_r, s', q_r') \in \delta_r$ s.t. SUBSUMPTION(s, s', *substs*) $\neq \emptyset$) **then**
      substs $\leftarrow$ SUBSUMPTION(s, s', *substs*) ▷ (for all such s')
      else $R \leftarrow R \setminus (q_l, q_r)$, *changed* $\leftarrow$ *true* **end if**
    end for
  end while
  return R
end function

in *substs* by first applying an existing substitution and then finding another one for yet uninstantiated abstract symbols. If there is no such substitution, e.g., since one would have to instantiate the same abstract symbol with different values, the original substitution is dropped. We do not further specify the process of finding substitutions; a naive approach could try to instantiate abstract symbols with all combinations of terms occurring as right-hand sides in the store of s. An example application of Algo. 1 is shown in the appendix. Lem. 4 below states correctness of the FINDSSR. The subsequent main theorem follows from Lems. 1 to 4 and the usage of correct symbolic lifting functions (Def. 5).

Lemma 4. *Function* FINDSSR *(Algo. 1) is correct: For SFAs A_1, A_2, it holds that any SSR R found by* FINDSSR(A_1, A_2) *satisfies (SR1).*

Theorem 1. *Function* EVALUATE *(Algo. 1) is correct: For all $s_0 \in$* SymState, EVALUATE$(s_0, [\mathcal{C}_1 \Vdash_\alpha \mathcal{C}_2])$ = YES *only if, for all $\sigma \in \mathsf{tval}(s_0)$, $\sigma \models [\mathcal{C}_1 \Vdash_\alpha \mathcal{C}_2]$.*

5 Related Work

We compare our work to(1) logics based on traces and (2) approaches unifying program verification techniques. De Giacomo & Vardi [9] propose a Regular Temporal Specification language RE_f that is syntactically similar to our symbolic traces, but ranges over *propositional* formulas while our atoms are first-order symbolic states. They show that RE_f has the same expressiveness as Monadic Second-order Logic (MSO) and is strictly more expressive than LTL on finite

traces. They define Linear-time Dynamic Logic LDL_f, having the same expressivity as RE_f, but allowing logical connectives like negation. Reasoning in LDL_f is also translated to automata. They mention, but do not detail, the possibility to "capture finite executions of programs [...] (in a propositional variant [...])", which is exactly what we do—but not restricted to a propositional variant. In addition, we incorporate *abstract programs* to reason about *classes* of programs. It would be interesting to investigate whether we could use a variant of LDL_f to embed symbolic traces conveniently into logic formulas.

Beckert & Bruns [5] combine dynamic logic and first-order temporal logic to a *Dynamic Trace Logic*. They have a trace-based semantics for a while language and provide a sequent calculus to reason about temporal properties (not preceded by symbolic lifting). The calculus rules depend on the top-level operator of the first-order LTL post condition. This leads quite complex loop invariant rules. Also, the approach is not directly applicable to other verification domains, e.g., relational verification. Our approach is more flexible, because there is no syntactic constraint between the left and right-hand side of the trace modality.

Din et al. [11] propose a trace semantics for the actor-based concurrent language ABS. Traces are "locally abstract, globally concrete": at the local (e.g., method) level, symbolic traces are used. These are primarily a *semantic* notion, facilitating a modular semantics for a concurrent language, while our symbolic traces are *syntactic* entities. The authors briefly sketch a program logic with trace formulas, but leave the notion of trace formulas abstract.

Regarding area (2), Kamburjan [20] proposes the *behavioral modality* aiming to integrate existing analyses and sharing some aspects with the trace modality. It asserts that a statement in a concurrent language meets a behavioral specification consisting of a *type* and a *translation* of the type into an MSO formula. This is the case if that formula holds for all traces generated by the statement. Important differences to our approach include: (a) The behavioral modality *syntactically* integrates analyses on the *same program class*, while the trace modality is mainly a *general semantic framework*, (b) the "translation" of [20] projects to MSO and is thus less expressive than lifting to arbitrary trace sets. The trace modality can also be used to combine verification techniques. Two specifications can semantically be combined by forming the intersection of the trace sets. For reasoning about combinations, we could use product constructions on SFAs.

Some systems do not provide a common semantics for verification domains, but a framework to *implement* different analyses. They usually represent verification problems in an Intermediate Language (IL) and interface to different provers. Boogie [2] and Why3 [6] both are an IL and tool for deductive program verification. They are used as backends by verifiers for languages like C and Java. Our "IL" is the regular symbolic trace language, which, compared to Boogie and WhyML, is less usable for direct programming, more abstract and less expressive (e.g., we cannot directly write loops, but have to use invariants). Yet, the syntactic notion of symbolic traces is closely related to the semantic notion of the trace modality, allowing *formalizing* and *proving* a problem in a closely related

framework. Moreover, the trace modality can easily express other problems than "standard" post condition verification. Our algorithm also interfaces to different provers: Which one to use in the subsumption step is left open.

6 Conclusion and Outlook

We presented the trace modality, a novel formalism for expressing many practical problems of sequential program verification. It relates two elements of the same or different domains, e.g., programs, first-order assertions, or temporal logic formulas. Programs can be abstract and represent classes of concrete programs. We demonstrate the usefulness of the trace modality by providing formalizations of various verification problems: Functional Verification, Information Flow Analysis, Model Checking, Program Synthesis, Compilation, and Program Evolution. Our uniform reasoning system translates programs and formulas to regular symbolic traces and then reduces the problem to the construction of simulation relations between finite automata with symbolic transitions. Similar to the semantics of the trace modality, this approach is parametric in the translation to symbolic traces and the abstraction operator. Although regular symbolic traces have already been proposed before as both a specification mechanism and semantic representation, our work is the first we know of connecting both aspects. This facilitates flexible reasoning about programs and specifications in different combinations: A program can even serve as the specification of a formula.

We hope that our uniform formalization helps to uncover synergy potential between so far separate areas in the field of program verification. Moreover, the practical potential of a system based on symbolic traces supporting different verification techniques, for example, program synthesis and deductive verification, is huge. For instance, after a failed proof attempt of a postcondition, one could try synthesis techniques for stepwise refinement of the postcondition to an abstract program. MC and deductive verification techniques could work hand in hand to treat loops, by unwinding, k-induction, abstract interpretation-based techniques, etc. Finally, the idea of "patch abstraction" for program evolution could help in proof reuse, by applying the patches also to existing proofs.

Apart from investigating these ideas, we plan to implement our reasoning algorithm for symbolic traces and to examine different existing trace languages, like linear-time dynamic logic, which might lead to more intuitive or more expressive representations. Also, we project to extend our framework to non-deterministic, in particular, to concurrent programming languages.

References

1. Ahrendt, W., Beckert, B., et al. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer (2016)
2. Barnett, M., Chang, B.Y.E., et al.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: Intern. Symp. on FMCO. pp. 364–387. Springer (2005)
3. Barthe, G., Crespo, J.M., et al.: Relational Verification Using Product Programs. In: Butler, M.J., Schulte, W. (eds.) Proc. 17th FM. pp. 200–214. Springer (2011)

4. Barthe, G., D'Argenio, P.R., et al.: Secure Information Flow by Self-Composition. In: Proc. CSFW-17. pp. 100–114. IEEE Computer Society (2004)
5. Beckert, B., Bruns, D.: Dynamic Logic with Trace Semantics. In: CADE-24 (2013)
6. Bobot, F., Filliâtre, J.C., et al.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on IVL. pp. 53–64 (2011)
7. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: 4th Symp. of POPL. pp. 238–252. ACM Press (Jan 1977)
8. Darvas, Á., Hähnle, R., et al.: A Theorem Proving Approach to Analysis of Secure Information Flow. In: Proc. 2nd Intern. Conf. on SPC. pp. 193–209 (2005)
9. De Giacomo, G., Vardi, M.Y.: Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In: Proc. 23rd IJCAI. pp. 854–860 (2013)
10. Dill, D.L., Hu, A.J., et al.: Checking for Language Inclusion Using Simulation Preorders. In: CAV '91. pp. 255–265. LNCS, Springer (1991)
11. Din, C.C., Hähnle, R., et al.: Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages. In: TABLEAUX 2017. pp. 22–43 (2017)
12. Garrido, A., Meseguer, J.: Formal Specification and Verification of Java Refactorings. In: Proc. 6th SCAM. pp. 165–174. IEEE Computer Society (2006)
13. Godlin, B., Strichman, O.: Regression Verification: Proving the Equivalence of Similar Programs. Softw. Test., Verif. Reliab. 23(3), 241–258 (2013)
14. Hähnle, R., Heisel, M., et al.: An Interactive Verification System based on Dynamic Logic. In: Siekmann, J.H. (ed.) 8th CADE, pp. 306–315. Springer (1986)
15. Harel, D., Tiuryn, J., et al.: Dynamic Logic. MIT Press (2000)
16. Heisel, M.: Formalizing and Implementing Gries' Program Development Method in Dynamic Logic. Sci. Comput. Program. 18(1), 107–137 (1992)
17. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576–580 (1969)
18. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. SE 23(5) (1997)
19. Jhala, R., Majumdar, R.: Software Model Checking. ACM Comput. Surv. 41(4), 21:1–21:54 (2009)
20. Kamburjan, E.: Behavioral Program Logic. In: Proc. 28th TABLEAUX (2019)
21. Leroy, X.: Formal Verification of a Realistic Compiler. Comm. ACM 52(7) (2009)
22. Monperrus, M.: Automatic Software Repair: A Bibliography. ACM Comput. Surv. 51(1), 17:1–17:24 (2018)
23. Pasareanu, C.S., Visser, W.: Verification of Java Programs Using Symbolic Execution and Invariant Generation. In: Proc. 11th Intern. SPIN Workshop on Model Checking Software. pp. 164–181 (2004)
24. Rauch Henzinger, M., Henzinger, T.A., et al.: Computing Simulations on Finite and Infinite Graphs. In: Proc. 36th Symp. on FoCS. pp. 453–462. IEEE (1995)
25. Reps, T.W., Horwitz, S., et al.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Proc. 22nd POPL. pp. 49–61 (1995)
26. Shankar, N.: Combining Model Checking and Deduction. In: Handbook of Model Checking., pp. 651–684. Springer (2018)
27. Srivastava, S., Gulwani, S., et al.: From Program Verification to Program Synthesis. In: Proc. 37th POPL. pp. 313–326 (2010)
28. Steinhöfel, D., Hähnle, R.: Modular, Correct Compilation with Automatic Soundness Proofs. In: Margaria, T., Steffen, B. (eds.) Proc. 8th ISOLA. LNCS (2018)
29. Visser, W., Havelund, K., et al.: Model Checking Programs. Autom. Softw. Eng. 10(2), 203–232 (2003)
30. Yang, H.: Relational Separation Logic. Theoretical CS 375(1-3), 308–334 (2007)