EPiC Series in Computing
Volume 94, 2023, Pages 73–95
Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Trace-based Deductive Verification*

Richard Bubel[1], Dilian Gurov[2], Reiner Hähnle[1], and Marco Scaletta[1]

[1] Technical University of Darmstadt, Darmstadt, Germany
{bubel,haehnle,scaletta}@cs.tu-darmstadt.de
[2] KTH Royal Institute of Technology, Stockholm, Sweden
dilian@kth.se

Abstract

Contracts specifying a procedure's behavior in terms of pre- and postconditions are essential for scalable software verification, but cannot express any constraints on the events occurring during execution of the procedure. This necessitates to annotate code with intermediate assertions, preventing full specification abstraction.

We propose a logic over symbolic traces able to specify recursive procedures in a modular manner that refers to specified programs only in terms of events. We also provide a deduction system based on symbolic execution and induction that we prove to be sound relative to a trace semantics.

Our work generalizes contract-based to trace-based deductive verification by extending the notion of state-based contracts to trace-based contracts.

1 Introduction

To make deductive verification scale, modular specification and verification is of essence [21]. In imperative programming languages, modularity manifests itself at the granularity of procedure calls. A procedure's behavior is specified in terms of a *contract* [34,35]: a pair of first-order pre- and postconditions. During verification each procedure call in the code is replaced with a check that the contract's precondition holds at this point, the call's effect is approximated by assuming the postcondition. In consequence, verification of the called code is replaced (or approximated) by first-order constraints. The overall verification effort for a program is proportional to its length, instead of the unfolded (and unbounded) call graph. The contract-based approach works for recursive procedures [26,46], it requires an induction principle [6,33], and it achieves *procedure-modular* verification in state-of-art deductive verification systems, such as [1,29,32].

The fundamental limitation of pre-/postcondition contracts, which in the following are called *state-based* contracts, is the inability to specify events happening *during* execution of a procedure. This is not only problematic for specifying concurrent programs, but already an issue for

*Partially funded by DFG project "KeY — A Deductive Software Analysis Tool for the Research Community" (BU 2924/3-1, HA 2617/9-1) and by the Hessian LOEWE initiative within the "Software-Factory 4.0" project and by the ATHENE funded project "Model-centric Deductive Verification of Smart Contracts" and the Swedish Vinnova FFI project "AVerT2: Automated Verification and Testing".

interactive programs or loop specification: For the latter, it is necessary to annotate code with intermediate assertions, which worsens readability and impedes full abstraction from the implementation during specification. Indeed, contracts written in popular specification languages for deductive verification [4,31] cannot be stated independently of the specified code.

The ability to specify recurring and intermediate assertions without referring to the code under verification is a feature of *temporal* logic, as commonly used in model checking [11]. However, even though there exist extensions of temporal logic that permit to record procedure calls and returns [2], these do not approximate procedure calls with contracts and, in consequence, do not allow procedure-modular verification in the same manner as deductive verification. Instead, specification refinement is used to combat state explosion [10] and to permit compositional verification [9,19] in model checking.

In this paper we propose a specification logic that generalizes both, state-based procedure contracts and trace-based modal logics, to a *trace-based contract language* that is sufficiently expressive to model recursive calls (and thus also loops). The central idea is to provide specification elements that permit explicit representation of the control structures embodied in the control flow graph (CFG) of a program. From the CFG point of view two requirements on a trace-based contract language are derived: (i) It must be possible to specify *scopes* relating to procedure calls and other control structures: To demarcate scopes we introduce *events* that are semantically connected with the specified code; (ii) it must be possible to represent cyclic edges in the CFG in a finite manner: this is achieved by a least fixed-point operator. Both aspects are not new, of course, but inspired by ideas first expressed in process logic [23], interval temporal logic [22], and the modal μ-calculus [30].

However, we use the resulting logic of *trace contracts* differently from temporal/modal logic: like in Hoare logic [25,26] and dynamic logic [24,41], we admit explicit *judgments* about a given program: if s is a program and Φ a trace contract, then $s : \Phi$ expresses that any possible execution of s results in one of the traces characterized by Φ. As mentioned above, we incorporate *events* into contracts to enable *full abstraction* from the specified code (cf. [28] who use *actions* to design an abstract semantics). Consequently, our logic generalizes Hoare/dynamic logic from state-based to trace-based contracts via events (which permit scoping) and recursive specifications. This sounds more complex than it is. Still, to keep the presentation intuitive and manageable, it is crucial to choose (i) a suitable deductive framework and (ii) an adequate semantics to demonstrate soundness.

Regarding the first point, we adapt a symbolic execution calculus for deductive verification in dynamic logic [1]: It reduces a program s via symbolic execution to a sequence of elementary symbolic state updates $\mathcal{U}_s$ (plus path conditions), hence, it reduces judgments of the form $s : \Phi$ to $\mathcal{U}_s : \Phi$. Adding events to this formalism is easy. Also, dynamic logic is closed under propositional and first-order operators, which enables formulation of a fixed-point induction rule over the syntactic structure of programs.

Concerning the second issue, an adequate semantics must be *trace-based*, i.e. it provides (a) for a given state σ and program s the trace of s when started in σ and (b) for a given trace formula Φ the set of all traces it characterizes. Our deduction rules for symbolic execution act on programs of the form $s; r$, where s is a single statement and r the trailing statements. In consequence, an adequate semantics evaluates *locally* each different kind of statement s and puts r into a continuation. Such a local, trace-based semantic framework was suggested in [14,15]—here, we specialize it to sequential programs with recursive procedures.

To summarize, the main contributions of this paper are: (1) A trace-based specification language that permits fully abstract, modular specification of recursive procedures; (2) the extension of a symbolic execution calculus for procedure-modular verification by structural

$$\begin{array}{rlrl} P \in \mathit{Prog} ::= & \overline{M}\,\{d\ s\} & M \in \mathit{ProcDecl} ::= & m(x)\ sc \\ d \in \mathit{VarDecl} ::= & \varepsilon \mid x;\ d & sc \in \mathit{Scope} ::= & \{d\ s;\ \mathbf{return}\ e\} \end{array}$$

$$s \in \mathit{Stmt} ::= \mathrm{skip} \mid x = e \mid x = m(e) \mid \mathbf{if}\ e\ \{\ s\ \} \mid s;s \mid \mathbf{while}\ e\ \{\ s\ \}$$

$$\text{Global lookup table: } \mathcal{G} = \{\langle\overline{m(x)\ sc}\rangle \mid m \in \mathrm{procedures}(P)\}$$

Figure 1: Syntax of an imperative language with recursive procedures

induction and trace abstraction; (3) a soundness proof of the deduction rules based on a local trace semantics.

In Sect. 2 we present the local trace semantics, then define syntax and semantics of trace contracts in Sect. 3. The deductive verification system is given in Sect. 4. Related work is in Sect. 5, conclusion with future work in Sect. 6.

2 Local Trace Semantics of Recursive Procedures

We provide an LAGC-style [14, 15] trace semantics for the sequential language with recursive procedures whose grammar is shown in Fig. 1.

The definition of integer and boolean expressions is standard, they are assumed to be well-typed. Scopes of procedure bodies may only write to local integer variables, initialized to zero, i.e. procedure calls have no side effects. We stress that this is not a fundamental limitation: to handle side effects and aliasing [1, 40] is orthogonal to the goals of our paper and this restriction greatly simplifies the technical issues and thus readability.

Example 1. We illustrate the concepts in this paper with the running example in Figure 2. The behavior of procedure m is the identity function for input k, but the control flow is non-trivial, because the result is computed by k many non-tail recursive calls.

Definition 1 (State, Modification)**.** Let *Var* be a set of program variables and *Val* a set of values, with typical elements x and v, respectively. A *state* $\sigma \in \Sigma$ is a partial mapping $\sigma : \mathit{Var} \rightharpoonup \mathit{Val}$ from variables to values. Notation $\sigma[x \mapsto v]$ expresses the *modification* of state σ at x with value v and is defined as $\sigma[x \mapsto v](y) = v$ if $x = y$ and $\sigma[x \mapsto v](y) = \sigma(y)$ otherwise.

We assume a standard evaluation function val_σ for expressions, for example, in a state $\sigma = [x \mapsto 0, y \mapsto 1]$ we have $\mathrm{val}_\sigma(x + y) = \mathrm{val}_\sigma(x) + \mathrm{val}_\sigma(y) = 0 + 1 = 1$.

Definition 2 (Context)**.** A *(call) context* is a pair $\mathit{ctx} = (m, \mathit{id})$, where m is a *procedure name* and id is a *call identifier*. Given a call identifier id we denote with $\mathit{res}_{\mathit{id}}$ the unique name of the return variable associated with the call.

```
m(k) {
  r; // initialized to 0
  if (k != 0){
    r = m(k−1);
    r = r + 1
  };
  return r
}
```

Figure 2: Running Example

We define events sufficient to characterize the scope and call structure of recursive procedures, but one could define further event types, for example, input events or, in a concurrent setting, scheduling events.

Definition 3 (Event Marker)**.** Let m be a procedure name, e a parameter, id a call identifier, and v a return value. Then $\mathsf{call}(m, e, id)$ and $\mathsf{ret}(v)$ are *event markers* associated with a procedure call and a return statement, respectively. We also introduce event markers associated with the start and end of a computation in context ctx, defined as $\mathsf{push}(ctx)$ and $\mathsf{pop}(ctx)$, respectively. We denote with $ev(\overline{e})$ a generic event marker over expressions $\overline{e}$.

Definition 4 (Trace)**.** A *trace* τ is defined by the following rules (where ε denotes the empty trace):

$$\tau ::= \varepsilon \mid \tau \curvearrowright t \qquad t ::= \sigma \mid ev(\overline{e})$$

This definition declares traces as sequences over events and states, but we need to associate an event $ev(\overline{e})$ uniquely with a specific state σ. This is done by inserting[1] event $ev(\overline{e})$ into a trace between two copies of σ: We define the *event trace* $ev_\sigma(\overline{e})$ as $ev_\sigma(\overline{e}) = \langle\sigma\rangle \curvearrowright ev(\mathrm{val}_\sigma(\overline{e})) \curvearrowright \sigma$. Events do not change a state.

Like in process logic [23], sequential composition "r ; s" of statements is semantically modeled as trace composition, where the trace from executing r ends in a state from which the execution trace of s proceeds. Thus the trace of r ends in the same state as where the trace of s begins. This motivates the *semantic chop* "$\underline{**}$" on traces [22, 23, 37] that we often use, instead of the standard concatenation operator "$\cdot$".

Definition 5 (Semantic Chop on Traces)**.** Let τ_1, τ_2 be traces, assume τ_1 is non-empty and finite. The *semantic chop* $\tau_1 \underline{**} \tau_2$ is defined as $\tau_1 \underline{**} \tau_2 = \tau \cdot \tau_2$, where $\tau_1 = \tau \curvearrowright \sigma$, $\tau_2 = \langle\sigma'\rangle \cdot \tau'$, and $\sigma = \sigma'$. When $\sigma \neq \sigma'$ the result is undefined.

Example 2. Let $\tau_1 = \langle\sigma\rangle \curvearrowright \sigma[\mathrm{x} \mapsto 1]$ and $\tau_2 = \langle\sigma[\mathrm{x} \mapsto 1]\rangle \curvearrowright \sigma[\mathrm{x} \mapsto 1, \mathrm{y} \mapsto 2]$, then $\tau_1 \underline{**} \tau_2 = \langle\sigma\rangle \curvearrowright \sigma[\mathrm{x} \mapsto 1] \curvearrowright \sigma[\mathrm{x} \mapsto 1, \mathrm{y} \mapsto 2]$.

Our language is deterministic, so we design local evaluation $\mathrm{val}_\sigma(s)$ of a statement s in state σ to return a single trace: The result of $\mathrm{val}_\sigma(s)$ is of the form $\tau \cdot \mathrm{K}(s')$, where τ is an initial (small-step) trace of s and $\mathrm{K}(s')$ contains the remaining, possibly empty, statement s' yet to be evaluated.

Definition 6 (Continuation Marker)**.** Let s be a program statement, then $\mathrm{K}(s)$ is a *continuation marker*. The empty continuation is denoted with $\mathrm{K}(\emptyset)$ and expresses that nothing remains to be evaluated.

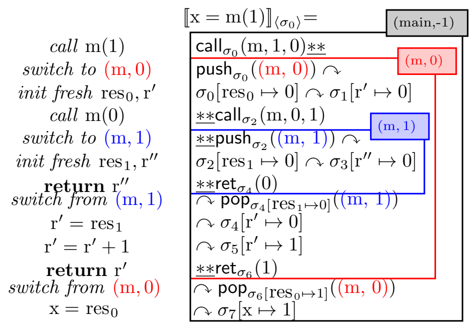
The local evaluation rules defining $\mathrm{val}_\sigma(s)$ are in Fig. 3a. The rules for call and return emit suitable events, the rule for sequential composition assumes that empty leading continuations are discarded, the remaining rules are straightforward.

We define *schematic traces* that allow us to characterize succinctly sets of traces (not) containing certain events via matching. The notation $\overset{\overline{ev}}{\cdots}$ represents the set of all non-empty, finite traces *without* events of type $ev \in \overline{ev}$. Symbol $\cdots$ is shorthand for $\overset{\emptyset}{\cdots}$ and $\underline{Ev}$ includes all event types in Def. 3. With $\tau_1 \overset{\overline{ev}}{\cdots} \tau_2$ we denote the set of all well-defined traces $\tau_1 \underline{**} \tau \underline{**} \tau_2$ such that $\tau \in \overset{\overline{ev}}{\cdots}$. Schematic traces make it easy to retrieve the most recent event and the current call context from a given trace:

[1]Alternatively, one could use state transitions labeled with (possibly empty) events.

$$\mathrm{val}_\sigma(\mathrm{skip}) = \langle\sigma\rangle \cdot \mathrm{K}(\mathbb{0}) \qquad \mathrm{val}_\sigma(\{\ s\ \}) = \mathrm{val}_\sigma(s)$$
$$\mathrm{val}_\sigma(x = e) = \langle\sigma\rangle \curvearrowright \sigma[x \mapsto \mathrm{val}_\sigma(e)] \cdot \mathrm{K}(\mathbb{0})$$
$$\mathrm{val}_\sigma(\textbf{return}\ e) = \mathsf{ret}_\sigma(\mathrm{val}_\sigma(e)) \cdot \mathrm{K}(\mathbb{0})$$
$$\mathrm{val}_\sigma(\textbf{if}\ e\ \{\ s\ \}) = \begin{cases} \langle\sigma\rangle \cdot \mathrm{K}(s), & \text{if } \mathrm{val}_\sigma(e) = \mathtt{tt} \\ \langle\sigma\rangle \cdot \mathrm{K}(\mathbb{0}) & \text{otherwise} \end{cases}$$
$$\mathrm{val}_\sigma(\textbf{while}\ e\ \{\ s\ \}) = \mathrm{val}_\sigma(\textbf{if}\ e\ \{\ s;\ \textbf{while}\ e\ \{\ s\ \}\})$$
$$\mathrm{val}_\sigma(r; s) = \tau \cdot \mathrm{K}(r'; s),$$
$$\text{where } \mathrm{val}_\sigma(r) = \tau \cdot \mathrm{K}(r') \text{ and } \mathbb{0}; s \rightsquigarrow s$$
$$\mathrm{val}_\sigma(\{\ x; d\ s\ \}) = \langle\sigma\rangle \curvearrowright \sigma[x' \mapsto 0] \cdot \mathrm{K}(\{d\ s[x \leftarrow x']\ \})$$
$$\text{with } x' \notin \mathit{dom}(\sigma)$$
$$\mathrm{val}_\sigma(x = m(e)) = \mathsf{call}_\sigma(m, \mathrm{val}_\sigma(e), id) \cdot \mathrm{K}(x = \mathrm{res}_{id})$$
$$\text{where } \mathrm{res}_{id} \notin \mathit{dom}(\sigma)$$

(a) Local Program Semantics



$[\![\mathrm{x} = \mathrm{m}(1)]\!]_{\langle\sigma_0\rangle} =$
(main,-1)
call m(1)
switch to (m, 0)
init fresh $\mathrm{res}_0, \mathrm{r}'$
call m(0)
switch to (m, 1)
init fresh $\mathrm{res}_1, \mathrm{r}''$
return r''
switch from (m, 1)
$\mathrm{r}' = \mathrm{res}_1$
$\mathrm{r}' = \mathrm{r}' + 1$
return r'
switch from (m, 0)
$\mathrm{x} = \mathrm{res}_0$
$\mathsf{call}_{\sigma_0}(\mathrm{m}, 1, 0)\underline{**}$
(m, 0)
$\mathsf{push}_{\sigma_0}((\mathrm{m}, 0)) \curvearrowright$
$\sigma_0[\mathrm{res}_0 \mapsto 0] \curvearrowright \sigma_1[\mathrm{r}' \mapsto 0]$
$\underline{**}\mathsf{call}_{\sigma_2}(\mathrm{m}, 0, 1)$
(m, 1)
$\underline{**}\mathsf{push}_{\sigma_2}((\mathrm{m}, 1)) \curvearrowright$
$\sigma_2[\mathrm{res}_1 \mapsto 0] \curvearrowright \sigma_3[\mathrm{r}'' \mapsto 0]$
$\underline{**}\mathsf{ret}_{\sigma_4}(0)$
$\curvearrowright \mathsf{pop}_{\sigma_4[\mathrm{res}_1 \mapsto 0]}((\mathrm{m}, 1))$
$\curvearrowright \sigma_4[\mathrm{r}' \mapsto 0]$
$\curvearrowright \sigma_5[\mathrm{r}' \mapsto 1]$
$\underline{**}\mathsf{ret}_{\sigma_6}(1)$
$\curvearrowright \mathsf{pop}_{\sigma_6[\mathrm{res}_0 \mapsto 1]}((\mathrm{m}, 0))$
$\curvearrowright \sigma_7[\mathrm{x} \mapsto 1]$

(b) Visualization of the semantics of $\mathrm{x} = \mathrm{m}(1)$ in σ_0

Figure 3: Local Program Semantics and Visualization

Definition 7 (Last Event, Current Context)**.** Let τ be a non-empty trace.

$$\mathit{last}(\tau) = \begin{cases} \mathit{ev}_\sigma(\overline{v}) & \tau \in \cdots \mathit{ev}_\sigma(\overline{v}) \overset{\underline{Ev}}{\cdots} \\ \mathit{NoEvent} & \text{otherwise} \end{cases}$$

$$\mathit{currCtx}(\tau) = \begin{cases} \mathit{ctx} & \tau \in \ \cdots \mathsf{push}_\sigma(\mathit{ctx}) \overset{\mathsf{push,pop}}{\cdots} \\ \mathit{currCtx}(\tau') & \tau \in \tau' \underline{**}\, \mathsf{push}_\sigma(\mathit{ctx}) \cdots \mathsf{pop}_{\sigma'}(\mathit{ctx}) \overset{\mathsf{push,pop}}{\cdots} \\ (\mathrm{main}, -1) & \mathit{otherwise} \end{cases}$$

Since local evaluation of a statement s yields a small step τ of s plus a continuation $\mathrm{K}(s')$, traces of composite statements can be generated by evaluating the continuation and joining the result with τ. This is performed by *trace composition rules* that operate on a configuration of the form $\tau, \mathrm{K}(s)$. The process terminates when all statements are evaluated, i.e. $\mathrm{K}(s) = \mathrm{K}(\mathbb{0})$. In this case, τ is the semantics resulting from evaluation of a program. There are three composition rules. The following rule (PROGRESS) evaluates a statement that has not been directly preceded by a call or a return and extends the current trace accordingly.

$$\frac{\tau \notin \cdots \mathsf{call}_\sigma(\_, \_, \_) \qquad \tau \notin \cdots \mathsf{ret}_\sigma(\_) \qquad \sigma = \mathrm{last}(\tau) \qquad \mathrm{val}_\sigma(s) = \tau' \cdot \mathrm{K}(s')}{\tau, \mathrm{K}(s) \to \tau \underline{**} \tau', \mathrm{K}(s')} \tag{1}$$

Separate rules are needed to handle procedure calls and returns to model the change of call context and the return of the computed result. Right after when a call statement is evaluated, i.e. if τ ends with a call event, the call context is switched to the new context ctx and the body s' of the called procedure is inlined in the following (CALL) rule. It also declares the result variable res_{id}:

$$\frac{\mathit{ctx} = (m, id) \qquad \tau \in \cdots \mathsf{call}_\sigma(m, v, id) \qquad s' = sc[x \leftarrow v],\ \text{where}\, \mathrm{lookup}(m, \mathcal{G}) = m(x)\ sc}{\tau, \mathrm{K}(s) \to \tau \underline{**} \mathsf{push}_\sigma(\mathit{ctx}), \mathrm{K}(\mathrm{res}_{id}; s'; s)} \tag{2}$$

Immediately after evaluating a return statement, the returned value is assigned to the *result variable* associated with the current context and that context is switched back to the old *ctx* retrieved from τ via matching in the (Return) rule:

$$\frac{ctx = currCtx(\tau) = (m, id) \qquad \tau \in \cdots \mathsf{ret}_\sigma(v)}{\tau, \mathrm{K}(s) \to \tau \curvearrowright \sigma[\mathrm{res}_{id} \mapsto v] \underline{**} \mathsf{pop}_{\sigma[\mathrm{res}_{id} \mapsto v]}(ctx), \mathrm{K}(s)} \tag{3}$$

Taken together, the rules (2)–(3) model synchronous procedure calls.

Example 3. We evaluate configuration $\langle\sigma_0\rangle, \mathrm{K}(\mathrm{x} = \mathrm{m}(1))$ for empty σ_0, with m as defined in Example 1. Let $\mathrm{s} = mb[\mathrm{k} \leftarrow 1]$, where mb is the body of m. Applying the progress and call rule we obtain the initial sequence of configurations:

$$\langle\sigma_0\rangle, \mathrm{K}(\mathrm{x} = \mathrm{m}(1)) \to \mathsf{call}_{\sigma_0}(\mathrm{m}, 1, 0), \mathrm{K}(\mathrm{x} = \mathrm{res}_0) \to \mathsf{call}_{\sigma_0}(\mathrm{m}, 1, 0) \underline{**} \mathsf{push}_{\sigma_0}((\mathrm{m}, 0)), \mathrm{K}(\mathrm{res}_0; \mathrm{s}; \mathrm{x} = \mathrm{res}_0)$$

It is straightforward to see that the local evaluation and the composition rules are exhaustive and deterministic, which justifies the following definition:

Definition 8 (Program Trace)**.** Given a program s and a state σ, the *trace* of s (with implicit lookup table) is the maximal sequence obtained by repeated application of composition rules, starting from $\langle\sigma\rangle, \mathrm{K}(s)$. If finite, it has the form $\langle\sigma\rangle, \mathrm{K}(s) \to \cdots \to \tau, \mathrm{K}(\emptyset)$, also written $\langle\sigma\rangle, \mathrm{K}(s) \overset{*}{\to} \tau, \mathrm{K}(\emptyset)$.

Definition 9 (Program Semantics)**.** The *semantics* of a program s is only defined for terminating programs as $[\![s]\!]_\tau = \tau'$ if $\tau, \mathrm{K}(s) \overset{*}{\to} \tau \underline{**} \tau', \mathrm{K}(\emptyset)$.

From this definition and the local semantics of sequential statements it is easy to prove by induction:

Proposition 1. $[\![r; s]\!]_\tau = \tau' \underline{**} [\![s]\!]_{\tau'}$*, with* $\tau' = [\![r]\!]_\tau$*.*

Example 4 (Continuing Example 3)**.** Fig. 3b visualizes the semantics $[\![\mathrm{x} = \mathrm{m}(1)]\!]_{\langle\sigma_0\rangle}$ with the intermediate states $\sigma_1 = \sigma_0[\mathrm{res}_0 \mapsto 0]$, $\sigma_2 = \sigma_1[\mathrm{r}' \mapsto 0]$, $\sigma_3 = \sigma_2[\mathrm{res}_1 \mapsto 0]$, $\sigma_4 = \sigma_3[\mathrm{r}'' \mapsto 0]$, $\sigma_5 = \sigma_4[\mathrm{r}' \mapsto 0]$, $\sigma_6 = \sigma_5[\mathrm{r}' \mapsto 1]$, and $\sigma_7 = \sigma_6[\mathrm{res}_0 \mapsto 1]$.

We only consider traces that are *adequate*, i.e. consistent with local evaluation and composition rules: context switches can only occur in event pairs call-push and ret-pop consistently with the current context, and the freshness of call identifiers has to be preserved. We define Traces as the set of all adequate traces. For details, see Appendix A.

3 A Logic for Trace Contracts

We present a logic for specifying properties over finite program traces. The logic is a temporal μ-calculus [43] with two binary temporal operators, corresponding to concatenation and chop over sets of traces, respectively. We consider the syntax fragment without negation, which guarantees that fixed-point formulas indeed denote fixed points of the corresponding semantic transformers, and with least fixed-point recursion only. Then we show this logic to be suitable for expressing relevant finite-trace properties of recursive programs.

3.1 Syntax

The formulas of our logic are built from a set LVar of first-order ("logical") variables and a set RecVar of recursion variables. The syntax of the logic is defined by the following grammar:

$$\Phi \ ::= \lceil P \rceil \mid X(\bar{t}) \mid \mathit{Ev} \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \cdot \Phi \mid \Phi ** \Phi \mid (\mu X(\overline{y}).\Phi)(\bar{t})$$

where P ranges over first-order predicates, X over recursion variables, $\overline{y}$ over tuples of first-order variables, $\bar{t}$ over tuples of terms over first-order variables, and where in the last clause the arities of $\overline{y}$ and $\bar{t}$ agree. We also admit result variables res_i to appear in trace formulas that contain the return value of the call with identifier i when the call is finished, and 0 when not.

Events Ev have the form $\mathsf{start}(m, e, i)$ or $\mathsf{finish}(m, e, i)$. There are no push or pop events, but the finish events record the context. As stated in the introduction, we aim at procedure-modular specification and verification. Therefore, it is not necessary to record all context switches in a global trace, as long as corresponding calls and returns can be uniquely identified.

Remark 1. Events Ev in the logic are *syntactic* representations of events $\underline{\mathit{Ev}}$ in the local semantics (Sect. 2), thus different entities. This permits to tailor events to the desired degree of abstraction of specifications and the deduction system. For example, if one is only interested in the interface behavior of a program, but not in internal state changes, one might choose abstract events of the form $\mathsf{start}(m)/\mathsf{finish}(m)$, where call identifiers and arguments are dropped.

Example 5. To illustrate our logic, we introduce a formula template (or pattern) that we make extensive use of later. Let $\overline{m}$ be a finite, possibly empty, list of procedure names, and let $\mathit{NoEv}(\overline{m})$ be a predicate that is true for any trace element which is not an event involving any $m \in \overline{m}$. Consider the recursive formula: $\Psi_{\overline{m}} = \mu X.(\mathit{NoEv}(\overline{m}) \vee \mathit{NoEv}(\overline{m}) \cdot X)$.

It recursively constructs a trace formula that holds exactly for finite traces not containing any event involving any $m \in \overline{m}$.

Using $\Psi_{\overline{m}}$, we define the binary operator $\Phi_1 \overset{\overline{m}}{\cdot\cdot} \Phi_2$ as shorthand for $\Phi_1 ** \Psi_{\overline{m}} ** \Phi_2$, expressing the trace property of satisfying Φ_1 initially, satisfying Φ_2 in the end, and not containing any event involving an $m \in \overline{m}$ in between. This is the logic equivalent of the semantic operator $\overset{\overline{ev}}{\cdot\cdot\cdot}$ on traces introduced earlier. In case $\overline{m}$ is empty, we simply write "$\cdot\cdot$".

3.2 Semantics

To define the semantics of our logic, we need a first-order interpretation I for predicate and function symbols, a first-order variable assignment $\beta : \mathsf{LVar} \to \mathsf{D}$, and a recursion variable assignment $\rho : \mathsf{RecVar} \to (\overline{\mathsf{D}} \to 2^{\mathsf{Traces}})$ that assigns each recursion variable to a set of traces. The (finite-trace) semantics $[\![\Phi]\!]_{\mathsf{I},\beta,\rho}$ of formulas as a set of traces is inductively defined in Fig. 4.

By a *trace formula* we mean a formula of our logic that is closed with respect to both first-order and recursion variables. Since the semantics of a trace formula does not depend on any variable assignment, we sometimes use $[\![\Phi]\!]$ to denote $[\![\Phi]\!]_{\mathsf{I},\beta,\rho}$ for arbitrary β and ρ. We also permit a slight extension, where we allow trace formulas to be syntactically closed with respect to first-order operators, as long as fixed-point formulas occur only in positive positions.

3.3 Specifying Procedure Contracts

In general, a (finite-trace) procedure contract captures the sequences of states and events that may be generated as a result of a call to the procedure. Since the procedure calls can be

$$[\![\mathsf{start}(m,e,i)]\!]_{\mathsf{I},\beta,\rho} = \{\mathsf{call}_\sigma(m,\mathrm{val}_\sigma(e),i)\underline{**}\mathsf{push}_\sigma((m,i)) \curvearrowright \sigma[\mathrm{res}_i \mapsto 0] \mid \sigma \in \Sigma\}$$
$$[\![\mathsf{finish}(m,e,i)]\!]_{\mathsf{I},\beta,\rho} = \{\mathsf{ret}_\sigma(\mathrm{val}_\sigma(e)) \cdot \mathsf{pop}_{\sigma[\mathrm{res}_i\mapsto\mathrm{val}_\sigma(e)]}((m,i)) \mid \sigma \in \Sigma\}$$
$$[\![\lceil P\rceil]\!]_{\mathsf{I},\beta,\rho} = \{\langle\sigma\rangle \mid \sigma \in \Sigma \wedge \sigma,\mathsf{I},\beta \models P\} \qquad [\![X(\bar{t})]\!]_{\mathsf{I},\beta,\rho} = \rho(X)([\![\bar{t}]\!]_{\mathsf{I},\beta})$$
$$[\![\Phi_1 \wedge \Phi_2]\!]_{\mathsf{I},\beta,\rho} = [\![\Phi_1]\!]_{\mathsf{I},\beta,\rho} \cap [\![\Phi_2]\!]_{\mathsf{I},\beta,\rho} \qquad [\![\Phi_1 \cdot \Phi_2]\!]_{\mathsf{I},\beta,\rho} = \{\tau_1 \cdot \tau_2 \mid \tau_1 \in [\![\Phi_1]\!]_{\mathsf{I},\beta,\rho} \wedge \tau_2 \in [\![\Phi_2]\!]_{\mathsf{I},\beta,\rho}\}$$
$$[\![\Phi_1 \vee \Phi_2]\!]_{\mathsf{I},\beta,\rho} = [\![\Phi_1]\!]_{\mathsf{I},\beta,\rho} \cup [\![\Phi_2]\!]_{\mathsf{I},\beta,\rho} \qquad [\![\Phi_1 ** \Phi_2]\!]_{\mathsf{I},\beta,\rho} = \{\tau_1 \underline{**}\, \tau_2 \mid \tau_1 \in [\![\Phi_1]\!]_{\mathsf{I},\beta,\rho} \wedge \tau_2 \in [\![\Phi_2]\!]_{\mathsf{I},\beta,\rho}\}$$
$$[\![(\mu X(\bar{y}).\Phi)(\bar{t})]\!]_{\mathsf{I},\beta,\rho} = [\![\mu X(\bar{y}).\Phi]\!]_{\mathsf{I},\beta,\rho}([\![\bar{t}]\!]_{\mathsf{I},\beta}) \qquad [\![\mu X(\bar{y}).\Phi]\!]_{\mathsf{I},\beta,\rho} = \sqcap\{F : \overline{\mathsf{D}} \to 2^{\mathsf{Traces}} \mid \lambda\bar{d}.[\![\Phi]\!]_{\substack{\mathsf{I},\beta[\bar{y}\mapsto\bar{d}],\\ \rho[X\mapsto F]}} \sqsubseteq F\}$$

$\sqsubseteq$ and $\sqcap$ denote point-wise set inclusion and intersection, respectively
$[\![t]\!]_{\mathsf{I},\beta}$ denotes the evaluation of term t under I and β

Figure 4: Semantics of trace formulas

recursive, such contracts need to be stated recursively. The base case(s) of a recursive contract should specify the traces that do not involve any procedure calls. The induction case(s), on the other hand, should specify the remaining traces, and use recursion variables, properly applied to arguments, at the points where calls to procedures are made.

It is not obvious how to define a general pattern for procedure contracts, so we illustrate the idea on a particular class of contracts that generalize Hoare-style, state-based contracts to trace-based ones. Like in Hoare logic, we use state predicates over logical variables to formulate pre- and postconditions expressing the intended relationship between the values of the program variables upon procedure call and return. But in addition, we capture the structure and arguments of recursive procedure calls. We propose pattern H_m below, where m is the name of the specified procedure, n the value of its (sole) formal parameter it is called with, and i the call identifier:

$$\begin{aligned}\mathsf{H}_m =\,& \mu X_m(n,i).\\ &\big(\lceil \mathit{pre}_m^{\mathit{base}}\rceil ** \mathsf{start}(m,n,i) \overset{m}{\cdot\cdot} \mathsf{finish}(m, f_m(n), i) ** \lceil \mathrm{res}_i \doteq f_m(n)\rceil \vee\\ &\ \lceil \mathit{pre}_m^{\mathit{step}}\rceil ** \mathsf{start}(m,n,i) \overset{m}{\cdot\cdot} X_m(\mathit{step}^{-1}(n), i+1) \overset{m}{\cdot\cdot} \mathsf{finish}(m, f_m(n), i) ** \lceil \mathrm{res}_i \doteq f_m(n)\rceil\big)\end{aligned}$$

Both base and inductive cases use state predicates $\mathit{pre}_m^{\mathit{base}/\mathit{step}}$ to establish the precondition. Either uses state predicate $\mathrm{res}_i \doteq f_m(n)$ to specify the return value res_i of m as a function f_m depending on the value n of the formal parameter. The inductive case additionally specifies a recursive call to m. The first argument of the recursion variable is the inverse of the recursive step function, for example, $\mathit{step}^{-1} = n - 1$ when $\mathit{step} = n + 1$.

Example 6. We illustrate pattern H_m to provide a contract for procedure m from Example 1. To specify that the returned value equals the value of the parameter, we define $\mathit{pre}_\mathrm{m}^{\mathit{base}} = \mathrm{n} \doteq 0$, f_m as the identity function, $\mathit{pre}_\mathrm{m}^{\mathit{step}} = \mathrm{n} > 0$, and $\mathit{step}(\mathrm{n}) = \mathrm{n} + 1$. As a result, we obtain contract Φ_m by instantiating the schema H_m:

$$\begin{aligned}\Phi_\mathrm{m} := \mu X_\mathrm{m}(\mathrm{n},i)\big(&\lceil \mathrm{n} \doteq 0\rceil **\mathsf{start}(\mathrm{m},0,i) \overset{\mathrm{m}}{\cdot\cdot} \mathsf{finish}(\mathrm{m},0,i) ** \lceil \mathrm{res}_i \doteq 0\rceil \vee\\ &\lceil \mathrm{n} > 0\rceil **\mathsf{start}(\mathrm{m},\mathrm{n},i) \overset{\mathrm{m}}{\cdot\cdot} X_\mathrm{m}(\mathrm{n}-1, i+1) \overset{\mathrm{m}}{\cdot\cdot} \mathsf{finish}(\mathrm{m},\mathrm{n},i) ** \lceil \mathrm{res}_i \doteq \mathrm{n}\rceil\big)\end{aligned}$$

Contract Φ_m exposes the structure of recursive calls when $\mathrm{m}(n)$ is executed. In contrast, the big-step semantics of state-based contracts can be expressed simply as[2]

$$\mathsf{H}_m \subseteq \lceil \mathit{pre}_m^{\mathit{base}} \vee \mathit{pre}_m^{\mathit{step}}\rceil \cdot\cdot \lceil \mathrm{res}_i \doteq f_m(n)\rceil\ .$$

[2]Formally, the trace formula on the right should be bound by $\mu X(n,i)$, even though there is no occurrence of X. We take the liberty of omitting such trivial variable binders.

In addition to the final state, we can also specify behavior related to intermediate states. The contract in Example 6 can be extended to relate the result of the internal recursive call to the final state by replacing the postcondition of the recursive call with $\lceil \mathrm{res}_i \doteq \mathrm{res}_{i+1} + 1 \rceil$.

Trace formulas (by design) completely abstract away from a specified program, to which they are only connected via events. The set Ev of events can be extended if needed. It is also conceivable to expose part of the program state in traces.

4 A Calculus for Deductive Verification

To verify whether a program fulfills its trace contract, we design a symbolic execution calculus. The most important feature of the calculus is that it permits *procedure-modular* verification: Each procedure m is specified with a trace formula Φ_m, relative to a precondition pre_m. A *trace contract* $\mathbf{C}_m$ for m expresses that any execution of m satisfying pre_m produces one of the traces in the semantics of Φ_m. Crucially, Φ_m may refer to *contracts* of procedures called in m.

Trace contracts are over *judgments* of the form $s : \Phi$, expressing that any run of the procedure body s is a trace in the semantics of Φ. Judgments permit to express correctness of contracts, hence the rules of our calculus decompose judgments via symbolic execution.

To prove correctness of a program statement s with calls to procedures $m \in \overline{m}$ against a trace specification Φ, one then needs to show that (1) the judgment $s : \Phi$ holds and (2) all contracts $\mathbf{C}_m$ for any $m \in \overline{m}$ hold.

To achieve modularity, during symbolic execution each encountered procedure call is approximated by the specification given by its contract. To avoid technical complications, it turns out that it is advisable to disentangle symbolic execution and contract application. Therefore, in a first phase, the calculus infers for a given statement s the symbolic state changes $\mathcal{U}_s$ it produces by applying symbolic execution rules that ignore recursive calls. In a second step, called *abstraction*, we replace each recursive call with the specification provided in its contract and show that this is sufficient to prove $s : \Phi$.

The strategy laid out above defines the structure of the present section: Symbolic state changes are syntactically represented in a notation called *update*, defined Sect. 4.1. Then we formally introduce judgments and give the symbolic execution rules for straight-line programs in Sect. 4.2. Next we formalize contracts and provide a rule for proving their correctness in Sect. 4.3, followed in Sect. 4.4 by the abstraction rule that uses a proven contract to approximate a call. Finally, in Sect. 4.5, we state and prove soundness of the calculus.

4.1 Updates

Updates [1] can be seen as explicit substitutions recording state changes. An *elementary update* assigns an expression e to variable v, denoted by $\{v := e\}$. We also admit *event updates* $\{\mathsf{ev}(\overline{e})\}$ to record that event Ev with parameters $\overline{e}$ occurred (here, ev is start or finish, but this is generalizable). A composite update $\mathcal{U}$ is a (possibly empty) sequence of elementary/event updates u, see the grammar on top of Fig. 5. Updates $\mathcal{U}$ precede statements s and have the meaning that in $\mathcal{U}s$ statement s is evaluated under the state changes embodied by $\mathcal{U}$.

Example 7. An expression such as "$\{\mathsf{start}(\mathrm{m}, 0, i)\}\{\mathrm{r} := 0\}$ **return** r" results after partial symbolic execution of a procedure m, where only the return statement is left to be executed in a state where a local variable r is initialized to 0.

$$\mathcal{U} ::= \epsilon \mid \{v := e\}\mathcal{U} \mid \{\mathsf{ev}(\overline{e})\}\mathcal{U} \quad (\epsilon \text{ empty sequence of updates})$$

$$\begin{aligned}
&\mathrm{val}_\sigma(u\mathcal{U}s) = \tau \cdot \mathrm{K}(\mathcal{U}s),\ \text{ if } \mathrm{val}_\sigma(u) = \tau \cdot \mathrm{K}(\emptyset)\\
&\mathrm{val}_\sigma(\{v := e\}) = \mathrm{val}_\sigma(v = e),\ \text{ with } v \neq \mathrm{res}_i\\
&\mathrm{val}_\sigma(\{\mathrm{res}_i := e\}) = \langle\sigma\rangle \cdot \mathrm{K}(\emptyset)\\
&\mathrm{val}_\sigma(\{v := m(e)\}) = [\![v = m(e)]\!]_{\langle\sigma\rangle} \cdot \mathrm{K}(\emptyset),\ \text{ when } [\![v = m(e)]\!]_{\langle\sigma\rangle} \text{ is defined}\\
&\mathrm{val}_\sigma(\{\mathsf{start}(m, e, i)\}) = \mathsf{call}_\sigma(m, \mathrm{val}_\sigma(e), i)\underline{**}\mathsf{push}_\sigma((m, i)) \curvearrowright \sigma[\mathrm{res}_i \mapsto 0] \cdot \mathrm{K}(\emptyset)\\
&\mathrm{val}_\sigma(\{\mathsf{finish}(m, e, i)\}) = \mathsf{ret}_\sigma(\mathrm{val}_\sigma(e)) \cdot \mathsf{pop}_{\sigma[\mathrm{res}_i \mapsto \mathrm{val}_\sigma(e)]}((m, i)) \cdot \mathrm{K}(\emptyset)
\end{aligned}$$

Figure 5: Update syntax and local evaluation of statements with leading updates

4.1.1 Local Evaluation

We extend the local semantic evaluation rules to programs with leading updates. Fig. 5 contains semantic rules for expressions of the form $u\mathcal{U}s$, where u is either an elementary update or an event update. If $\mathcal{U}$ is empty then after the evaluation of u the rules from Sect. 2 apply. The semantic rules for updates are similar to those for statements. The first rule is similar to the rule for sequential composition. Three rules are needed to evaluate different cases of elementary updates: (a) the case corresponding to plain assignments; (b) if a result variable res_i is assigned, then the update is simply ignored: it is redundant, because its evaluation always follows $\mathsf{finish}(m, e, i)$; (c) the semantics of procedure calls invokes the program semantics. The final two rules in Fig. 5 evaluate event updates occurring in the deduction rules. They represent the start and end of the execution of m, respectively, and generate suitable events.

Example 8. One step of local evaluation of Example 7 in a state σ yields the expression "$\mathsf{call}_\sigma(\mathrm{m}', 0, i) \underline{**} \mathsf{push}_\sigma((\mathrm{m}', i)) \cdot \mathrm{K}(\{\mathrm{r} := 0\}\textbf{return } \mathrm{r})$".

4.1.2 Updates over Expressions

Expressions e are evaluated in the current program state, represented by a preceding composite update $\mathcal{U}$. To evaluate e under $\mathcal{U}$, written $\mathcal{U}(e)$, we apply updates from inner- to outermost. Applying an elementary update $\{v := e'\}$ to an expression e corresponds to syntactic substitution. Events have no effect on the value of expressions. We obtain the following rules:

$$\mathcal{U}'u(e) = \mathcal{U}'(u(e)) \qquad \{v := e'\}(e) = e[v/e']) \qquad \{Ev(\_)\}(e) = \epsilon(e) = e$$

Trace composition is performed by applying rule Progress (1), overloaded to support updates. The semantics of programs with updates is defined in analogy to Def. 9:

Definition 10 (Semantics of Programs with Updates)**.** We define the *semantics* of a (possibly empty) terminating program s (undefined else) with leading updates as

$$[\![\mathcal{U}\, s]\!]_\tau = \tau',\ \text{ if } \tau, \mathrm{K}(\mathcal{U}\, s) \xrightarrow{*} \tau\underline{**}\tau', \mathrm{K}(\emptyset).$$

When symbolically executing the return statement we need to generate a finish event whose context matches the current context of the leading update. We retrieve that call context from the leading update with a helper function:

$$\textsf{Assign}\ \frac{\Gamma \vdash \mathcal{U}\{v := e\}s : \Phi}{\Gamma \vdash \mathcal{U}\, v = e; s : \Phi} \qquad \textsf{VarDecl}\ \frac{\Gamma \vdash \mathcal{U}\{v' := 0\}\{s[v'/v]\} : \Phi \qquad v' \text{ fresh for } s, \Gamma, \Phi}{\Gamma \vdash \mathcal{U}\{v; s\} : \Phi}$$

$$\textsf{Cond}\ \frac{\Gamma, \mathcal{U}(e) \vdash \mathcal{U}s; s' : \Phi \qquad \Gamma, \mathcal{U}(!e) \vdash \mathcal{U}s' : \Phi}{\Gamma \vdash \mathcal{U}\ \textbf{if}\ e\ \{\ s\ \}; s' : \Phi} \qquad \textsf{Prestate}\ \frac{\Gamma \vdash Q \qquad \Gamma \vdash \mathcal{U}s : \Phi}{\Gamma \vdash \mathcal{U}s : \lceil Q \rceil \ ** \Phi}$$

$$\textsf{Return}\ \frac{\text{currCtx}(\mathcal{U}) = (m, i) \qquad \Gamma \vdash \mathcal{U}\{\textsf{finish}(m, e, i)\}\text{res}_i = e : \Phi}{\Gamma \vdash \mathcal{U}\,\textbf{return}\ e : \Phi} \qquad \textsf{Scope}\ \frac{\Gamma \vdash \mathcal{U}s : \Phi}{\Gamma \vdash \mathcal{U}\{s\} : \Phi}$$

Figure 6: Sequent rules for straight-line programs

Definition 11 (Current Context for Updates)**.**

$$currCtx(\mathcal{U}) = \begin{cases} (m, i) & \mathcal{U} = \mathcal{U}'\{\textsf{start}(m, \_, i)\} \\ currCtx(\mathcal{U}') & \mathcal{U} = \mathcal{U}'\{v := e\} \text{ or } \mathcal{U} = \mathcal{U}'\{\textsf{start}(m, \_, i)\}\mathcal{U}''\{\textsf{finish}(m, \_, i)\} \\ (main, -1) & otherwise \end{cases}$$

Example 9. The context in which the return statement of Example 7 executes is: $currCtx(\{\textsf{start}(\mathrm{m}, 0, i)\}\{\mathrm{r}{:=}0\}) = currCtx(\{\textsf{start}(\mathrm{m}, 0, i)\}) = (\mathrm{m}, i)$.

4.2 Calculus for Straight-line Programs

We present a calculus for programs that contain neither loops nor procedure calls. Our deductive proof system is a Gentzen-style sequent calculus, where partial symbolic execution of a program is represented by $\mathcal{U}s$, with $\mathcal{U}$ the executed part and s the remaining part. If we *judge* $\mathcal{U}s$ to conform to a trace specification Φ we write $\mathcal{U}\, s : \Phi$, where the *judgment* $\mathcal{U}\, s : \Phi$ is evaluated in a current state σ, formally:

Definition 12 (Judgment, Assertion, Sequent)**.** A *judgment* has the shape $\mathcal{U}\, s : \Phi$, where $\mathcal{U}$ is an update, s a program statement, and Φ a trace formula. An *assertion* is either a closed first-order predicate P or a judgment. A *sequent* has the shape $\Gamma \vdash \mathcal{U}\, s : \Phi$, where Γ is a set of assertions.

Definition 13 (Semantics of Sequents)**.** Let σ be a state. A first-order predicate P is true in σ if $\sigma \models P$, as usual in first-order logic. A judgment $\mathcal{U}\, s : \Phi$ is true in σ, denoted $\sigma \models \mathcal{U}\, s : \Phi$, when $[\![\mathcal{U}\, s]\!]_{\langle\sigma\rangle}$ is undefined or $[\![\mathcal{U}\, s]\!]_{\langle\sigma\rangle} \in [\![\Phi]\!]$. A sequent $\Gamma \vdash \mathcal{U}\, s : \Phi$ is true in σ if one of the assertions in Γ is not true in σ or $\sigma \models \mathcal{U}\, s : \Phi$. A sequent is *valid* if it is true in all states σ.

The schematic rules of the symbolic execution calculus for straight-line programs are in Fig. 6. For space reasons, we omit the standard rules for Gentzen-style first-order sequent calculi. In addition to the symbolic execution rules, we need a rule for unfolding fixed-point formulas:

$$\textsf{Unfold}\ \frac{\Gamma \vdash \mathcal{U}s : \Phi\big[(\mu X(\overline{y}).\Phi)/X, \overline{t}/\overline{y}\big]}{\Gamma \vdash \mathcal{U}s : (\mu X(\overline{y}).\Phi)(\overline{t})}$$

Example 10. We show in Fig. 7 the symbolic execution phase of the straight-line program

$$\mathrm{r};\ \textbf{if}\ \ (\mathrm{k}{!=}0)\ \{\ \mathrm{r}{=}\mathrm{k}{-}1;\ \mathrm{r}{=}\mathrm{r}{+}1;\ \}\ \textbf{return}\ \mathrm{r}$$

with premise $\mathrm{k} > 0$ and under a similar event update $\{\textsf{start}(\mathrm{m}, \mathrm{k}, \mathrm{i})\}$ as in Example 7. First the local variable r is initialised (VarDecl) generating the update $\{\mathrm{r} := 0\}$. Then, the following

$$\begin{array}{lc}
 & \text{(symbolic execution finished)} \\ \hline
\mathsf{Assign} & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}\{\mathrm{r} := \mathrm{k} - 1\}\{\mathrm{r} := \mathrm{r} + 1\}\{\mathsf{finish}(\mathrm{m},\mathrm{r},\mathrm{i})\}\{\mathrm{res}_{\mathrm{i}} := \mathrm{r}\} : \Phi \\ \hline
\mathsf{Return} & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}\{\mathrm{r} := \mathrm{k}{-}1\}\{\mathrm{r} := \mathrm{r}{+}1\}\{\mathsf{finish}(\mathrm{m},\mathrm{r},\mathrm{i})\}\mathrm{res}_{\mathrm{i}}{=}\mathrm{r} : \Phi \\ \hline
\mathsf{Assign} & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}\{\mathrm{r} := \mathrm{k} - 1\}\{\mathrm{r} := \mathrm{r}{+}1\}\mathbf{return}\ \mathrm{r} : \Phi \\ \hline
\mathsf{Assign} & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}\{\mathrm{r} := \mathrm{k}{-}1\}\mathrm{r}{=}\mathrm{r}{+}1;\ \mathbf{return}\ \mathrm{r} : \Phi \\ \hline
\mathsf{Cond} & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}\mathrm{r}{=}\mathrm{k}{-}1;\ \mathrm{r}{=}\mathrm{r}{+}1;\ \mathbf{return}\ \mathrm{r} : \Phi \\ \hline
\mathsf{VarDecl} & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}\mathbf{if}\ \ (\mathrm{k}!{=}0)\ \{\mathrm{r}{=}\mathrm{k}{-}1;\ \mathrm{r}{=}\mathrm{r}{+}1;\}\ \mathbf{return}\ \mathrm{r} : \Phi \\ \hline
 & \mathrm{k} > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\mathrm{r};\ \ \mathbf{if}\ \ (\mathrm{k}!{=}0)\ \{\mathrm{r}{=}\mathrm{k}{-}1;\ \mathrm{r}{=}\mathrm{r}{+}1;\}\ \mathbf{return}\ \mathrm{r} : \Phi
\end{array}$$

Figure 7: Proof Tree for Example 10

statement is a conditional, so rule Cond is applied. Observe that $\{\mathsf{start}(\mathrm{m},\mathrm{k},\mathrm{i})\}\{\mathrm{r} := 0\}(\mathrm{k}!{=}0)$ evaluates to $\mathrm{k}\ != 0$ and is subsumed by $\mathrm{k} > 0$. We show only the left premise as the right premise is immediately closed due to $\mathrm{k} > 0$. The final sequent is the result of applying rule Assign twice, followed by an application of the Return rule.

4.3 Procedure Contracts

Specifying and verifying trace contracts for each procedure allows us to verify procedure calls in a modular way. First, we show how to specify a procedure m with a contract $\mathbf{C}_m$, then we present the rule ProcedureContract that is used to prove $\mathbf{C}_m$ in our calculus. Let

$$\mathbf{C}_m = \forall n.(pre_m(n) \rightarrow m(n){:}\Phi_m(n)) \tag{4}$$

specify a stand-alone procedure call outside of an assignment. The local evaluation of such a stand-alone call is given as $\mathrm{val}_\sigma(m(e)) = \mathsf{call}_\sigma(m, \mathrm{val}_\sigma(e), id) \cdot \mathrm{K}(\emptyset)$ and the same as for procedure calls with an assignment, except for the empty continuation (cf. Fig. 3a). When proving correctness of trace contracts $\mathbf{C}_m$ of recursive procedures m, one establishes that Φ_m is a *specification invariant* for the implementation of m under the given precondition.

Example 11. A valid contract for procedure m from Example 1 is obtained from $pre_m(k) \equiv k \geq 0$ and $\Phi_m \equiv \exists i.\,(\cdot\cdot\mathsf{start}(\mathrm{m}, 0, i)\cdot\cdot)$, which expresses that when m is started with a non-negative k, then in the resulting trace there is a recursive call with argument 0.

In the special case $\Phi_m(n) \equiv \Phi'_m(n) ** \lceil \mathrm{res} \doteq f_m(n) \rceil$, where $f_m(n)$ is the result of m given input n, equation (4) can be seen as the generalization of the Hoare-style state-based contract $\forall n.(\{pre_m(n)\}m(n)\{\mathrm{res} \doteq f_m(n)\})$, where n is a first-order parameter.

To prove a contract in state-based deductive verification [1,26,27,46], one shows by induction that the inlined body of m respects the contract, under the assumption that when verification of the method body encounters a recursive call to m, one can assume the contract holds already for that call. This yields partial correctness. We generalize this method to *traces* in the following rule for recursive self-calls:

$$\mathsf{ProcCon}\ \frac{\Gamma, pre_m(n'),\ \mathbf{C}_m \vdash \mathrm{inline}(m, n', i') : \Phi_m(n', i')}{\Gamma \vdash \mathbf{C}_m} \qquad \text{(with } i', n' \text{ fresh)}$$

The rule expresses that for any parameter n' and any recursion depth i', the trace specification $\Phi_m(n', i')$ is an invariant for the inlined procedure body, where $\mathbf{C}_m$ can be assumed in recursive calls. To represent inlining succinctly, p being the procedure parameter, we use

$$\mathrm{inline}(m, n, i) = \{\mathsf{start}(m, n, i)\}\, mb[n/p] \ . \tag{5}$$

In general, other procedures might be called in m, so the assumption in rule $\mathsf{ProcCon}$ should be generalized to $\bigwedge_{m \in \mathrm{procedures}(P)} \mathbf{C}_m$.

Example 12. Symbolic execution for procedure m from Example 1 with contract $\mathbf{C}_\mathrm{m} = \forall n, i.(\mathrm{m}(n) : \Phi_\mathrm{m}(n, i) ** \lceil \mathrm{res}_i \doteq n \rceil)$:

$$\textsf{ProcCon}\ \dfrac{\dfrac{\textsf{VarDecl}\ \dfrac{\dfrac{\text{(here symbolic execution of procedure body starts, similar to Example 10)}}{\mathrm{n}' \geq 0,\ \mathbf{C}_\mathrm{m} \vdash \{\mathsf{start}(\mathrm{m}, \mathrm{n}', \mathrm{i}')\}\{\mathrm{r}' := 0\}\mathrm{s}[\mathrm{n}'/\mathrm{k}, \mathrm{r}'/\mathrm{r}] : \Phi_\mathrm{m}(\mathrm{n}', \mathrm{i}')}}{\mathrm{n}' \geq 0,\ \mathbf{C}_\mathrm{m} \vdash \{\mathsf{start}(\mathrm{m}, \mathrm{n}', \mathrm{i}')\}\{\mathrm{r}'; \mathrm{s}[\mathrm{n}'/\mathrm{k}, \mathrm{r}'/\mathrm{r}]\} : \Phi_\mathrm{m}(\mathrm{n}', \mathrm{i}')}}{\mathrm{n}' \geq 0,\ \mathbf{C}_\mathrm{m} \vdash \mathit{inline}(\mathrm{m}, \mathrm{n}', \mathrm{i}') : \Phi_\mathrm{m}(\mathrm{n}', \mathrm{i}')}}{\vdash \mathbf{C}_\mathrm{m}}$$

4.4 Procedure Calls

As Example 12 shows, after applying rule $\mathsf{ProcCon}$, a procedure body can be fully symbolically executed, where recursive calls in assignments are simply handled by the Assign rule. The semantics of elementary updates with a call on the right ensures that this is sound under our assumption that a procedure call has no side effects. In this way, we can retrofit standard, state-based verification into our trace-based framework.[3] Complete symbolic execution of a procedure body with a recursive call to m yields a judgment of the form

$$\mathcal{U}_1\{r := m(e)\}\mathcal{U}_2 : \Phi_1 ** \Phi_m(e, k) ** \Phi_2 \ . \tag{6}$$

The updates are here followed by the "empty" program, indicating that symbolic execution finished. For the trace specification, the above shape is also justified, because typically we have $\Phi_m = \mu X(y).(\Phi_1(y) \vee \cdots \vee \Phi_n(y))$. This specification one strengthens into $\mu X(y).\Phi_i(y)$, where Φ_i is the specification case corresponding to the current symbolic execution path. At this point, we use pattern (6) and the proof of $\mathbf{C}_m$ to justify the *trace abstraction rule*

$$\textsf{TrAbs}\ \dfrac{\Gamma \vdash \mathcal{U}_1 : \Phi_1 \qquad\qquad \Gamma \vdash \mathcal{U}_1(\mathit{pre}_m(e)) \qquad\qquad \mathbf{C}_m \vdash \{v := f_m(\mathcal{U}_1(e))\}\mathcal{U}_2 : \Phi_2}{\Gamma, \mathbf{C}_m \vdash \mathcal{U}_1\{v := m(e)\}\mathcal{U}_2 : \Phi_1 ** \Phi_m(e, k) ** \Phi_2}$$

where $f_m(\cdot)$ is the function computed by m. The trace abstraction rule decomposes the conclusion into three premises. The first premise verifies that the trace represented by $\mathcal{U}_1$ conforms to Φ_1. The second premise guarantees that the precondition of contract $\mathbf{C}_m$ is satisfied. The proof of contract $\mathbf{C}_m$ is now implicitly used to guarantee that the trace of update $\{v := m(e)\}$ conforms to $\Phi_m(e, k)$, up to assignment of the procedure's result to program variable v. The task of the third premise is then to ensure that the trace consisting of the latter assignment followed by update $\mathcal{U}_2$ conforms to Φ_2.

Example 13. Trace abstraction is applied in the continuation of Example 12, after the recursive call in the body was symbolically executed and moved to an update. At this point the goal sequent has the form

$$\mathrm{n}' > 0, \mathbf{C}_\mathrm{m} \vdash \mathcal{U}_1\{\mathrm{r}' := \mathrm{m}(\mathrm{n}' - 1)\}\mathcal{U}_2 : \Phi_1 \overset{\mathrm{m}}{\cdot\cdot} \Phi_\mathrm{m}(\mathrm{n}' - 1, \mathrm{i}' + 1) \overset{\mathrm{m}}{\cdot\cdot} \Phi_2$$

[3]We stress that the absence of side effects is for ease of presentation and not a fundamental limitation of our approach. How to model side effects in symbolic execution is well-known (for example: [1]).

with abbreviations $\mathcal{U}_1 \equiv \{\mathsf{start}(\mathrm{m},\mathrm{n}',\mathrm{i}')\}\{\mathrm{r}' := 0\}$, $\mathcal{U}_2 \equiv \{\mathrm{r}'{:=}\mathrm{r}'{+}1\}\{\mathsf{finish}(\mathrm{m},\mathrm{r}',\mathrm{i}')\}\{\mathrm{res}_{\mathrm{i}'}{:=}\mathrm{r}'\}$, $\Phi_1 \equiv \lceil \mathrm{n}' > 0 \rceil \ast\!\ast \mathsf{start}(\mathrm{m},\mathrm{n}',\mathrm{i}')\overset{\mathrm{m}}{\cdot\cdot}$, $\Phi_\mathrm{m}(\mathrm{n}'{-}1,\mathrm{i}'{+}1) \equiv X_\mathrm{m}(\mathrm{n}'{-}1,\mathrm{i}'{+}1)$ and $\Phi_2 \equiv \overset{\mathrm{m}}{\cdot\cdot}\mathsf{finish}(\mathrm{m},\mathrm{n}',\mathrm{i}') \cdot \lceil \mathrm{res}_{\mathrm{i}'} \doteq \mathrm{n}' \rceil$.

Applying the trace abstraction rule results in the following three sequents:

$$\begin{array}{l}
\mathrm{n}' > 0 \vdash \{\mathsf{start}(\mathrm{m},\mathrm{n}',\mathrm{i}')\}\{\mathrm{r}' := 0\} : \lceil \mathrm{n}' > 0 \rceil \ast\!\ast \mathsf{start}(\mathrm{m},\mathrm{n}',\mathrm{i}')\overset{\mathrm{m}}{\cdot\cdot} \\
\mathrm{n}' > 0 \vdash \mathrm{n}' > 0, \\
\mathrm{res}_{\mathrm{i}'+1} \doteq \mathrm{n}'-1 \vdash \\
\qquad \{\mathrm{r}' := \mathrm{res}_{\mathrm{i}'+1}\}\{\mathrm{r}' := \mathrm{r}' + 1\}\{\mathsf{finish}(\mathrm{m},\mathrm{r}',\mathrm{i}')\}\{\mathrm{res}_{\mathrm{i}'} := \mathrm{r}'\} : \overset{\mathrm{m}}{\cdot\cdot}\mathsf{finish}(\mathrm{m},\mathrm{n}',\mathrm{i}') \cdot \lceil \mathrm{res}_{\mathrm{i}'} \doteq \mathrm{n}' \rceil
\end{array}$$

The first subgoal is provable with rule Prestate. Like the third premise, it additionally requires the rules found in Appendix B; the second premise is trivial.

Remark 2 (Weakening). It is not possible to prove the contract in Example 11 with the rules so far. The reason is that the TrAbs rule expects that the contract of m *exactly* matches Φ_m in the conclusion, but often we only need a trace formula that is *implied* by the specification of m. This motivates the Cons rule, shown in Figure 8.

This rule assumes there is a calculus $\vdash_t$ to prove consequence of trace formulas. This is unsurprising, because deductive verification with state contracts assumes one can weaken pre- and postconditions (which then boils down to first-order consequence). To provide a sound and complete calculus for $\vdash_t$ is out of scope of this paper and deferred to future work.

$$\mathsf{Cons}\ \frac{\Gamma \vdash s : \Phi' \qquad \Phi' \vdash_t \Phi}{\Gamma \vdash s : \Phi}$$

Figure 8: The Cons rule

Remark 3 (Loops). For space reasons, we do not provide rules for dealing with loops. Conceptually, it is well-known that contracts can be used to specify loops, because they can be expressed as tail-recursive procedures. A systematic overview and comparison between invariant-based and contract-based loop specification is in [16]. A suitable adaptation of our contract rules to the case of loops will be the topic of future work.

4.5 Soundness

We conclude the section about our calculus by discussing and proving its correctness. First, we define the notion of soundness for individual rule schemata, then lift the notion to the calculus as a whole.

Definition 14 (Soundness)**.** A rule of the calculus is *sound* if the validity of the conclusion follows from the validity of the premises. A calculus is sound if it can prove only valid statements.

The proofs for the local soundness of the calculus rules make use of some recurring equations. To streamline their presentation, we state them separately in the following proposition:

Proposition 2. *1.* $[\![\mathcal{U}s]\!]_\tau = [\![\mathcal{U}]\!]_\tau \underline{\ast\ast}\, [\![s]\!]_{[\![\mathcal{U}]\!]_\tau}$

2. $[\![\mathcal{U}\mathcal{U}']\!]_\tau = [\![\mathcal{U}]\!]_\tau \underline{\ast\ast}\, [\![\mathcal{U}']\!]_{[\![\mathcal{U}]\!]_\tau}$

3. $[\![\{v := m(e)\}]\!]_\tau = [\![\{v := m(e)\}]\!]_{last(\tau)}$.

We state and prove now that our proof system is *sound*, since every rule of the system is *locally sound*.

Theorem 1 (Calculus soundness)**.** *The presented sequent calculus is sound.*

Proof. The result is a direct consequence of the (local) soundness of each rule, which we show here for selected rules. Recall that a rule is sound if its conclusion is a valid sequent whenever all its premises are. For some rules we show even *reversibility* (also called *backward soundness*), which means that whenever the conclusion of the rule is a valid sequent, the rule can be applied backwards in a way so that all premises are valid.

Rule Assign We prove soundness and reversibility of the rule. Let σ be a state. We have, with $\tau = [\![\mathcal{U}]\!]_{\langle\sigma\rangle}$:

$$\begin{array}{lll}
 & \sigma \models \mathcal{U}\{v := e\}s : \Phi & \\
\Leftrightarrow & [\![\mathcal{U}\{v := e\}s]\!]_{\langle\sigma\rangle} \in [\![\Phi]\!] & \{\text{Def. } 13\} \\
\Leftrightarrow & \tau \mathbin{\underline{**}} [\![\{v := e\}]\!]_\tau \mathbin{\underline{**}} [\![s]\!]_{[\![\{v:=e\}]\!]_\tau} \in [\![\Phi]\!] & \{\text{Prop. } 2\} \\
\Leftrightarrow & \tau \mathbin{\underline{**}} [\![v = e]\!]_\tau \mathbin{\underline{**}} [\![s]\!]_{[\![v=e]\!]_\tau} \in [\![\Phi]\!] & \{\text{Def. } \mathrm{val}_\sigma(\{v := e\})\} \\
\Leftrightarrow & [\![\mathcal{U} v = e; s]\!]_{\langle\sigma\rangle} \in [\![\Phi]\!] & \{\text{Prop. } 1,2\} \\
\Leftrightarrow & \sigma \models \mathcal{U} v = e; s : \Phi & \{\text{Def. } 13\}
\end{array}$$

We therefore have that the sequent $\Gamma \vdash \{v := e\}s : \Phi$ is valid if and only if $\Gamma \vdash v = e; s : \Phi$ is valid.

Rule Cond We prove soundness and reversibility of the rule. Let σ be a state. We have, with $\tau = [\![\mathcal{U}]\!]_{\langle\sigma\rangle}$ and $\sigma' = last(\tau)$:

$$\begin{array}{ll}
\ \sigma \models \mathcal{U}(e) \Rightarrow \sigma \models \mathcal{U}s; s' : \Phi \wedge\ \sigma \models \mathcal{U}(!e) \Rightarrow \sigma \models \mathcal{U}s' : \Phi & \\
\Leftrightarrow \sigma \models \mathcal{U}(e) \Rightarrow [\![\mathcal{U}\, s; s']\!]_{\langle\sigma\rangle} \in [\![\Phi]\!] \wedge\ \sigma \models \mathcal{U}(!e) \Rightarrow [\![\mathcal{U}\, s']\!]_{\langle\sigma\rangle} \in [\![\Phi]\!] & \{\text{Def. } 13\} \\
\Leftrightarrow \sigma \models \mathcal{U}(e) \Rightarrow \tau \mathbin{\underline{**}} [\![s; s']\!]_\tau \in [\![\Phi]\!] \wedge\ \sigma \models \mathcal{U}(!e) \Rightarrow \tau \mathbin{\underline{**}} [\![s']\!]_\tau \in [\![\Phi]\!] & \{\text{Prop. } 2\} \\
\Leftrightarrow \mathrm{val}_{\sigma'}(e) = \mathtt{tt} \Rightarrow \tau \mathbin{\underline{**}} [\![s; s']\!]_\tau \in [\![\Phi]\!] \quad \wedge\ \mathrm{val}_{\sigma'}(e) = \mathtt{ff} \Rightarrow \tau \mathbin{\underline{**}} [\![s']\!]_\tau \in [\![\Phi]\!] & \{\text{Def. } \sigma \models \mathcal{U}(e)\} \\
\Leftrightarrow \tau \mathbin{\underline{**}} [\![\mathbf{if}\ e\ \{\ s\ \}; s']\!]_\tau \in [\![\Phi]\!] & \{\text{Fig. 3a, Def. } 9\} \\
\Leftrightarrow [\![\mathcal{U}\,\mathbf{if}\ e\ \{\ s\ \}; s']\!]_{\langle\sigma\rangle} \in [\![\Phi]\!] & \{\text{Prop. } 2\} \\
\Leftrightarrow \sigma \models \mathcal{U}\,\mathbf{if}\ e\ \{\ s\ \}; s' : \Phi & \{\text{Def. } 13\}
\end{array}$$

Explanation: Given Fig. 3a, Def. 9 we have

$$\tau,\ K(\mathbf{if}\ e\ \{\ s\ \}; s') \xrightarrow{*} \tau,\ K(s; s'),\ \text{if } \mathrm{val}_{last(\tau)}(e) = \mathtt{tt},\ \text{and}$$
$$\tau,\ K(\mathbf{if}\ e\ \{\ s\ \}; s') \xrightarrow{*} \tau,\ K(s'),\ \text{if } \mathrm{val}_{last(\tau)}(e) = \mathtt{ff}$$

Therefore

$$[\![\mathbf{if}\ e\ \{\ s\ \}; s']\!]_\tau = [\![s; s']\!]_\tau \text{ if } \mathrm{val}_{last(\tau)}(e) = \mathtt{tt},\ \text{and}$$
$$[\![\mathbf{if}\ e\ \{\ s\ \}; s']\!]_\tau = [\![s']\!]_\tau \text{ if } \mathrm{val}_{last(\tau)}(e) = \mathtt{ff}$$

We therefore have that the sequent $\Gamma \vdash \mathcal{U}\,\mathbf{if}\,(e)\,s; s' : \Phi$ is valid if and only if the sequents $\Gamma, \mathcal{U}(e) \vdash \mathcal{U}s; s' : \Phi$ and $\Gamma, \mathcal{U}(!e) \vdash \mathcal{U}s' : \Phi$ are valid.

Rule Unfold We prove soundness and reversibility of the rule. By Tarski's fixed-point theorem for complete lattices [45], the semantics $[\![(\mu X(\overline{y}).\Phi)(\overline{t})]\!]_{\mathsf{I},\beta,\rho}$ of a fixed-point predicate $\mu X(\overline{y}).\Phi$ is indeed a fixed point of the trace predicate transformer $\lambda F.\lambda\overline{d}.[\![\Phi]\!]_{\beta[\overline{y}\mapsto\overline{d}],\rho[X\mapsto F]}$. We therefore have the following *fixed-point unfolding* equivalence:

$$(\mu X(\overline{y}).\Phi)(\overline{t}) \;\equiv\; \Phi\big[(\mu X(\overline{y}).\Phi)/X, \overline{t}/\overline{y}\big]$$

where $\Phi_1 \equiv \Phi_2$ is defined to hold whenever $[\![\Phi_1]\!]_{\mathsf{I},\beta,\rho} = [\![\Phi_2]\!]_{\mathsf{I},\beta,\rho}$ for all β and ρ. The soundness and reversibility of the rule is a direct consequence of this equivalence.

Rule ProcedureContract Since the details are somewhat technical, soundness is only sketched here. We follow the approach taken in [46] to prove the soundness of a similar rule, but there in the context of Hoare logic. The essence of the approach is to find a suitable notion of validity of sequents that allows to capture an inductive argument on the recursive depth of procedure calls. In [46], this is achieved by augmenting the notion of sequent validity with an explicit parameter n of that depth, in turn relying on a modified version of the operational semantics that is also parameterised on n as a bound on the maximal recursion depth when going from an initial state to a final one. For our case, we use the same approach, but apply it to traces.

Rule TrAbs We prove soundness of the rule.

$$\mathsf{TrAbs}\ \frac{\Gamma \vdash \mathcal{U}_1 : \Phi_1 \qquad \Gamma \vdash \mathcal{U}_1(pre_m(e)) \qquad \mathbf{C}_m \vdash \{v := f_m(\mathcal{U}_1(e))\}\mathcal{U}_2 : \Phi_2}{\Gamma, \mathbf{C}_m \vdash \mathcal{U}_1\{v := m(e)\}\mathcal{U}_2 : \Phi_1 ** \Phi_m(e,k) ** \Phi_2}$$

where $f_m(\cdot)$ is the function computed by m.

Assuming the premisses (I)–(III) (from left to right) are valid, we have to show that the conclusion is valid. This means that for all states σ

$$\sigma \models (\Gamma \wedge \mathbf{C}_m) \rightarrow \mathcal{U}_1\{v := m(e)\}\mathcal{U}_2 : \Phi_1 ** \Phi_m(e,k) ** \Phi_2$$

holds. We consider only the non-trivial case where $\sigma \models \Gamma \wedge \mathbf{C}_m$ holds. This means we have to prove that

$$[\![\mathcal{U}_1\{v := m(e)\}\mathcal{U}_2]\!]_{\langle\sigma\rangle} \in [\![\Phi_1 ** \Phi_m(e,k) ** \Phi_2]\!]$$

We can decompose the left side as follows:

$$[\![\mathcal{U}_1\{v := m(e)\}\mathcal{U}_2]\!]_{\langle\sigma\rangle} = \underbrace{\tau \cdot [\![v = m(e)]\!]_\tau}_{\tau'} \cdot \tau'',$$
$$\text{with } [\![\mathcal{U}_1]\!]_{\langle\sigma\rangle} = \tau \text{ and } [\![\mathcal{U}_2]\!]_{\tau'} = \tau''.$$

Validity of premise (I) ensures already that $\tau \in [\![\Phi_1]\!]_{\langle\sigma\rangle}$.
For the middle part, we observe that

$$\begin{aligned}[\![v = m(e)]\!]_\tau &= \overbrace{\mathsf{call}_{\mathrm{last}(\tau)}(m,e,k)\underline{**}\mathsf{push}_{\mathrm{last}(\tau)}((m,i))}^{\bar{\tau}}\,\underline{**}[\![mb[p \mapsto e]; v = \mathrm{res}_k]\!]_{\bar{\tau}} \\ &= [\![m(e);]\!]_\tau\underline{**}[\![v = \mathrm{res}_k;]\!]_{\hat{\tau}}\end{aligned}$$

with $\hat{\tau} = [\![m(e);]\!]_\tau$.
By assumption $\sigma \models \mathbf{C}_m$, i.e.,

$$\sigma \models \forall n, i.(pre_m(n) \rightarrow m(n) : \Phi_m(n,i) ** \lceil \mathrm{res}_i \doteq f_m(n)\rceil)$$

and hence,

$$\sigma \models (pre_m(e_1) \to m(e_1) : \Phi_m(e_1, k) **\lceil \mathrm{res}_k \doteq f_m(e_1)\rceil)$$

with $\mathrm{val}_\sigma(e_1) = \mathrm{val}_\sigma(\mathcal{U}_1 e)$ and e_1 fresh rigid constant symbol and i instantiated with k. Validity of premise (II) asserts that

$$\sigma \models \mathcal{U}_1 pre_m(e) \Leftrightarrow \sigma \models pre_m(\mathcal{U}_1 e) \Leftrightarrow \sigma \models pre_m(e_1) \ \ .$$

Consequently (modus ponens),

$$\begin{aligned} &\sigma \models m(e_1) : \Phi_m(e_1, k) **\lceil \mathrm{res}_k \doteq f_m(e_1)\rceil \\ \Leftrightarrow &[\![m(e_1)]\!]_{\langle\sigma\rangle} \in [\![\Phi_m(e_1, k) **\lceil \mathrm{res}_k \doteq f_m(e_1)\rceil]\!]_{\langle\sigma\rangle} \\ \Rightarrow &[\![m(e_1)]\!]_{\langle\sigma\rangle} \in [\![\Phi_m(e_1, k)]\!]_{\langle\sigma\rangle} \end{aligned}$$

Because there are no side effects from procedure calls on the state, we have that if for any two states σ, σ'

$$\mathrm{val}_\sigma(e) = \mathrm{val}_{\sigma'}(e) \quad \text{and} \quad \mathrm{val}_\sigma(k) = \mathrm{val}_{\sigma'}(k)$$

then

$$[\![m(e)]\!]_{\langle\sigma\rangle} = [\![m(e)]\!]_{\langle\sigma'\rangle} \text{ and } [\![\Phi_m(e, k)]\!]_{\langle\sigma\rangle} = [\![\Phi_m(e, k)]\!]_{\langle\sigma'\rangle} \ \ .$$

Thus we can deduce:

$$[\![m(e_1)]\!]_{\langle\sigma\rangle} = [\![m(e)]\!]_{\langle \mathrm{last}(\tau)\rangle} = [\![m(e)]\!]_\tau = \hat{\tau} \ \ ,$$

and further,

$$\hat{\tau} \in [\![\Phi_m(e, k)]\!]_\tau \ \ .$$

In summary, we have now that $\tau \cdot \hat{\tau} \in [\![\Phi_1 **\Phi_m(e, k)]\!]_{\langle\sigma\rangle}$. We have not yet considered the whole trace of the procedure call update. Recall that:

$$[\![v = m(e)]\!]_\tau = [\![m(e); v = \mathrm{res}_k]\!]_\tau = [\![m(e);]\!]_\tau \underline{**} [\![v = \mathrm{res}_k;]\!]_{\hat{\tau}}$$

It remains to show that

$$[\![v = \mathrm{res}_k;]\!]_{\hat{\tau}} \underline{**} [\![\mathcal{U}_2]\!]_{\tau'} \in [\![\Phi_2]\!]_{\hat{\tau}}$$

This is a direct consequence of premise (III), the only critical point being the equality of the value of variable v. This follows from the fact that in the conclusion, v has the value computed by the procedure when called with parameters $\mathcal{U}_1(e)$, which is the same value to which $f_m(\mathcal{U}_1(e))$ evaluates by definition of f_m. □

5 Related Work

We specify symbolic traces, so it is unsurprising to find related work in extensions of LTL model checking and program synthesis. CaReT logic [2] can specify the call structure of programs which is modeled as pushdown automata. It has abstract versions of the temporal next and until operators that jump over balanced calls. The call and event structure is fixed. *Systems of procedural automata* [18] model procedure calls with context-free rules for atomic actions and procedure calls. The goal is to learn automata from observed traces. Temporal Stream Logic [17] features uninterpreted function terms and updates in addition to standard LTL operators, aiming at program synthesis of Büchi stream automata and FPGA programs. Müller-Olm [36] defines a fixed-point logic with chop that is close to a propositional version of our logic, while

Bruse & Lange [7] extend propositional temporal logic with a recursive operator. In each case the setup is finite or propositional, not first-order. The connection to specification of recursive procedures is not made.

Cousot and Cousot [12] investigate abstract intepretation of temporal logics. In particular they apply it to a generalization of the μ-calculus, which provides the possibility to express also reversal and state abstraction modalities (beyond the classical CTL and LTL modalities). They differ from our approach in the following significant ways: (i) their reasoning is based on abstract interpretation and model checking, trading precision for full automation, while our approach allows for high precision, maintains high (but not full) automation; (ii) their trace specification language uses temporal logic to implicitly characterize the traces, while we admit explicit trace specification; most importantly, (iii) our main contribution is modular contract-based specification and verification of programs, which is not addressed by their approach.

Process logic [23] and interval temporal logic [22] feature the chop operator, which was taken up by Nakata & Uustalu [37], who used infinite symbolic traces to characterize non-terminating loops. These were extended to a rich dynamic logic [8] and equipped with events and a local trace semantics [15].

Cyclic proof systems to prove inductive claims, including contracts of recursive procedures, date back to Hoare's axiomatization of recursive procedures [26]. Gurov & Westman [20] provide an abstract framework based on denotational semantics to formally justify (cyclic) procedure-modular verification, while the paper [6] investigates the expressive power of sequent calculi for cyclic and infinite arguments that are often used as a basis in deductive verification. Recursive predicate specifications are standard, for example, in separation logic [39], but not used to specify program traces. Several papers [13, 42] present a first-order μ-calculus, but do not feature explicit programs or events.

Logical frameworks [5,38] feature abstract specifications in terms of typed higher-order logic formulas, but they target functional languages and feature neither events nor traces.

The authors of [47] are concerned with the specification of closures in Rust. One particular problem they encounter is the need to address the evolution of the captured state in a closure. They use history invariants to express invariant relations between call sites of the closure. History invariants are naturally subsumed by our proposed logic and can be embedded as conditions of the trace after repeated calls.

The paper [44] is about specification and reasoning on delegates. For named delegates they universally quantify over Hoare triples used as the delegates' specification. This provides plug-in specifications at call sites, similar to, but more specific than what our approach achieves with recursion variables. Their approach does not allow to talk about traces.

Ancona et al. [3] present a runtime verification approach that features trace-based specification and a trace-based rewrite calculus, implemented as part of runtime monitors (or their synthesis). Their work allows one to specify program traces that consist of events (method invocations and returns, both might carry symbolic data expressions) and they allow recursive specification equations. Our approach differs in the following points: (i) we admit to specify general state properties in the traces; (ii) we use first-order deduction to simplify and weaken properties; (iii) our specification language can define method contracts in terms of pre- and postconditions.

6 Conclusions and Future Work

In this paper we established the fundamental theory of trace-based contracts, generalizing specification and deductive verification with state-based contracts. The ingredients are (i) a fixed-point logic to characterize event structures over recursive procedures; (ii) a uniform, local trace-event semantics for programs, state updates, and the trace logic; (iii) a sound symbolic execution calculus with rules to prove trace contracts and programs with recursive procedures. Programs and trace contracts communicate semantically via a configurable set of events. This permits fully abstract specification of programs and a highly flexible approach to specify concepts like user input, concurrency, etc.

To fully harvest the opportunities arising from trace-based specification and verification is the goal of follow-up papers, where we will look at concurrent programs, explicitly defined loops, and more complex case studies, for example, from protocol verification. This requires to provide a calculus for consequence between trace formulas and to add support for heap data with side effects. The latter is also interesting, because it will permit an experimental comparison of our approach to the more specific setting of [44, 47]. Specifically, paper [47] is interesting, as they keep aliasing manageable by relying on Rust's ownership type system—something we will keep in mind for possible adoption.

Also the trace construction operators of [3] for shuffling (interleaving) of distributed programs, as well as their prefix closure operator might be interesting to look at, but it remains to be seen whether the latter is as useful for static verification than for runtime verification.

On the theoretical side, we would like to characterize the expressiveness of our trace logic and investigate its relation to temporal logic and standard mu-calculus on finite traces. Finally, it would be interesting to prove (relative) completeness.

References

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification—The KeY Book: From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th Intl. Conf., TACAS, Barcelona, Spain*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.

[3] D. Ancona, L. Franceschini, A. Ferrando, and V. Mascardi. RML: theory and practice of a domain specific language for runtime verification. *Sci. Comput. Program.*, 205:102610, 2021.

[4] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification. Technical Report Version 1.17, CEA and INRIA, 2021.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[6] J. Brotherston and A. Simpson. Sequent calculi for induction and infinite descent. *J. Logic and Computation*, 21(6):1177–1216, 2011.

[7] F. Bruse and M. Lange. Temporal logic with recursion. *Information and Computation*, 281:104804, 2021.

[8] R. Bubel, C. C. Din, R. Hähnle, and K. Nakata. A dynamic logic with traces and coinduction. In H. D. Nivelle, editor, *Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Wroclaw, Poland*, volume 9323 of *LNCS*, pages 303–318. Springer, 2015.

[9] A. Cimatti, R. Demasi, and S. Tonetta. Tightening the contract refinements of a system architecture. *Formal Methods in System Design*, 52(1):88–116, 2018.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th Inter-*

national Conference, Chicago/IL, USA, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[12] P. Cousot and R. Cousot. Temporal abstract interpretation. In M. N. Wegman and T. W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 12–25. ACM, 2000.

[13] M. Dam and D. Gurov. μ-calculus with explicit points and approximations. *J. of Logic and Computation*, 12(2):255–269, Apr. 2002.

[14] C. C. Din, R. Hähnle, L. Henrio, E. B. Johnsen, V. K. I. Pun, and S. L. Tapia Tarifa. LAGC semantics of concurrent programming languages, 2022. arXiv preprint 2202.12195.

[15] C. C. Din, R. Hähnle, E. B. Johnsen, V. K. I. Pun, and S. L. Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In C. Nalon and R. Schmidt, editors, *Proc. 26th Intl. Conf. on Automated Reasoning with Tableaux and Related Methods*, volume 10501 of *LNCS*, pages 22–43. Springer, Sept. 2017.

[16] G. Ernst. Loop verification with invariants and contracts. In B. Finkbeiner and T. Wies, editors, *Verification, Model Checking, and Abstract Interpretation: 23rd Intl. Conf., VMCAI, Philadelphia, PA, USA*, volume 13182 of *LNCS*, pages 69–92. Springer, 2022.

[17] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Temporal stream logic: Synthesis beyond the bools. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification: 31st Intl. Conf., CAV, New York City, NY, USA, Part I*, volume 11561 of *LNCS*, pages 609–629. Springer, 2019.

[18] M. Frohme and B. Steffen. Compositional learning of mutually recursive procedural systems. *Intl. J. Software Tools for Technology Transfer*, 23(4):521–543, 2021.

[19] D. Gurov and M. Huisman. Interface abstraction for compositional verification. In B. K. Aichernig and B. Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 414–424. IEEE Computer Society, 2005.

[20] D. Gurov and J. Westman. A Hoare logic contract theory: An exercise in denotational semantics. In P. Müller and I. Schaefer, editors, *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 119–127. Springer, 2018.

[21] R. Hähnle and M. Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In B. Steffen and G. Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 345–373. Springer, Cham, Switzerland, 2019.

[22] J. Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. *Journal of the ACM*, 38(4):935–962, 1991.

[23] D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 129–142. IEEE Computer Society, 1980.

[24] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 583, Oct. 1969.

[26] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.

[27] M. Hofmann. Semantik und Verifikation. Lecture notes, Technical University Darmstadt, Fachbereich Mathematik, 64289 Darmstadt, 1997. In German, `http://www.dcs.ed.ac.uk/home/mxh/teaching/marburg.ps.gz`.

[28] A. Jeffrey and J. Rathke. Java Jr: Fully Abstract Trace Semantics for a Core Java Language. In

S. Sagiv, editor, *Programming Languages and Systems, 14th European Symp. on Programming, ESOP, Edinburgh, UK*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.

[29] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: a software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[30] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[31] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual*, May 2013. Draft revision 2344.

[32] K. R. M. Leino and V. Wüstholz. The Dafny integrated development environment. In C. Dubois, D. Giannakopoulou, and D. Méry, editors, *Proc. 1st Workshop on Formal Integrated Development Environment, F-IDE, Grenoble, France*, volume 149 of *EPTCS*, pages 3–15, 2014.

[33] C. Lidström and D. Gurov. An abstract contract theory for programs with procedures. In E. Guerra and M. Stoelinga, editors, *Fundamental Approaches to Software Engineering: 24th Intl. Conf., FASE, Luxembourg City, Luxembourg*, volume 12649 of *LNCS*, pages 152–171. Springer, 2021.

[34] B. H. Liskov. Modular program construction using abstractions. In D. Bjørner, editor, *Abstract Software Specifications, 1979 Copenhagen Winter School, Proceedings*, volume 86 of *LNCS*, pages 354–389. Springer, 1979.

[35] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

[36] M. Müller-Olm. A modal fixpoint logic with chop. In C. Meinel and S. Tison, editors, *STACS 99, 16th Annual Symp. on Theoretical Aspects of Computer Science, Trier, Germany*, volume 1563 of *LNCS*, pages 510–520. Springer, 1999.

[37] K. Nakata and T. Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. *Logical Methods in Computer Science*, 11(1):1–32, 2015.

[38] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[39] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Long Beach, California, USA*, pages 247–258. ACM, 2005.

[40] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logićal Methods in Computer Science*, 8(3), 2012.

[41] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symp. on Found. of Comp. Sci., Houston, Texas, USA*, pages 109–121. IEEE Computer Society, 1976.

[42] C. Sprenger and M. Dam. On global induction mechanisms in a μ-calculus with explicit approximations. *Theoretical Informatics and Applications*, 37(4):365–391, 2003.

[43] C. Stirling. *Modal and Temporal Logics*, pages 477–563. Oxford University Press, Inc., USA, 1993.

[44] K. Svendsen, L. Birkedal, and M. J. Parkinson. Verifying generics and delegates. In T. D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 175–199. Springer, 2010.

[45] A. Tarski. A lattice-theoretical fixedpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[46] D. von Oheimb. Hoare logic for mutual recursion and local variables. In C. P. Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science, 19th Conf., Chennai, India*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.

[47] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers. Modular specification and verification of closures in rust. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021.

A Trace Adequacy

We only consider traces consistent with local evaluation and composition rules: at most one variable update occurs between two consecutive states and the events in a trace properly record call events, context switches, and return events.

Definition 15 (Trace Adequacy)**.** We say $\tau = \tau_1 \underline{**} \tau_2$ is an *adequate* trace if either $\tau_1 = \tau_2 = \langle\sigma\rangle$ or τ_1 is adequate and exactly one of the following holds:

1. $\tau_2 = \langle\sigma\rangle \curvearrowright \sigma[x \mapsto v]$
2. $\tau_2 = \mathsf{call}_\sigma(\_,\_,id)$, $\tau_1 \notin \overset{\mathsf{push}((\_,id)),\,\mathsf{pop}((\_,id)),\,\mathsf{call}(\_,\_,id)}{\cdots}$ and $\mathit{last}(\tau_1) \notin \{\mathsf{call}, \mathsf{ret}\}$
3. $\tau_2 = \mathsf{ret}_\sigma(\_)$, $\mathit{last}(\tau_1) \notin \{\mathsf{call}, \mathsf{ret}\}$
4. $\tau_2 = \mathsf{push}_\sigma((m, id))$, $\tau_1 \in \cdots \mathsf{call}_\sigma(m,\_,id)$
5. $\tau_2 = \mathsf{pop}_\sigma((m, id))$, $\tau_1 \in \cdots \mathsf{ret}_\sigma(m)$, $\mathit{currCtx}(\tau_1) = (m, id)$.

Traces is the set of all adequate traces.

The second clause expresses that call identifiers are unique as well as one part of the requirement that call/return events are followed immediately by push/pop events. That requirement is ensured together with the remaining clauses.

The events call and ret are always followed by push and pop respectively. Therefore they can be the last event of a trace only if they occur at the end of the trace. This result is formalized in the following lemma.

Lemma 1. *If $\langle\sigma\rangle, K(s) \overset{n}{\to} \tau, K(s')$ and $\mathsf{call} = \mathit{lastEv}(\tau)$, or $\mathsf{ret} = \mathit{lastEv}(\tau)$, then $\tau = \cdots \mathsf{call}_{\mathit{last}(\tau)}$ or $\tau = \cdots \mathsf{ret}_{\mathit{last}(\tau)}$, respectively.*

If $\tau, K(s) \overset{*}{\to} \tau', K(s')$ in exactly n steps then we write $\tau, K(s) \overset{n}{\to} \tau', K(s')$.

Lemma 2. *Given a program s, if $\langle\sigma\rangle, K(s) \overset{n}{\to} \tau, K(s')$ then τ is adequate.*

Proof. We prove the lemma by induction on n, i.e. on the number of applications of composition rules.

Base case (n=1). For non trivial programs, s has the form "x $d; r$", i.e. at least a variable x is declared. Otherwise, assignment and procedures calls cannot occur. Therefore if $n = 1$ then the rule applied is the *progress rule*.

$$\langle\sigma\rangle, K(\mathrm{x}\, d; r) \to \langle\sigma\rangle \curvearrowright \sigma[\mathrm{x} \mapsto 0], K(d; s)$$

where $\langle\sigma\rangle \curvearrowright \sigma[\mathrm{x} \mapsto 0]$ satisfies condition (1) of Def. 15.

Inductive Step. Let's assume as IH that $\langle\sigma\rangle, K(s) \overset{n}{\to} \tau, K(s')$ with τ adequate and $s' \neq \emptyset$. Let also $\sigma' = \mathit{last}(\tau)$. There are three main cases:

- $\tau = \tau' \underline{**} \mathsf{call}_{\sigma'}(m,\_,i)$: condition (4) is satisfied since applying *call rule* we have $\tau, K(s') \to \tau' \underline{**} \mathsf{call}_{\sigma'}(m,\_,i) \underline{**} \mathsf{push}_{\sigma'}((m,i)), K(s'')$
- $\tau = \tau' \underline{**} \mathsf{ret}_{\sigma'}(\_)$: condition (5) is satisfied since applying *return rule* we have

$$\tau, K(s') \to \tau' \underline{**} \mathsf{ret}_{\sigma'}(\_) \underline{**} \mathsf{pop}_{\sigma'}((m, id)), K(s'')$$

where $\mathit{currCtx}(\tau') = (m, id)$ by definition.

- $\tau \neq \tau' \underline{**}\mathsf{call}_{\sigma'}$ and $\tau \neq \tau' \underline{**}\mathsf{ret}_{\sigma'}$: the *progress rule* applies and we have three cases
  - $\langle\sigma\rangle,\, K(s) \overset{n}{\to} \tau,\, K(x := m(\overline{e}); r))$: a call with a fresh call identifier is generated. Since, by Lemma 1 we have that $last(\tau) \notin \{\mathsf{call}, \mathsf{ret}\}$ condition (2) is satisfied.
  - $\langle\sigma\rangle,\, K(s) \overset{n}{\to} \tau,\, K(\mathbf{return}\ e; r))$: a ret is generated. Since, by Lemma 1 we have that $last(\tau) \notin \{\mathsf{call}, \mathsf{ret}\}$ condition (2) is satisfied.
  - Otherwise we have $\tau,\, K(s') \overset{n}{\to} \tau',\, K(s''))$ where either $\tau' = \tau$ adequate by IH, or $\tau' = \tau \curvearrowright \sigma'[x \mapsto v]$, that satisfies condition (1).

Therefore, given

$$\langle\sigma\rangle,\, K(s) \overset{n}{\to} \tau,\, K(s')$$

with τ adequate we have

$$\langle\sigma\rangle,\, K(s) \overset{n+1}{\to} \tau',\, K(s'')$$

where τ' is adequate. □

Theorem 2 (Adequacy of Program Semantics)**.** *The program semantics* $[\![s]\!]_{\langle\sigma\rangle}$ *is an adequate trace for any state* σ *and terminating program* s*.*

Proof. From Lemma 2 it follows that given a program s if $\langle\sigma\rangle,\, K(s) \overset{*}{\to} \tau,\, K(\lozenge)$ then τ is adequate. In other words $[\![\sigma]\!]_s$ is adequate. □

B Selected Update Simplification Rules

We provide some of the needed update simplification rules:

$$\mathsf{elimUpdate_1}\ \frac{\Gamma \vdash \mathcal{U} : \Phi,\ \Delta \qquad \Gamma \vdash \mathcal{U}\{x := e\}\varphi,\ \Delta}{\Gamma \vdash \mathcal{U}\{x := e\} : \Phi \cdot \lceil\varphi\rceil,\ \Delta}$$

$$\mathsf{elimUpdate_2}\ \frac{\Gamma \vdash \mathcal{U} : \Phi,\ \Delta \qquad \Gamma \vdash ((\mathcal{U}e)\ \text{boolOp}\ \ e') \land (\mathcal{U}i) \doteq i',\ \Delta}{\Gamma \vdash \mathcal{U}\{\text{res}_i := e\} : \Phi **\lceil \text{res}_{i'}\ \text{boolOp}\ \ e'\rceil,\ \Delta}$$

$$\mathsf{elimUpdate_3}\ \frac{\begin{array}{c}\Gamma \vdash \mathcal{U} : \Phi,\ \Delta \\ \Gamma \vdash (\mathcal{U}\overline{e}) \doteq \overline{e'},\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}\{ev(\overline{e})\} : \Phi **ev(\overline{e'}),\ \Delta}$$

$$\mathsf{subsumeUpdates_1}\ \frac{\Gamma \vdash \mathcal{U}_1 : \Phi,\ \Delta}{\Gamma \vdash \mathcal{U}_1\mathcal{U}_2 : \Phi \overset{m}{\cdot\cdot},\ \Delta} \quad \text{if } \mathcal{U}_2 \text{ does not contain any update dependent of } m$$

In $\mathsf{elimUpdate_2}$ boolOp is a placeholder for boolean operators, as "$\doteq$" or "$>$".