Derivative-Guided Symbolic Execution

YONGWEI YUAN, Purdue University, USA
ZHE ZHOU, Purdue University, USA
JULIA BELYAKOVA, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

We consider the formulation of a symbolic execution (SE) procedure for functional programs that interact with effectful, opaque libraries. Our procedure allows specifications of libraries and abstract data type (ADT) methods that are expressed in *Linear Temporal Logic over Finite Traces* (LTL_f), interpreting them as *symbolic finite automata* (SFAs) to enable intelligent specification-guided path exploration in this setting. We apply our technique to facilitate the falsification of complex data structure safety properties in terms of effectful operations made by ADT methods on underlying opaque representation type(s). Specifications naturally characterize admissible traces of temporally-ordered events that ADT methods (and the library methods they depend upon) are allowed to perform. We show how to use these specifications to construct feasible symbolic input states for the corresponding methods, as well as how to encode safety properties in terms of this formalism. More importantly, we incorporate the notion of *symbolic derivatives*, a mechanism that allows the SE procedure to intelligently underapproximate the set of precondition states it needs to explore, based on the automata structures latent in the provided specifications and the safety property that is to be falsified. Intuitively, derivatives enable symbolic execution to exploit temporal constraints defined by trace-based specifications to quickly prune unproductive paths and discover feasible error states. Experimental results on a wide-range of challenging ADT implementations demonstrate the effectiveness of our approach.

CCS Concepts: • **Theory of computation** → **Regular languages**; **Automated reasoning**; *Modal and temporal logics*; • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: symbolic execution, regular expression derivatives

ACM Reference Format:
Yongwei Yuan, Zhe Zhou, Julia Belyakova, and Suresh Jagannathan. 2025. Derivative-Guided Symbolic Execution. *Proc. ACM Program. Lang.* 9, POPL, Article 50 (January 2025), 31 pages. https://doi.org/10.1145/3704886

1 Introduction

Symbolic execution [Baldoni et al. 2018; Cadar and Sen 2013] (SE) is a well-studied program analysis technique whose goal is to statically explore a bounded set of (symbolic) program executions in search of one that yields a symbolic state inconsistent with a given safety property. The states generated during the course of these executions consist of a set of path constraints; a violation is identified if the conjunction of these constraints with the negation of the safety property is logically satisfiable. By knowing the prestate under which a method may be invoked, SE can be performed on individual methods in a *compositional* fashion. Oftentimes, however, the program being analyzed interacts with libraries whose implementations are unavailable for analysis. In this case, we can

Authors' Contact Information: Yongwei Yuan, Purdue University, West Lafayette, USA, yuan311@purdue.edu; Zhe Zhou, Purdue University, West Lafayette, USA, zhou956@purdue.edu; Julia Belyakova, Purdue University, West Lafayette, USA, julbinb@gmail.com; Suresh Jagannathan, Purdue University, West Lafayette, USA, suresh@cs.purdue.edu.

ACM 2475-1421/2025/1-ART50
https://doi.org/10.1145/3704886

augment the SE procedure to interpret models [Chipounov et al. 2011] or specifications [Tobin-Hochstadt and Van Horn 2012; Xu et al. 2009] attached to library methods that describe the intended behavior of their implementation in a form suitable for symbolic reasoning.

In this paper, we consider the design of an SE procedure for functional programs that interface with effectful, opaque libraries. Since we cannot express the behavior of library methods directly in terms of how they manipulate their hidden state (since their implementations are opaque), we instead reason about the interaction of clients with these methods in terms of *traces*, sequences of method invocations and return values that constrain the shape of allowed symbolic states that the symbolic interpreter needs to consider. Our primary contribution is a formalization of symbolic execution in this setting that directly leverages the temporal ordering constraints latent in these traces to intelligently guide path exploration.

A particular useful setting in which this style of symbolic reasoning is likely to be effective are abstract data type (ADT) implementations whose specifications and safety properties are often couched in terms of temporal modalities that constrain how datatype instances can be constructed and used. For example, to establish that an implementation of a functional `Set` datatype, implemented using an effectful list representation, correctly respects the semantics of a mathematical set (e.g., $|S \cup \{x\}| = |S|$ if $x \in S$) necessitates showing that any element added to its list representation is different from any *previously* added element. Because the list implementation is potentially effectful, but does not expose the state it manages to its clients, we can only reason about the `Set` ADT methods that use it behaviorally, in terms of how inputs to the list type's setters affect the values returned by its getters that are subsequently consumed.

In our running example, the representation type `List`, defined as a library, may provide a number of operations on a list instance, some of which are pure like `mem` that checks for list membership, and others of which are effectful, such as `append!` that destructively appends its argument to its instance. The `Set` ADT might provide methods like `in` that simply uses the `mem` method from `List` to check if an element is included in a set instance, or `insert` that adds a new element to the set using `append!`. Suppose `insert`'s implementation incorrectly adds a new element by simply invoking `append!`, without first checking if the element is already present. Constructing a set using this implementation would violate our desired safety property, namely that every element in the set is unique. Our goal is to use symbolic execution to identify such errors.

Given the availability of specifications on ADT and representation-type methods, symbolic execution of an ADT then involves: (1) the generation of feasible (aka constructible) precondition states for an ADT method being analyzed in the form of *symbolic traces* of method calls (and return values) on the representation type that is nonetheless consistent with the ADT's specification, and (2) devising an effective search procedure from this precondition state that identifies a feasible execution path, again expressed as a symbolic trace over symbolic invocations of methods on the representation type, whose final state violates the safety property.

In this work, we develop an SE procedure for a class of behavioral specifications that can be concisely expressed in linear temporal logic (LTL_f [Bansal et al. 2023; De Giacomo and Vardi 2013, 2015]); these specifications correspond to symbolic finite automata (SFA [D'Antoni and Veanes 2014, 2017; Veanes 2013]), in which automata transitions represent effectful and opaque operations made by the ADT on its representation type(s). Our SE procedure exploits the latent SFA structure through *symbolic derivatives*. By computing the residual language after consuming a prefix, Brzozowski derivatives simplify membership checking of regular and context-free languages [Might et al. 2011]. In our setting, symbolic derivatives compute the residual specification after observing a sequence of ADT operations, enabling both the extraction of admissible temporally ordered symbolic events (i.e., method invocations and returns expressed in terms of symbolic variants of program variables)

of the ADT's representation type and the prediction of future admissible events by progressively refining the space of safe behaviors.

When equipped with such derivatives, our SE procedure is capable of (1) generating precondition traces whose interpretation yields a prestate consistent with method specifications, (2) correlating pre- and post-invocation events with the safety property, and (3) guiding exploration along paths likely to lead to a falsification of the safety property. By viewing the set of traces prior to and after method invocations from the lens of the safety property we wish to falsify, our SE procedure intelligently performs path exploration. In the case of Set ADT, our SE procedure may, in the presence of a past append! event, actively look for another append! of the identical element, in an attempt to accelerate the falsification of the unique-element property. As a result, we oftentimes observe many orders-of-magnitude improvement in path enumeration times, enabling it to scale favorably with specification complexity.

In summary, this paper makes the following contributions:

(1) We formalize an SE framework suitable for falsifying safety properties of effectful ADT implementations that manage hidden states. Specifications are expressed as LTL_f formulae and capture temporal dependencies over a history of interactions between an ADT implementation and its underlying representation type(s).
(2) We identify the latent SFA structures within these specifications and treat them as executable representations that enable the formalization of an SE procedure in terms of the traces characterized by these automata.
(3) We propose to integrate a notion of symbolic derivatives as part of our SE procedure that intelligently underapproximates trace-based symbolic states and accelerates the search for a falsification witness.
(4) We describe an implementation of these ideas in OCaml and show its effectiveness on a challenging set of data structure programs.

The remainder of the paper is organized as follows. Motivation and informal explanation of our ideas is provided in the next section. Section 3 provides preliminaries and details about LTL_f, SFAs, and derivatives. The syntax of a core language and a naïve (derivative-free) semantics is given in Section 4. The semantics of deriviative-based execution is provided in Section 5. We show how to translate the declarative semantics of derivatives into an efficient algorithm in Section 6. Implementation details and evaluation results are provided in Section 7. Related work and conclusions are given in Section 8 and 9, resp.

2 Motivation

To motivate our ideas, consider the program shown in Fig. 1. The function `remove` is a method in a linked-list ADT that uses two effectful key-value stores as its representation type, one to maintain an ordering relation among nodes in the list (named Nxt), and the other to record the elements associated with these nodes (named Val). The implementation of the store is opaque to the ADT. Given a node `curr` in a linked-list instance containing argument value `v`, `remove` removes `curr` from the list by first initializing its successor field to `null` (given as the shaded statement at Line 18), and then adjusting the link from its predecessor `prev` to point to its successor `next`.

2.1 Specifications

The specification associated with `remove` is expressed as LTL_f [De Giacomo and Vardi 2013][1] formulae given in the comment above its definition. Informally, we can think of such formulae as characterizing a set of admissible *traces*, event sequences defined in terms of method invocations

[1]Linear temporal logic over finite sequences.

```
module type KVStore
  (K: Key) (V: Value) : T =
sig
  (** (k: K.t) → (v: V.t)
      ghost (v': V.t)
      context stored(T, k, v')
      effect ⟨get k v⟩
      ensure v = v' *)
  val get : K.t → V.t

  (** (k:K.t) → (v:V.t) → unit
      effect ⟨put k v⟩ *)
  val put : K.t → V.t → unit
end
module type Node = sig
  type t
  val null : t  ...
end
module type Elem = sig
  type t ...
end
```

```
1 module Nxt = KVStore (Node) (Node)
2 module Val = KVStore (Node) (Elem)
3
4 (** (hd:Node.t) → (v:Elem.t) → Node.t
5     ghost (a: Node.t), (b: Node.t)
6     context stored(Nxt, a, b)
7     effect ⌣⟨Nxt.put a̸ b⟩W⟨Nxt.put a b̸⟩ *)
8 let remove (hd: Node.t) (elem: Elem.t) =
9   if hd = null then hd
10  else if Val.get hd = elem then
11    Nxt.get hd
12  else
13    let rec loop prev =
14       let curr = Nxt.get prev in
15      if curr = null then ()
16      else if Val.get curr = elem then
17        let next = Nxt.get curr in
18        Nxt.put curr null;
19        Nxt.put prev next
20      else loop curr
21    in loop hd; hd
```

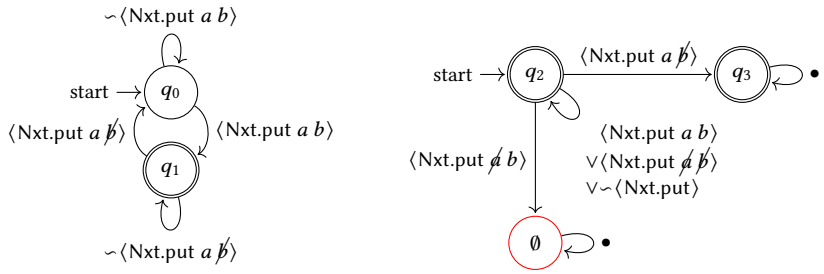
Fig. 1. An implementation of a node `remove` operation in a linked-list ADT using two key-value stores.

and results. The specification has several elements. In the case of `remove`, it introduces ghost variables a and b; these variables represent an arbitrary pair of nodes in the list, constrained by the method's precondition (identified by the keyword `context`) and postcondition (identified by the keyword `effect`). The precondition characterizes all traces that construct a linked-list in terms of the underlying key-value store representation type, identifying an arbitrary consecutive pair of nodes using the introduced ghost variables a and b; it uses the following definition:

$$\text{stored}(\text{Store}, k, v) \doteq \mathbf{F}(\langle \text{Store.put } k\ v\rangle \wedge \mathbf{XG}\backsim\langle \text{Store.put } k\ \_\rangle)$$

The postcondition reflects the actions performed by the method: it specifies that node b can be linked to a predecessor other than a (as denoted by $\not{a}$) only after a is linked to a successor other than b (as denoted by $\not{b}$). Both the pre- and post-condition use LTL modalities. The precondition uses the *finally* modality (**F**) to represent the eventual establishment of a link between a and b in a trace, and next (**X**) and global (**G**) modalities to prevent subsequent actions in the trace from modifying that link; similarly, the postcondition uses the weak-until (**W**) modality to specify a conditional action, namely that b can be linked to a predecessor other than a *only after* a is no longer b's predecessor.

The specification for the key-value stores used by the linked-list ADT is given in the left of Fig. 1. As before, we capture the effectful behavior of these methods using LTL_f specifications. The precondition for **`get`** requires that it be invoked in a state constructed from a sequence of actions that include a **`put`** operation which associates key k to value v'; it leverages the definition of stored defined above, except using the key-value store instance in which the **`get`** is performed. The method's postcondition ensures that this property holds upon return. Additionally, the specification establishes an equality constraint, using the `ensure` annotation, between the value returned (v) and the value previously **`put`** on key k (v'). Note that specifications used in this way constrain the set of precondition states that a symbolic execution engine should consider; in particular, the specification ignores any state that does not contain a binding for k. The specification for **`put`**



(a) Admissible traces prior to `remove`. (b) New actions allowed from `remove`.

Fig. 2. The SFA representation of `remove`'s trace-based specification.

imposes no structure on the store that must hold before it can execute, and only guarantees that the **`put`** action is performed upon return.

Trace Specification as Safety Property. To reiterate, specifications written in this way characterize admissible execution *traces* whose effects determine the context in which a function can execute as well as the behavior manifested by the function upon return from a call, allowing us to reason about the behavior of the ADT without having to expose implementation details about its underlying representation type. Together, a pair of such pre- and post-condition traces captures a safety property against which the function must be checked. In the case of `remove`, an execution under the specified context (precondition trace) that does not satisfy the post-condition trace serves as a witness of a violation of the *predecessor uniqueness* safety property.

SFA Representation of Trace Specifications. The set of traces characterized by LTL_f specifications can be naturally represented by (symbolic) finite automata [Veanes 2013] (SFA) structures whose labels are events representing ADT method invocations and their return values, and whose transitions reflect control dependencies over these actions, defined by modalities used in the specification. Fig. 2 shows how the LTL_f specifications given in Fig. 1 can be represented as SFAs. The automaton in Fig. 2a captures the precondition for `remove`. The start state q_0 admits traces which contain an arbitrary number of **`get`** or **`put`** operations, not involving **`put`** operations with key a or value b; it allows such traces to be augmented with **`put`** operations that store a binding of a to b, thus establishing the required shape of lists to which `remove` can be applied. The store can be subsequently updated with the effects of other **`put`** operations on key a that bind the key to nodes other than b, leading to a transition that exits the accepting state q_1. Traces accepted by the precondition automaton encapsulate program states that can be used as the basis for a successful symbolic execution run of `remove`. The postcondition for `remove` can be represented as the automaton shown in Fig. 2b. Here, the initial state of the postcondition q_2 presumes the precondition, namely ghost nodes a and b such that a is the predecessor of b in the list. A safe implementation of remove is allowed to repeatedly (re)link a to b ($\langle\text{Nxt.put } a\ b\rangle$), link other nodes besides a to b ($\langle\text{Nxt.put } \not{a}\ b\rangle$), or perform **`get`** operations ($\smile\langle\text{Nxt.put}\rangle$). An event that links another node to b *without* first removing the link from a results in a violation of the safety property, however, depicted by the error state with a red circle containing $\emptyset$. State q_3 represents another accepting state corresponding to a linked-list in which node a no longer points to b. The traces admitted by these automata correspond to the *hidden states* constructible by method invocations to the underlying `Nxt` and `Val` store instances.

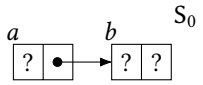
Symbolic Derivatives and SFAs. To reveal the latent SFA representations of trace specifications that qualify over symbolic variables, we propose to compute a variant of Brzozowski derivatives [Brzozowski 1964], dubbed a *symbolic derivative*. Let's revisit the postcondition for `remove` in connection with its SFA representations in Fig. 2b: its LTL$_f$ formula $\backsim\langle \text{Nxt.put } \not{a}\ b\rangle \mathbf{W} \langle \text{Nxt.put } a\ \not{b}\rangle$ admits (1) action $\langle \text{Nxt.put } a\ b\rangle$, $\langle \text{Nxt.put } \not{a}\ \not{b}\rangle$, or $\backsim\langle \text{Nxt.put}\rangle$, followed by traces admissible by the formula itself ($q_2 \rightarrow q_2$), or (2) action $\langle \text{Nxt.put } a\ \not{b}\rangle$ followed by any trace of actions ($q_2 \rightarrow q_3$). Additionally, it does not admit $\langle \text{Nxt.put } \not{a}\ b\rangle$ regardless of the following actions ($q_2 \rightarrow \emptyset$). The symbolic derivatives of the postcondition over $\langle \text{Nxt.put } a\ b\rangle \vee \langle \text{Nxt.put } \not{a}\ \not{b}\rangle \vee \backsim\langle \text{Nxt.put}\rangle$, $\langle \text{Nxt.put } a\ \not{b}\rangle$, and $\langle \text{Nxt.put } \not{a}\ b\rangle$ corresponds to states q_2, q_3, and $\emptyset$ respectively. Such derivatives allow symbolic execution to, as we will discuss in Section 2.3, "execute" trace specifications following their latent SFA representations and make the put operation at Line 19 a witness of the action $\langle \text{Nxt.put } \not{a}\ b\rangle$ that leads to the dead state $\emptyset$.

2.2 Trace-Based Symbolic Execution

Expressing Hidden States as Traces. While our specification language can express a rich set of behaviors that can be exhibited by the ADT, it is not immediately obvious how to incorporate such specifications as part of an efficient symbolic execution procedure. Yet, it is clear that `remove`'s specification naturally entails the uniqueness property that we wish to check, albeit in terms of traces over the representation type's operations, rather than directly in terms of the method's implementation.

Conventionally, symbolic execution explores symbolic states along a program's CFG to find a reachable path that ends at an erroneous state; however, in our setting, the linked list maintained by the key-value stores Nxt and Val does not have an explicit state representation that can be trivially constructed from the program; it is instead manifested by traces extracted from SFAs associated with the ADT's specification. We will show shortly in Section 2.3 how to precisely relate the trace structure described by the specification with the execution paths explored by symbolic execution to manifest these hidden states. Establishing this relation via the use of symbolic derivatives, which will also be described shortly, enables a novel form of property-directed exploration that can be exploited by a symbolic execution procedure to avoid searching over unproductive execution paths.

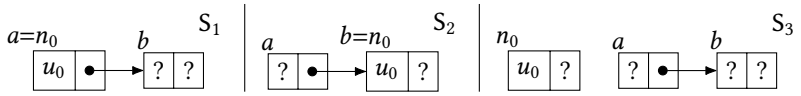
A relatively straightforward approach is to encapsulate symbolic states into a set of traces that records the temporally ordered events produced along the current execution path being explored. Such a set of traces, like our specifications, has a natural representation in SFA. Take the precondition of `remove` as an example; stored(Nxt, a, b) encapsulates a symbolic state S_0 as follows:



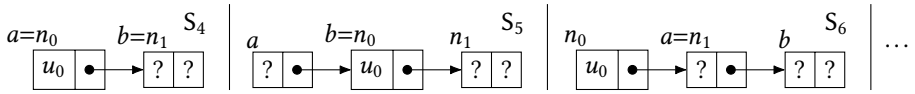
Since a and b are two symbolic variables denoting two arbitrary nodes as long as the former is the predecessor of the latter, the *path condition* is $\top$ initially. Before symbolically executing `remove`, we introduce two additional symbolic variables n_0 and u as the arguments passed to `remove`, respectively denoting the first node in the input linked-list, and the element that the node to be removed stores. Since the precondition of `remove` places no constraint on these variables, the path condition is $\top$ initially. Substituting n_0 for `hd` (and u for `elem`) in the body of `remove`, symbolic execution may choose to enter the first branch at Line 9, augment the path condition with n_0 = `null`, and return n_0. An *empty* symbolic trace (of length zero) is produced from this execution and ghost nodes a and node b are left untouched in the symbolic state. Therefore, the execution complies with the predecessor uniqueness safety property as specified by the method's postcondition.

Conventionally, symbolic states are constrained by path conditions whose satisfiability *directly* manifests the reachability of the respective state. In this setting, however, events executed along the path may also quantify over symbolic variables like path conditions and thus may impose additional constraints on the associated symbolic state. Take S_0 as an example: the SFA representation of stored(Nxt, a, b), as shown in Fig. 2a, admits the traces of executed events allowed upon entering remove. Transitions labeled by $\langle \text{Nxt.put } a \not b \rangle$ admit event Nxt.put key val only if $key = a \wedge val \neq b$ holds. As a result, path condition $\top$ and the SFA *synergistically* encapsulate the symbolic state S_0, reflecting the conditions necessary for the execution paths that lead to remove to be feasible.

Refinement of Trace-Based Symbolic States. We further illustrate this synergy by considering other feasible execution paths in `remove` along which symbolic states are refined. A symbolic execution procedure may also assume that the linked list is not empty, i.e., $n_0 \neq$ `null`, and can thus take the branch at Line 10 where a symbolic method invocation Val.get takes place with n_0 as its argument. Just as path conditions are refined by branching conditions, SFAs are refined by the effectful operations performed along execution paths, consistent with the constraints given by the preconditions of executed operations. Here, the precondition of Val.get, when applied to n_0, is stored(Val, n_0, u_0), where u_0 is a fresh symbolic variable representing the result of the invocation. That is, u_0 is the element stored at n_0. Noticing that n_0 is not even mentioned in S_0, n_0 may be a, b, or some other node not explicitly specified in S_0. The refined SFA representation, as denoted by stored(Nxt, a, b) $\wedge$ stored(Val, n_0, u_0), encapsulates the prestate of the method invocation $\langle u_0 \leftarrow \text{Val.get } n_0 \rangle$, which can be cleanly dissected into the following three states:

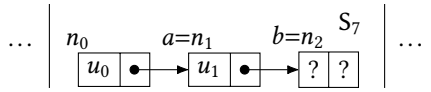


Augmenting the path condition with $u_0 = u$ as we enter Line 11, symbolically executing the invocation of Nxt.get with n_0 first introduces a new symbolic variable n_1 denoting the successor of n_0, and then further refines the traces of executed events to be $((\text{stored}(\text{Nxt}, a, b) \wedge \text{stored}(\text{Val}, n_0, u_0)) \cdot \langle u_0 \leftarrow \text{Val.get } n_0 \rangle) \wedge \text{stored}(\text{Nxt}, n_0, n_1)$, where $\cdot\langle u_0 \leftarrow \text{Val.get } n_0 \rangle$ records the previous method invocation. Within the refined state, similarly, node n_1 could be a, b, some other node, or even n_0. Three ordinary cases are depicted below:

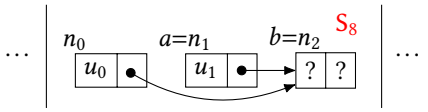


where S_4, S_5 and S_6 refines the above dissected states S_1, S_2, and S_3 respectively. Then, node n_1 is returned at (Line 11). Although the symbolic state is refined, node a and node b are still left intact to comply with the constraints defined by the method's specification. Since the safety property holds over all these possible symbolic states, this execution path is also deemed to satisfy the postcondition.

The violation of the postcondition may happen within the loop (Line 13) if we do not execute the shaded operation at (Line 18). Substituting n_0 for `prev` in the loop body, the invocation of Nxt.get with n_0 then takes us to the same symbolic states, S_4, S_5, and S_6, generated in the earlier explored branch. If we continue the execution from S_6 up until Line 17, one possible symbolic state that holds after the invocation of Val.get and Nxt.get with n_1 would be:



where u_1 and n_2 are fresh symbolic variables that denotes the element stored in n_1 and the successor of n_1 respectively. Without executing Line 18, the invocation of Nxt.put at Line 19 makes node b (i.e., n_2) the successor of both n_0 and a (i.e., n_1):



This symbolic state (S_8), among other possible states that we omit here, happens to manifest a violation of the method's safety property because S_8 with path condition $n_0 \neq \texttt{null} \wedge u_0 \neq u \wedge n_1 \neq \texttt{null} \wedge u_1 = u$ is obviously reachable and it is obvious from the above illustration that any trace of events encapsulated in S_8 is rejected by the method's postcondition.

2.3 Symbolic Execution with Symbolic Derivatives

Efficiency is a serious problem that must be considered by any symbolic execution procedure. Conventional techniques can prune infeasible paths [Baldoni et al. 2018] by leveraging the control structure in programs along with provided preconditions to consider an underapproximation of program behavior that follows a single execution path at a time. For example, if a precondition requires the input list to be not empty, path exploration can ignore paths that contradict this constraint (e.g., Line 9 in Fig. 1). A trace-aware symbolic execution procedure can additionally discover the shape of linked-lists automatically from the SFA structures latent in specifications that induce these traces, and thus can introduce new opportunities for pruning unproductive paths. For example, as currently described, although the precondition of $\langle u_0 \leftarrow \text{Val.get } n_0 \rangle$ refines S_0, the refined symbolic state does not specify whether n_0 is equal to a, b, or some other node. More notably, when the symbolic method invocation $\langle n_1 \leftarrow \text{Nxt.get } n_0 \rangle$ is made, the SFA encapsulating the symbolic state after the invocation includes contradicting paths not depicted among S_4, S_5, and S_6, in which n_0 can be both equal to and not equal to a. Explicitly leveraging the control structure latent in SFAs would enable us to correlate traces (and the hidden states they induce) to specific program paths, exposing new pruning opportunities that would otherwise not be possible. We propose to compute *symbolic derivatives* over the specifications to explore and exploit the latent SFA structures within the trace-based specifications.

Derivative-Guided Path Exploration. Our symbolic execution procedure employs symbolic derivatives to explore the SFA structures latent in the specifications and intelligently enumerate admissible traces that encapsulate prestates along execution paths, lowering the cost of path feasibility check and enabling effective path pruning. In our running example, the initial symbolic state S_0 of remove requires that some node a is the predecessor of some node b but does not specify when b is made the successor of a via Nxt.put. Recall the precondition automaton (Fig. 2), which denotes the traces encapsulating S_0: the relevant Nxt.put operation may be performed ($q_0 \rightarrow q_1$) after some indefinite number of irrelevant actions ($q_0 \rightarrow q_0$), or after a previously executed $\langle \text{Nxt.put } a\ b \rangle$ is invalidated ($q_1 \rightarrow q_0$). A symbolic derivative computation helps explore this automaton structure by sampling paths from the start state q_0 to the accepting state q_1. In the case of `remove`, its behavior happens to be invariant to when the call $\langle \text{Nxt.put } a\ b \rangle$ is performed. Therefore, in order to reach the erroneous symbolic state S_8, it is sufficient to begin the symbolic execution of `remove` with a precondition trace that consists of $\langle \text{Nxt.put } a\ b \rangle$ followed by actions that do not invalidate this operation.

Our symbolic execution procedure further exploits the latent SFA structures to make informed decisions in choosing the precondition trace that favors the efficient exploration of feasible execution paths. To minimize the complexity of reasoning about the behavior of method invocations performed, one straightforward strategy is to choose the "simplest" symbolic state based on the length of the corresponding trace induced from the precondition automaton. For example, `remove`

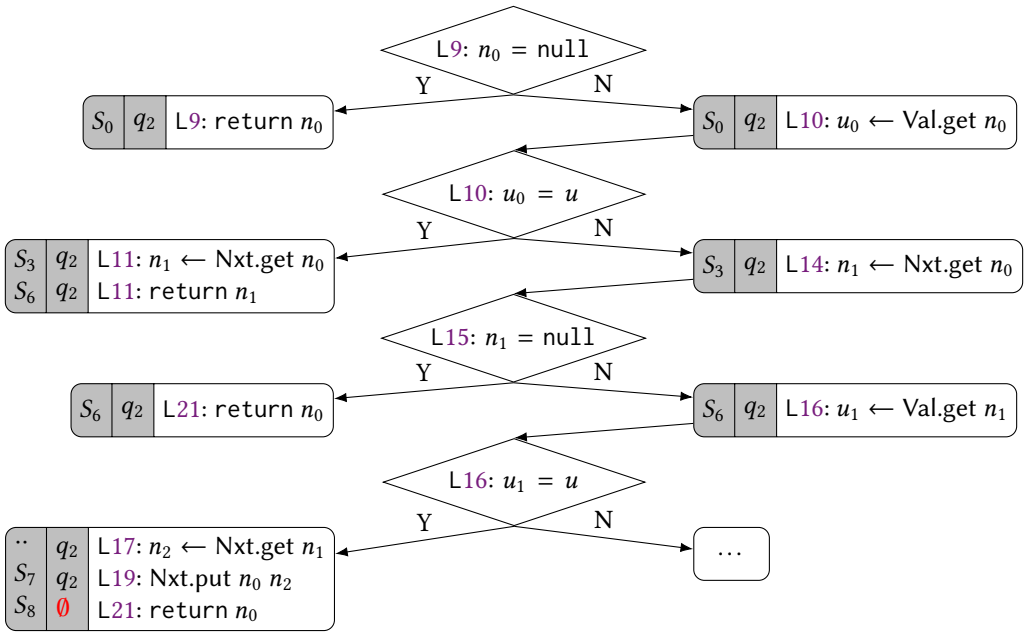


Fig. 3. Derivative-guided symbolic execution of remove.

may be invoked under a state encapsulated by the singleton trace $\langle \text{Nxt.put } a\ b \rangle$. This trace, however, cannot be refined to admit a Val.put event, which is required by the invocation of Val.get at Line 10 in Fig. 1, thus failing to reach the error state S_8. As we proceed to consider longer precondition traces, the simplicity criteria soon becomes insufficient to distinguish between precondition traces. Here are two traces of four symbolic events that can be induced from the precondition automaton and thus equally encapsulate a valid prestate of remove:

$$\langle \text{Nxt.put } a\ b \rangle (\smile \langle \text{Nxt.put } a\ \not{b} \rangle)^3 \quad \text{(repeated 3 times)}$$

$$\langle \text{Nxt.put } a\ b \rangle \langle \text{Nxt.put } a\ \not{b} \rangle \langle \text{Nxt.put } a\ b \rangle (\smile \langle \text{Nxt.put } a\ \not{b} \rangle)$$

We know from before that the erroneous execution path leading to S_8 requires at least two Val.put events but the later trace can only admit one Val.put event. In this case, symbolic derivatives guide SE to first consider the former trace when executing remove. This behavior arises from symbolic derivatives' tendency to maximize "progress" when inducing traces from the precondition automaton. As symbolic derivatives facilitate the exploration within the latent SFA structures of precondition automata, this tendency manifests in several ways: (1) consistently select the states closer to the accepting state; (2) avoid unnecessarily transiting back to non-accepting states, and; (3) steer clear of generating the stagnation pattern of "setting", "unsetting", and "resetting". In the case of the precondition automaton shown in Fig. 2a, the execution tends to move from q_0 to q_1 via $\langle \text{Nxt.put } a\ b \rangle$ and stays at q_1. Intuitively, the former trace induced in the described fashion encapsulates a relatively more permissive state that potentially leads to more interesting feasible paths being explored, including the one that leads to the error state S_8.

Derivative-Guided Falsification. In addition to intelligently enumerating precondition traces, symbolic derivatives can guide the symbolic execution of remove itself, by again exploiting the latent SFA structure of the specification. Specifically, they allow our symbolic execution to relate method invocations in remove to the transitions in the postcondition automaton (Fig. 2b). Consider

the symbolic execution tree of `remove` in terms of operations over symbolic variables, as depicted in Fig. 3. Each program point is associated with a state in the postcondition automaton that effectively determines the set of future traces (i.e., sequences of future actions) that are (un)safe to explore. The symbolic execution of `remove` begins with the initial state q_2 of the postcondition automaton because it admits all traces of actions that `remove` is safe to perform. As `remove` traverses the input linked list via repetitive get invocations before unsafely invoking Nxt.put, symbolic derivatives intelligently determine that the future safe traces after those get invocations are also represented by q_2. This is because $q_2 \longrightarrow q_2$ is the only outgoing transition from q_2 that is compatible with get actions. Recall the frontier state S_7 before the unsafe invocation of Nxt.put from before: the past traces of actions already determine $n_0 \neq a$ and $n_2 = b$. Then our SE procedure can indeed determine that $\langle \text{Nxt.put } n_0\ n_2 \rangle$ is unsafe because it is compatible with the transition $q_2 \longrightarrow \emptyset$, which happens to be the only compatible outgoing transition from q_2. Note that even if `remove` continues traversing the linked list after removing the first found element via a recursive call to `loop`, the symbolic derivatives guide SE to avoid unprofitably unrolling the `loop` because any future action is unsafe. In contrast, the naïve trace-based SE described in Section 2.2 would wastefully relate each explored execution path of `remove` with traces in the postcondition automaton.

To conclude this section, the main contribution of this paper is a new symbolic execution procedure that computes such symbolic derivatives as symbolic execution proceeds to maintain a trace of symbolic events that witness the current execution path and its relationship with the safety property, in an attempt to accelerate the search for a feasible execution that violates the property.

3 Preliminaries

The SFA representations of specifications expressed in LTL_f [De Giacomo and Vardi 2013] facilitate the discussion in Section 2. To properly formulate the relationship between traces admissible by LTL_f formulae and SFAs (see Section 5), however, we need to introduce regular expressions. The language of regular expressions (RE) is strictly more expressive for representing traces than LTL_f and enjoys an important closure property under the classic derivative computation [Brzozowski 1964]. In this section, we present symbolic regular expressions (SREs) whose atoms are predicates, show how to express common temporal modalities from LTL_f in terms of RE operations, and relate classic derivatives with states in finite state automata.

Effective Boolean Algebras. Tuple $(\Sigma, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ defines *Effective Boolean Algebra* (EBA [Veanes 2013]) where Σ is a set of domain elements and Ψ is a set of *predicates*, closed under the Boolean connectives with $\bot, \top \in \Psi$. The denotation of $\phi, \psi \in \Psi$ are provided by $[\![\_]\!]{:}\Psi \longrightarrow 2^\Sigma$ where

$$[\![\bot]\!] = \{\} \qquad [\![\top]\!] = \Sigma \qquad [\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!] \qquad [\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!] \qquad [\![\neg\phi]\!] = \Sigma / [\![\phi]\!]$$

Traces. Finite sequences of elements α, β from domain Σ are called *traces* π. Let ϵ be the empty trace and $\pi_1 \cdot \pi_2$ be the associative concatenation of π_1 and π_2. We write $\pi_1\pi_2$ for $\pi_1 \cdot \pi_2$ when it is clear from the context that juxtaposition stands for concatenation. Following the convention, we further denote that $\Sigma^{(0)} = \{\epsilon\}$, $\Sigma^{(k+1)} = \Sigma \cdot \Sigma^{(k)}$, for $k \geq 0$, and $\Sigma^* = \bigcup_{k \geq 0} \Sigma^{(k)}$, where $L_1 \cdot L_2 = \{\pi_1\pi_2 \mid \pi_1 \in L_1, \pi_2 \in L_2\}$ for $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$. Lastly, we write L^* for the closure of L under concatenation when it is clear from the context that $L \subseteq \Sigma^*$.

Symbolic Regular Expressions. We define *Symbolic Regular Expressions* (SRE) modulo Boolean Algebra $(\Sigma, \Psi, [\![\_]\!], \bot, \bullet, \sqcup, \sqcap, \backsim)$ such that SREs use *literals* ℓ from Ψ as predicates over these characters, i.e., $[\![\ell]\!] \subseteq \Sigma$, and accept traces of characters from alphabet Σ. The top literal is denoted by $\bullet$ following the convention of regular expressions. Note that, to avoid later confusion with boolean predicates, we adopt a different set of notations for the boolean connectives. The syntax of SREs is then defined by the following operations: empty set ($\varnothing$), null (ε), literals (ℓ), Kleene Star ($\mathcal{R}^*$),

concatenation ($\mathcal{R}_1 \cdot \mathcal{R}_2$), negation ($\neg\mathcal{R}$), conjunction ($\mathcal{R}_1 \wedge \mathcal{R}_2$), and disjunction ($\mathcal{R}_1 \vee \mathcal{R}_2$).

$$\mathcal{R} \quad ::= \quad \varnothing \mid \varepsilon \mid \ell \mid \mathcal{R}^* \mid \mathcal{R}_1 \cdot \mathcal{R}_2 \mid \neg\mathcal{R} \mid \mathcal{R}_1 \wedge \mathcal{R}_2 \mid \mathcal{R}_1 \vee \mathcal{R}_2$$

Abusing the notation $[\![\_]\!]$, the denotation of SREs, $[\![\mathcal{R}]\!] \subseteq \Sigma^*$, is recursively defined as:

$$[\![\varnothing]\!] = \{\} \qquad [\![\varepsilon]\!] = \{\epsilon\} \qquad [\![\mathcal{R}^*]\!] = [\![\mathcal{R}]\!]^* \qquad [\![\mathcal{R}_1 \cdot \mathcal{R}_2]\!] = [\![\mathcal{R}_1]\!] \cdot [\![\mathcal{R}_2]\!]$$
$$[\![\neg\mathcal{R}]\!] = \Sigma^* \setminus [\![\mathcal{R}]\!] \qquad [\![\mathcal{R}_1 \wedge \mathcal{R}_2]\!] = [\![\mathcal{R}_1]\!] \cap [\![\mathcal{R}_2]\!] \qquad [\![\mathcal{R}_1 \vee \mathcal{R}_2]\!] = [\![\mathcal{R}_1]\!] \cup [\![\mathcal{R}_2]\!]$$

Following the denotation of SREs, we write $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ for $[\![\mathcal{R}_1]\!] \subseteq [\![\mathcal{R}_2]\!]$ and $\mathcal{R}_1 \equiv \mathcal{R}_2$ for $[\![\mathcal{R}_1]\!] = [\![\mathcal{R}_2]\!]$.

Conversion from LTL$_f$ to SRE. Interestingly, common temporal modalities from LTL$_f$ can be expressed in SRE. As the leaf nodes in LTL$_f$ formulae are literals ℓ , which is expressible in SRE, we provide the translation semantics of common temporal operators, assuming that the operands have already been converted to SREs, as follows:

$$\mathbf{X}\mathcal{R} \doteq \bullet \cdot \mathcal{R} \quad \ell\mathbf{U}\mathcal{R} \doteq \ell^* \cdot \mathcal{R} \quad \mathbf{F}\mathcal{R} \doteq \bullet^* \cdot \mathcal{R} \quad \mathbf{G}\mathcal{R} \doteq \neg(\bullet^* \cdot \neg\mathcal{R}) \quad \ell\mathbf{W}\mathcal{R} \doteq \neg(\bullet^* \cdot \mathcal{R}) \vee (\ell^* \cdot \mathcal{R})$$

That is, $\mathbf{X}\mathcal{R}$ (next) holds if $\mathcal{R}$ accepts the trace starting from the next position; $\ell\mathbf{U}\mathcal{R}$ (until) holds if there exists such a position that $\mathcal{R}$ accepts the following trace and ℓ holds until that position; $\mathbf{F}\mathcal{R}$ (finally) holds if there exists such a position that $\mathcal{R}$ accepts the following trace; $\mathbf{G}\mathcal{R}$ holds if there does not exist such a position that $\mathcal{R}$ rejects the following trace, and; $\ell\mathbf{W}\mathcal{R}$ (weak until) holds if either there does not exist such a position that $\mathcal{R}$ accepts the following trace, or ℓ holds until such a position. For simplicity, we limit the first operand of **U** and **W** to be a single literal ℓ , which suffices for common cases found in the ADT specifications we consider, including the modalities used in our evaluation (Section 7). We directly use SRE in the rest of the paper.

Derivatives of SRE. A *derivative* is a notion from language theory. Given a language, say defined by an SRE $\mathcal{R}$, and a string π, the derivative operation returns a new language accepting all strings that are accepted by $\mathcal{R}$ when appended to π, which can be thought of as a *prefix* to those strings.

$$[\![\mathrm{d}_\pi\mathcal{R}]\!] = \{\pi' \mid \pi \cdot \pi' \in [\![\mathcal{R}]\!]\}$$

Following the literature on derivatives of regular expressions [Antimirov 1995; Berry and Sethi 1986; Brzozowski 1964], we first inductively define a *nullable* predicate $\nu(\mathcal{R})$ that determines if $\mathcal{R}$ accepts the empty string. That is, $\nu(\mathcal{R})$ iff $\epsilon \in [\![\mathcal{R}]\!]$.

$$\nu(\varepsilon) = \nu(\mathcal{R}^*) = \top \qquad \nu(\varnothing) = \nu(\ell) = \bot \qquad \nu(\neg\mathcal{R}) = \neg\nu(\mathcal{R})$$
$$\nu(\mathcal{R}_1 \cdot \mathcal{R}_2) = \nu(\mathcal{R}_1 \wedge \mathcal{R}_2) = \nu(\mathcal{R}_1) \wedge \nu(\mathcal{R}_2) \qquad \nu(\mathcal{R}_1 \vee \mathcal{R}_2) = \nu(\mathcal{R}_1) \vee \nu(\mathcal{R}_2)$$

Then the derivatives of SREs follow and can be computed recursively via the following rules:

$$\mathrm{d}_\varepsilon\mathcal{R} = \mathcal{R} \qquad \mathrm{d}_{\alpha\pi}\mathcal{R} = \mathrm{d}_\pi\mathrm{d}_\alpha\mathcal{R} \qquad \mathrm{d}_\alpha\varnothing = \mathrm{d}_\alpha\varepsilon = \varnothing \qquad \mathrm{d}_\alpha\left(\mathcal{R}^*\right) = \mathrm{d}_\alpha\mathcal{R} \cdot \mathcal{R}^*$$

$$\mathrm{d}_\alpha\ell = \begin{cases} \varepsilon & \text{if } \alpha \in [\![\ell]\!] \\ \varnothing & \text{if } \alpha \notin [\![\ell]\!] \end{cases} \qquad \mathrm{d}_\alpha\left(\mathcal{R}_1 \cdot \mathcal{R}_2\right) = \begin{cases} (\mathrm{d}_\alpha\mathcal{R}_1 \cdot \mathcal{R}_2) \vee \mathrm{d}_\alpha\mathcal{R}_2 & \text{if } \nu(\mathcal{R}_1) \\ \mathrm{d}_\alpha\mathcal{R}_1 \cdot \mathcal{R}_2 & \text{if } \neg\nu(\mathcal{R}_1) \end{cases}$$

$$\mathrm{d}_\alpha\left(\neg\mathcal{R}\right) = \neg\mathrm{d}_\alpha\mathcal{R} \qquad \mathrm{d}_\alpha\left(\mathcal{R}_1 \wedge \mathcal{R}_2\right) = \mathrm{d}_\alpha\mathcal{R}_1 \wedge \mathrm{d}_\alpha\mathcal{R}_2 \qquad \mathrm{d}_\alpha\left(\mathcal{R}_1 \vee \mathcal{R}_2\right) = \mathrm{d}_\alpha\mathcal{R}_1 \vee \mathrm{d}_\alpha\mathcal{R}_2$$

Computing the derivative of a regular expression is a well-known technique for constructing an automaton that accepts the same language as the given regular expression. The construction closely follows a property of regular expressions — every SRE $\mathcal{R}$ can be written in the form of a disjunction as follows:

$$\mathcal{R}_{\mathsf{nullable}} \vee \bigvee_{\alpha \in \Sigma} \alpha\mathrm{d}_\alpha\mathcal{R} \quad \text{where } \mathcal{R}_{\mathsf{nullable}} \text{ is } \varepsilon \text{ if } \nu(\mathcal{R}), \varnothing \text{ otherwise.}$$

Informally, starting with the initial state, each disjunct $\alpha\mathrm{d}_\alpha\mathcal{R}$ denotes a transition to a new state with label α. If $\nu(\mathcal{R})$, then we mark the current state as an accepting state. Iteratively, we repeat the same procedure on the new states with the corresponding derivative $\mathrm{d}_\alpha\mathcal{R}$ until no new state can be added. Intuitively, each state q_i in the constructed automaton is denoted by a derivative (also in SRE) of the original $\mathcal{R}$– the derivative accepts the same language as the constructed automaton

with its initial state set to q_i. This can be manifested by a different disjunctive form $\bigvee_{\pi \in \Sigma^*} \pi \mathrm{d}_\pi \mathcal{R}$. For each disjunct $\pi \mathrm{d}_\pi \mathcal{R}$, if $\mathrm{d}_\pi \mathcal{R}$ is not empty, then π denotes a path from the accepting state to the state denoted by $\mathrm{d}_\pi \mathcal{R}$. Hence, whether $\mathrm{d}_\pi \mathcal{R}$ denotes an accepting state determines if $\mathcal{R}$ accepts π:

$$\pi \in [\![\mathcal{R}]\!] \text{ iff } \nu(\mathrm{d}_\pi \mathcal{R})$$

However, Σ often contains a large if not infinite number of symbols and thus enumerating over all symbols to build an automata is inefficient at best, and impossible in the general case. *Mintermization* solves this problem by constructing a finite set of equivalence classes over the infinite domain Σ such that all literals ℓ can be mapped to elements in this finite set ([D'Antoni and Veanes 2014; Veanes et al. 2010]). Then, following a similar procedure of constructing automata from regular expressions, one may construct an equivalent SFA where transtitions between states are labeled by equivalence classes. We will present the characterization of *symbolic derivatives* in Section 5 as a device to exploit SREs' latent SFA structures without upfront mintermization and later its computation in Section 6.

4 Trace-Based Symbolic Execution

Variable $x, y, \ldots$ Symbolic Variable $\hat{x}_\tau, \hat{y}_\tau, \ldots$ Data Constructor d
Primitive Operator op Effectful API of Representation Type $\mathrm{f}, \mathrm{g} \in \Delta$

Simple Type	τ	::=	$\mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \ldots \mid \times\, \overline{\tau} \mid +\, \overline{d\ \tau}$
Constant	c	::=	$() \mid \mathbb{B} \mid \mathbb{Z} \mid \ldots \mid (\overline{c}) \mid d\ c$
Symbolic First-Order Value	q	::=	$c \mid x \mid \hat{x}_\tau \mid (\overline{q}) \mid d\ q \mid op\ \overline{q}$
Boolean Formula	ϕ, Φ	::=	$q \mid \bot \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$
Symbolic Event	ℓ	::=	$\langle x_{\mathrm{ret}} \leftarrow \mathrm{f}\ \overline{x_{\mathrm{arg}}} \mid \phi \rangle \mid \backsim \ell \mid \ell \sqcap \ell \mid \ell \sqcup \ell$
Symbolic Value	v	::=	$x \mid q \mid (\overline{v}) \mid \mathtt{fun}\ x.\, e \mid \mathtt{fix}\ f.\, \mathtt{fun}\ x.\, e$
Symbolic Expression	e	::=	$v \mid ?_\tau \mid \mathtt{abort} \mid \mathtt{assume}\ \phi \mid \mathtt{admit}\ \mathcal{R} \mid \mathtt{append}\ \mathcal{R}$
			$\mid \mathtt{let}\ x = v\ v\ \mathtt{in}\ e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid e \otimes e$

$e_1; e_2 \doteq \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ for fresh x
$\mathtt{assert}\ \phi \doteq (\mathtt{assume}\ \neg\phi; \mathtt{abort}) \otimes \mathtt{assume}\ \phi$
$\mathtt{affirm}\ \mathcal{R} \doteq (\mathtt{admit}\ \neg\mathcal{R}; \mathtt{abort}) \otimes \mathtt{admit}\ \mathcal{R}$

Fig. 4. Syntax of the core language.

We first introduce a naïve variant of our symbolic execution framework for falsifying functional ADT implementations that interact with an underlying effectful representation type. Symbolic execution is defined on a core functional language with explicit constructs for generating symbolic values and expressing specifications of two kinds: formulae Φ from decidable theories amenable to SMT solving, which are standard for symbolic execution techniques, and trace-based specifications $\mathcal{R}$ expressed as symbolic regular expressions, which is the novelty of our framework. Fig. 4 presents the syntax of our core language, where the expression e is expressed in monadic normal form (MNF) [Hatcliff and Danvy 1994], a variant of A-normal form (ANF) [Flanagan et al. 1993] that permits nested let-bindings. Recursive functions take the form of $\mathtt{fix}\ f.\, \mathtt{fun}\ x.\, e$ using an explicit fixpoint construction, and control flow is modeled by nondeterministic choice $\otimes$ together with an `assume` construct. The symbolic constructs in this language, including `assume`, will be discussed throughout the rest of the section. Fig. 5 formalizes the naive symbolic execution of the core language as a small-step, substitution-based operational semantics over symbolic states $(\Phi, \mathcal{R}, e)$. In this section, we first introduce symbolic execution of pure functional programs and then extend it with the capability to reason over traces that interact with an ADT's underlying representation type.

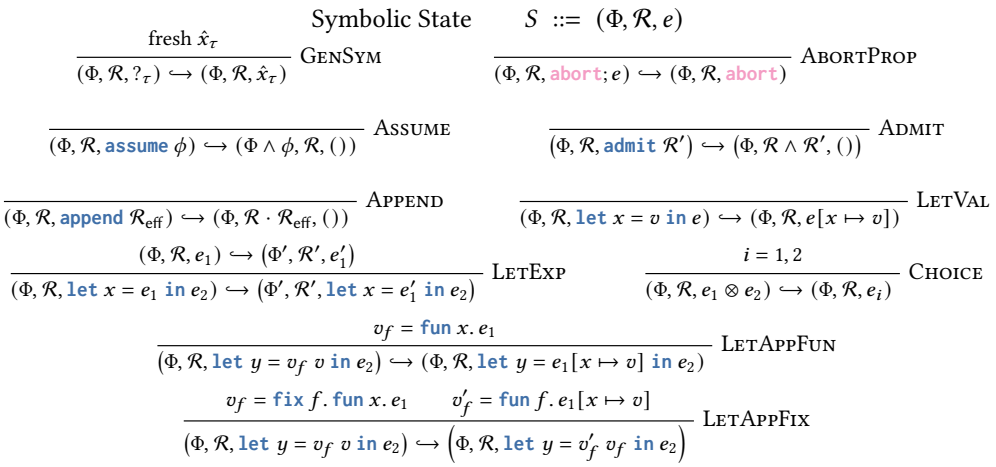


Fig. 5. Naive trace-augmented semantics.

To enable symbolic reasoning, the language supports symbolic variables $\hat{x}_\tau$, which stand for constants c of type τ. In contrast to program variables x, symbolic variables are internal to symbolic execution: they are never written by developers but are generated by the $?_\tau$ construct during symbolic execution (Rule GenSym). The τ subscript can be omitted whenever it is clear from the context; τ denotes simple types (primitive types, e.g., *unit* and *int*, product types $\times\ \overline{\tau}$, and user-defined data types $+\ \overline{d\ \tau}$) but not function types. Variables, symbolic variables, and constants, when composed by tuple constructors $(\ldots)$, data constructors d, and primitive operators op, build up to symbolic (first-order) values. Then, Boolean symbolic values, when composed by logical connectives, build up to Boolean formulae Φ. Since primitive operators are drawn from decidable first-order theories, e.g., arithmetic operators, or uninterpreted functions with user-provided axioms, the satisfiability of Boolean formulae can be straightforwardly discharged to SMT queries.

Definition 4.1 (Denotation of Boolean Formulae)**.** Let σ denote an interpretation of symbolic variables as constants and $\sigma(\Phi)$ denote the Boolean formula Φ with its symbolic variables substituted for constants according to σ. Then, the denotation of a *closed Boolean formula* Φ, where all variables are symbolic, is the set of interpretations σ such that $\sigma(\Phi)$ holds, i.e., $[\![\Phi]\!] = \{\sigma \mid \sigma(\Phi)\}$.

Now, we may introduce the symbolic execution of pure functional programs, in which case each symbolic state (Φ, e) consists of a closed Boolean formula Φ representing the current *path condition*, and a *closed* expression e — all variables are bound by `let`, `fun`, or `fix`. The path condition collects all the conditions that need to be satisfied for the symbolic state to be reachable, i.e., have a corresponding concrete state. The initial path condition is true, denoted by $\top$. Let e be the expression to be reduced. Then the initial symbolic state is $(\top, e)$. The reduction rules between symbolic states are described in Fig. 5 if we omit rules related to $\mathcal{R}$. Rule Assume describes the augmentation of the current path condition with the argument of `assume`, which is also a Boolean formula ϕ. In contrast to path conditions, we use ϕ for Boolean formulae that may involve program variables bound in expressions, which will be substituted for closed symbolic values via Rule LetVal. Each `assume` along an execution path further restricts the state space represented by the path condition. Suppose a path condition Φ is satisfiable, i.e., there exists $\sigma \in [\![\Phi]\!]$. Then, a sequence of reductions from $(\top, e)$ to $(\Phi, \texttt{abort})$, where `abort` represents a failure in execution, witnesses a *feasible* execution path of e that leads to a failure. In practice, `abort` is rarely written by developers and can be expressed

using **assert**, which is defined as syntactic sugar (Fig. 4). Whether a path condition Φ passes an assertion **assert** ϕ is effectively determined by the satisfiability of $\Phi \wedge \neg\phi$.

To reason about an ADT's interaction with underlying representation types, we equip symbolic execution with the capability to model such interactions extensionally, by recording the history of calls to the representation type's methods, along with their argument and return values. In particular, interactions are captured in symbolic regular expressions (SRE) $\mathcal{R}$ whose literals denote sets of such API calls to the representation types. Recall in Section 3, such literals are elements from EBA $(\Sigma, \Psi, [\![\_]\!], \bot, \bullet, \sqcup, \sqcap, \smile)$. Here, Σ stands for the domain of *events*, denoted by $c_{\text{ret}} \leftarrow \mathsf{f}\ \overline{c_{\text{arg}}}$, and Ψ includes all the *symbolic events* ℓ, each denoting set of events, according to the syntax shown in Fig. 4. An *atomic symbolic event* $\langle x_{\text{ret}} \leftarrow \mathsf{f}\ \overline{x_{\text{arg}}} \mid \phi\rangle$ denotes the calls to f such that the arguments $\overline{c_{\text{arg}}}$ and the return value c_{ret} satisfy the qualifier ϕ:

$$[\![\langle x_{\text{ret}} \leftarrow \mathsf{f}\ \overline{x_{\text{arg}}} \mid \phi\rangle]\!] \doteq \{c_{\text{ret}} \leftarrow \mathsf{f}\ \overline{c_{\text{arg}}} \mid [\overline{x_{\text{arg}} \mapsto c_{\text{arg}}}, x_{\text{ret}} \mapsto c_{\text{ret}}]\phi\}$$

The boolean connectives have standard denotation as shown in Section 3. Notice that the scope of $\overline{x_{\text{arg}}}$ and x_{ret} is limited to the qualifier ϕ of the symbolic event. We omit such variables *local* to the symbolic event when they are either obvious from or irrelevant to the context. For example, we always use *key* and *val* to denote keys and values of the calls to put and get from key-value stores, with the result of put omitted. And similar to Section 2, we write $\langle \text{put}\ \hat{k}\ \hat{v}\rangle$ for $\langle \text{put}\ \mathit{key}\ \mathit{val} \mid \mathit{key} = \hat{k} \wedge \mathit{val} = \hat{v}\rangle$ and $\langle \text{put}\ \not{\hat{k}}\ \hat{v}\rangle$ for $\langle \text{put}\ \mathit{key}\ \mathit{val} \mid \mathit{key} \neq \hat{k} \wedge \mathit{val} = \hat{v}\rangle$. An atomic symbolic event is *closed* if all variables in its qualifier are either symbolic or local to the event; a symbolic event ℓ is *closed* if all its atomic symbolic events are, and; an SRE is *closed* if all its symbolic events are. The denotation does not apply to all closed SREs but only those SREs without symbolic variables. For SREs that reference symbolic variables, we can only interpret them after interpreting these symbolic variables in a way consistent with the path condition if any.

By augmenting symbolic states with SRE $\mathcal{R}_{\text{curr}}$ to represent the events that have happened, we define a reduction semantics over $(\Phi, \mathcal{R}_{\text{curr}}, e)$ as shown in Fig. 5. We refer to such an SRE $\mathcal{R}_{\text{curr}}$ as the *current context* of the execution from the associated symbolic state. In addition, we refer to SREs as *contexts* or *effects* of a method (ADT's or representation type's) depending on whether they describe admissible traces prior to calling the method or traces the method is supposed to produce. Similar to path conditions, the SREs that represent the current context of symbolic states are always closed. Definition 4.2 gives the reachability of a symbolic state S based on the satisfiability of its path condition Φ and its current context $\mathcal{R}_{\text{curr}}$.

Definition 4.2 (Reachability)**.** isSat $(\Phi, \mathcal{R}_{\text{curr}})$ iff there exists $\sigma \in [\![\Phi]\!]$ such that $[\![\sigma(\mathcal{R}_{\text{curr}})]\!] \neq \emptyset$.

Henceforth, we omit the carat ($\hat{}$) on symbolic variables $\hat{x}$ and assume all variables are symbolic except for those variables bound in expressions.

For a symbolic state $(\Phi, \mathcal{R}_{\text{curr}}, e)$, its path condition Φ effectively captures the history of pure computation up to this state while its current context $\mathcal{R}_{\text{curr}}$ captures the history of effectful computation. Because the events in $\mathcal{R}_{\text{curr}}$ are qualified with reference to the symbolic variables in Φ, both structures synergyistically enable the recording of a sufficient condition that allows a computation to reach the symbolic state.

***Example* 4.3.** The `remove` method from Fig. 1 can be rewritten in our core language, with the conditional expression represented by a combination of **assume** and choice operation $\otimes$, as follows:

$$e_{\texttt{remove}} \doteq \textbf{fun}\ \mathit{hd}.\ \textbf{fun}\ \mathit{elem}.\ (\textbf{assume}\ \mathit{hd} = \texttt{null}; \mathit{hd}) \otimes (\textbf{assume}\ \mathit{hd} \neq \texttt{null};\ \ldots)$$

Recall that the specification in Fig. 1 requires `remove` to be called in a symbolic state where node a is linked to node b as its successor. Its precondition can be written as an SRE thus:

$$\mathcal{R}_{a \rightsquigarrow b} \doteq \bullet^* \cdot \langle \text{Nxt.put}\ a\ b\rangle \cdot (\smile\langle \text{Nxt.put}\ a\ \_\rangle)^*$$

This context admits traces in which a call to Nxt.put is made on key a and value b, followed by subsequent events that do not include calls to Nxt.put with key a. Hence, the context encapsulates the intended requirement on symbolic states prior to calling `remove`.

The specification in Fig. 1 also requires that `remove`, when called under the specified context, can link a new node other than a to b *only* when b has been unlinked from a. The postcondition can be written as an SRE parameterized by a and b thus:

$$\mathcal{R}_{a\nrightarrow b} \doteq ((\smile\langle \text{Nxt.put } \not{a}\ b\rangle)^* \cdot \langle \text{Nxt.put } a\ \not{b}\rangle \cdot \bullet^*) \vee (\smile\langle \text{Nxt.put } a\ \not{b}\rangle)^*$$

The effect of `remove` admits traces where no node other than a is linked to b (via $\smile\langle \text{Nxt.put } \not{a}\ b\rangle$) before a is unlinked from b (via $\langle \text{Nxt.put } a\ \not{b}\rangle$), or a is never unlinked from b (via $\smile\langle \text{Nxt.put } a\ \not{b}\rangle$). If an execution of `remove` produces traces not admissible to $\mathcal{R}_{a\nrightarrow b}$, then we conclude that some node, as witnessed by b, may unexpectedly have two predecessors at some point during the execution. □

Similar to how path conditions are augmented by **`assume`** constructs, the current contexts of executions are augmented by two constructs, **`admit`** $\mathcal{R}_{\text{past}}$ and **`append`** $\mathcal{R}_{\text{eff}}$. The former, **`admit`**, combines the current context $\mathcal{R}_{\text{curr}}$ and its argument $\mathcal{R}_{\text{past}}$ with conjunction, as described by Rule Admit; thus, it restricts the traces of past events in $\mathcal{R}_{\text{curr}}$ to only those admissible by $\mathcal{R}_{\text{past}}$. In contrast, **`append`** concatenates the current context $\mathcal{R}_{\text{curr}}$ with the argument $\mathcal{R}_{\text{eff}}$, as described by Rule Append; thus it records new events produced during symbolic execution. The initial context before starting the symbolic execution is ε, indicating that no event has happened yet.

Now, we illustrate that a pair of **`append`** and **`affirm`** constructs the translation of the specification attached to an ADT method, capturing the safety property. Recall from Fig. 1 that the specification includes three key components, ghost variables, required context (`context`), and expected effect (`effect`). Intuitively, the specification states that when being executed in a required context (with possible reference to both ghost variables and method parameters), the method with the specification attached should produce events in compliance with the expected effect. The **`append`** helps set up this required context while the **`affirm`** is responsible for affirming that the context upon exiting the method complies with its argument $\mathcal{R}_{\text{post}}$ by conjoining the context $\mathcal{R}_{\text{curr}}$ with $\neg\mathcal{R}_{\text{post}}$. The satisfiability of $\mathcal{R}_{\text{curr}} \wedge \neg\mathcal{R}_{\text{post}}$ then witnesses a violation of $\mathcal{R}_{\text{post}}$ in the execution manifested by $\mathcal{R}_{\text{curr}}$. To falsify the implementation of the ADT method against its specification, we construct a harness $e_{harness}$[2] that wraps a call to the ADT method with such a pair of **`append`** and **`affirm`**.

***Example* 4.4.** Continuing from Example 4.3, the specification of `remove` is converted into a harness $e_{harness}$. First, symbolic variables a and b are generated (by "?" with the same name as the program variables) to denote two arbitrary nodes. Second, symbolic variables hd and u are generated (again by "?") to denote the input to `remove`. Third, the required context $\mathcal{R}_{a\rightharpoonup b}$ of `remove` is appended to the initial context ε. Lastly, after calling `remove`, the postcondition of the harness is affirmed to check for any violation during the execution of the harness. Notably, the postcondition prepends the required context $\mathcal{R}_{a\rightharpoonup b}$ to the expected effect $\mathcal{R}_{a\nrightarrow b}$. □

$e_{harness} \doteq$
`let` $a, b = ?_{\texttt{node}}, ?_{\texttt{node}}$ **`in`**
`let` $hd, u = ?_{\texttt{node}}, ?_{\texttt{elem}}$ **`in`**
`append` $\mathcal{R}_{a\rightharpoonup b}$;
`remove` $hd\ u$;
`affirm` $\mathcal{R}_{a\rightharpoonup b} \cdot \mathcal{R}_{a\nrightarrow b}$

In contrast, pair(s) of **`admit`** and **`append`** construct the translation of the specifications attached to APIs of representation types, providing an extensional and underapproximate model for their behavior. And the **`admit`** relates the context require by the API with the current context by conjoining them while the following **`append`** records the expected traces of events produced by the API.

***Example* 4.5.** Taking the same form as the required context $\mathcal{R}_{a\rightharpoonup b}$ of `remove` from Example 4.3, the required context of Nxt.get is $\mathcal{R}_{s\rightharpoonup t}$, admitting traces where node s (the argument) is linked to

[2]See supplementary material for details on the translation of source language expressions to core language ones.

node t (the return value). And the expected effect of the Nxt.get is a symbolic event $\langle t \leftarrow \mathsf{Nxt.get}\ s\rangle$. And thus, calls to Nxt.get can be replaced by a function $e_{\mathsf{Nxt.get}}$ defined as follows:

$$e_{\mathsf{Nxt.get}} \doteq \mathbf{fun}\ s.\ \mathbf{let}\ t = ?_{\mathtt{node}}\ \mathbf{in}\ \mathbf{admit}\ \mathcal{R}_{s\rightsquigarrow t};\ \mathbf{append}\ \langle t \leftarrow \mathsf{Nxt.get}\ s\rangle;\ t$$

Similarly, let $\mathcal{R}_{n:u} \doteq \bullet^* \cdot \langle \mathsf{Val.put}\ n\ u\rangle \cdot (\backsim\langle \mathsf{Val.put}\ n\ \_\rangle)^*$ denote the required context of Val.get, where node n (the argument) stores an element u (the return value). Correspondingly, calls to Val.get are replaced by a function $e_{\mathsf{Val.get}}$ defined as follows:

$$e_{\mathsf{Val.get}} \doteq \mathbf{fun}\ n.\ \mathbf{let}\ u = ?_{\mathtt{elem}}\ \mathbf{in}\ \mathbf{admit}\ \mathcal{R}_{n:u};\ \mathbf{append}\ \langle u \leftarrow \mathsf{Val.get}\ n\rangle;\ u$$

Here, the calls to get always succeed because our goal is to falsify the implementation of `remove` with respect to the specified safety property. □

By replacing API calls in the direct translation of the ADT method, e.g., $e_{\mathtt{remove}}$ from Example 4.3, with symbolic expressions that augment the context of execution using **admit** and **append**, we now have an implementation e_{remove} of the ADT method `remove` that is ready to be plugged in $e_{harness}$ for symbolic execution.

***Example* 4.6.** By substituting the call to `remove` for $e_{\mathtt{remove}}$ (Example 4.3) with Val.get and Nxt.get respectively substituted for $e_{\mathsf{Val.get}}$ and $e_{\mathsf{Nxt.get}}$ (Example 4.5), the harness $e_{harness}$ (Example 4.4) is closed and ready for symbolic execution. Initially, the symbolic state is $(\top, \varepsilon, e_{harness})$. The required context $\mathcal{R}_{a\rightsquigarrow b}$ of `remove` is first appended to ε. Following the second branch, $n_0 \neq \mathtt{null}$ augments the path condition. As Val.get is called on n_0, n_0 substitutes n in the body of $e_{\mathsf{Val.get}}$ and a fresh symbolic variable u_0 is generated to represent the element stored in n_0. The symbolic state becomes

$$(n_0 \neq \mathtt{null}, \mathcal{R}_{a\rightsquigarrow b}, \mathbf{let}\ u' = \mathbf{admit}\ \mathcal{R}_{n_0:u_0};\ \mathbf{append}\ \langle u_0 \leftarrow \mathsf{Val.get}\ n_0\rangle;\ u_0\ \mathbf{in}\ \ldots)$$

As u_0 is returned to the top level and substitutes u', the current context becomes $\mathcal{R}_{a\rightsquigarrow b} \wedge \mathcal{R}_{n_0:u_0} \cdot \langle u_0 \leftarrow \mathsf{Val.get}\ n_0\rangle$ ($\wedge$ binds SREs tighter than $\cdot$). Following the nested second branch, the path condition becomes $n_0 \neq \mathtt{null} \wedge u_0 \neq u$. As we enter the `loop` and follow the execution path illustrated in Section 2.2, we (1) get the successor of n_0, n_1, (2) get the element stored in n_1, u_1, (3) assume $u_1 = u$, (4) get the successor of n_1, n_2, and (5) remove n_1 by linking n_0 to n_2. The symbolic state becomes $(\Phi_{\mathsf{bad}}, \mathcal{R}_{\mathsf{bad}}, \mathbf{affirm}\ \mathcal{R}_{a\rightsquigarrow b} \cdot \mathcal{R}_{a\not\rightsquigarrow b})$, where

$$\begin{aligned}
\Phi_{\mathsf{bad}} &\doteq n_0 \neq \mathtt{null} \wedge u_0 \neq u \wedge n_1 \neq \mathtt{null} \wedge u_1 = u \quad \text{and} \\
\mathcal{R}_{\mathsf{bad}} &\doteq ((((\mathcal{R}_{a\rightsquigarrow b} \wedge \mathcal{R}_{n_0:u_0} \cdot \langle u_0 \leftarrow \mathsf{Val.get}\ n_0\rangle) \wedge \mathcal{R}_{n_0\rightsquigarrow n_1} \cdot \langle n_1 \leftarrow \mathsf{Nxt.get}\ n_0\rangle) \\
&\qquad \wedge \mathcal{R}_{n_1:u_1} \cdot \langle u_1 \leftarrow \mathsf{Val.get}\ n_1\rangle) \wedge \mathcal{R}_{n_1\rightsquigarrow n_2} \cdot \langle n_2 \leftarrow \mathsf{Nxt.get}\ n_1\rangle) \cdot \langle \mathsf{Nxt.put}\ n_0\ n_2\rangle
\end{aligned}$$

$\mathcal{R}_{\mathsf{bad}}$ denotes the traces that can be produced following the execution path. To show that the affirmation may fail, it is sufficient to find an interpretation for the symbolic variables such that the path condition Φ_{bad} holds and there exists a trace included in $\mathcal{R}_{\mathsf{bad}}$ but excluded from the postcondition of the harness, i.e., $\mathsf{isSat}\ (\Phi_{\mathsf{bad}}, \mathcal{R}_{\mathsf{bad}} \wedge \neg(\mathcal{R}_{a\rightsquigarrow b} \cdot \mathcal{R}_{a\not\rightsquigarrow b}))$. □

As illustrated in Example 4.6, the size of the SRE that represents the current context of the execution quickly blows up during symbolic execution. This in turn makes the symbolic affirmation check at the end of each execution path potentially very expensive as we quantify in Section 7.

To conclude this section, we lift the **affirm** check at the end of the harness progress and regard it as a falsification query on the harness program without the **affirm** statement. Then Definition 4.7 describes a falsification problem in terms of this trace-based symbolic execution framework.

Definition 4.7 (Naïve Falsification)**.** Given a safety property $\mathcal{R}_{\mathsf{post}}$. If $(\top, \varepsilon, e) \hookrightarrow^* (\Phi, \mathcal{R}_{\mathsf{curr}}, v)$ and $\mathsf{isSat}\ (\Phi, \mathcal{R}_{\mathsf{curr}} \wedge \neg\mathcal{R}_{\mathsf{post}})$ then this execution of e is falsified with respect to $\mathcal{R}_{\mathsf{post}}$.

5 Symbolic Execution with Symbolic Derivatives

The inefficiency of the naive semantics stems from its failure to recognize *regularity* — the capacity of specifications to be represented as automata structures — during symbolic execution. We can exploit this regularity by underapproximating the required context or the expected effect of method calls. This approximation facilitates a derivative computation, effectively emulating state transitions in the SFAs associated with SREs. Specifically, the underapproximation takes the form of *symbolic traces* Π, where only a subset of operations from the SRE (with the same denotation) are allowed: empty trace (ε), symbolic event (ℓ), and concatenation ($\Pi_1 \cdot \Pi_2$). Then, a derivative-based notion of symbolic state $S_{\mathcal{D}}$ that underapproximates a symbolic state S, besides the expression e under execution, is given by (i) Φ and Π to encapsulate the execution path that leads to $S_{\mathcal{D}}$; along with (ii) $\mathcal{R}_{\text{cont}}$ that predicts the traces allowed to be produced in the continuation of the execution in compliance with the safety property, dubbed *continuation effect*.

$$\text{Symbolic Trace } \Pi \ ::= \ \varepsilon \mid \ell \mid \Pi \cdot \Pi \qquad \text{Derivative-Based Symbolic State } S_{\mathcal{D}} \ ::= \ (\Phi, \Pi, \mathcal{R}_{\text{cont}}, e)$$

In this section, we present (1) *symbolic derivatives* that allow us to effectively explore and thus exploit the automata structures of specifications, without appealing to their calculation (see Section 6), and (2) a *derivative-based semantics* that leverages this notion to facilitate symbolic execution over $S_{\mathcal{D}}$ as well as the falsification of a given safety property that is both sound and complete with respect to the naïve semantics given in the previous section.

5.1 Symbolic Derivatives

SREs that represent the context of the current execution or the arguments to **admit** and **append** may refer to symbolic variables that are also constrained by path conditions, as discussed in Section 4. In what follows, we first revisit notions on SREs from Section 3 with such symbolic variables left uninterpreted, i.e., treating symbolic variables as abstract symbols whose interpretation is unknown. We then define *symbolic derivatives* of such SREs, which may also refer to symbolic variables involved in those SREs.

First, the inclusion and equivalence relationship between two SREs $\mathcal{R}_1$ and $\mathcal{R}_2$ is given by Definition 5.1 and Definition 5.2 such that the relationship holds under any interpretation of symbolic variables involved in $\mathcal{R}_1$ and $\mathcal{R}_2$.

Definition 5.1 (Inclusion). $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ iff $[\![\sigma(\mathcal{R}_1)]\!] \subseteq [\![\sigma(\mathcal{R}_2)]\!]$ for all σ.

Definition 5.2 (Equivalence). $\mathcal{R}_1 \equiv \mathcal{R}_2$ iff $[\![\sigma(\mathcal{R}_1)]\!] = [\![\sigma(\mathcal{R}_2)]\!]$ for all σ.

Second, the nullable operation ν defined over SREs in Section 3, when applied to any symbolic event ℓ, returns false irrespective of ℓ 's qualifiers and any symbolic variables involved. Hence, the nullable operation $\nu(\mathcal{R})$ determines if $\mathcal{R}$ accepts the empty trace ϵ regardless of the interpretation of its symbolic variables (Lemma 5.3).

Lemma 5.3. $\nu(\mathcal{R})$ iff $\nu(\sigma(\mathcal{R}))$ for all σ. [3]

Third, we need to revisit the notion of prefixes of SREs. Recall in Section 3, for an SRE $\mathcal{R}$ that does not involve symbolic variables, any concrete trace π can be a prefix of $\mathcal{R}$ since the derivative $\mathrm{d}_\pi\mathcal{R}$, which contains all concrete traces that are accepted by $\mathcal{R}$ when appended to π, is always well-defined. To account for symbolic variables involved in $\mathcal{R}$, we consider symbolic traces Π, which may also involve these symbolic variables, as a consolidated form of prefixes of $\mathcal{R}$. Definition 5.4 gives the criteria that a valid prefix Π of $\mathcal{R}$ has to meet.

Definition 5.4. Π is a prefix of $\mathcal{R}$ iff there exists $\mathcal{R}'$ s.t. $\mathrm{d}_\pi\sigma(\mathcal{R})=\sigma(\mathcal{R}')$ for all $\pi \in [\![\sigma(\Pi)]\!]$ for all σ.

[3] All proofs are deferred to the full version of this paper [Yuan et al. 2024b].

Intuitively, Π qualifies as a prefix of $\mathcal{R}$ if it represents a collection of partial runs of an SFA associated with $\mathcal{R}$. These partial runs must begin with the SFA's start state and end at any arbitrary state. Notably, the ending state does not need to be accepting and may even be a *dead state*, from which no accepting state is accessible. For example, recall the postcondition automaton from Fig. 2: $\langle \text{Nxt.put } a \not\flat \rangle$ and $\langle \text{Nxt.put } \not\flat\, b \rangle$ are both valid prefixes but their disjunction is not because some runs end at q_3 while the others end at the dead state denoted by $\varnothing$. We dub such singleton prefixes as *next events*.

Following the definition of prefixes, we introduce symbolic derivatives in Definition 5.5.

Definition 5.5 (Symbolic Derivative)**.** $\mathsf{D}_\Pi \mathcal{R} = \mathcal{R}'$ iff $\mathsf{d}_\pi \sigma(\mathcal{R}) = \sigma(\mathcal{R}')$ for all $\pi \in [\![\sigma(\Pi)]\!]$ for all σ.

In contrast to conventional derivatives discussed in Section 3, symbolic derivatives of $\mathcal{R}$ are well defined *only* over its prefixes (Definition 5.4) but not arbitrary symbolic traces. And the property of prefixes ensures that symbolic derivatives can still be succinctly expressed as SREs with references to symbolic variables if any. Notably, $\mathcal{R}$, its prefix Π, and its symbolic derivative $\mathcal{R}$' over Π shall interpret any referenced symbolic variables in a consistent way; Definition 5.5 serves as a guard against inconsistent interpretations.

Since each valid prefix Π of $\mathcal{R}$ establishes an equivalence class where all π denoted by Π produce the same derivative, a symbolic derivative $\mathsf{D}_\Pi \mathcal{R}$ is not only a *quotient* but also a *residual* of $\mathcal{R}$ with respect to Π. As noted by [Pratt 1991], the quotient of $\mathcal{R}$ contains traces that are accepted by $\mathcal{R}$ when appended to *some* π denoted by prefix Π, while the residual of $\mathcal{R}$ contains traces that are accepted by $\mathcal{R}$ when appended to *any* π denoted by prefix Π. This residuality is manisfest by Corollary 5.6, i.e., the concatenation of prefix Π and $\mathsf{D}_\Pi \mathcal{R}$ is included in $\mathcal{R}$ itself.

Corollary 5.6 (Residuality)**.** Let $\mathcal{R}' = \mathsf{D}_\Pi \mathcal{R}$. Then $\Pi \cdot \mathcal{R}' \sqsubseteq \mathcal{R}$.

***Example* 5.7.** Consider the expected effect $\mathcal{R}_{a \hookrightarrow b}$ of `remove` from Example 4.3, admitting traces where either no node other than a may be linked to b before b is unlinked from a, or a is linked to b during the course of execution. Its next events include $\langle \text{Nxt.put } a \not\flat \rangle$, $\langle \text{Nxt.put } \not\flat\, b \rangle$, and $\langle \text{Nxt.put } a\ b \rangle \sqcup \langle \text{Nxt.put } \not\flat \not\flat \rangle \sqcup\!\smile \langle \text{Nxt.put} \rangle$. Their symbolic derivatives are defined as follows, with their respective residuality manifested: (1) $\mathsf{D}_{\langle \text{Nxt.put } a \not\flat \rangle} \mathcal{R}_{a \hookrightarrow b} = \bullet^*$ because any event is allowed once b is unlinked from a; (2) $\mathsf{D}_{\langle \text{Nxt.put } \not\flat\, b \rangle} \mathcal{R}_{a \hookrightarrow b} = \varnothing$ because it is unsafe to link node other than a to b with a linked to b, and; (3) $\mathsf{D}_{\langle \text{Nxt.put } a\ b \rangle \sqcup \langle \text{Nxt.put } \not\flat \not\flat \rangle \sqcup\!\smile \langle \text{Nxt.put} \rangle} \mathcal{R}_{a \hookrightarrow b} = \mathcal{R}_{a \hookrightarrow b}$ because linking a to b again, linking nodes other than a and b, or **`get`** calls have no effect on subsequent traces admissible by $\mathcal{R}_{a \hookrightarrow b}$. □

Recall that the nullablility of derivative $\mathsf{d}_\pi \mathcal{R}$ determines if a concrete trace π is accepted by $\mathcal{R}$. Given a prefix Π of an SRE $\mathcal{R}$, the nullability of symbolic derivative $\mathsf{D}_\Pi \mathcal{R}$ determines, as established by Corollary 5.8, whether the symbolic trace Π is included in $\mathcal{R}$, i.e., all runs of Π in $\mathcal{R}$ end at an accepting state irrespective of the interpretation of symbolic variables.

Corollary 5.8. Let $\mathcal{R}' = \mathsf{D}_\Pi \mathcal{R}$. Then (1) $\Pi \sqsubseteq \mathcal{R}$ iff $\nu(\mathcal{R}')$ and (2) $\Pi \sqsubseteq \neg\mathcal{R}$ iff $\neg\nu(\mathcal{R}')$.

Therefore, by enumerating prefixes of $\mathcal{R}$, we may sample symbolic traces included in $\mathcal{R}$.

***Example* 5.9.** Consider the required context $\mathcal{R}_{a \rightsquigarrow b}$ of `remove` from Example 4.3, admitting traces where a is linked to b and stays pointing to b. The prefixes of $\mathcal{R}_{a \rightsquigarrow b}$ include $\langle \text{Nxt.put } a\ b \rangle \cdot (\smile \langle \text{Nxt.put } a \not\flat \rangle)^n$ and $\langle \text{Nxt.put } a\ b \rangle \cdot \langle \text{Nxt.put } a \not\flat \rangle \cdot \langle \text{Nxt.put } a\ b \rangle \cdot (\smile \langle \text{Nxt.put } a \not\flat \rangle)^n$ for any number n of repetitions, which all lead to the same symbolic derivative:

$$(\smile \langle \text{Nxt.put } a \not\flat \rangle)^* \vee (\langle \text{Nxt.put } a \not\flat \rangle \cdot \mathcal{R}_{a \rightsquigarrow b})$$

admitting traces where either no subsequent event invalidates the link between a and b, or a is linked to b again after being unlinked. The symbolic derivative is nullable because its first disjunct is a Kleene Star. Therefore, all these prefixes are included in $\mathcal{R}_{a \rightsquigarrow b}$. □

$$\frac{\Pi_{\text{past}} \sqsubseteq \mathcal{R}_{\text{past}} \qquad \Pi' \equiv \Pi \wedge \Pi_{\text{past}}}{\left(\Phi, \Pi, \mathcal{R}_{\text{cont}}, \texttt{admit}\ \mathcal{R}_{\text{past}}\right) \hookrightarrow_{\mathcal{D}} \left(\Phi, \Pi', \mathcal{R}_{\text{cont}}, ()\right)}\ \textsc{DAdmit}$$

$$\frac{\Pi_{\text{eff}} \sqsubseteq \mathcal{R}_{\text{eff}} \qquad \Pi_{\text{new}} \equiv \Pi_{\text{eff}} \wedge \Pi_{\text{prefix}}}{\left(\Phi, \Pi, \mathcal{R}_{\text{cont}}, \texttt{append}\ \mathcal{R}_{\text{eff}}\right) \hookrightarrow_{\mathcal{D}} \left(\Phi, \Pi \cdot \Pi_{\text{new}}, \mathrm{D}_{\Pi_{\text{prefix}}} \mathcal{R}_{\text{cont}}, ()\right)}\ \textsc{DAppend}$$

Fig. 6. Selected rules of derivatve-based semantics.

5.2 Derivative-Based Semantics

Now, we facilitate symbolic execution with symbolic derivatives. As **admit** $\mathcal{R}_{\text{past}}$ and **append** $\mathcal{R}_{\text{eff}}$ are the only two constructs that augment contexts in symbolic states, we only present their reduction rules in Fig. 6, exhibiting the complete set of rules in the supplementary material. In contrast to the naïve semantics, a derivative-based semantics begins symbolic execution with the postcondition, denoted by $\mathcal{R}_{\text{post}}$. Recall from Section 4, $\mathcal{R}_{\text{post}}$ is the concatenation of the required context and the expected effect attached to the ADT method to be falsified. Effectively, $\mathcal{R}_{\text{post}}$ predicts that the context will be set up before calling the method, and the execution of the method complies with its specified effect. During symbolic execution, we maintain the continuation effect $\mathcal{R}_{\text{cont}}$ such that it precisely predicts the safe traces to be produced as execution continues.

***Example* 5.10.** Consider the harness program $e_{harness}$ from Example 4.4. Regarding the trailing **affirm** as a postcondition to be affirmed upon finishing each execution path, we assume some symbolic variables a and b and discharge **affirm** from $e_{harness}$ as part of the symbolic state. In a derivative-based semantics, the initial symbolic state is

$$(\top, \varepsilon, \mathcal{R}_{a \rightharpoonup b} \cdot \mathcal{R}_{a \not\rightharpoonup b}, \texttt{let}\ hd, u = ?_{\texttt{node}}, ?_{\texttt{elem}}\ \texttt{in}\ \texttt{append}\ \mathcal{R}_{a \rightharpoonup b};\ \texttt{remove}\ hd\ u)$$

where hd and u will then immediately be replaced by symbolic variables with the same name for demonstration's purposes. □

Rule DAdmit describes the semantics of **admit** $\mathcal{R}_{\text{past}}$. Given a symbolic trace Π, a sequence of symbolic events, as an underapproximation of what has happened so far following the current execution, **admit** $\mathcal{R}_{\text{past}}$ imposes constraint on Π, also in an underapproximated fashion. The underapproximation of $\mathcal{R}_{\text{past}}$ can be found by sampling symbolic traces $\Pi_{\text{past}} \sqsubseteq \mathcal{R}_{\text{past}}$ via symbolic derivatives. Then the execution is forked on each Π_{past} and its conjunction Π' with Π. Intuitively, the conjunction Π' is the pairwise conjunction of events in Π and Π_{past} (see Section 6 for details). A straightforward pruning strategy then is to discard Π_{past} with (1) a different number of events than Π or (2) an event associated with a different effectful function than the corresponding event in Π. In both cases, the conjunction Π' trivially denotes an empty set. We describe such Π_{past} as *incompatible* with Π. Note that we deliberately exclude from the compatibility check the consistency check between qualifiers of paired symbolic events to avoid generating an excessive number of SMT queries.

***Example* 5.11.** Consider the naive symbolic state prior to calling Val.get on n_0 from Example 4.6, $(n_0 \neq \texttt{null}, \mathcal{R}_{a \rightharpoonup b}, \texttt{let}\ u' = \texttt{admit}\ \mathcal{R}_{n_0:u_0};\ \texttt{append}\ \langle u_0 \leftarrow \text{Val.get}\ n_0 \rangle;\ u_0\ \texttt{in}\ \ldots)$. A derivative-based symbolic state that underapproximates this naive state is $(n_0 \neq \texttt{null}, \Pi_{a \rightharpoonup b}, \mathcal{R}_{a \not\rightharpoonup b}, \ldots)$, where $\Pi_{a \rightharpoonup b} \doteq \langle \text{Nxt.put}\ a\ b \rangle \cdot (\backsim \langle \text{Nxt.put}\ a\ \not b \rangle)^3 \sqsubseteq \mathcal{R}_{a \rightharpoonup b}$ as shown in Example 5.9, and $\mathcal{R}_{a \not\rightharpoonup b}$ is the continuation effect prior to the call as we will discuss shortly in Example 5.12. The **admit** $\mathcal{R}_{n_0:u_0}$ operation enforces the required context of the call to Val.get. A symbolic trace that underapproximates $\mathcal{R}_{n_0:u_0}$ and is compatible with the current context $\Pi_{a \rightharpoonup b}$ is $\Pi_{n_0:u_0} \doteq \backsim \langle \text{Val.put}\ n_0\ u_0 \rangle \cdot$

$\langle \text{Val.put } n_0\ u_0\rangle \cdot (\backsim\langle \text{Val.put } n_0\ \not{u_0}\rangle)^2$. Augmented by $\Pi_{n_0:u_0}$, the context in the symbolic state then becomes $\langle \text{Nxt.put } a\ b\rangle \cdot \langle \text{Val.put } n_0\ u_0\rangle \cdot (\backsim\langle \text{Val.put } n_0\ \not{u_0}\rangle \sqcap \backsim\langle \text{Nxt.put } a\ \not{b}\rangle)^2$. □

Rule DAppend describes the semantics of **append** $\mathcal{R}_{\text{eff}}$, where new events are to be appended to the current symbolic trace Π. First, we underapproximate events to be produced by **append** $\mathcal{R}_{\text{eff}}$, again by sampling symbolic traces $\Pi_{\text{eff}} \sqsubseteq \mathcal{R}_{\text{eff}}$. Second, we enumerate prefixes Π_{prefix} of $\mathcal{R}_{\text{cont}}$ that are compatible with Π_{eff}, along with the symbolic derivative $\mathrm{D}_{\Pi_{\text{prefix}}}\mathcal{R}_{\text{cont}}$. The execution can be forked for each pair of Π_{eff} and Π_{prefix}. Recall that $\mathcal{R}_{\text{cont}}$ imposes constraints on the events produced during symbolic execution, including those produced by **append**. As long as the behavior of **append**, in this case, the underapproximation Π_{eff}, complies with the constraints imposed by Π_{prefix}, $\mathrm{D}_{\Pi_{\text{prefix}}}\mathcal{R}_{\text{cont}}$ represents the constraint on events to be produced after **append** and thus can safely replace $\mathcal{R}_{\text{cont}}$ in the next symbolic state. To enforce this compliance, we take the conjunction of Π_{eff} and Π_{prefix} and append the result Π_{new} to the current symbolic path Π. Effectively, we relate an underapproximated behavior of **append** with the postcondition of the method to be falsified, from which $\mathcal{R}_{\text{cont}}$ is derived, and track this relation in the symbolic state.

***Example* 5.12.** Consider the initial symbolic state from Example 5.10. The **append** construct requires the traces of past events to be admissible to its argument $\mathcal{R}_{a\rightsquigarrow b}$ before calling `remove`. Then `remove` can be called in a context represented by any symbolic trace $\Pi_{a\rightsquigarrow b} \sqsubseteq \mathcal{R}_{a\rightsquigarrow b}$. Furthermore, each such $\Pi_{a\rightsquigarrow b}$ is also a prefix of the postcondition $\mathcal{R}_{a\rightsquigarrow b} \cdot \mathcal{R}_{a\not\rightsquigarrow b}$. The symbolic derivative $\mathrm{D}_{\Pi_{a\rightsquigarrow b}}(\mathcal{R}_{a\rightsquigarrow b} \cdot \mathcal{R}_{a\not\rightsquigarrow b}) = \mathcal{R}_{a\not\rightsquigarrow b}$ becomes the continuation effect after evaluating the **append** operation. The symbolic state prior to calling `remove` is $(\top, \varepsilon, \mathcal{R}_{a\not\rightsquigarrow b}, e_{\texttt{remove}}\ n_0\ u)$. □

***Example* 5.13.** Continuing from Example 5.11, the symbolic state after reducing the **admit** is $(n_0 \neq \texttt{null}, \Pi_{a\rightsquigarrow b} \wedge \Pi_{n_0:u_0}, \mathcal{R}_{a\not\rightsquigarrow b}, \textbf{append}\ \langle u_0 \leftarrow \text{Val.get } n_0\rangle)$. The **append** construct records the call to Val.get and appends a singleton event $\langle u_0 \leftarrow \text{Val.get } n_0\rangle$ to the context. Correspondingly, the continuation effect $\mathcal{R}_{a\not\rightsquigarrow b}$ is updated by its symbolic derivative over the Val.get event, $\mathrm{D}_{\langle u_0 \leftarrow \text{Val.get } n_0\rangle}\mathcal{R}_{a\not\rightsquigarrow b}$, which is $\mathcal{R}_{a\not\rightsquigarrow b}$ itself as shown in Example 5.7. □

Now we leverage derivative-based semantics to falsify program e with respect to the postcondition $\mathcal{R}_{\text{post}}$ on the symbolic trace produced by e and show that the falsification is sound. Consider an execution that is recorded by a reduction from the initial symbolic state to some final symbolic state, $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e) \hookrightarrow^*_{\mathcal{D}} (\Phi, \Pi, \mathcal{R}_{\text{cont}}, v)$. This execution is falsified by $\mathcal{R}_{\text{post}}$ if the final state is reachable, i.e., isSat (Φ, Π), and its continuation effect is not nullable, i.e., $\neg\nu(\mathcal{R}_{\text{cont}})$. The soundness of the falsification relies on two key properties of a derivative-based semantics: ① the continuation effect is properly updated to denote future traces that are safe to produce as the execution continues, as established by Lemma 5.14.

Lemma 5.14. If $\mathcal{R}_{\text{cont}}{=}\mathrm{D}_{\Pi}\mathcal{R}_{\text{post}}$ and $(\Phi, \Pi, \mathcal{R}_{\text{cont}}, e) \hookrightarrow_{\mathcal{D}} (\Phi', \Pi', \mathcal{R}'_{\text{cont}}, e')$ then $\mathcal{R}'_{\text{cont}}{=}\mathrm{D}_{\Pi'}\mathcal{R}_{\text{post}}$.

That is, $\mathcal{R}_{\text{cont}}$ in the final symbolic state, given that Π records past events, correctly predicts future events to be produced in compliance with the postcondition $\mathcal{R}_{\text{post}}$, i.e., $\mathcal{R}_{\text{cont}} = \mathrm{D}_{\Pi}\mathcal{R}_{\text{post}}$. Then $\neg\nu(\mathcal{R}_{\text{cont}})$ suggests that without new events being produced, Π fails to comply with $\mathcal{R}_{\text{post}}$, i.e., $\Pi \sqsubseteq \neg\mathcal{R}_{\text{post}}$ by Corollary 5.8. Because the execution stops at value v and no more events are to be produced, the execution is indeed falsified by $\mathcal{R}_{\text{post}}$. ② Execution, including a falsified one, underapproximate those of the non-derivative based naïve semantics, as established by Lemma 5.15.

Lemma 5.15. If $\Pi \sqsubseteq \mathcal{R}_{\text{curr}}$ and $(\Phi, \Pi, \mathcal{R}_{\text{cont}}, e) \hookrightarrow_{\mathcal{D}} (\Phi', \Pi', \mathcal{R}'_{\text{cont}}, e')$ then there exists $\mathcal{R}'_{\text{curr}}$ such that $\Pi' \sqsubseteq \mathcal{R}'_{\text{curr}}$ and $(\Phi, \mathcal{R}_{\text{curr}}, e) \hookrightarrow (\Phi', \mathcal{R}'_{\text{curr}}, e')$.

That is, there exists an execution paths $(\top, \varepsilon, e) \hookrightarrow^* (\Phi, \mathcal{R}_{\text{curr}}, v)$ in the naïve semantics such that $\Pi \sqsubseteq \mathcal{R}_{\text{curr}}$. As all traces denoted by Π fail to comply with $\mathcal{R}_{\text{post}}$, there exist some trace in $\mathcal{R}_{\text{curr}}$

that fails to comply with $\mathcal{R}_{\text{post}}$. We conclude with Theorem 5.16, establishing that given a falsified execution in derivative-based semantics, there exists a corresponding execution in naive semantics that overapproximates this execution and thus can also be falsified.

Theorem 5.16 (Soundness of $\hookrightarrow^*_{\mathcal{D}}$)**.** Assume isSat (Φ, Π). If $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e) \hookrightarrow^*_{\mathcal{D}} (\Phi, \Pi, \mathcal{R}_{\text{cont}}, v)$ and $\neg\nu(\mathcal{R}_{\text{cont}})$ then e is falsified against $\mathcal{R}_{\text{post}}$.

Proof sketch.

(1) First, the single-step reduction in Lemma 5.14 can be extended to multi-step. Since $\mathcal{R}_{\text{post}} = \mathrm{D}_{\varepsilon}\mathcal{R}_{\text{post}}$, we have $\mathcal{R}_{\text{cont}} = \mathrm{D}_{\Pi}\mathcal{R}_{\text{post}}$.
(2) Then, by Corollary 5.8 on $\neg\nu(\mathcal{R}_{\text{cont}})$, we have $\Pi \sqsubseteq \neg\mathcal{R}_{\text{post}}$.
(3) Additionally, the single-step reduction in Lemma 5.15 can also be extended to multi-step. Since $\varepsilon \sqsubseteq \varepsilon$, there exists $\mathcal{R}_{\text{curr}}$ such that $\Pi \sqsubseteq \mathcal{R}_{\text{curr}}$ and $(\top, \varepsilon, e) \hookrightarrow^* (\Phi, \mathcal{R}_{\text{curr}}, v)$.
(4) Now that $\Pi \sqsubseteq \mathcal{R}_{\text{curr}} \wedge \neg\mathcal{R}_{\text{post}}$ and isSat (Φ, Π), we have isSat $\left(\Phi, \mathcal{R}_{\text{curr}} \wedge \neg\mathcal{R}_{\text{post}}\right)$.
(5) Lastly, by Definition 4.7, e is falsified against $\mathcal{R}_{\text{post}}$. □

***Example* 5.17.** Consider the execution path from Example 4.6. Through calls to Nxt.get and Val.get, the execution iterates over two nodes, n_0 and n_1, of the given linked list before finding a node storing element u, i.e., n_1. Then n_1 is removed by linking n_0 to its successor i.e., n_2. Following Examples 5.11 and 5.13, the execution before the removal can be manifested in a derivative-based symbolic state: $(\Phi_{\text{bad}}, \Pi_{\text{prestate}}, \mathcal{R}_{a\not\hookrightarrow b}, \mathsf{Nxt.put}\ n_0\ n_2;\ n_0)$, where the path condition is Φ_{bad} from Example 4.6 and

$$\Pi_{\text{prestate}} \doteq \langle \mathsf{Nxt.put}\ key\ val \mid key{=}a{=}n_1 \wedge val{=}b{=}n_2\rangle \cdot \langle \mathsf{Val.put}\ n_0\ u_0\rangle \cdot \langle \mathsf{Nxt.put}\ n_0\ n_1\rangle \cdot \langle \mathsf{Val.put}\ n_1\ u_1\rangle$$
$$\cdot \langle u_0 \leftarrow \mathsf{Val.get}\ n_0\rangle \cdot \langle n_1 \leftarrow \mathsf{Nxt.get}\ n_0\rangle \cdot \langle u_1 \leftarrow \mathsf{Val.get}\ n_1\rangle \cdot \langle u_2 \leftarrow \mathsf{Nxt.get}\ n_2\rangle$$

To relate the event $\langle \mathsf{Nxt.put}\ n_0\ n_2\rangle$ with $\mathcal{R}_{a\not\hookrightarrow b}$, we consider $\mathcal{R}_{a\not\hookrightarrow b}$'s next event $\langle \mathsf{Nxt.put}\ \not\in b\rangle$, leading to a symbolic derivative of $\varnothing$ as shown in Example 5.7. Hence, the conjunction between $\langle \mathsf{Nxt.put}\ n_0\ n_2\rangle$ and $\langle \mathsf{Nxt.put}\ \not\in b\rangle$ witnesses this relation and is appended to the context Π_{prestate}. The symbolic state becomes: $(\Phi_{\text{bad}}, \Pi_{\text{bad}}, \varnothing, n_0)$, where

$$\Pi_{\text{bad}} \doteq \Pi_{\text{prestate}} \cdot \langle \mathsf{Nxt.put}\ key\ val \mid key = n_0 \neq a \wedge val = n_2 = b\rangle$$

$a = n_1$ and $b = n_2$ witnesses the reachability of the final symbolic state. In combination with $\neg\nu(\varnothing)$, the execution is falsified. In fact, this execution underapproximates the execution shown in Example 4.6, i.e., $\Pi_{\text{bad}} \sqsubseteq \mathcal{R}_{\text{bad}}$, which could have been falsified but proves too costly using naive semantics. □

Furthermore, this refined semantics guarantees completeness with respect to falsification. Consider an execution in the naïve semantics that is manifested by a reduction from the initial symbolic state to some final symbolic state, $(\top, \varepsilon, e) \hookrightarrow^* (\Phi, \mathcal{R}_{\text{curr}}, v)$. According to Definition 4.7, the execution is falsified with respect to the postcondition $\mathcal{R}_{\text{post}}$ as long as isSat $\left(\Phi, \mathcal{R}_{\text{curr}} \wedge \neg\mathcal{R}_{\text{post}}\right)$ holds. Looking backward from the final state, it is sufficient to falsify the execution if there exists some underapproximation of the execution, encapsulated by a symbolic trace $\Pi_{\text{curr}} \sqsubseteq \mathcal{R}_{\text{curr}}$, and some prefix $\Pi_{\text{prefix}} \sqsubseteq \neg\mathcal{R}_{\text{post}}$ such that $\Pi_{\text{curr}} \wedge \Pi_{\text{prefix}}$ represents a viable execution. Effectively, all compatible pairs of symbolic paths $\Pi_{\text{curr}} \sqsubseteq \mathcal{R}_{\text{curr}}$ and prefixes Π_{prefix} of $\mathcal{R}_{\text{post}}$ are exhaustively explored by executions in a derivative-based semantics. Lemma 5.18 establishes this exhaustiveness on each reduction step.

Lemma 5.18. Given a safety property $\mathcal{R}_{\text{post}}$. If $(\Phi, \mathcal{R}_{\text{curr}}, e) \hookrightarrow (\Phi', \mathcal{R}'_{\text{curr}}, e')$ then for all $\Pi'_{\text{curr}} \sqsubseteq \mathcal{R}'_{\text{curr}}$, prefix Π'_{prefix} of $\mathcal{R}_{\text{post}}$, and $\Pi' \equiv \Pi'_{\text{curr}} \wedge \Pi'_{\text{prefix}}$, there exists $\Pi_{\text{curr}} \sqsubseteq \mathcal{R}_{\text{curr}}$, prefix Π_{prefix} of $\mathcal{R}_{\text{post}}$, and $\Pi \equiv \Pi_{\text{curr}} \wedge \Pi_{\text{prefix}}$ such that $(\Phi, \Pi, \mathrm{D}_{\Pi_{\text{prefix}}}\mathcal{R}_{\text{post}}, e) \hookrightarrow_{\mathcal{D}} (\Phi', \Pi', \mathrm{D}_{\Pi'_{\text{prefix}}}\mathcal{R}_{\text{post}}, e')$.

As a result, Theorem 5.19 establishes that given a falsified execution manifested using the naïve semantics, there exists an underapproximating execution in a derivative-based semantics that can also be falsified.

Theorem 5.19 (Completeness of $\hookrightarrow^*_{\mathcal{D}}$)**.** If $(\top, \varepsilon, e) \hookrightarrow^* (\Phi, \mathcal{R}_{\text{curr}}, v)$ and isSat $(\Phi, \mathcal{R}_{\text{curr}} \wedge \neg\mathcal{R}_{\text{post}})$ then there exists $\Pi \sqsubseteq \mathcal{R}_{\text{curr}}$ and $\neg\nu(\mathcal{R}_{\text{cont}})$ such that $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e) \hookrightarrow^*_{\mathcal{D}} (\Phi, \Pi, \mathcal{R}_{\text{cont}}, v)$ and isSat (Φ, Π).

Proof sketch.

(1) First, by Definition 4.7, there exists Φ and $\mathcal{R}_{\text{curr}}$ such that isSat $(\Phi, \mathcal{R}_{\text{curr}} \wedge \neg\mathcal{R}_{\text{post}})$ and $(\top, \varepsilon, e) \hookrightarrow^* (\Phi, \mathcal{R}_{\text{curr}}, v)$.
(2) Then, let $\Pi_{\text{curr}} \sqsubseteq \mathcal{R}_{\text{curr}}$ and $\Pi_{\text{prefix}} \sqsubseteq \neg\mathcal{R}_{\text{post}}$ such that isSat $(\Phi, \Pi_{\text{curr}} \wedge \Pi_{\text{prefix}})$.
(3) Furthermore, the single-step reduction in Lemma 5.18 can be extended to multi-step. As a result, $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e) \hookrightarrow^*_{\mathcal{D}} (\Phi, \Pi_{\text{curr}} \wedge \Pi_{\text{prefix}}, \mathrm{D}_{\Pi_{\text{prefix}}} \mathcal{R}_{\text{post}}, v)$.
(4) Lastly, we have $\neg\nu(\mathrm{D}_{\Pi_{\text{prefix}}} \mathcal{R}_{\text{post}})$ from $\Pi_{\text{prefix}} \sqsubseteq \neg\mathcal{R}_{\text{post}}$. □

The completeness argument requires the symbolic execution to exhaustively relate the events produced during execution and the safe events required by the postcondition. In the hope of finding a falsified execution at the earliest, symbolic derivative enables strategic exploration of this relationship during the symbolic execution. Consider an unfinished execution $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e_0) \hookrightarrow^*_{\mathcal{D}} (\Phi, \Pi, \mathcal{R}_{\text{cont}}, e)$. Recall that the continuation effect $\mathcal{R}_{\text{cont}}$ predicts future traces that are safe to produce if we finish the execution from the current symbolic state $(\Phi, \Pi, \mathcal{R}_{\text{cont}}, e)$. Hence, the concatenation of the current symbolic trace Π and the continuation effect $\mathcal{R}_{\text{cont}}$ gives an optimistic overapproximation of the safe behavior of the execution when finished. Then $\neg$isSat $(\Phi, \Pi \cdot \mathcal{R}_{\text{cont}})$ essentially states that all behavior is unsafe following this execution. Therefore, without finishing the execution, we may determine it is falsified. In theory, we also require that the execution can be finished in a satisfiable state, as stated in Theorem 5.20.

Theorem 5.20 (Soundness of $\varnothing$)**.** Assume $(\Phi, \Pi, \mathcal{R}_{\text{cont}}, e) \hookrightarrow^*_{\mathcal{D}} (\Phi', \Pi', \mathcal{R}'_{\text{cont}}, v)$ and isSat (Φ', Π'). If $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e_0) \hookrightarrow^*_{\mathcal{D}} (\Phi, \Pi, \mathcal{R}_{\text{cont}}, e)$ and $\neg$isSat $(\Phi, \Pi \cdot \mathcal{R}_{\text{cont}})$ then e is falsified against $\mathcal{R}_{\text{post}}$.

Proof sketch.

(1) First, by transitivity of $\hookrightarrow^*_{\mathcal{D}}$, $(\top, \varepsilon, \mathcal{R}_{\text{post}}, e_0) \hookrightarrow^*_{\mathcal{D}} (\Phi', \Pi', \mathcal{R}'_{\text{cont}}, v)$.
(2) Then, by Theorem 5.16, it is sufficient to prove $\neg\nu(\mathcal{R}'_{\text{cont}})$.
(3) By multi-step variant of Lemma 5.14 on $\mathcal{R}_{\text{cont}} = \mathrm{D}_{\Pi}(\Pi \cdot \mathcal{R}_{\text{cont}})$, $\mathcal{R}'_{\text{cont}} = \mathrm{D}_{\Pi'}(\Pi \cdot \mathcal{R}_{\text{cont}})$.
(4) $\neg$isSat $(\Phi, \Pi \cdot \mathcal{R}_{\text{cont}})$ suggests that $\Pi \cdot \mathcal{R}_{\text{cont}}$ is equivalent to $\varnothing$ under the path condition Φ or its refined path condition Φ'. So does its derivative $\mathcal{R}'_{\text{cont}}$. We have $\neg\nu(\mathcal{R}'_{\text{cont}})$. □

However, in practice, as long as the current symbolic state is satisfiable, i.e., isSat (Φ, Π), it is safe to assume that the execution can be finished in a satisfiable symbolic state, which in turn witnesses the falsification. Another practical concern is that checking $\neg$isSat $(\Phi, \Pi \cdot \mathcal{R}_{\text{cont}})$ can be expensive as discussed in Section 4. Instead, we check whether $\mathcal{R}_{\text{cont}}$ is *syntactically* equal to $\varnothing$, which implies $\neg$isSat $(\Phi, \Pi \cdot \mathcal{R}_{\text{cont}})$. If not, we continue the execution without compromising soundness. With a standard set of rewriting rules, e.g., $\varnothing \vee \mathcal{R} \equiv \varnothing$, the syntactic approach is effective in falsifying unfinished execution for programs considered in Section 7. In fact, Example 5.17 is such a case – had `remove` not stopped at the first node found to store the given element, we can still conclude that the execution is falsified without needing to finish iterating over the remaining linked list.

Intuitively, we exploit the existence of a dead state in the automaton associated with the post-condition $\mathcal{R}_{\text{post}}$. As events produced during symbolic execution are related to transitions in the automaton, it is sufficient to falsify a execution if the events produced can be related to transitions in the automaton that leads to a dead state. This is similar to the recognition of a string

in a deterministic automaton, where it is sufficient to determine the string cannot be accepted if a character causes the automata to enter a dead state. However, each event produced may still nondeterministically be related to transtions from the current state in the postcondition automaton.

We would like to further exploit the structure of the postcondition automaton by *actively* looking for a dead state. Again we consider the symbolic state of an unfinished execution, its continuation symbolic derivative $\mathcal{R}_{\mathsf{cont}}$ is a symbolic derivative of the postcondition $\mathcal{R}_{\mathsf{post}}$ and thus $\mathcal{R}_{\mathsf{cont}}$ represents a state in the automaton associated with $\mathcal{R}_{\mathsf{post}}$. The minimal distance of the state denoted by $\mathcal{R}_{\mathsf{cont}}$ to a dead state gives us a lower bound on the number of events that the current execution needs to produce in order to be falsified. It is also the minimal length of $\mathcal{R}_{\mathsf{cont}}$'s prefixes such that the derivative over them denotes a dead state, i.e., $\mathsf{DistToDead}(\mathcal{R}_{\mathsf{cont}})$, where

$$\mathsf{DistToDead}(\mathcal{R}) \doteq \min_{\mathsf{D}_\Pi \mathcal{R}=\emptyset} |\Pi|$$

When new events Π_{eff} are produced during the execution as in Rule DAppend, among all $\mathcal{R}_{\mathsf{cont}}$'s prefixes Π_{prefix} that are compatible with Π_{eff}, we prioritize relating Π_{eff} with prefixes that brings us closer to a dead state, according to $\mathsf{DistToDead}(\mathsf{D}_{\Pi_{\mathsf{prefix}}} \mathcal{R}_{\mathsf{cont}})$. In practice, we set a cut-off constant to limit the depth of such exploitation.

***Example* 5.21.** Consider a different execution from what is shown in Example 5.17. $\langle \mathsf{Nxt.put}\ a\ \not{b} \rangle$ is also a next event of $\mathcal{R}_{a \nrightarrow b}$ but leads to a symbolic derivative of $\bullet^*$. Correspondingly, the event $\langle \mathsf{Nxt.put}\ key\ val \mid key = n_0 = a \wedge val = n_2 \neq b \rangle$ is appended to the symbolic trace. While the symbolic state happens to becomes unreachable ($n_2 \neq b$ contradicts $n_2 = b$ from Π_{prestate}) and thus can be pruned, it does not have to be the case and nondeterministic time may be spent on this infeasible execution before it is pruned. □

6 Algorithm

In this section, we substitute the declarative components of derivative-based semantics with their algorithmic equivalents, thus demonstrating the derivative-based semantics is a sound and relatively complete procedure for falsification.

First, we show that the reachability check (Definition 4.2) of derivative-based symbolic states, i.e., $\mathsf{isSat}\ (\Phi, \Pi)$, can be straightforwardly discharged to SMT queries like conventional symbolic execution techniques. Intuitively, since a symbolic path Π is a sequence of symbolic events, we would like to collect constraints from each symbolic event. The constraint of an atomic symbolic event can be built as:

$$\mathsf{constr}(\langle x_{\mathsf{ret}} \leftarrow \mathsf{f}\ \overline{x_{\mathsf{arg}}} \mid \phi \rangle) = [\overline{x_{\mathsf{arg}} \mapsto \hat{x_{\mathsf{arg}}}}, x_{\mathsf{ret}} \mapsto \hat{x_{\mathsf{ret}}}]\phi \quad \text{for fresh } \overline{\hat{x_{\mathsf{arg}}}} \text{ and } \hat{x_{\mathsf{ret}}}$$

To facilitate constraint collection, we give a stratified representation of symbolic events ℓ as a disjunction of atomic symbolic events associated with *disjoint* effectful functions:

$$\ell \doteq \cdots \parallel \langle x_{\mathsf{ret}} \leftarrow \mathsf{f}\ \overline{x_{\mathsf{arg}}} \mid \phi \rangle \parallel \ldots \qquad \text{such that} \quad [\![\ell]\!] = \bigcup_{\langle x_{\mathsf{ret}} \leftarrow \mathsf{f}\ \overline{x_{\mathsf{arg}}} \mid \phi \rangle \in \ell} [\![\langle x_{\mathsf{ret}} \leftarrow \mathsf{f}\ \overline{x_{\mathsf{arg}}} \mid \phi \rangle]\!]$$

Since the effectful functions f associated with the disjuncts in ℓ are different, the constraint of a symbolic event ℓ is simply the disjunction of constraints from ℓ 's atomic symbolic events, and the constraint of a symbolic path Π is the conjunction of constraints from Π's symbolic events:

$$\mathsf{constr}(\varepsilon) = \top \qquad \mathsf{constr}(\ell) = \bigvee_{\langle \mathsf{f} \mid \phi \rangle \in \ell} \mathsf{constr}(\langle \mathsf{f} \mid \phi \rangle) \qquad \mathsf{constr}(\Pi_1 \cdot \Pi_2) = \mathsf{constr}(\Pi_1) \wedge \mathsf{constr}(\Pi_2)$$

It immediately follows that, as established by Corollary 6.1, the reachability of a symbolic state can be determined by the satisfiabiliy of the conjunction between its path condition Φ and the constraints from its current symbolic path Π.

Corollary 6.1. $\mathsf{isSat}\ (\Phi, \Pi)$ iff $\sigma \in [\![\Phi \wedge \mathsf{constr}(\Pi)]\!]$.

In response to the stratified representation of symbolic events, we discharge their boolean connectives using Definition 6.2, which was part of the syntax in Section 4.

Definition 6.2 (Events Algebra)**.** The boolean operations on symbolic events can be defined as:

$$\backsim(\|_i \langle \mathsf{f}_i \mid \phi_i\rangle) = (\|_i \langle \mathsf{f}_i \mid \neg\phi_i\rangle) \parallel (\|_{\mathsf{g}\in\Delta/\overline{\mathsf{f}_i}} \langle \mathsf{g}\rangle)$$
$$(\|_i \langle \mathsf{f}_i \mid \phi_i\rangle) \sqcap (\|_j \langle \mathsf{g}_j \mid \psi_j\rangle) = \|_{\mathsf{f}_i=\mathsf{g}_j} \langle \mathsf{f}_i \mid \phi_i \wedge \psi_j\rangle$$
$$(\|_i \langle \mathsf{f}_i \mid \phi_i\rangle) \sqcup (\|_j \langle \mathsf{g}_j \mid \psi_j\rangle) = (\|_{\mathsf{f}_i=\mathsf{g}_j} \langle \mathsf{f}_i \mid \phi_i \vee \psi_j\rangle) \parallel (\|_{\mathsf{f}_i\notin\overline{\mathsf{g}_j}} \langle \mathsf{f}_i \mid \phi_i\rangle) \parallel (\|_{\mathsf{g}_j\notin\overline{\mathsf{f}_i}} \langle \mathsf{g}_j \mid \psi_j\rangle)$$

Again, all atomic symbolic events in ℓ are associated with different effectful functions and thus are disjoint. The negation of ℓ includes atomic symbolic events from ℓ with their qualifiers negated and atomic symbolic events out of ℓ with $\top$ qualifier. The conjunction of ℓ_1 and ℓ_2 includes atomic symbolic events included by both ℓ_1 and ℓ_2 with the qualifier being their conjunctions. The disjunction of ℓ_1 and ℓ_2 includes atomic symbolic events included by both ℓ_1 and ℓ_2 with the qualifier being their disjunctions, as well as atomic symbolic events included only in ℓ_1 or ℓ_2. Definition 6.2 preserves the disjointness requirement in the result and is consistent with the denotation $[\![\ell]\!]$ above.

Before providing algorithms for computing prefixes and symbolic derivatives, we first demonstrate a procedure for finding next events of a given SRE $\mathcal{R}$ by rediscovering the notion of "next literals" presented in [Keil and Thiemann 2014]. For $\varnothing$ and ε, their next event can only be bottom. For ℓ, its next event is simply ℓ itself. For $\mathcal{R}^*$, its next events are the same as those of $\mathcal{R}$. For $\neg\mathcal{R}$, its next events include those of $\mathcal{R}$ and the complement of their disjunction. For $\mathcal{R}_1 \wedge \mathcal{R}_2$, its next events includes the conjunction of events included in both $\mathcal{R}_1$ and $\mathcal{R}_2$. For $\mathcal{R}_1 \vee \mathcal{R}_2$, its next events includes not only the conjunction of events included in both $\mathcal{R}_1$ and $\mathcal{R}_2$, but also the conjunction of each event from $\mathcal{R}_1$ and the negation of the disjunction of $\mathcal{R}_2$'s next events, and vice versa, defined as a join operation $\bowtie$ between two sets of events. As a result, the disjunction of $\mathcal{R}_1 \vee \mathcal{R}_2$'s next events is equivalent to the disjunction of $\mathcal{R}_1$'s and $\mathcal{R}_2$'s. For $\mathcal{R}_1 \cdot \mathcal{R}_2$, its next events are determined by the join of those of $\mathcal{R}_1$ and those of $\mathcal{R}_2$ if $\mathcal{R}_1$ is nullable. Otherwise, its next events includes only those of $\mathcal{R}_2$.

Definition 6.3 (Admissible Next Events)**.** The set $\mathfrak{L}$ of events admissible to $\mathcal{R}$ can be computed as:

$$\mathsf{next}(\varnothing) = \mathsf{next}(\varepsilon) = \{\bot\} \qquad \mathsf{next}(\ell) = \{\ell\} \qquad \mathsf{next}(\mathcal{R}^*) = \mathsf{next}(\mathcal{R})$$
$$\mathsf{next}(\mathcal{R}_1 \cdot \mathcal{R}_2) = \begin{cases} \mathsf{next}(\mathcal{R}_1) \bowtie \mathsf{next}(\mathcal{R}_2) & \nu(\mathcal{R}_1) \\ \mathsf{next}(\mathcal{R}_1) & \text{otherwise} \end{cases} \qquad \mathsf{next}(\neg\mathcal{R}) = \mathsf{next}(\mathcal{R}) \cup \left\{\mathsf{next}(\mathcal{R})^{\complement}\right\}$$
$$\mathsf{next}(\mathcal{R}_1 \wedge \mathcal{R}_2) = \mathsf{next}(\mathcal{R}_1) \sqcap \mathsf{next}(\mathcal{R}_2) \qquad \mathsf{next}(\mathcal{R}_1 \vee \mathcal{R}_2) = \mathsf{next}(\mathcal{R}_1) \bowtie \mathsf{next}(\mathcal{R}_2)$$

where the dual of an event set $\mathfrak{L}$ is $\mathfrak{L}^{\complement} \doteq \backsim \bigsqcup_{\ell\in\mathfrak{L}} \ell$ and the join of two event sets $\mathfrak{L}_1$ and $\mathfrak{L}_2$ is $\mathfrak{L}_1 \bowtie \mathfrak{L}_2 \doteq \{\ell_1 \sqcap \ell_2, \ell_1 \sqcap \mathfrak{L}_2^{\complement}, \mathfrak{L}_1^{\complement} \sqcap \ell_2 \mid \ell_1 \in \mathfrak{L}_1, \ell_2 \in \mathfrak{L}_2\}$.

Definition 6.3 provides such a next operation such that each symbolic event $\ell \in \mathsf{next}(\mathcal{R})$ is a singleton prefix of $\mathcal{R}$ (Definition 5.4). Due to the negation rule, the disjunction of $\mathsf{next}(\mathcal{R})$ overapproximates the set of events admissible to $\mathcal{R}$. Then the negation of this disjunction, i.e., $\mathsf{next}(\mathcal{R})^{\complement}$ is also a next event of $\mathcal{R}$, the derivative over which is $\varnothing$. $\mathsf{next}(\mathcal{R}) \cup \{\mathsf{next}(\mathcal{R})^{\complement}\}$ gives us a set of symbolic events that covers the entire space of possible events and are all amenable to symbolic derivative computation of $\mathcal{R}$. Now, the symbolic derivative of $\mathcal{R}$ over its next events can be computed inductively in a similar fashion to Section 3 by the following lemma:

Lemma 6.4. Given $\mathcal{R}$ and its prefix Π, the symbolic derivative $\mathsf{D}_\Pi\mathcal{R}$ can be computed via:

$$\mathsf{D}_\varepsilon\mathcal{R} = \mathcal{R} \qquad \mathsf{D}_{\Pi_1\cdot\Pi_2}\mathcal{R} = \mathsf{D}_{\Pi_2}\mathsf{D}_{\Pi_1}\mathcal{R} \qquad \mathsf{D}_\ell\varnothing = \mathsf{D}_\ell\varepsilon = \varnothing \qquad \mathsf{D}_\ell(\mathcal{R}^*) = \mathsf{D}_\ell\mathcal{R} \cdot \mathcal{R}^*$$
$$\mathsf{D}_{\ell'}\ell = \begin{cases} \varepsilon & \ell' \sqsubseteq \ell \\ \varnothing & \ell' \sqsubseteq \backsim\ell \end{cases} \qquad \mathsf{D}_\ell(\mathcal{R}_1 \cdot \mathcal{R}_2) = \begin{cases} (\mathsf{D}_\ell\mathcal{R}_1 \cdot \mathcal{R}_2) \vee \mathsf{D}_\ell\mathcal{R}_2 & \nu(\mathcal{R}_1) \\ \mathsf{D}_\ell\mathcal{R}_1 \cdot \mathcal{R}_2 & \text{otherwise} \end{cases}$$
$$\mathsf{D}_\ell(\neg\mathcal{R}) = \neg\mathsf{D}_\ell\mathcal{R} \qquad \mathsf{D}_\ell(\mathcal{R}_1 \wedge \mathcal{R}_2) = \mathsf{D}_\ell\mathcal{R}_1 \wedge \mathsf{D}_\ell\mathcal{R}_2 \qquad \mathsf{D}_\ell(\mathcal{R}_1 \vee \mathcal{R}_2) = \mathsf{D}_\ell\mathcal{R}_1 \vee \mathsf{D}_\ell\mathcal{R}_2$$

The main difference is, when computing the symbolic derivative of a symbolic event ℓ over another ℓ', we need to perform an inclusion check between them. If all events denoted by ℓ' are included by ℓ, then the symbolic derivative is ε. If all events denoted by ℓ' are not included by ℓ, then the symbolic derivative is $\varnothing$. Since ℓ' is guaranteed to be a singleton prefix of ℓ, it is impossible that some events denoted by ℓ' are included by ℓ while some are excluded. Thus, checking whether $\ell' \sqsubseteq \ell$ is sufficient. The inclusion check essentially involves checking the validity of $\mathsf{constr}(\backsim \ell' \sqcup \ell)$, which is well-suited for SMT solving.

Using the next operation and the computation of symbolic derivatives over symbolic events, we may enumerate prefixes of arbitrary length from an SRE $\mathcal{R}$ along with their corresponding symbolic derivatives by following the rules in Fig. 7. Rule Pfx-ε states that ε is a prefix of $\mathcal{R}$ and the corresponding symbolic derivative is $\mathcal{R}$ itself. Rule Pfx-ℓ states that all next events of $\mathcal{R}$ is a prefix of $\mathcal{R}$. Rule Pfx-$\cdot$ states that given any prefix Π_1 of $\mathcal{R}$ along with the corresponding symbolic derivative $\mathcal{R}_1$, and any prefix Π_2 of $\mathcal{R}_1$ along with the corresponding symbolic derivative $\mathcal{R}_2$, the concatenation of Π_1 and Π_2 is still a prefix of $\mathcal{R}$ with $\mathcal{R}_2$ being the corresponding symbolic derivative. Intuitively,

$$\frac{}{(\varepsilon, \mathcal{R}) \triangleright \mathcal{R}}\ \textsc{Pfx}\text{-}\varepsilon \qquad \frac{\ell \in \mathsf{next}(\mathcal{R}) \cup \left\{\mathsf{next}(\mathcal{R})^{\complement}\right\}}{(\ell, \mathrm{D}_\ell \mathcal{R}) \triangleright \mathcal{R}}\ \textsc{Pfx}\text{-}\ell \qquad \frac{(\Pi_1, \mathcal{R}_1) \triangleright \mathcal{R} \qquad (\Pi_2, \mathcal{R}_2) \triangleright \mathcal{R}_1}{(\Pi_1 \cdot \Pi_2, \mathcal{R}_2) \triangleright \mathcal{R}}\ \textsc{Pfx}\text{-}\cdot$$

Fig. 7. Enumerate prefixes of a given $\mathcal{R}$ and compute their symbolic derivatives.

these rules allow us to construct a deterministic SFA that accepts the same set of traces as $\mathcal{R}$ and all paths in the SFA are enumerated, including those that lead to dead states. As established by Lemma 6.5, each enumerated prefix is indeed a prefix of $\mathcal{R}$.

Lemma 6.5 (Soundness of Prefix Enumeration)**.** If $(\Pi, \mathcal{R}') \triangleright \mathcal{R}$ then $\mathcal{R}' = \mathrm{D}_\Pi \mathcal{R}$.

A completeness result then states that all paths in the SFA can be enumerated. As an SRE $\mathcal{R}$ may have different SFA representations, a prefix Π of $\mathcal{R}$ may not correspond to a path in the SFA constructed by our prefix enumeration. However, it is guaranteed that, as established by Lemma 6.6, a set of prefixes, i.e., a set of paths in the SFA, can be found by enumeration such that their disjunction includes all traces denoted by such a prefix Π.

Lemma 6.6 (Completeness of Prefix Enumeration)**.** If $\mathcal{R}' = \mathrm{D}_\Pi \mathcal{R}$ then there exists $\overline{\Pi_i}^i$ such that $(\Pi_i, \mathcal{R}') \triangleright \mathcal{R}$ for all i and $\Pi \sqsubseteq \bigvee_i \Pi_i$.

Sampling symbolic traces Π from a given SRE $\mathcal{R}$ is a special case of enumerating prefixes whose symbolic derivative is nullable, as shown in Corollary 5.8. Intuitively, the sampled symbolic traces correspond to the paths that lead to an accepting state in the SFA. For the purpose of sampling symbolic traces, we may ignore paths that lead to a dead state without compromising the completeness of sampling. That is, when applying Rule PFX-ℓ for trace sampling, we ignore $\mathsf{next}(\mathcal{R})^{\complement}$, whose corresponding symbolic derivative is always $\varnothing$.

Lastly, we show how to relate symbolic traces of the same length by computing their conjunction. The following rules effectively perform pairwise conjunction between symbolic events (Definition 6.2) from two symbolic traces Π_1 and Π_2:

$$\varepsilon \wedge \varepsilon \equiv \varepsilon \qquad \ell_1 \wedge \ell_2 \equiv \ell_1 \sqcap \ell_2 \qquad (\Pi_{11} \cdot \Pi_{12}) \wedge (\Pi_{21} \cdot \Pi_{22}) \equiv (\Pi_{11} \wedge \Pi_{21}) \cdot (\Pi_{12} \wedge \Pi_{22})$$

where $|\Pi_{11}| = |\Pi_{21}|$ and $|\Pi_{12}| = |\Pi_{22}|$. A symbolic trace equivalent to the conjunction of Π_1 and Π_2 is returned following the rules.

To conclude this section, the prefix enumeration algorithm gives us a sound and relatively complete equivalent for the premises of Rules DAdmit and DAppend. By enumerating prefixes in increasing length, minimal traces of events are produced and appended along symbolic execution.

Table 1. Falsification of a variety of safety property violations in ADT implementations.

ADT	Repr. Type	Safety Property	Violation to the Safety Property	Time (s) To Falsify	Speedup over Naïve	Speedup over Verifier
Stack	LinkedList	Elements are stored at unique locations.	Overwrite an existing node when pushing.	0.51	×4.9	×3.2
			Make the linked list circular during concatenation.	0.25	O/M	×13.2
	KVStore	Elements are linked linearly.	Push the new element in the middle of the stack.	1.11	T/O	×4.6
			Concatenate elements to the middle of a stack.	0.94	O/M	×6.5
Queue	LinkedList	Elements are stored at unique locations.	Overwrite an existing node when enqueueing.	0.73	×2.5	×2.7
	Graph	Degrees of vertices are at most one.	Overwrite an existing vertex when enqueueing.	1.75	T/O	×7.4
Set	KVStore	Each key is associated with a distinct value.	Put a duplicated element.	0.87	T/O	×1.4
	Tree	The underlying tree is a binary search tree.	Insert a smaller element to the right subtree.	1.10	×40.7	×11.1
Heap	LinkedList	Elements are stored at unique locations, sorted.	Insert after a node with a larger value.	0.11	×12.9	×13.2
	Tree	Parents are smaller than their children.	Insert a smaller element to the right subtree.	1.00	×2.4	×2.5
Min Set	Set	The cached element has been inserted and is no larger than other elements.	Record the minimum without inserting it.	1.14	×1.3	×1.3
			Insert a new minimum without recording it.	1.32	×9.0	×9.9
	KVStore	The cached element has been put and is no larger than other elements.	Record the minimum without putting it.	0.66	T/O	×4.3
			Overwrite an existing element when putting.	1.95	×10.7	×14.9
Lazy Set	Tree	The underlying tree is a binary search tree.	Insert a smaller element to the right subtree.	1.09	×4.8	×11.5
	Set	The same element is never inserted twice.	Insert a duplicated element.	0.49	×1.2	×1.3
	KVStore	Each key is associated with a distinct value.	Put a duplicated element.	0.88	×49.8	×1.5
DFA	KVStore	Each state is associated with a non-empty list of next states via unique labels.	Put an overlapping transtion with the same label.	0.66	×29.9	×29.9
			A transition is reversed instead of deleted.	1.04	×15.0	×15.2
	Graph	The outgoing edges of each state are labeled by different characters.	Connect two connected nodes with the same label.	0.98	×12.9	×12.8
			Connect two nodes instead of disconnecting them.	0.97	×16.5	×16.5
Connected Graph	LinkedList	Edges (pairs of vertices) are uniquely stored with connected vertices being valid.	Insert a vertex pair twice during initialization.	0.27	T/O	×16.4
			Insert a vertex without ensuring its connectivity.	1.31	O/M	×9.9
			Insert a duplicated vertex pair.	1.44	O/M	×11.2
	Graph	All vertices are connected in the graph.	Create a duplicated edge during initialization.	1.13	×1.7	×1.8
			Create a vertex without ensuring its connectivity.	2.05	×6.0	×6.1
			Disconnect a vertex from the rest of the graph.	2.38	×20.0	×16.2
Colored Graph	Graph	Vertices are colored before being connected to vertices with different colors.	Create an edge between two vertices colored the same.	3.68	T/O	T/O
	KVStore	Each vertex is associated with a list of vertices with different colors.	Put an edge between two vertices with the same color.	7.82	T/O	T/O
Linked List	KVStore	Each node has at most one predecessor.	Put a new predecessor to a node before deleting its old predecessor.	7.03	O/M	T/O

7 Implementation and Evaluation

We have implemented a symbolic execution engine in OCaml based on a derivative-based semantics, called HATch that targets the falsification of OCaml-like ADT implementations that interact with their underlying representation types via API calls. HATch takes as input the implementation of an ADT's method, its behavioral specification, and the behavioral specifications of the underlying representation types, and performs symbolic execution against an execution harness as described in Example 4.4. Symbolic execution is performed in increasing depth of explored execution traces. HATch performs two additional optimizations that are not discussed in Section 6. First, it not only tracks the atomic symbolic events that are included in a symbolic event ℓ but also tracks those that are excluded. This frees us from enumerating all other available APIs when computing the negation. Second, since the prefixes to be enumerated are combined with a given symbolic trace, the enumeration of prefixes is interleaved with a compatibility check against the trace. This interleaving helps avoid enumerating prefixes that are known to be incompatible.

In our evaluation, we consider the following research questions: **Q1.** Can HATch's behavioral specifications effectively capture interesting safety properties? **Q2.** Can HATch's use of symbolic derivatives improve trace exploration for falsification? **Q3.** Can HATch enhance assurance through falsification when verification is challenging?

We evaluate HATch on stateful variants of functional ADTs (see Table 1) drawn from different sources [Miltner et al. 2020; Okasaki 1999; Zhou et al. 2024]. The ADTs we consider are implemented using different effectful representation types (i.e., **Repr. Type** column), including key-value stores, linked lists, sets, trees, and graphs. We introduce artificial bugs in their methods as summarized in the Violation column, and evaluate HATch's capability to falsify these buggy implementations. The next column reports the time HATch takes to falsify the violation.

To demonstrate the effectiveness of symbolic derivatives, we implement a variant of HATch following the description given in Section 4, and report HATch's speedup over this variant. The satisfiability of a path condition Φ and a SRE $\mathcal{A}$ (Definition 4.7) is checked by first replacing logical formulae with the elements from a finite equivalence class. An $\mathcal{A}$ then becomes an ordinary regular expression amenable to SMT solving, whose non-emptiness, along with the satisfiability of the logical formulae, witness its satisfiability. Our results show that without using derivatives, symbolic execution is unable to falsify (1) 7 violations (out of a possible 20) within 60 seconds, resulting in timeouts (T/O) due to excessive calls to the SMT solver, and (2) 5 violations under an 8 GB memory limit, leading to out-of-memory errors (O/M) due to the complexity involved in constructing equivalence classes.

To demonstrate the effectiveness of HATch against a verification procedure, we compare its performance with recent work on representation invariant verification [Zhou et al. 2024], and report its speedup over that verifier in terms of the time taken to identify a violation. Overall, HATch demonstrates significant improvement in performance, measured in orders of magnitude, compared to both the non-derivative aware engine and the verifier. It is noteworthy that it is able to efficiently handle two challenging ADTs, colored graphs and linked lists, falsifying their buggy implementation in a small (< 8) number of seconds, whereas the other approaches are unable to provide any result within the given resource bound (60 seconds, 8 GB).

8 Related Work

Symbolic Execution for Functional Languages. While symbolic execution has been typically used in the context of imperative languages for bug finding [Baldoni et al. 2018], there have been recent efforts that apply SE in a functional programming setting. [Xu et al. 2009] and [Nguyen et al. 2014] use SE to verify contracts in Haskell and pure Racket, respectively, with [Nguyễn et al. 2017] extending contract verification to handle Racket programs with mutable state. SE has also been used for underapproximate reasoning to identify weak library specifications that lead to type-checking failures of client programs in LiquidHaskell [Hallahan et al. 2019]. Our goals in this paper are substantially different, focused on falsifying safety properties of functional ADTs that interact with opaque and effectful libraries.

Temporal Verification. Model checking has been applied for software verification against temporal specifications, e.g., LTL and CTL. Early work shows how transition systems can be extracted from programs to abstract their behavior in a form amenable for automata-based inclusion checking to validate temporal specifications [Clarke et al. 1994]. More recently, type and effect systems have been proposed to infer a conservative overapproximation of effects produced during execution of higher-order functional programs [Skalka and Smith 2004]. The granularity of effects inferred has been improved by regarding past effects as a handle for reasoning about hidden states [Nanjo et al. 2018; Sekiyama and Unno 2023; Song et al. 2022]. The use of SFAs as a basis for specification and

verification has also been explored in [Zhou et al. 2024] that introduces Hoare Automata Types (HATs) as a new refinement type abstraction for verifying programs against effectful trace-based temporal specifications. In contrast to these efforts, HATch considers this style of specification in the context of underapproximate reasoning, exploiting the structure of SFAs to enable efficient falsification.

Derivatives of Regular Expressions. The classic notion of derivatives of regular expressions provides a lazy and algebraic approach for constructing automaton-based recognizers from given regular expressions, effectively relating automaton states to their regular expression counterparts. Brzozowski's derivative [Brzozowski 1964] initially introduced this concept for constructing deterministic finite automata, followed by Antimirov's partial derivative [Antimirov 1995] for nondeterministic finite automata, later extended to handle complement and intersection operations [Caron et al. 2011]. While it is known that the classic derivative approach either overapproximates or underapproximates with predicates in regular expressions, computing "next literals" has been proposed as a remedy [Keil and Thiemann 2014]. Our formulation of symbolic derivatives, while largely inspired by this work, accounts for universally quantified variables in regular expressions, which are ubiquitous in program analysis tasks. However, the "next literal" approach can generate an exponential number of transitions in worst cases. Recent work on *transition regexes* [Stanford et al. 2021] introduces a novel form of symbolic derivatives that embeds potentially exponential choices within nested conditionals, enabling lazy exploration of transitions and algebraic simplification. Incorporating these symbolic derivatives into our symbolic execution engine thus may benefit the reasoning of specifications with richer control structures, presenting a promising avenue for future research.

Dynamic Trace-Based Reasoning. Traces as a form of (in)correctness specification have been widely adopted by dynamic analysis techniques. Various runtime monitoring systems rely on a language of traces [Avgustinov et al. 2007; Chen and Roşu 2007; Goldsmith et al. 2005; Havelund and Roşu 2001; Meredith et al. 2008]. Regular properties over traces are also used to guide path exploration in dynamic symbolic execution [Zhang et al. 2015]. Arbitrary trace predicates are now supported in Racket contracts [Moy and Felleisen 2023]. We leave for future work the exploration of non-regular trace languages amenable for derivative computation to enable the falsification of even richer safety properties.

9 Conclusions

This paper presents a new symbolic execution procedure that integrates trace-based temporal specifications to reason about ADTs that interact with effectful libraries. We demonstrate how to leverage these specifications, specifically their latent SFA representations, to manifest the hidden state maintained by an ADT's underlying representation. More significantly, we introduce the concept of a symbolic derivative, a new encoding of symbolic states that relate admissible specification traces with path exploration decisions, and show how they enable significant efficiency gains by allowing paths that are irrelevant to the falsification of a given safety property to be quickly pruned by a symbolic execution engine. Our ideas provide new insight into how trace-guided specifications can enable effective reachability-based program analyses.

Acknowledgments

We thank the anonymous POPL reviewers for their detailed comments and constructive feedback. We also thank Guannan Wei for stimulating discussions and suggestions on the draft of the paper. This material is based upon work supported by the Defense Advanced Research Projects Agency

(DARPA) and the Naval Information Warfare Center Pacific (NIWC Pacific) under N6600 1-22-C-4027, STR Research with funding from the US government, and the National Science Foundation under CCF-SHF 2321680.

Data-Availability Statement

Our implementation of HATch, the benchmark suite used, and the instructions for reproducing results are available at Yuan et al. [2024a].

References

Valentin Antimirov. 1995. Partial Derivatives of Regular Expressions and Finite Automata Constructions. In *STACS 95 (Lecture Notes in Computer Science)*, Ernst W. Mayr and Claude Puech (Eds.). Springer, Berlin, Heidelberg, 455–466. https://doi.org/10.1007/3-540-59042-0_96

Pavel Avgustinov, Julian Tibble, and Oege de Moor. 2007. Making Trace Monitors Feasible. *SIGPLAN Not.* 42, 10 (Oct. 2007), 589–608. https://doi.org/10.1145/1297105.1297070

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (May 2018), 50:1–50:39. https://doi.org/10.1145/3182657

Suguman Bansal, Yong Li, Lucas M. Tabajara, Moshe Y. Vardi, and Andrew Wells. 2023. Model Checking Strategies from Synthesis over Finite Traces. In *Automated Technology for Verification and Analysis (Lecture Notes in Computer Science)*, Étienne André and Jun Sun (Eds.). Springer Nature Switzerland, Cham, 227–247. https://doi.org/10.1007/978-3-031-45329-8_11

Gerard Berry and Ravi Sethi. 1986. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science* 48 (1986), 117–126. https://doi.org/10.1016/0304-3975(86)90088-5

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. 2011. Partial Derivatives of an Extended Regular Expression. In *Language and Automata Theory and Applications (Lecture Notes in Computer Science)*, Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide (Eds.). Springer, Berlin, Heidelberg, 179–191. https://doi.org/10.1007/978-3-642-21254-3_13

Feng Chen and Grigore Roşu. 2007. Mop: An Efficient and Generic Runtime Verification Framework. *SIGPLAN Not.* 42, 10 (Oct. 2007), 569–588. https://doi.org/10.1145/1297105.1297069

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Newport Beach California USA, 265–278. https://doi.org/10.1145/1950365.1950396

Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems* 16, 5 (Sept. 1994), 1512–1542. https://doi.org/10.1145/186025.186051

Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. *ACM SIGPLAN Notices* 49, 1 (Jan. 2014), 541–553. https://doi.org/10.1145/2578855.2535849

Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 47–67. https://doi.org/10.1007/978-3-319-63387-9_3

Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI '13)*. AAAI Press, Beijing, China, 854–860.

Giuseppe De Giacomo and Moshe Y. Vardi. 2015. Synthesis for LTL and LDL on Finite Traces. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, Buenos Aires, Argentina, 1558–1564.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. 2005. Relational Queries over Program Traces. *SIGPLAN Not.* 40, 10 (Oct. 2005), 385–402. https://doi.org/10.1145/1103845.1094841

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(PLDI 2019). Association for Computing Machinery, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. Association for Computing Machinery, New York, NY, USA, 458–471. https://doi.org/10.1145/174675.178053

Klaus Havelund and Grigore Roşu. 2001. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science* 55, 2 (Oct. 2001), 200–217. https://doi.org/10.1016/S1571-0661(04)00253-1

Matthias Keil and Peter Thiemann. 2014. Symbolic Solving of Extended Regular Expression Inequalities. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.FSTTCS.2014.175*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175

Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Rosu. 2008. Efficient Monitoring of Parametric Context-Free Patterns. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 148–157. https://doi.org/10.1109/ASE.2008.25

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with Derivatives: A Functional Pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ACM, Tokyo Japan, 189–195. https://doi.org/10.1145/2034773.2034801

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3385412.3385967

Cameron Moy and Matthias Felleisen. 2023. Trace Contracts. *Journal of Functional Programming* 33 (Jan. 2023), e14. https://doi.org/10.1017/S0956796823000096

Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, New York, NY, USA, 759–768. https://doi.org/10.1145/3209108.3209204

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/2628136.2628156

Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2017. Soft Contract Verification for Higher-Order Stateful Programs. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 51:1–51:30. https://doi.org/10.1145/3158139

Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Vaughan Pratt. 1991. Action Logic and Pure Induction. In *Logics in AI*, J. van Eijck (Ed.). Springer, Berlin, Heidelberg, 97–120. https://doi.org/10.1007/BFb0018436

Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 71:2079–71:2110. https://doi.org/10.1145/3571264

Christian Skalka and Scott Smith. 2004. History Effects and Verification. In *Programming Languages and Systems*, Wei-Ngan Chin (Ed.). Vol. 3302. Springer Berlin Heidelberg, Berlin, Heidelberg, 107–128. https://doi.org/10.1007/978-3-540-30477-7_8

Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification for Algebraic Effects. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.). Springer Nature Switzerland, Cham, 88–109. https://doi.org/10.1007/978-3-031-21037-2_5

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 620–635. https://doi.org/10.1145/3453483.3454066

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-Order Symbolic Execution via Contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 537–554. https://doi.org/10.1145/2384616.2384655

Margus Veanes. 2013. Applications of Symbolic Finite Automata. In *Implementation and Application of Automata*, Stavros Konstantinidis (Ed.). Springer, Berlin, Heidelberg, 16–23. https://doi.org/10.1007/978-3-642-39274-0_3

Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic Regular Expression Explorer. In *Verification and Validation 2010 Third International Conference on Software Testing*. 498–507. https://doi.org/10.1109/ICST.2010.15

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/1480881.1480889

Yongwei Yuan, Zhe Zhou, Julia Belyakova, and Suresh Jagannathan. 2024a. Artifact for "Derivative-Guided Symbolic Execution". Zenodo. https://doi.org/10.5281/zenodo.13800040

Yongwei Yuan, Zhe Zhou, Julia Belyakova, and Suresh Jagannathan. 2024b. Derivative-Guided Symbolic Execution. https://doi.org/10.48550/arXiv.2411.02716 arXiv:2411.02716

Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 643–653. https://doi.org/10.1109/ICSE.2015.80

Zhe Zhou, Qianchuan Ye, Benjamin Delaware, and Suresh Jagannathan. 2024. A HAT Trick: Automatically Verifying Representation Invariants Using Symbolic Finite Automata. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 1387–1411. https://doi.org/10.1145/3656433

Received 2024-07-11; accepted 2024-11-07