Noname manuscript No.
(will be inserted by the editor)

Symbolic Automata for Representing Big Code

Hila Peleg · Sharon Shoham · Eran Yahav · Hongseok Yang

Received: date / Accepted: date

Abstract Analysis of massive codebases ("big code") presents an opportunity for drawing insights about programming practice and enabling code reuse. One of the main challenges in analyzing big code is finding a representation that captures sufficient semantic information, can be constructed efficiently, and is amenable to meaningful comparison operations. We present a formal framework for representing code in large codebases. In our framework, the semantic descriptor for each code snippet is a *partial temporal specification* that captures the sequences of method invocations on an API. The main idea is to represent partial temporal specifications as symbolic automata—automata where transitions may be labeled by variables, and a variable can be substituted by a letter, a word, or a regular language. Using symbolic automata, we construct an abstract domain for static analysis of big code, capturing both the *partialness* of a specification and the *precision* of a specification. We show interesting relationships between lattice operations of this domain and common operators for manipulating partial temporal specifications, such as building a more informative specification by consolidating two partial specifications, and comparing partial temporal specifications.

A preliminary version of this paper appeared in [20].

H. Peleg
Tel Aviv University, Israel,
E-mail: hila.peleg@gmail.com

S. Shoham
Tel Aviv-Yaffo Academic College, Israel
E-mail: sharon.shoham@gmail.com

E. Yahav
Technion, Israel
E-mail: yahave@gmail.com

H. Yang
University of Oxford, UK
E-mail: hongseok00@gmail.com

1 Introduction

Analysis of massive codebases ("big code") presents an opportunity for drawing insights about programming practice and enabling code reuse. One of the main challenges in analyzing big code is finding a representation that captures sufficient semantic information, can be constructed efficiently, and is amenable to meaningful comparison operations.

In this paper, we present a formal framework for representing big code—a large number of partial programs. The main idea is to extract temporal specifications from code. To analyze big code, it is critical to allow analysis of *code snippets*, i.e., code fragments with unknown parts. A natural approach for capturing gaps in the snippets is to use gaps in the specification. For example, when the code contains an invocation of an unknown method, this approach reflects this fact in the extracted specification as well (we elaborate on this point later). These *partial* specifications serve as a basis for a semantic index of the code snippets. Such partial specifications can be used to construct a probabilistic model of the observed behaviors (similar to the PFSA used in [4, 16] or the language model of [22]).

Technically, we represent *partial temporal specifications* as symbolic automata, where transitions may be labeled by variables representing unknown information. Using symbolic automata, we present an abstract domain for representing big code, and show interesting relationships between the *partialness* and the *precision* of a specification. In this paper, we focus on representation, and operations over symbolic automata, and do not address the probabilistic aspects of the problem.

Representing Partial Specifications using Symbolic Automata We focus on generalized typestate specifications [24, 16]. Such specifications capture legal sequences of method invocations on a given API, and are usually expressed as finite-state automata where a state represents an internal state of the underlying API, and transitions correspond to API method invocations.

Our *symbolic automaton* is conceived in order to represent partial information in specifications. It is a finite-state machine where transitions may be labeled by variables and a variable can be substituted by a letter, a word, or a regular language in a context sensitive manner—when a variable appears in multiple strings accepted by the state machine, it can be replaced by different words in all these strings.

An Abstract Domain for Representing Partial Specifications One challenge for forming an abstract domain with symbolic automata is to find appropriate operations that capture the subtle interplay between the partialness and the precision of a specification. Let us explain this challenge using a preorder over symbolic automata.

When considering non-symbolic automata, we typically use the notion of language inclusion to model "precision"—we can say that an automaton A_1 overapproximates an automaton A_2 when its language includes that of A_2. However, this standard approach is not sufficient for symbolic automata, because the use of variables introduces *partialness* as another dimension for relating the (symbolic) automata. Intuitively, in a preorder over symbolic automata, we would like to capture the notion of a symbolic automaton A_1 being *more complete* than a symbolic automaton A_2 when A_1 has fewer variables that represent unknown information. In Section 4, we describe

an interesting interplay between *precision* and *partialness*, and define a preorder between symbolic automata, which we later use as a basis for an abstract domain of symbolic automata.

Consolidating Partial Specifications After mining a large number of partial specifications from code snippets, it is desirable to combine consistent partial information to yield consolidated temporal specifications. This leads to the question of *combining consistent symbolic automata*. In Section 7, we show how the join operation of our abstract domain leads to an operator for consolidating partial specifications.

Completion of Partial Specifications Having constructed consolidated specifications, we can use symbolic automata as queries for code completion and search. Treating one symbolic automaton as a query being matched against a database (index) of consolidated specifications, we show how to use simulation over symbolic automata to find automata that match the query (Section 5), and how to use *unknown elimination* to find completions of the query automaton (Section 6).

Main Contributions The contributions of this paper are as follows:

- We present a formal framework for the analysis and representation of large scale codebases. The idea is to provide a representation that captures rich semantic information but can be still constructed efficiently and enable comparison, indexing, and other common operations (e.g., completion) of code.
- A central aspect of analyzing big code is the partialness of the code snippets being analyzed. To capture this aspect, we formally define the notion of *partial temporal specification* based on a new notion of *symbolic automata*.
- We explore relationships between partial specifications along two dimensions: (i) *precision* of symbolic automata, a notion that roughly corresponds to containment of non-symbolic automata; and (ii) *partialness* of symbolic automata, a notion that roughly corresponds to an automata having fewer variables, which represent unknown information.
- We present an abstract domain of symbolic automata where operations of the domain correspond to key operators for manipulating partial temporal specifications.
- We define the operations required for algorithms for consolidating two partial specifications expressed in terms of our symbolic automata, for comparing them, and for completing certain partial parts of such specifications.

1.1 Related Work

Semantic Representations of Code There has been a lot of work on semantic representations of code for detecting similarities [11,5,13] and for semantic code search [16]. Using program dependence graphs was shown effective in comparing procedures [12]. Our approach instead focuses on API calls and on handling *partial information*. Recently, David and Yahav presented a technique for measuring similarity between binaries using tracelets [8]. Their tracelets are (partial) sequences of instructions, but can be extended to be tracelets of function calls, similar to the linear case of our partial automata. Raychev et al. [22] present an elegant and efficient semantic represen-

tation based on sequences of calls including call parameters. Their representation is practical and efficient, and generalizes our symbolic automata by including method parameters. However, it uses explicit a priori bounds on the number and length of tracked sequences, and loses the soundness guarantee as a result. They also present a code completion technique that is based on statistical language models. Other semantic notions of code similarity focus on integer programs [18, 19]. In contrast, we focus on programs using library APIs.

Specification Mining Static specification mining techniques (e.g., [23, 16, 3, 27]) have emerged as a way to obtain a succinct description of usage scenarios when working with a library. However, although they demonstrated great practical value, these techniques do not address many interesting and challenging technical questions. In particular, they do not address the question of analyzing a large number of code snippets, where each snippet is (potentially) a *partial* program.

Mishne et al. [16] present a practical framework for static specification mining and query matching based on automata. Their framework imposes restrictions on the structure of automata and could be viewed as a special case of the formal framework introduced in this paper. In contrast to their informal treatment, this paper presents the notion of symbolic automata with an appropriate abstract domain. Shoham et al. [23] use a whole-program analysis to statically analyze clients using a library. Their approach is not applicable in the setting of partial programs and partial specification since they rely on the ability to analyze the complete program for complete alias analysis and for type information.

Many static works on specification mining learn specifications consisting of pairs of events $\langle a, b \rangle$, where a and b are method calls, and do not consider larger automata; Weimer and Necula [26] use a lightweight static analysis to infer such simple specifications from a given codebase. Wasylkowski et al. [25] use an intraprocedural static analysis to automatically mine object usage patterns and identify usage anomalies. Their approach is based on identifying usage patterns, in the restricted form of pairs of events, reflecting the order in which events should be used. Gruska et al. [10] considers limited specifications that are only pairs of events. The work [2] also mines pairs of events in an attempt to mine a partial order between events.

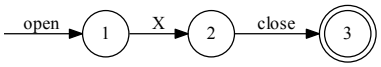
Monperrus et al. [17] attempt to identify missing method calls when using an API by mining a codebase. Their approach deals with identical histories minus k method calls, where histories are modeled as (unordered) sets of method calls. Unlike our approach, it cannot handle incomplete programs, method call sequences where the order is important, and histories that exhibit branching behavior

Many *dynamic* specification mining approaches also extract various forms of temporal specifications (e.g. [6, 4, 14, 27, 29, 7, 15]). Unlike static analysis of partial code snippets, dynamic approaches do not need to handle unknown sequences of events, which are a key ingredient in our approach. On the other hand, because our focus on this paper is on analysis of code snippets, employing dynamic analysis would be extremely challenging.

Word Equations Plandowski [21] solves *word equations*, which define equality conditions on strings with variable portions. Ganesh et al. [9] extend this work with quantifiers and additional predicates (such as length constraints on the assignment

```
1  void process(File f) {
2    f.open();
3    doSomething(f);
4    f.close();
5  }
```

(a)



(b)

Fig. 1 (a) Simple code snippet using `File`. The methods `open` and `close` are API methods, and the method `doSomething` is unknown. (b) Symbolic automaton mined from this snippet. The transition corresponding to `doSomething` is represented using the variable `X`. Transitions corresponding to API methods are labeled with method name.

size). Solving such equations can be thought of as performing unknown elimination on symbolic words. In comparison to our symbolic automata, word equations use variables to represent unknown parts of words rather than automata. Unlike our automata, they cannot capture a branching behavior. In addition, while word equations allow all predicate arguments to have symbolic components, the equation is solved by a completely concrete assignment, disallowing the concept of assigning a symbolic language. More importantly, the assignments considered are context-free. They map variables to words, making the set of solutions define a relation over the set of words. In contrast, in our work assignments are context-sensitive. Namely, the same variable can be assigned different words (or languages) depending on the word that it appears in. As a result, in our case a set of assignments (solutions) does not represent a simple relation on words. Instead, it defines a set of languages, each of which results from one assignment.

2 Overview

We start with an informal overview of our approach by using a simple `File` example.

2.1 Illustrative Example

Consider the example snippet of Fig. 1(a). We would like to extract a temporal specification that describes how this snippet uses the `File` component. The snippet invokes `open` and then an unknown method `doSomething(f)` the code of which is not available as part of the snippet. Finally, it calls `close` on the component. Analyzing this snippet using our approach yields the partial specification of Fig. 1(b). Technically, this is a symbolic automaton, where transitions corresponding to API methods are labeled with method name, and the transition corresponding to the unknown method `doSomething` is labeled with a variable `X`. The use of a variable indicates that some operations may have been invoked on the `File` component between `open` and `close`, but that this operation or sequence of operations is unknown.

Now consider the specifications of Fig. 2, obtained as the result of analyzing similar fragments using the `File` API. Both of these specifications are *more complete* than the specification of Fig. 1(b). In fact, both of these automata do not contain variables, and they represent non-partial temporal specifications. These three separate

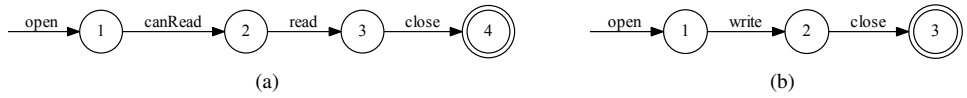


Fig. 2 Automata mined from programs using `File` to (a) read after canRead check; (b) write.

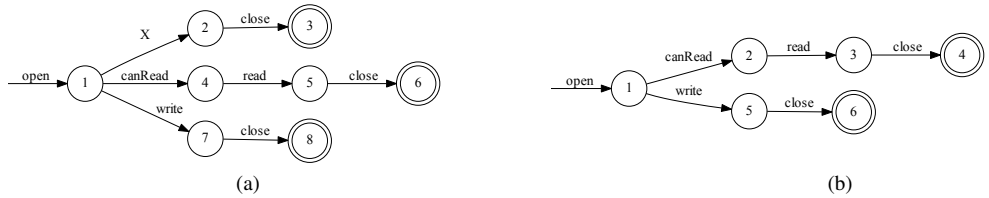


Fig. 3 (a) Automaton resulting from combining all known specifications of the `File` API, and (b) the `File` API specifications after partial paths have been subsumed by more concrete ones.

Fig. 4 (a) Symbolic automaton representing the query for the behavior around the method `read` and (b) the assignment to its symbolic transitions which answers the query.

specifications come from three pieces of code, but all contribute to our knowledge of how the `File` API is used. As such, we would like to be able to compare them to each other and to combine them, and in the process to eliminate as many of the unknowns as possible using other, more complete examples.

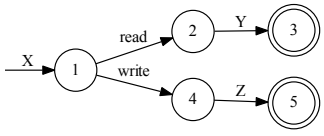
Our first step is to *consolidate* the specifications into a more comprehensive specification, describing as much of the API as possible, while losing no behavior represented by the original specifications.

Next, we would like to *eliminate unknown operations* based on the extra information that the new temporal specification now contains with regard to the full API. For instance, where in Fig. 1 we had no knowledge of what might happen between `open` and `close`, the specification in Fig. 3(a) suggests it might be either `canRead` and `read`, or `write`. Thus, the symbolic placeholder for the unknown operation is now no longer needed, and the path leading through `X` becomes redundant (as shown in Fig. 3(b)).

We may now note that all three original specifications are still *included* in the specification in Fig. 3(b), even after the unknown operation was removed; the concrete paths are fully there, whereas the path with the unknown operation is represented by both the remaining paths.

The ability to find the inclusion of one specification with unknowns within another is useful for performing queries. A user may wish to use the `File` object in order to read, but be unfamiliar with it. He can query the specification, marking any portion he does not know as an unknown operation, as in Fig. 4(a).

As this very partial specification is included in the API's extracted specification (Fig. 3(b)), there will be a match. Furthermore, we can deduce what should replace



(a)

(*,X,read) ↦ open,canRead
(*,X,write) ↦ open
Y ↦ close
Z ↦ close

(b)

Fig. 5 (a) Symbolic automaton representing the query for the behavior around `read` and `write` methods and (b) the assignment with contexts to its symbolic transitions which answers the query.

the symbolic portions of the query. This means the user can get the reply to his query that `X` should be replaced by `open,canRead` and `Y` by `close`.

Fig. 5 shows a more complex query and its assignment. The assignment to the variable `X` is made up of two different assignments for the different contexts surrounding `X`: when followed by `write`, `X` is assigned `open`, and when followed by `read`, `X` is assigned the word `open,canRead`. Even though the branching point in Fig. 3(b) is not identical to the one in the query, the query can still return a correct result using contexts.

2.2 An Abstract Domain of Symbolic Automata

To provide a formal background for the operations we demonstrated here informally, we define an abstract domain based on symbolic automata. Operations in the domain correspond to natural operators required for effective specification mining and answering code search queries. Our abstract domain serves a dual goal: (i) it is used to represent partial temporal specification during the analysis of each individual code snippet; (ii) it is used for consolidation and answering code search queries across multiple snippets.

In its first role—used in the analysis of a single snippet—the abstract domain can further employ a quotient abstraction to guarantee that symbolic automata do not grow without a bound due to loops or recursion [23]. In Section 4.2, we show how to obtain a lattice based on symbolic automata.

In the second role—used for consolidation and answering code-search queries—query matching can be understood in terms of *unknown elimination* in symbolic automata (explained in Section 6), and consolidation can be understood in terms of *join* in the abstract domain, followed by "minimization" (explained in Section 7).

3 Symbolic Automata

We represent partial typestate specifications using symbolic automata defined below.

Definition 1 A deterministic symbolic automaton (DSA) is a tuple $\langle \Sigma, Q, \delta, \iota, F, \mathit{Vars} \rangle$ where:

- Σ is a finite alphabet $a, b, c, \ldots$;
- Q is a finite set of states $q, q', \ldots$;

- δ is a partial function from $Q \times (\Sigma \cup \mathit{Vars})$ to Q, representing a transition relation;
- $\iota \in Q$ is an initial state;
- $F \subseteq Q$ is a set of final states;
- *Vars* is a finite set of variables $x, y, z, \ldots$

Our definition mostly follows the standard notion of deterministic finite automata. Two differences are that transitions can be labeled not just by alphabet symbols but by variables, and that they are partial functions, instead of total ones. Hence, an automaton might get stuck at a letter in a state, because the transition for the letter at the state is not defined.

We write $(q, l, q') \in \delta$ for a transition $\delta(q, l) = q'$ where $q, q' \in Q$ and $l \in \Sigma \cup \mathit{Vars}$. If $l \in \mathit{Vars}$, the transition is called *symbolic*. We extend δ to words over $\Sigma \cup \mathit{Vars}$ in the usual way. Note that this extension of δ over words is a partial function, because of the partiality of the original δ. When we write $\delta(q, s) \in Q_0$ for such words s and a state set Q_0 in the rest of the paper, we mean that $\delta(q, s)$ is defined and belongs to Q_0.

A *path* π of a DSA is a sequence of states $q_0, \ldots, q_n$ such that $q_i \in Q$ for every $0 \leq i \leq n$, and for every $0 \leq i \leq n-1$, there exists $l_i \in \Sigma \cup \mathit{Vars}$ such that $\delta(q_i, l_i) = q_{i+1}$. We then say that the symbolic word $l_0 \ldots l_{n-1}$ labels the path π. If, in addition, $q_0 = \iota$ and $q_n \in F$, we say that π is *accepting*.

From now on, we fix Σ and *Vars* and omit them from the notation of a DSA.

3.1 Semantics

For a DSA A, we define its *symbolic language*, denoted $SL(A)$, to be the set of all words over $\Sigma \cup \mathit{Vars}$ accepted by A, i.e.,

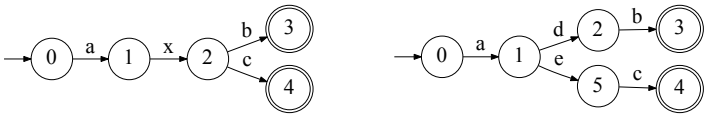
$$SL(A) = \{s \in (\Sigma \cup \mathit{Vars})^* \mid \delta(\iota, s) \in F\}.$$

Words over $\Sigma \cup \mathit{Vars}$ are called *symbolic words*, whereas words over Σ are called *concrete words*. Similarly, languages over $\Sigma \cup \mathit{Vars}$ are *symbolic*, whereas languages over Σ are *concrete*.

The symbolic language of a DSA can be interpreted in different ways, depending on the semantics of variables: (i) a variable represents a sequence of letters from Σ; (ii) a variable represents a regular language over Σ; (iii) a variable represents *different* sequences of letters from Σ under different contexts.

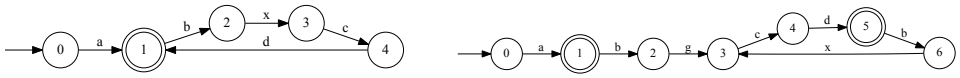
All above interpretations of variables, except for the last, assign some value to a variable while ignoring the context in which the variable lies. This is not always desirable. For example, consider the DSA in Fig. 6(a). We want to be able to interpret x as d when it is followed by b, and to interpret it as e when it is followed by c (Fig. 6(b)). Motivated by this example, we focus here on the last possibility of interpreting variables, which also considers their context. Formally, we consider the following definitions.

Definition 2 A *context-sensitive assignment*, or in short assignment, σ is a function from $(\Sigma \cup \mathit{Vars})^* \times \mathit{Vars} \times (\Sigma \cup \mathit{Vars})^*$ to the set of nonempty regular languages on $(\Sigma \cup \mathit{Vars})$.



(a) (b)

Fig. 6 DSAs (a) and (b).



(a) (b)

Fig. 7 DSA before and after assignment

When σ maps (s_1, x, s_2) to SL, we refer to (s_1, s_2) as the *context* of x. The meaning is that an occurrence of x in the context (s_1, s_2) is to be replaced by SL (i.e., by any word from SL). Thus, it is possible to assign multiple words to the same variable in different contexts. The context used in an assignment is the *full* context preceding and following x. In particular, it is not restricted in length and it can be symbolic, i.e., it can contain variables. Note that these assignments consider a *linear* context of a variable. A more general definition would consider the branching context of a variable (or a symbolic transition).

Formally, applying σ to a symbolic word behaves as follows. For a symbolic word $s = l_1 l_2 \dots l_n$, where $l_i \in \Sigma \cup \mathit{Vars}$ for every $1 \leq i \leq n$,

$$\sigma(s) = SL_1 SL_2 \dots SL_n$$

where (i) $SL_i = \{l_i\}$ if $l_i \in \Sigma$; and (ii) $SL_i = SL$ if $l_i \in \mathit{Vars}$ is a variable x and $\sigma(l_1 ... l_{i-1}, x, l_{i+1} ... l_n) = SL$.

Accordingly, for a symbolic language SL, $\sigma(SL) = \bigcup\{\sigma(s) \mid s \in SL\}$.

Definition 3 An assignment σ is *concrete* if its image consists of concrete languages only. Otherwise, it is *symbolic*.

If σ is concrete, then $\sigma(SL)$ is a concrete language; whereas if σ is symbolic, then $\sigma(SL)$ can still be symbolic.

In the sequel, when σ maps some x to the same language in several contexts, we sometimes write $\sigma(C_1, x, C_2) = SL$ as an abbreviation for $\sigma(s_1, x, s_2) = SL$ for every $(s_1, s_2) \in C_1 \times C_2$. We also write $*$ as an abbreviation for $(\Sigma \cup \mathit{Vars})^*$.

Example 1 Consider the DSA A from Fig. 6(a). Its symbolic language is $\{axb, axc\}$. Now consider the concrete assignment $\sigma : (*, x, b*) \mapsto d, (*, x, c*) \mapsto e$. Then $\sigma(axb) = \{adb\}$ and $\sigma(axc) = \{aec\}$, which means that $\sigma(SL(A)) = \{adb, aec\}$. If we consider $\sigma : (*, x, b*) \mapsto d^*, (*, x, c*) \mapsto (e|b)^*$, then $\sigma(axb) = ad^*b$ and $\sigma(axc) = a(e|b)^*c$, which means that $\sigma(SL(A)) = (ad^*b)|(a(e|b)^*c)$.

Example 2 Consider the DSA A depicted in Fig. 7(a) and consider the symbolic assignment σ which maps $(*ab, x, *)$ to g, and maps x in any other context to x. The assignment is symbolic since in any incoming context other than $*ab$, x is assigned x. Then Fig. 7(b) presents a DSA for $\sigma(SL(A))$.

Completions of a DSA Each concrete assignment σ to a DSA A results in some "completion" of $SL(A)$ into a language over Σ (c.f. Example 1). We define the semantics of a DSA A, denoted $[\![A]\!]$, as the set of all languages over Σ obtained by concrete assignments:

$$[\![A]\!] = \{\sigma(SL(A)) \mid \sigma \text{ is a concrete assignment}\}.$$

We call $[\![A]\!]$ the set of *completions* of A.

For example, for the DSA from Fig. 6(a), $\{adb, aec\} \in [\![A]\!]$ (see Example 1). Note that if a DSA A has no symbolic transition, i.e. $SL(A) \subseteq \Sigma^*$, then $[\![A]\!] = \{SL(A)\}$.

Remark 1 The interested reader might wonder about the expressive power of the languages accepted by symbolic automata, and about their closure properties. We note, however, that when talking about symbolic automata, our main interest is not their languages but their sets of completions. The language of a DSA is simply a regular language (over an alphabet extended by the set of variables). The set of completions of a DSA, on the other hand, is not a single language, but a set of languages. In this sense, the discussion of expressive power and closure properties in their traditional formulation is not natural.

4 An Abstract Domain for Specification Mining

In this section we lay the ground for defining common operations on DSAs by defining a preorder on DSAs. In later sections, we use this preorder to define an algorithm for query matching (Section 5), completion of partial specification (Section 6), and consolidation of multiple partial specification (Section 7).

The definition of a preorder over DSAs is motivated by two concepts. The first is *precision*. We are interested in capturing that one DSA is an overapproximation of another, in the sense of describing more behaviors (sequences) of an API. When DFAs are considered, language inclusion is suitable for capturing a precision (abstraction) relation between automata. The second is *partialness*. We would like to capture that a DSA is "more complete" than another in the sense of having less variables that stand for unknown information.

4.1 Preorder on DSAs

Our preorder combines precision and partialness. Since the notion of partialness is less standard, we first explain how it is captured for symbolic words. The consideration of symbolic words rather than DSAs allows us to ignore the dimension of precision and focus on partialness, before we combine the two.

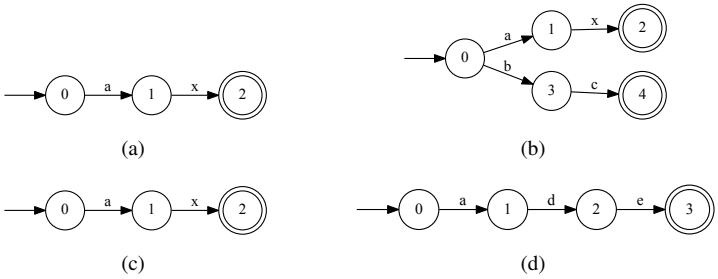


Fig. 8 Dimensions of the preorder on DSAs

4.1.1 Preorder on Symbolic Words

Definition 4 Let s_1, s_2 be symbolic words. $s_1 \leq s_2$ if for every concrete assignment σ_2 to s_2, there is a concrete assignment σ_1 to s_1 such that $\sigma_1(s_1) = \sigma_2(s_2)$.

This definition captures the notion of a symbolic word being "more concrete" or "more complete" than another: Intuitively, the property that no matter how we fill in the unknown information in s_2 (using a concrete assignment), the same completion can also be obtained by filling in the unknowns of s_1, ensures that every unknown of s_2 is also unknown in s_1 (which can be filled in the same way), but s_1 can have additional unknowns. Thus, s_2 has "no more" unknowns than s_1. In particular, $\{\sigma(s_1) \mid \sigma \text{ is a concrete assignment}\} \supseteq \{\sigma(s_2) \mid \sigma \text{ is a concrete assignment}\}$. Note that when considering two concrete words $w_1, w_2 \in \Sigma^*$ (i.e., without any variable), $w_1 \leq w_2$ iff $w_1 = w_2$. In this sense, the definition of $\leq$ over symbolic words is a relaxation of equality over words.

For example, $abxcd \geq ayd$ according to our definition. Intuitively, this relationship holds because $abxcd$ is more complete (carries more information) than ayd.

4.1.2 Symbolic Inclusion of DSAs

We now define the preorder over DSAs that combines precision with partialness. On the one hand, we say that a DSA A_2 is "bigger" than A_1, if A_2 describes more possible behaviors of the API, as captured by standard automata inclusion. For example, see the DSAs (a) and (b) in Fig. 8. On the other hand, we say that a DSA A_2 is "bigger" than A_1, if A_2 describes "more complete" behaviors, in terms of having less unknowns. For example, see the DSAs (c) and (d) in Fig. 8.

However, these examples are simple in the sense of "separating" the precision and the partialness dimensions. Each of these examples demonstrates one dimension only. We are also interested in handling cases that combine the two, such as cases where A_1 and A_2 represent more than one word, thus the notion of completeness of symbolic words alone is not applicable, and in addition the language of A_1 is not included in the language of A_2 per se, e.g., since some of the words in A_1 are less complete than those of A_2. This leads us to the following definition.

Definition 5 (symbolic-inclusion) A DSA A_1 is symbolically-included in a DSA A_2, denoted by $A_1 \preceq A_2$, if for every concrete assignment σ_2 of A_2 there exists a concrete assignment σ_1 of A_1, such that $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$.

The above definition ensures that for each concrete language L_2 that is a completion of A_2, A_1 can be assigned in a way that will result in its language being included in L_2. This means that the "concrete" parts of A_1 and A_2 admit the inclusion relation, and A_2 is "more concrete" than A_1. Note that A_1 is symbolically-included in A_2 iff for every $L_2 \in [\![A_2]\!]$ there exists $L_1 \in [\![A_1]\!]$ such that $L_1 \subseteq L_2$.

Example 3 The DSA depicted in Fig. 6(a) is symbolically-included in the one depicted in Fig. 6(b), since for any assignment σ_2 to (b), the assignment σ_1 to (a) that will yield a language included in the language of (b) is $\sigma : (*, x, b*) \mapsto d, (*, x, c*) \mapsto e$. Note that if we had considered assignments to a variable without a context, the same would not hold: If we assign to x the sequence d, the word adc from the assigned (a) will remain unmatched. If we assign e to x, the word aeb will remain unmatched. If we assign to x the language $d|e$, then both of the above words will remain unmatched. Therefore, when considering context-free assignments, there is no suitable assignment σ_1.

Theorem 1 $\preceq$ *is reflexive and transitive.*

Proof Reflexivity: Let A be a DSA. Then $A \preceq A$ since for every concrete assignment σ to A, σ will fulfill $\sigma(SL(A)) \subseteq \sigma(SL(A))$.

Transitivity: Let A_1, A_2, A_3 be DSAs such that $A_1 \preceq A_2$ and $A_2 \preceq A_3$. We show that A_1 is symbolically-included in A_3. Let σ_3 be a concrete assignment to A_3. Then there exists a concrete assignment σ_2 to A_2 such that $\sigma_2(SL(A_2)) \subseteq \sigma_3(SL(A_3))$ (since $A_2 \preceq A_3$). Similarly, since $A_1 \preceq A_2$, then in particular for σ_2 there exists a concrete assignment σ_1 to A_1 such that $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$. By transitivity of language inclusion, we get that $\sigma_1(SL(A_1)) \subseteq \sigma_3(SL(A_3))$. □

4.1.3 Structural Inclusion

As a basis for an algorithm for checking if symbolic-inclusion holds between two DSAs, we note that provided that any alphabet Σ' can be used in assignments, the following definition is equivalent to Definition 5.

Definition 6 A_1 is *structurally-included* in A_2 if there exists a symbolic assignment σ to A_1 such that $\sigma(SL(A_1)) \subseteq SL(A_2)$. We say that σ *witnesses* the structural inclusion of A_1 in A_2.

Theorem 2 *Let A_1, A_2 be DSAs. Then $A_1 \preceq A_2$ iff A_1 is structurally-included in A_2.*

Before we turn to prove Theorem 2, we note that if there is a witnessing assignment for structural inclusion, then in particular there exists one that assigns a single sequence to each variable in every context, as formalized by the following lemma. It will be used in the proof of Theorem 2.

Lemma 1 *A_1 is* structurally-included *in A_2 iff there exists a symbolic assignment σ to A_1, such that $\sigma(SL(A_1)) \subseteq SL(A_2)$, and the image of σ consists of singleton languages only.*

Proof Suppose there exists a symbolic assignment σ to A_1, such that $\sigma(SL(A_1)) \subseteq SL(A_2)$. We show that there is also such an assignment σ' whose image consists of singleton languages only. We define σ' by arbitrarily keeping one sequence from each language participating in the image of σ. Clearly, $\sigma'(SL(A_1)) \subseteq \sigma(SL(A_1)) \subseteq SL(A_2)$. □

Proof (Theorem 2) We first show that structural inclusion implies symbolic-inclusion. Suppose A_1 is structurally included in A_2. Then there exists a symbolic assignment σ such that $\sigma(SL(A_1)) \subseteq SL(A_2)$, i.e., for every $s \in SL(A_1)$, $\sigma(s) \subseteq SL(A_2)$. Without loss of generality (by Lemma 1), σ assigns to each variable a single sequence under each context. To show that $A_1 \preceq A_2$, consider some assignment σ_2 of A_2. To find a corresponding assignment σ_1 to A_1 such that $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$, we consider the composition of σ_2 on σ, defined as follows.

Let $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m \in \Sigma \cup \mathit{Vars}$ and $x \in \mathit{Vars}$. Then

$$\sigma_1(a_1 a_2 \ldots a_n, x, b_1 b_2 \ldots b_m) = \{w_1\} L_1 \{w_2\} \ldots \{w_k\} L_k \{w_{k+1}\}$$

where $w_1, w_2, \ldots, w_{k+1} \in \Sigma^*$, $L_1, \ldots, L_k \subseteq \Sigma^*$, and there exist

- $x_1, \ldots, x_k \in \mathit{Vars}$ and
- $s_1, \ldots, s_n, s'_1, \ldots s'_m \in (\Sigma \cup \mathit{Vars})^*$

such that:

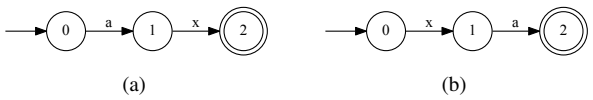
- $\sigma(a_1 a_2 \ldots a_n, x, b_1 b_2 \ldots b_m) = \{w_1 x_1 w_2 \ldots w_k x_k w_{k+1}\}$.
- For every $1 \leq i \leq n$ such that $a_i \in \mathit{Vars}$:
  $\sigma(a_1 \ldots a_{i-1}, a_i, a_{i+1} \ldots a_n x b_1 b_2 \ldots b_m) = \{s_i\}$.
- For every $1 \leq i \leq n$ such that $a_i \in \Sigma$: $s_i = a_i$.
- For every $1 \leq i \leq m$ such that $b_i \in \mathit{Vars}$:
  $\sigma(a_1 \ldots a_n x b_1 \ldots b_{i-1}, b_i, b_{i+1} \ldots b_m) = \{s'_i\}$.
- For every $1 \leq i \leq m$ such that $b_i \in \Sigma$: $s'_i = b_i$.
- For every $1 \leq i \leq k$:
  $\sigma_2(s_1 \ldots s_n w_1 x_1 \ldots w_i, x_i, w_{i+1} \ldots x_k w_{k+1} s'_1 \ldots s'_m) = L_i$.

In other words, if

$$\begin{aligned}\sigma(a_1 a_2 \ldots a_n x b_1 b_2 \ldots b_m) &= \{s_1 \ldots s_n w_1 x_1 w_2 \ldots w_k x_k w_{k+1} s'_1 \ldots s'_m\} \\ \wedge\, \sigma_2(s_1 \ldots s_n w_1 x_1 w_2 \ldots w_k x_k w_{k+1} s'_1 \ldots s'_m) &\\ = L' \{w_1\} L_1 \{w_2\} \ldots \{w_k\} L_k \{w_{k+1}\} L'',&\end{aligned}$$

then $\sigma_1(a_1 \ldots a_n x b_1 \ldots b_m)$ is also equal to $L' \{w_1\} L_1 \{w_2\} \ldots \{w_k\} L_k \{w_{k+1}\} L''$. Therefore, the definition of σ_1 ensures for every symbolic word $s \in SL(A_1)$, $\sigma_1(s) = \sigma_2(\sigma(s))$. Since $\sigma(s) \subseteq SL(A_2)$ (by the choice of σ), we conclude that $\sigma_1(s) = \sigma_2(\sigma(s)) \subseteq \sigma_2(SL(A_2))$, and hence $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$.

We now show that symbolic-inclusion implies structural inclusion. Suppose $A_1 \preceq A_2$. For each variable $x \in \mathit{Vars}$, we introduce a new letter $a_x \in \Sigma'$. We now define an



(a) (b)

Fig. 9 Example for a case where symbolic-inclusion does not imply structural inclusion

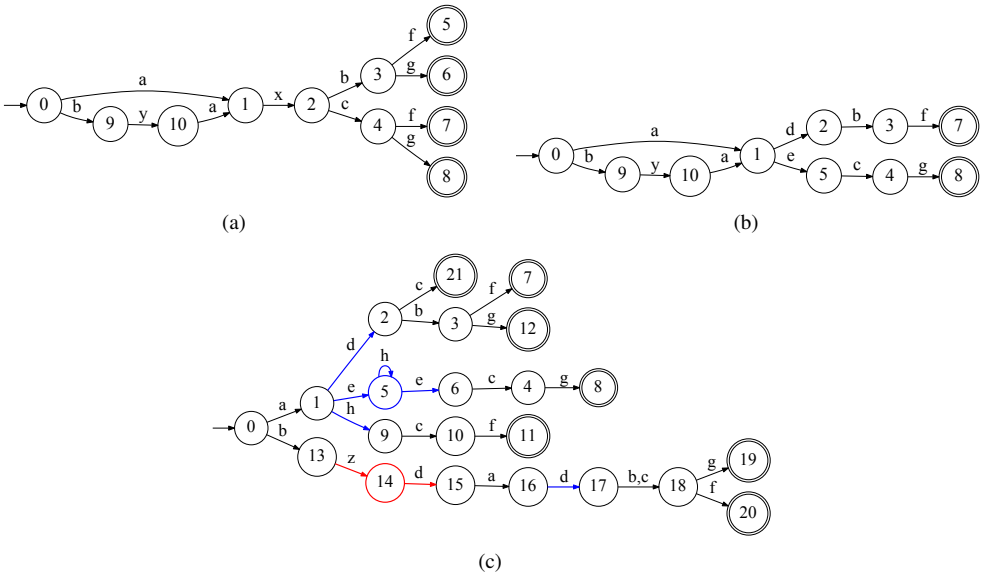


Fig. 10 Example for a case where there is no assignment to either (a) or (b) to show (a) $\preceq$ (b) or (b) $\preceq$ (a), and where there is such an assignment for (a) so that (a) $\preceq$ (c).

assignment σ_2 to A_2 such that for every $x \in \mathit{Vars}$, and for every $s_1, s_2 \in (\Sigma' \cup \mathit{Vars})^*$, $\sigma_2 : (s_1, x, s_2) \mapsto a_x$. Since $A_1 \preceq A_2$, there exists an assignment σ_1 such that $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$. To obtain a symbolic assignment σ to A_1 such that $\sigma(SL(A_1)) \subseteq SL(A_2)$, we replace every occurrence of a_x in σ_1 by x. □

To understand why we need to consider assignments over an extended alphabet Σ', consider the following example.

Example 4 Consider the DSAs in Fig. 9. Structural inclusion does not hold between (a) and (b). However, when considering assignments over $\Sigma = \{a\}$ only, (a) is symbolically-included in (b). Note that if we also consider assignments over, say $\{a, b\}$, then, symbolic-inclusion ceases to hold as well, regaining the relation between structural inclusion and symbolic-inclusion.

The corollary below provides another sufficient condition for symbolic-inclusion:

Corollary 1 *If* $SL(A_1) \subseteq SL(A_2)$*, then* $A_1 \preceq A_2$.

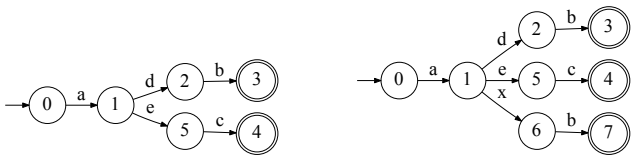


Fig. 11 Equivalent DSAs w.r.t. symbolic-inclusion

Example 5 The DSA depicted in Fig. 10(a) is not symbolically-included in the one depicted in Fig. 10(b) since no symbolic assignment to (a) will substitute the symbolic word $axbg$ by a (symbolic) word (or set of words) in (b). This is because assignments cannot "drop" any of the contexts of a variable (e.g., the outgoing bg context of x). Such assignments are undesirable since removal of contexts amounts to removal of observed behaviors.

On the other hand, the DSA depicted in Fig. 10(a) is symbolically-included in the one depicted in Fig. 10(c), since there is a witnessing assignment that maintains all the contexts of x, namely, $\sigma : (a, x, b*) \mapsto d, (a, x, cf*) \mapsto h, (a, x, cg*) \mapsto eh^*e, (bya, x, *) \mapsto d, (*, y, *) \mapsto zd$. Assigning σ to (a) results in a DSA whose symbolic language is strictly included in the symbolic language of (c). Note that symbolic-inclusion holds despite of the fact that in (c) there is no longer a state with an incoming c event and both an outgoing f and an outgoing g events while being reachable from the state 1. This example demonstrates our interest in linear behaviors, rather than in branching behavior. Note that in this example, symbolic-inclusion would not hold if we did not allow to refer to contexts of any length (and in particular length > 1).

4.2 A Lattice for Specification Mining

As stated in Theorem 1, $\preceq$ is reflexive and transitive, and therefore a preorder. However, it is not antisymmetric. This is not surprising, since for DFAs $\preceq$ collapses into standard automata inclusion, which is also not antisymmetric (due to the existence of different DFAs with the same language). In the case of DSAs, symbolic transitions are an additional source of problem, as demonstrated by the following example.

Example 6 The DSAs in Fig. 11 satisfy $\preceq$ in both directions even though their symbolic languages are different. DSA (a) is trivially symbolically-included in (b) since the symbolic language of (a) is a subset of the symbolic language of (b) (see Corollary 1). For the other direction, symbolic inclusion holds by Lemma 1 when assigning d to x in (b) in any context.

Examining the example closely shows that the reason that symbolic-inclusion holds in both directions even though the DSAs have different symbolic languages is that the symbolic language of DSA (b) contains the symbolic word axb, as well as the concrete word adb, which is a completion of axb. In this sense, axb is subsumed by the rest of the DSA, which amounts to DSA (a).

In order to obtain a partial order, we follow a standard construction of turning a preordered set to a partially ordered set. We first define the following equivalence relation based on $\preceq$:

Definition 7 DSAs A_1 and A_2 are symbolically-equivalent, denoted by $A_1 \equiv A_2$, iff $A_1 \preceq A_2$ and $A_2 \preceq A_1$.

Theorem 3 *$\equiv$ is an equivalence relation over the set* **DSA** *of all DSAs.*

We now lift the discussion to the quotient set $\mathbf{DSA}/\equiv$, which consists of the equivalence classes of **DSA** w.r.t. the $\equiv$ equivalence relation.

Definition 8 Let $[A_1], [A_2] \in \mathbf{DSA}/\equiv$. Then $[A_1] \sqsubseteq [A_2]$ if $A_1 \preceq A_2$.

Theorem 4 *$\sqsubseteq$ is a partial order over $\mathbf{DSA}/\equiv$.*

Proof Reflexivity and transitivity follow immediately from the properties of $\preceq$ over DSAs. We prove that unlike the latter, $\sqsubseteq$ over $\mathbf{DSA}/\equiv$ is also antisymmetric. Suppose $[A_1] \sqsubseteq [A_2]$ and $[A_2] \sqsubseteq [A_1]$. Then $A_1 \preceq A_2$ and vice versa, hence $A_1 \equiv A_2$, and thus $[A_1] = [A_2]$. □

Definition 9 For DSAs A_1 and A_2, we use $union(A_1, A_2)$ to denote a union DSA for A_1 and A_2, which is defined similarly to the definition of union of DFAs. That is, $union(A_1, A_2)$ is a DSA such that $SL(union(A_1, A_2)) = SL(A_1) \cup SL(A_2)$.

Theorem 5 *Let $[A_1], [A_2] \in \mathbf{DSA}/\equiv$ and let $union(A_1, A_2)$ be a union DSA for A_1 and A_2. Then $[union(A_1, A_2)]$ is the least upper bound of $[A_1]$ and $[A_2]$ w.r.t. $\sqsubseteq$.*

Proof We first show that $[union(A_1, A_2)] \sqsupseteq [A_i]$ for every $i \in \{1, 2\}$. This follows since $SL(union(A_1, A_2)) = SL(A_1) \cup SL(A_2) \supseteq SL(A_i)$ and hence by Corollary 1, $A_i \preceq union(A_1, A_2)$.

We now show that if $[A] \sqsupseteq [A_i]$ for every $i \in \{1, 2\}$, then

$$[A] \sqsupseteq [union(A_1, A_2)].$$

It suffices to show that $union(A_1, A_2) \preceq A$. Since $[A] \sqsupseteq [A_i]$, we conclude that $A_i \preceq A$. Consider a concrete assignment σ to A. Since $[A] \sqsupseteq [A_i]$, $A_i \preceq A$, thus there exists an assignment σ_i such that $\sigma_i(SL(A_i)) \subseteq \sigma(SL(A))$. This means that $\sigma_1(SL(A_1)) \cup \sigma_2(SL(A_2)) \subseteq \sigma(SL(A))$. Consider the assignment σ' to $union(A_1, A_2)$ obtained by $\sigma'_1 \cup \sigma'_2$, where σ'_1 is identical to σ_1, except that it is undefined for symbolic words in $SL(A_2)$, and σ'_2 is identical to σ_2, except that it is defined only for symbolic words in $SL(A_2)$. This ensures that the assignment is well defined. In addition,

$$\begin{aligned}
\sigma'(SL(union(A_1, A_2)) &= \sigma'(SL(A_1) \cup SL(A_2)) \\
&= \sigma'(SL(A_1)) \cup \sigma'(SL(A_2)) \\
&= \sigma_1(SL(A_1) \setminus SL(A_2)) \cup \sigma_2(SL(A_2)) \\
&\subseteq \sigma_1(SL(A_1)) \cup \sigma_2(SL(A_2)) \subseteq \sigma(SL(A)).
\end{aligned}$$

We conclude that $union(A_1, A_2) \preceq A$. □

Corollary 2 *$(\mathbf{DSA}/\equiv, \sqsubseteq)$ is a join semi-lattice.*

The $\bot$ element in the lattice is the equivalence class of a DSA for $\emptyset$. The $\top$ element is the equivalence class of a DSA for Σ^*.

5 Query Matching Using Symbolic Simulation

Given a query in the form of a DSA, and a database of other DSAs, query matching attempts to find DSAs in the database that symbolically include the query DSA. In this section, we describe a notion of simulation for DSAs, which precisely captures the preorder on DSAs and serves a basis of core algorithms for manipulating symbolic automata. In particular, in Section 5.2, we provide an algorithm for computing symbolic simulation that can be directly used to determine when symbolic inclusion holds.

5.1 Symbolic Simulation

Let A_1 and A_2 be DSAs $\langle Q_1, \delta_1, \iota_1, F_1\rangle$ and $\langle Q_2, \delta_2, \iota_2, F_2\rangle$, respectively.

Definition 10 A relation $H \subseteq Q_1 \times (2^{Q_2} \setminus \{\emptyset\})$ is a *symbolic simulation* from A_1 to A_2 if it satisfies the following conditions:

(a) $(\iota_1, \{\iota_2\}) \in H$;
(b) for every $(q, B) \in H$, if q is a final state, some state in B is final;
(c) for every $(q, B) \in H$ and $q' \in Q_1$, if $q' = \delta_1(q, a)$ for some $a \in \Sigma$,

$$\exists B' \text{ s.t. } (q', B') \in H \ \wedge \ B' \subseteq \{q_2' \mid \exists q_2 \in B \text{ s.t. } q_2' = \delta_2(q_2, a)\};$$

(d) for every $(q, B) \in H$ and $q' \in Q_1$, if $q' = \delta_1(q, x)$ for $x \in \mathit{Vars}$,

$$\exists B' \text{ s.t. } (q', B') \in H \ \wedge \ B' \subseteq \{q_2' \mid \exists q_2 \in B \text{ s.t. } q_2' \text{ is reachable from } q_2\}.$$

We say that (q', B') in the third or fourth item above is a *witness* for $((q, B), l)$, or an *l-witness* for (q, B) for $l \in \Sigma \cup \mathit{Vars}$. Finally, A_1 is *symbolically simulated* by A_2 if there exists a symbolic simulation H from A_1 to A_2.

In this definition, a state q of A_1 is simulated by a nonempty set B of states from A_2, with the meaning that their union overapproximates all of its outgoing behaviors. In other words, the role of q in A_1 is "split" among the states of B in A_2. A "split" arises from symbolic transitions, but the "split" of the target of a symbolic transition can be propagated forward for any number of steps, thus allowing states to be simulated by sets of states even if they are not the target of a symbolic transition. This accounts for splitting that is performed by an assignment with a context longer than one. Note that since we consider *deterministic* symbolic automata, the sizes of the sets used in the simulation are monotonically decreasing, except for when a target of a symbolic transition is considered, in which case the set increases in size.

Note that a state q_1 of A_1 can participate in more than one simulation pair in the computed simulation, as demonstrated by the following example.

Example 7 Consider the DSAs in Fig. 10(a) and (c). In this case, the simulation will be

$$\begin{aligned} H = \{\ & (0,\{0\}), (1,\{1\}), (2,\{2,6,9\}), (3,\{3\}), (4,\{4,10\}), (5,\{7\}), (6,\{12\}) \\ & (7,\{11\}), (8,\{8\}), (9,\{13\}), (10,\{15\}), (1,\{16\}), (2,\{17\}), (4,\{18\}), \\ & (7,\{20\}), (8,\{19\}), (3,\{18\}), (5,\{20\}), (6,\{19\})\ \}. \end{aligned}$$

One can see that state 2 in (a), which is the target of the transition labeled x, is "split" between states 2, 6 and 9 of (c). In the next step, after seeing b from state 2 in (a), the target state reached (state 3) is simulated by a singleton set. On the other hand, after seeing c from state 2 in (a), the target state reached (state 4), is still "split", however this time to only two states: 4 and 10 in (c). In the next step, no more splitting occurs.

Note that the state 1 in (a) is simulated both by $\{1\}$ and by $\{16\}$. Intuitively, each of these sets simulates the state 1 in another incoming context (a and b, respectively).

Theorem 6 (Soundness) *For all DSAs A_1 and A_2, if there is a symbolic simulation H from A_1 to A_2, then $A_1 \preceq A_2$.*

Our proof of this theorem uses Theorem 2 and constructs a desired symbolic assignment σ that witnesses structural inclusion of A_1 in A_2 explicitly from H. This construction shows, for any symbolic word in $SL(A_1)$, the assignment (completion) to all variables in it (in the corresponding context). Taken together with our next completeness theorem (Theorem 7), this construction supports a view that a symbolic simulation serves as a finite representation of symbolic assignment in the preorder. We develop this further in Section 6.

Proof Let H be a symbolic simulation from A_1 to A_2. We show that there exists a symbolic assignment σ such that $\sigma(SL(A_1)) \subseteq SL(A_2)$. Recall that $\sigma(SL(A_1)) = \bigcup\{\sigma(s) \mid s \in SL(A_1)\}$. Hence, it suffices to find σ such that for every $s \in SL(A_1)$, $\sigma(s) \subseteq SL(A_2)$.

Let $s = l_1 \dots l_n \in SL(A_1)$ where each $l_i \in \Sigma \cup \mathit{Vars}$. First, we define a sequence of simulation pairs $\overline{h} = h_0 \dots h_n$, where $h_0 = (\iota_1, \{\iota_2\})$, and for every $1 \leq i \leq n$, $h_i \in H$ is an l_i-witness for h_{i-1}. The sequence is well defined, since by the definition of H, a corresponding witness always exists. Note that if several witnesses exist, one of them is chosen arbitrarily.

The idea is to track a symbolic word in A_2 that matches s, up to variables, by following the simulation pairs in $\overline{h}$. For this purpose, we first need to minimize the simulation pairs to include only states that are relevant to the simulation of the particular word s. Once this is done, any path in A_2 through $\overline{h}$ defines a symbolic word that matches s up to variables, and accordingly defines a possible assignment for the variables in s.

We therefore first apply the following minimization algorithm on $\overline{h}$: The algorithm updates the second component of the h_i's from $i = n-1$ to $i = 0$. For $h_n = (q_n, B_n)$, we remove all non-final states from B_n, and set $\tilde{B}_n$ to be the set of the remaining ones in B_n. Suppose $h_i = (q_i, B_i)$ and $h_{i+1} = (q_{i+1}, B_{i+1})$, where h_{i+1} is an l_{i+1}-witness for h_i. Let $\tilde{B}_{i+1}$ denote the updated B_{i+1}. Then we remove from B_i all the states that contributed no states to $\tilde{B}_{i+1}$. More specifically:

- If $l_{i+1} \in \Sigma$, we let $\tilde{B}_i = \{q_2 \in B_i \mid \exists q_2' \in \tilde{B}_{i+1} \text{ s.t. } \delta_2(q_2, l_{i+1}) = q_2'\}$.
- If $l_{i+1} \in \mathit{Vars}$, we let $\tilde{B}_i = \{q_2 \in B_i \mid \exists q_2' \in \tilde{B}_{i+1} \text{ s.t. } q_2' \text{ is reachable from } q_2\}$.

Importantly, no set becomes empty as a result of the update, since $\tilde{B}_{i+1} \subseteq B_{i+1}$, which ensures that every state in $\tilde{B}_{i+1}$ is contributed by at least one state in B_i.

Once $\overline{h}$ is minimized as above, we greedily choose states

$$q_0^2 = \iota^2 \in \tilde{B}_0, q_1^2 \in \tilde{B}_1, \dots, q_n^2 \in \tilde{B}_n$$

such that if $l_i \in \Sigma$, then $\delta_2(q^2_{i-1}, l_i) = q^2_i$, and otherwise, q^2_i is reachable from q^2_{i-1}. This choice of states defines a symbolic word s' that matches s up to variables. Moreover, given this choice, for $l_i \in \mathit{Vars}$, each symbolic word $\tilde{s}_i$ such that $\delta_2(q^2_{i-1}, \tilde{s}_i) = q^2_i$ is a possible match for l_i. We denote such $\tilde{s}_i$ by $match(l_i)$.

Now we are ready to define the desired symbolic assignment σ. Suppose that $s = w_1x_1w_2 \dots w_kx_kw_{k+1}$ where $x_1, \dots, x_k \in \mathit{Vars}$ and $w_1, \dots, w_{k+1} \in \Sigma^*$. We let s_j and s'_j be the following prefix and suffix of s:

$$s_j = w_1x_1w_2 \dots w_{j-1}x_{j-1}w_j, \qquad s'_j = w_{j+1}x_{j+1}w_{j+2} \dots w_kx_kw_{k+1}.$$

Then for every $1 \leq j \leq n$,

$$\sigma(s_j, x_j, s'_j) = \{match(x_j)\}.$$

We now get that $\sigma(s) = \{s'\} \subseteq SL(A_2)$, as required. □

Theorem 7 (Completeness) *For all DSAs A_1 and A_2, if $A_1 \preceq A_2$, then there is a symbolic simulation H from A_1 to A_2.*

Proof Let σ be a symbolic assignment such that $\sigma(SL(A_1)) \subseteq SL(A_2)$. Without loss of generality, we can assume that σ maps all variables and contexts to singleton languages. This is because otherwise, we can reduce σ such that the resulting assignment satisfy this singleton-language requirement. Since σ satisfies this singleton-language requirement, we have that for every $s \in SL(A_1)$, $\sigma(s) = \{s'\}$ for some $s' \in SL(A_2)$. We define a symbolic simulation H from A_1 to A_2 based on σ.

We start by removing from A_1 and A_2 all states that do not lie on any path from an initial to an accepting state. In the rest of the proof, by A_1 and A_2, we refer to the pruned DSAs. Clearly, $SL(A_1)$ and $SL(A_2)$ remain unchanged, thus $\sigma(SL(A_1)) \subseteq SL(A_2)$ still holds.

The idea is to consider for each state $q_1 \in Q_1$, all the symbolic words that lead to it in A_1, and for each of these words s, let q_1 be simulated by the set of all states in A_2 reached when traversing the symbolic words resulting from assigning σ to s when considering s as a prefix of s', for every $s' \in SL(A_1)$ that extends s. The consideration of s' is needed since the result of applying σ on a variable depends on the (full) context.

Technically, for a symbolic word $s' = l_1l_2 \dots l_n$, and for $0 \leq k \leq n$ we define $\sigma \downarrow_k (s')$, which describes the result of assigning σ to the prefix of s' of length k while taking into consideration the full context based on s'. The symbolic word $\sigma \downarrow_k (s')$ is defined as follows. $\sigma \downarrow_k (l_1l_2 \dots l_n) = L_1L_2 \dots L_k$ where $L_i = \{l_i\}$ if $l_i \in \Sigma$, and $L_i = L$ if l_i is a variable x and $\sigma(l_1 \dots l_{i-1}, x, l_{i+1} \dots l_n) = L$. In particular, for $k = 0$, $\sigma \downarrow_0 (l_1l_2 \dots l_n) = \{\epsilon\}$, and for $k = n$, $\sigma \downarrow_n (l_1l_2 \dots l_n) = \sigma(l_1l_2 \dots l_n)$.

Since we consider an assignment σ whose image consists of singleton languages, we abuse the notation and write $\sigma(s)$ or $\sigma \downarrow_k (s)$ as a shorthand for the single word in the set.

For a symbolic word s, we define

$$ext(s) = \{s' \in SL(A_1) \mid s \text{ is a prefix of } s'\},$$

which denotes the set of extensions of s in $SL(A_1)$. Also, if s is of length k, we define

$$B(s) = \{\delta_2(\iota_2, \sigma \downarrow_k (s')) \in Q_2 \mid s' \in ext(s)\}$$

to be the set of states reached in A_2 when following the symbolic word obtained by applying σ to some extension s' of s up to the k'th symbol.

Let

$$H = \{(\delta_1(\iota_1, s), B(s)) \mid s \in (\Sigma \cup \mathit{Vars})^* \wedge \delta_1(\iota_1, s) \text{ is defined}\}.$$

In the rest of the proof, we will show that H is a symbolic simulation from A_1 to A_2.

The first requirement to check is that $H \subseteq Q_1 \times (2^{Q_2} \setminus \{\emptyset\})$. Pick $s \in (\Sigma \cup \mathit{Vars})^*$ such that $\delta_1(\iota_1, s)$ is defined. Since we keep only those states in A_1 that lie in a path from the initial state to an accepting state, there exists at least one $s' \in ext(s) \subseteq SL(A_1)$, which in turn implies that $\sigma(s') \in SL(A_2)$. Then, $\delta_2(\iota^2, \sigma \downarrow_k (s'))$ is defined, and belongs to $B(s)$. Hence, $B(s) \neq \emptyset$.

The next requirement is that $(\iota_1, \{\iota_2\}) \in H$. This holds because

$$\delta_1(\iota_1, \epsilon) = \iota_1 \wedge B(\epsilon) = \{\delta_2(\iota_2, \sigma \downarrow_0 (s')) \in Q_2 \mid s' \in ext(\epsilon)\} = \{\iota_2\}.$$

To prove the remaining requirements, consider $(q_1, B_2) \in H$. We show that all the remaining requirements of a symbolic simulation hold. Let $s \in (\Sigma \cup \mathit{Vars})^*$ be a symbolic word such that $q_1 = \delta_1(\iota^1, s)$, and $B_2 = B(s)$. Also, let k be the length of s.

If q_1 is a final state, the word s is in $SL(A_1)$. Hence, in this case, $\sigma(s) \in SL(A_2)$, so $\delta(\iota_2, \sigma(s))$ is a final state. But, $\delta(\iota_2, \sigma(s)) \in B(s)$. It means that $B(s)$ contains a final state, as required.

Suppose $\delta_1(q_1, a) = q_1'$ for some $a \in \Sigma$. Then $(\delta_1(\iota_1, sa), B(sa)) \in H$ is an a-witness for (q_1, B_2). First, $\delta_1(\iota_1, sa) = \delta_1(\delta_1(\iota_1, s), a) = \delta_1(q_1, a) = q_1'$, and hence it is defined. It remains to show that

$$B(sa) \subseteq \{\delta_2(q_2, a) \mid q_2 \in B_2\} = \{\delta_2(q_2, a) \mid q_2 \in B(s)\}.$$

Let $q_2' \in B(sa)$. We need to show that $q_2' \in \{\delta_2(q_2, a) \mid q_2 \in B(s)\}$, i.e. that there exists $q_2 \in B(s)$ such that $q_2' = \delta_2(q_2, a)$. Since $q_2' \in B(sa)$, there exists $s' \in ext(sa)$ such that

$$q_2' = \delta_2(\iota_2, \sigma \downarrow_{k+1} (s')) = \delta_2(\iota_2, \sigma \downarrow_k (s')a) = \delta_2(\delta_2(\iota_2, \sigma \downarrow_k (s')), a).$$

Moreover, $s' \in ext(s)$ since $ext(sa) \subseteq ext(s)$. So, $\delta_2(\iota_2, \sigma \downarrow_k (s')) \in B(s)$. Thus for $q_2 = \delta_2(\iota_2, \sigma \downarrow_k (s')) \in B(s)$, we have that $q_2' = \delta_2(q_2, a)$.

Suppose $\delta_1(q_1, x) = q_1'$ for some $x \in \mathit{Vars}$. Then $(\delta_1(\iota_1, sx), B(sx)) \in H$ is an x-witness for (q_1, B_2). First,

$$\delta_1(\iota_1, sx) = \delta_1(\delta_1(\iota_1, s), x) = \delta_1(q_1, x) = q_1'.$$

Hence it is defined. It remains to show that every state $q_2' \in B(sx)$ is reachable in A_2 from some $q_2 \in B(sx)$. Let $q_2' \in B(sx)$. Since $q_2' \in B(sx)$, there exist $s' \in ext(sx)$ and s_x such that

$$q_2' = \delta_2(\iota_2, \sigma \downarrow_{k+1} (s')) = \delta_2(\iota_2, \sigma \downarrow_k (s')s_x) = \delta_2(\delta_2(\iota_2, \sigma \downarrow_k (s')), s_x).$$

Moreover, $s' \in ext(s)$ since $ext(sx) \subseteq ext(s)$. So, $\delta_2(\iota_2, \sigma \downarrow_k (s')) \in B(s)$. Thus for $q_2 = \delta_2(\iota_2, \sigma \downarrow_k (s')) \in B(s)$, we have that $q_2' = \delta_2(q_2, s_x)$, which means q_2' is reachable from q_2. □

5.2 Algorithm for Checking Simulation

A maximal symbolic simulation relation can be computed using a greatest fixpoint algorithm (similarly to the standard simulation). A naive implementation would consider all sets in 2^{Q_2}, making it exponential.

More efficiently, we obtain a symbolic simulation relation H by an algorithm that traverses both DSAs simultaneously, starting from $(\iota_1, \{\iota_2\})$, similarly to a computation of a product automaton (see Algorithm 1). For each pair (q_1, B_2) that we explore, we make sure that if $q_1 \in F_1$, then $B_2 \cap F_2 \neq \emptyset$. If this is not the case, the pair cannot be part of the simulation. Otherwise, we traverse all the outgoing transitions of q_1, and for each one, we look for a *witness* in the form of another simulation pair, as required by Definition 10 (see below). If a witness is found, it is added to the list of simulation pairs that need to be explored. If no witness is found, the pair (q_1, B_2) cannot participate in the simulation.

Consider a candidate simulation pair (q_1, B_2). When looking for a witness for some transition of q_1, a crucial observation is that if some set $B_2' \subseteq Q_2$ simulates a state $q_1' \in Q_1$, then any superset of B_2' also simulates q_1'. Therefore, as a witness we add the *maximal* set that fulfills the requirement: if we fail to prove that q_1' is simulated by the maximal candidate for B_2', then we will also fail with any other candidate, making it unnecessary to check.

Specifically, for an a-transition, where $a \in \Sigma$, from q_1 to q_1', the witness is (q_1', B_2') where $B_2' = \{q_2' \mid \exists q_2 \in B_2 \text{ s.t. } q_2' = \delta_2(q_2, a)\}$. If $B_2' = \emptyset$, then no witness exists. For a symbolic transition from q_1 to some q_1', the witness is (q_1', B_2') where B_2' is the set of all states reachable from the states in B_2 (note that $B_2' \neq \emptyset$ as it contains at least the states of B_2).

If it turns out that some explored pair (q_1, B_2) cannot participate in the simulation (either due to the final states or because no witness was found for some transition), we conclude that no simulation exists. This is because the potential witnesses that we consider in each step are maximal. As a result, when it turns out that a witness cannot be in the simulation, we conclude that no witness exists. This goes back all the way to the initial pair $(\iota_1, \{\iota_2\})$. If no more pairs are to be explored, the algorithm concludes that there is a symbolic simulation, and it is returned.

As an optimization, if a simulation pair (q_1', B_2') is to be added as a witness for some pair where $B_2' \supseteq B_2''$ and $(q_1', B_2'') \in H$, we can automatically conclude that (q_1', B_2') will also be verified.[1] We therefore do not need to explore it. The witnesses of (q_1', B_2'') will also serve as witnesses for (q_1', B_2'). Note that in this case, the obtained witnesses are not maximal. Alternatively, it is possible to simply use (q_1', B_2'') instead of (q_1', B_2'). Since this optimization damages the maximality of the witnesses,

[1] Similar optimizations have been used in the antichain-based algorithms for checking automata inclusion [28, 1].

Algorithm 1: Symbolic simulation checking

Input: DSAs $A_1 = \langle Q_1, \delta_1, \iota_1, F_1 \rangle, A_2 = \langle Q_2, \delta_2, \iota_2, F_2 \rangle$
Output: Symbolic simulation $H \subseteq Q_1 \times (2^{Q_2} \setminus \{\emptyset\})$, or $\emptyset$ if does not exist

$toExplore = \{(\iota_1, \{\iota_2\})\}$;
$H = \emptyset$;
repeat
  (q_1, B_2) = $pop(toExplore)$;
  $H = H \cup (q_1, B_2)$;
  if $q_1 \in F_1$ *AND* $B_2 \cap F_2 = \emptyset$ **then**
    return $\emptyset$;
  `// Find witnesses`
  for $l : \delta_1(q_1, l)$ *is defined* **do**
    if $l \in \Sigma$ **then**
      `// find witness for concrete transition`
      $B_2' = \{q_2' \mid \exists q_2 \in B_2$ s.t. $q_2' = \delta_2(q_2, l)\}$;
      if $B_2' = \emptyset$ **then**
        return $\emptyset$;
      else
        if $(\delta_1(q_1, l), B_2') \notin H$ **then**
          $toExplore = toExplore \cup \{(\delta_1(q_1, l), B_2')\}$;
    else
      `// find witness for symbolic transition`
      $B_2' = \{q_2' \mid \exists q_2 \in B_2$ s.t. q_2' is δ_2-reachable from $q_2\}$;
      if $(\delta_1(q_1, l), B_2') \notin H$ **then**
        $toExplore = toExplore \cup \{(\delta_1(q_1, l), B_2')\}$;
until $toExplore = \emptyset$;
return H;

it is not used when maximal witnesses are desired (e.g., when looking for all possible unknown elimination results).

Example 8 Consider the DSAs depicted in Fig. 10(a) and (c). A simulation between these DSAs was presented in Example 7. We now present the simulation computed by the above algorithm, where "maximal" sets are used as the sets simulating a given state.

$$H = \{(0, \{0\}), (1, \{1\}), (2, \{1, \ldots, 12, 21\}), (3, \{3\}), (4, \{4, 10, 21\}), (5, \{7\}), (6, \{12\}), (7, \{11\}), (8, \{8\}), (9, \{13\}), (10, \{13, \ldots, 20\}), (1, \{16\}), (2, \{16, \ldots, 20\}), (3, \{18\}), (4, \{18\}), (5, \{20\}), (6, \{19\}), (7, \{20\}), (8, \{19\})\}.$$

For example, the pair $(2, \{1, \ldots, 12, 21\})$ is computed as an x-witness for $(1, \{1\})$, even though the subset $\{2, 6, 9\}$ of $\{1, \ldots, 12, 21\}$ suffices to simulate state 2.

6 Completion Using Unknown Elimination

Let A_1 be a DSA that is symbolically-included in A_2. This means that the "concrete parts" of A_1 exist in A_2 as well, and the "partial" parts of A_1 have some comple-

tion in A_2. Our goal is to be able to eliminate (some of) the unknowns in A_1 based on A_2. This amounts to finding a (possibly symbolic) assignment to A_1 such that $\sigma(SL(A_1)) \subseteq SL(A_2)$ (whose existence is guaranteed by Theorem 2).

As already shown in the proof of Theorem 6, such an assignment can be obtained from a simulation relation H. There, we described σ by considering each variable and each possible context separately, showing which symbolic word is assigned to the variable in this context.

We are interested in providing some *finite* representation of an assignment σ derived from a simulation H. Namely, for each variable $x \in \mathit{Vars}$, we would like to represent in some finite way the assignments to x in *every* possible context in A_1. When the set of contexts in A_1 is finite, this can be performed for every symbolic word (context) separately as described in the proof of Theorem 6. However, we also wish to handle cases where the set of possible contexts in A_1 is infinite.

Basic idea: Let $x \in \mathit{Vars}$ be a variable. To identify the possible completions of x, we identify all the symbolic transitions labeled by x in A_1, and for each such transition, we identify all the states of A_2 that participate in simulating its source and target states, q_1 and q_1', respectively. The states simulating q_1 and q_1' are given by states in simulation pairs $(q_1, B_2) \in H$ and $(q_1', B_2') \in H$, respectively. The paths in A_2 between states in B_2 and B_2' will provide the completions (assignments) of x; the corresponding contexts will be obtained by tracking the paths in A_1 that lead to (and from) the corresponding simulation pairs, where we make sure that the sets of contexts are pairwise disjoint.

Example 9 As a simple example, consider the simulation H from Example 7, computed for the DSAs from Fig. 10(a) and (c), and consider the variable y. The only symbolic transition labeled y in (a) is the transition from state 9 to state 10. The relevant simulation pairs in H are $(9, \{13\})$ and $(10, \{15\})$. Based on them, we define an assignment to y where the completion is the symbolic word zd leading from 13 to 15 in (c), and the incoming and outgoing contexts are obtained by examining the symbolic words that lead to 9 and the ones that exit 10 in (a). The result is the assignment $\sigma(b, y, ax(b|c)(f|g)) = zd$.

In Example 9, the derivation of an assignment for y from H was simple since each of the source and target states of the corresponding symbolic transition appeared only in one simulation pair. When we consider the variable x, on the other hand, we realize that its source state, 1, appears in two simulation pairs, and so does its target state, 2. The former means that under different incoming contexts leading to 1, a different set from (c) might be needed to simulate 1. Specifically, when reached following a, 1 needs to be simulated by $\{1\}$, and when reached following bya, it is simulated by $\{16\}$. Similarly, since 2 participates in several simulation pairs, it means that under different outgoing contexts exiting 2, a different set might be needed to simulate it. We therefore need to take these contexts into account when defining the completions of x. Technically, we need to "split" the incoming symbolic words of 1 and the outgoing symbolic words of 2 in (a) between four contexts under which x is assigned, where each context is given by another combination of incoming and outgoing con-

texts. Below we describe the general algorithm for computing an assignment, which handles such cases as well.

6.1 Algorithm for Deriving an Assignment from a Symbolic Simulation

Let H be a symbolic simulation from A_1 to A_2. We arbitrarily choose a unique *designated* witness for every simulation pair (q_1, B_2) in H and every transition labeled $l \in \Sigma \cup \mathit{Vars}$ from q_1. Whenever we refer to an l-witness of (q_1, B_2) in the rest of this section, we mean this chosen designated witness. The reason for making this choice will become clear later on. Note that we do not change H, i.e. other witnesses remain in H, but we no longer consider them as witnesses.

For all $x \in \mathit{Vars}$ and for every $q_1, q_1' \in Q_1$ such that $\delta_1(q_1, x) = q_1'$, we do the following:

(a) For every simulation pair $(q_1, B_2) \in H$, we compute a set of incoming contexts, denoted $in(q_1, B_2)$, as described in Section 6.1.1. Intuitively, these contexts represent the incoming contexts of q_1 under which it is simulated by B_2. The sets $in(q_1, B_2)$ are computed such that the sets of different B_2 sets are pairwise-disjoint, and form a partition of the set of incoming contexts of q_1 in A_1.
(b) For every $(q_1', B_2') \in H$ which is the x-witness of some $(q_1, B_2) \in H$, and for every $q_2' \in B_2'$, we compute a set of outgoing contexts, denoted $out(q_1', B_2', q_2')$, as described in Section 6.1.3. These contexts represent the outgoing contexts of q_1' under which it is simulated by the state q_2' of B_2'. The sets $out(q_1', B_2', q_2')$ are computed such that the sets of different states $q_2' \in B_2'$ are pairwise-disjoint and form a partition of the set of outgoing contexts of q_1' in A_1.
(c) For every pair of simulation pairs $(q_1, B_2), (q_1', B_2') \in H$ where (q_1', B_2') is the x-witness of (q_1, B_2), and for every pair of states $q_2 \in B_2$ and $q_2' \in B_2'$ such that q_2 "contributes" q_2' to the witness (see Section 6.1.3), we compute the set of words leading from q_2 to q_2' in A_2. We denote this set by $lang(q_2, q_2')$ (see Section 6.1.2).
(d) Finally, for every pair of simulation pairs $(q_1, B_2), (q_1', B_2') \in H$ where (q_1', B_2') is the x-witness of (q_1, B_2), and for every pair of states $q_2 \in B_2$ and $q_2' \in B_2'$ where q_2 contributes q_2' to the witness, if $in(q_1, B_2) \neq \emptyset$ and $out(q_1', B_2', q_2') \neq \emptyset$, then we define $\sigma(in(q_1, B_2), x, out(q_1', B_2', q_2')) = lang(q_2, q_2')$. For all other contexts, σ is defined arbitrarily.

Note that in step (d), for all the states $q_2 \in B_2$, the same set of incoming contexts is used ($in(q_1, B_2)$), whereas for every $q_2' \in B_2'$, a separate set of outgoing contexts is used ($out(q_1, B_2', q_2')$). This means that assignments to x that result from states in the same B_2 do not differ in their incoming context, but they differ by their outgoing contexts, which is ensured by the property that the sets $out(q_1', B_2', q_2')$ of different states $q_2' \in B_2'$ are pairwise-disjoint. Assignments to x that result from states in different B_2 sets differ in their incoming context, which is ensured by the property that the sets $in(q_1, B_2)$ of different B_2 sets are pairwise-disjoint. Assignments to x that result from different transitions labeled by x also differ in their incoming contexts,

which is ensured by the property that A_1 is deterministic, and hence the set of incoming contexts of each state in A_1 are pairwise disjoint. Altogether, there is a unique combination of incoming and outgoing contexts for each assignment of x.

In the following we formally define in, out, $lang$ and their computation.

6.1.1 Computation of Incoming Contexts

To compute the set $in(q_1, B_2)$ of incoming contexts of q_1 under which it is simulated by B_2, we define the *witness graph* $G_W = (Q_W, \delta_W)$. This is a labeled graph[2] whose states Q_W are all simulation pairs, and whose transitions δ_W are given by the witness relation: $((q_1', B_2'), l, (q_1'', B_2'')) \in \delta_W$ iff (q_1'', B_2'') is the l-witness of (q_1', B_2').

To compute $in(q_1, B_2)$, we derive from G_W a DSA, denoted $A_W(q_1, B_2)$, by setting the initial state to $(\iota^1, \{\iota^2\})$ and the final state to (q_1, B_2). We then define $in(q_1, B_2)$ to be $SL(A_W(q_1, B_2))$, describing all the symbolic words leading from $(\iota^1, \{\iota^2\})$ to (q_1, B_2) along the witness relation. These are the contexts in A_1 for which this witness is relevant.

By our choice of unique witnesses for H, the witness graph is deterministic and hence each incoming context in it will lead to at most one simulation pair. Thus, the sets $in(q_1, B_2)$ partition the incoming contexts of q_1, making the incoming contexts $in(q_1, B_2)$ of different sets B_2 pairwise-disjoint.

6.1.2 Computation of Completions

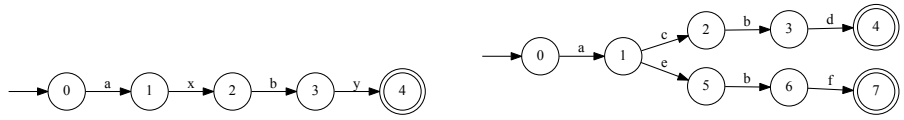
To compute the set $lang(q_2, q_2')$ of completions that arise from paths leading from q_2 to q_2' in A_2, we consider the DSA $A_{q_2 q_2'} = \langle Q_2, \delta_2, q_2, \{q_2'\}\rangle$, whose states and transition relation are as in A_2, but its initial state is q_2, and its only final state is q_2'. We define $lang(q_2, q_2') = SL(A_{q_2 q_2'})$.

6.1.3 Computation of Outgoing Contexts

To compute the set $out(q_1', B_2', q_2')$ of outgoing contexts of q_1' under which it is simulated by the state q_2' of B_2', we define a *contribution relation* based on the witness relation, and accordingly a *contribution graph* G_C. Namely, for $(q_1, B_2), (q_1'', B_2'') \in H$ such that (q_1'', B_2'') is the l-witness of (q_1, B_2), we say that $q_2 \in B_2$ "contributes" $q_2'' \in B_2''$ to the witness if q_2 has a corresponding l-transition (if $l \in \Sigma$) or a corresponding path (if $l \in \mathit{Vars}$) to q_2''. If two states $q_2 \neq q_2'$ in B_2 contribute the same state $q_2'' \in B_2''$ to the witness, then we keep only one of them in the contribution relation.

The *contribution graph* is a labeled graph $G_C = (Q_C, \delta_C)$ whose states Q_C are triples (q_1, B_2, q_2) where $(q_1, B_2) \in H$ and $q_2 \in B_2$. In this graph, a transition $((q_1, B_2, q_2), l, (q_1'', B_2'', q_2'')) \in \delta_C$ exists iff (q_1'', B_2'') is the l-witness of (q_1, B_2) and q_2 contributes q_2'' to the witness. Note that G_C refines G_W in the sense that its states are substates of G_W and so are its transitions. However, unlike G_W, G_C

[2] We use "states" and "transitions" to denote the components of a graph, rather than the traditional notions of "nodes" and "edges", in order to make the transition from graphs to DSAs and vice versa smoother.



(a) (b)

Fig. 12 Example demonstrating the need for consistency in computation of outgoing contexts during unknown elimination.

is nondeterministic since for an x-witness (where $x \in \mathit{Vars}$), a state $q_2 \in B_2$ can contribute multiple states $q_2'' \in B_2''$ (all of which are reachable from it).

To compute $out(q_1', B_2', q_2')$, we derive from G_C a nondeterministic version of our symbolic automaton, denoted $A_C(q_1', B_2', q_2')$, by setting the initial state to (q_1', B_2', q_2') and the final states to triples (q_1, B_2, q_2) where q_1 is a final state of A_1 and q_2 is a final state in A_2. Then $out(q_1', B_2', q_2') = SL(A_C(q_1', B_2', q_2'))$. This is the set of outgoing contexts of q_1' in A_1 for which the state q_2' of the simulation pair (q_1', B_2') is relevant. That is, it is used to simulate some outgoing path of q_1' leading to a final state.

However, the sets $SL(A_C(q_1', B_2', q_2'))$ of different $q_2' \in B_2'$ are not necessarily disjoint since multiple states $q_2' \in B_2'$ can have outgoing transitions that are labeled the same. In order to ensure disjoint sets of outgoing contexts $out(q_1', B_2', q_2')$ for different states q_2' within the same B_2', we need to associate contexts in the intersection of the outgoing contexts of several triples with one of them. Importantly, in order to ensure "consistency" in the outgoing contexts associated with different, but related triples, we require the following *consistency property*: If $\delta_W((q_1, B_2), s) = (q_1', B_2')$, then for every $q_2' \in B_2'$, $\{s\} \cdot out(q_1', B_2', q_2') \subseteq \bigcup\{out(q_1, B_2, q_2) \mid q_2 \in B_2 \wedge (q_1', B_2', q_2') \in \delta_C((q_1, B_2, q_2), s)\}$.

The consistency property ensures that the outgoing contexts associated with some triple (q_1', B_2', q_2') are a subset of the outgoing contexts of triples that lead to it in G_C, truncated by the corresponding word that leads to (q_1', B_2', q_2').

Example 10 To understand the importance of the consistency requirement of the outgoing contexts in the unknown elimination algorithm, consider the DSAs in Fig. 12. DSA (a) is symbolically-included in (b). A possible simulation between the two is:

$$H = \{\ (0, \{0\}), (1, \{1\}), (2, \{2, 5\}), (3, \{3, 6\}), (4, \{4, 7\})\}.$$

We use H to perform unknown elimination in (a) based on (b). When considering the symbolic transition $(1, x, 2)$ in (a), we obtain $in(1, \{1\}) = a$, and $out(2, \{2, 5\}, 2) = out(2, \{2, 5\}, 5) = by$. Similarly, based on the symbolic transition $(3, y, 4)$, we obtain $in(3, \{3, 6\}) = axb$, and $out(4, \{4, 7\}, 4) = out(4, \{4, 7\}, 7) = \epsilon$. In both cases, the out sets are not pairwise disjoint. If we arbitrarily eliminate the intersections, we can get, for example $out(2, \{2, 5\}, 2) = by, out(2, \{2, 5\}, 5) = \emptyset$, and $out(4, \{4, 7\}, 4) = \emptyset, out(4, \{4, 7\}, 7) = \epsilon$. This will result in an assignment $\sigma(a, x, by) = c, \sigma(axb, y, \epsilon) = f$. However, $\sigma(axby) = acbf$ which is not included in the symbolic language of (b). This happens since the out sets computed for the x-transition are not consistent with those computed for the y-transition, even though the latter is reachable from the former. A consistent update of the out sets

can be: $out(2, \{2,5\}, 2) = by$, $out(2, \{2,5\}, 5) = \emptyset$, $out(4, \{4,7\}, 4) = \epsilon$, and $out(4, \{4,7\}, 7) = \emptyset$, resulting in

$$\sigma(a, x, by) = c, \quad \sigma(axb, y, \epsilon) = d,$$

in which case $\sigma(axby) = acbd$.

Note that if $out(q'_1, B'_2, q'_2) = SL(A_C(q'_1, B'_2, q'_2))$, the consistency property holds trivially, as is the case if these sets are already pairwise-disjoint and no additional manipulation is needed. The following lemma ensures that if the intersections of the out sets of different q'_2 states in the same set B'_2 are eliminated in a way that satisfies the consistency property, then correctness is guaranteed.

Lemma 2 *If for all $(q'_1, B'_2, q'_2) \in Q_C$, $out(q'_1, B'_2, q'_2) \subseteq SL(A_C(q'_1, B'_2, q'_2))$, and for all $(q'_1, B'_2) \in Q_W$, $\bigcup_{q'_2 \in B'_2} out(q'_1, B'_2, q'_2) = \bigcup_{q'_2 \in B'_2} SL(A_C(q'_1, B'_2, q'_2))$, and the consistency property holds, then the assignment σ defined as above satisfies $\sigma(SL(A_1)) \subseteq SL(A_2)$.*

Proof We show that the consistency property ensures that σ defined as described above indeed satisfies $\sigma(s) \subseteq SL(A_2)$ for every $s \in SL(A_1)$. To see this, consider $s = w_1 x_1 w_2 x_2 w_3 \dots w_n x_n w_{n+1} \in SL(A_1)$. Let $q_i = \delta_1(\iota^1, w_1 \dots w_i)$ be the state reached in A_1 after traversing the prefix of s up to w_i (before traversing x_i), and let $q'_i = \delta_1(\iota^1, w_1 \dots w_i x_i)$ be the state reached after traversing x_i as well. Let B_i be the unique subset of Q_2 such that $w_1 \dots w_i \in in(q_i, B_i)$, let B'_i be the unique subset of Q_2 such that (q'_i, B'_i) is an x_i-witness for (q_i, B_i), and let $\tilde{q}'_i \in B'_i$ be the unique state of Q_2 such that $w_{i+1} \dots w_{n+1} \in out(q'_i, B'_i, \tilde{q}'_i)$. Moreover, let $\tilde{q}_i \in B_i$ be the unique state that contributed $\tilde{q}'_i$ to B'_i. Then, by definition of σ, σ assigns $lang(\tilde{q}_i, \tilde{q}'_i)$ to x_i in its context in s, where by the choice of $\tilde{q}_i$ and $\tilde{q}'_i$, $lang(\tilde{q}_i, \tilde{q}'_i) \neq \emptyset$. We denote this assignment $\sigma(x_i)$, omitting the corresponding (unique) context. Thus, $\tilde{q}_1, \tilde{q}'_1, \dots, \tilde{q}_i, \tilde{q}'_i, \dots, \tilde{q}_n, \tilde{q}'_n$ is a sequence of states of Q_2, where we know that for every i and for every $s_i \in \sigma(x_i)$, $\delta_2(\tilde{q}_i, s_i) = \tilde{q}'_i$. In order to show that $\sigma(s) = \{w_1\}\sigma(x_1)\{w_2\} \dots \{w_n\}\sigma(x_n\{\{w_{n+1}\} \subseteq SL(A_2)$, it remains to show that (1) $\delta_2(\iota^2, w_1) = \tilde{q}_1$, (2) $\delta_2(\tilde{q}'_i, w_{i+1}) = \tilde{q}_{i+1}$, and (3) $\delta_2(\tilde{q}'_n, w_{n+1}) \in F_2$.

Properties (1) and (3) follow immediately from the definition of in and out and the properties that $w_1 \in in(q_1, B_1)$ and $w_{n+1} \in out(q'_n, B'_n, \tilde{q}'_n)$. We show (2), i.e. that $\delta_2(\tilde{q}'_i, w_{i+1}) = \tilde{q}_{i+1}$. By definition of the in sets based on the witness graph, we know that (q_i, B_i) is the unique $w_1 \dots w_i$-witness for $(\iota^1, \{\iota^2\})$, and (q_{i+1}, B_{i+1}) is the unique $w_1 \dots w_i x_i w_{i+1}$-witness for $(\iota^1, \{\iota^2\})$. Moreover, recall that (q'_i, B'_i) is the unique x_i-witness for (q_i, B_i). This means that (q'_i, B'_i) is also the unique $w_1 \dots w_i x_i$-witness for $(\iota^1, \{\iota^2\})$. Due to our particular choice of witnesses for H, we conclude that (q_{i+1}, B_{i+1}) is the unique w_{i+1}-witness of (q'_i, B'_i).

Since $\delta_W((q'_i, B'_i), w_{i+1}) = (q_{i+1}, B_{i+1})$, the consistency requirement implies that for $\tilde{q}_{i+1} \in B_{i+1}$, $\{w_{i+1}\} \cdot out(q_{i+1}, B_{i+1}, \tilde{q}_{i+1}) \subseteq \bigcup_{\tilde{q} \in B'_i} \{out(q'_i, B'_i, \tilde{q}) \mid (q_{i+1}, B_{i+1}, \tilde{q}_{i+1}) \in \delta_C((q'_i, B'_i, \tilde{q}), w_{i+1})\}$. In our case,

$$x_{i+1} w_{i+2} \dots w_{n+1} \in out(q_{i+1}, B_{i+1}, \tilde{q}_{i+1}).$$

Therefore,

$$w_{i+1}x_{i+1}w_{i+2}\dots w_{n+1} \in \bigcup_{\tilde{q}\in B'_i} \{out(q'_i, B'_i, \tilde{q}) \mid (q_{i+1}, B_{i+1}, \tilde{q}_{i+1}) \in \delta_C((q'_i, B'_i, \tilde{q}), w_{i+1})\}.$$

This means that for some $\tilde{q} \in B'_i$, $w_{i+1}x_{i+1}w_{i+2}\dots w_{n+1} \in out(q'_i, B'_i, \tilde{q})$ and $(q_{i+1}, B_{i+1}, \tilde{q}_{i+1}) \in \delta_C((q'_i, B'_i, \tilde{q}), w_{i+1})$. Since out is a partition and

$$w_{i+1}x_{i+1}w_{i+2}\dots w_{n+1} \in out(q'_i, B'_i, \tilde{q}'_i),$$

we conclude that $\tilde{q} = \tilde{q}'_i$ satisfies the above, and in particular $\tilde{q}_{i+1} = \delta_2(\tilde{q}'_i, w_{i+1})$, which concludes (2). □

In general, eliminating the intersections of the $out(q'_1, B'_2, q'_2)$ sets of different states $q'_2 \in B'_2$ in a consistent way might be hard. However, in many cases this can be achieved by simple heuristics. For example, if no two symbolic transitions are reachable from each other in A_1, then the intersections can be eliminated by arbitrarily choosing one of the triples and associating the intersection with it. If A_1 contains no loops, then a greedy algorithm can be used, where triples are ordered by reachability, and handled according to this order. Specifically, if $\delta_W((q_1, B_2), s) = (q'_1, B'_2)$ for some s then triples of the form (q'_1, B'_2, q'_2) are handled before triples of the form (q_1, B_2, q_2). Triples (q'_1, B'_2, q'_2) that are handled first in such a (acyclic) sequence of dependencies, eliminate the intersections of the out sets arbitrarily. The following triples in the chain of dependencies eliminate intersections of the out sets while maintaining consistency with respect to the consequent triples. This is the case in Example 10, where out is first defined for $(4, \{4, 7\}, 4)$ and $(4, \{4, 7\}, 7)$, and only later defined for $(2, \{2, 5\}, 2)$ and $(2, \{2, 5\}, 5)$ in a consistent way.

Example 11 Consider the simulation H from Example 7, computed for the DSAs from Fig. 10(a) and (c). Unknown elimination based on H will yield the following assignment: $\sigma(bya, x, (b|c)(f|g)) = d$, $\sigma(b, y, ax(b|c)(f|g)) = zd$, $\sigma(a, x, b(f|g)) = d$, $\sigma(a, x, cg) = eh^*e$, and $\sigma(a, x, cf) = h$. All other contexts are irrelevant and assigned arbitrarily. The assignments to x are based on the symbolic transition $(1, x, 2)$ in (a) and on the simulation pairs $(1, \{1\})$ and $(1, \{16\})$, as well as their x-witnesses $(2, \{2, 6, 9\})$ and $(2, \{17\})$, respectively. Concretely, consider the simulation pair $(q_1, B_2) = (1, \{1\})$ and its witness $(q'_1, B'_2) = (2, \{2, 6, 9\})$. Then $B_2 = \{1\}$ contributed the incoming context $in(1, \{1\}) = a$, and each of the states $2, 6, 9 \in B'_2 = \{2, 6, 9\}$ contributed the outgoing contexts $out(2, \{2, 6, 9\}, 2) = b(f|g)$, $out(2, \{2, 6, 9\}, 6) = cg$ and $out(2, \{2, 6, 9\}, 9) = cf$, respectively. In this example, the out sets are pairwise-disjoint, thus no further manipulation is needed. Note that had we considered the simulation computed in Example 8, where the x-witness for $(1, \{1\})$ is $(2, \{2, \dots 12, 20\})$, we would still get the same assignment because for any $q \neq 2, 6, 9$, $out(2, \{2, \dots 12, 20\}, q) = \emptyset$. Similarly, $(1, \{16\})$ contributed $in(1, \{16\}) = bya$, and the (only) state $17 \in \{17\}$ contributed $out(2, \{17\}, 17) = (b|c)(f|g)$. The assignment to y is based on the symbolic transition $(9, x, 10)$ and the corresponding simulation pair $(9, \{13\})$ and its y-witness $(10, \{15\})$ (see also Example 9).

7 Consolidation Using Join and Minimization

Consolidation consists of (1) union, which corresponds to join in the lattice over equivalence classes, and (2) choosing a "most complete" representative from an equivalence class, where "most complete" is captured by having a minimal set of completions.

Note that DSAs A, A' in the same equivalence class do not necessarily have the same set of completions. Therefore, it is possible that $[\![A]\!] \neq [\![A']\!]$ (as is the case in Example 6). A DSA A is "most complete" in its equivalence class if there is no equivalent DSA A' such that $[\![A']\!] \subset [\![A]\!]$. Thus, A is most complete if its set of completions is minimal.

Let A be a DSA for which we look for an equivalent DSA A' that is most complete. If $[\![A]\!]$ itself is not minimal, there exists A' such that A' is equivalent to A but $[\![A']\!] \subset [\![A]\!]$. Equivalence means that (1) for every $L' \in [\![A']\!]$ there exists $L \in [\![A]\!]$ such that $L \subseteq L'$, and (2) conversely, for every $L \in [\![A]\!]$ there exists $L' \in [\![A']\!]$ such that $L' \subseteq L$. Requirement (1) holds trivially since $[\![A']\!] \subset [\![A]\!]$. Requirement (2) is satisfied iff for every $L \in [\![A]\!] \setminus [\![A']\!]$ (a completion that does not exist in the minimal DSA), there exists $L' \in [\![A']\!]$ such that $L' \subseteq L$ (since for $L \in [\![A]\!] \cap [\![A']\!]$ this holds trivially).

Namely, our goal is to find a DSA A' such that $[\![A']\!] \subset [\![A]\!]$ and for every $L \in [\![A]\!] \setminus [\![A']\!]$ there exists $L' \in [\![A']\!]$ such that $L' \subseteq L$. Clearly, if for some $L \in [\![A]\!]$ there is no $L' \in [\![A]\!]$ such that $L' \subset L$, then if $L \notin [\![A']\!]$, the requirement will not be satisfied. This means that the only completions L that can be removed from $[\![A]\!]$ are themselves non-minimal, i.e., are supersets of other completions in $[\![A]\!]$.

Note that it is in general impossible to remove from $[\![A]\!]$ all non-minimal languages: as long as $SL(A)$ contains at least one symbolic word $s \in (\Sigma \cup \mathit{Vars})^* \setminus \Sigma^*$, there are always multiple completions in $[\![A]\!]$ that are comparable by language inclusion (and hence are not all minimal). Namely, if assignments σ and σ' differ only on their assignment to some variable x in s (with the corresponding context), where σ assigns to it L_x and σ' assigns to it L'_x where $L_x \supset L'_x$, then $L = \sigma(SL(A)) = \sigma(SL(A) \setminus \{s\}) \cup \sigma(s) \supset \sigma'(SL(A) \setminus \{s\}) \cup \sigma'(s) = \sigma'(SL(A')) = L'$. Therefore $L \supset L'$ where both $L, L' \in [\![A]\!]$. On the other hand, not every DSA has an equivalent concrete DSA, whose language contains no symbolic word. For example, consider a DSA A_x such that $SL(A_x) = \{x\}$, i.e. $[\![A_x]\!] = 2^{\Sigma^*} \setminus \{\emptyset\}$. Then for every concrete DSA A_c with $[\![A_c]\!] = \{SL(A_c)\}$, there is $L_x \in [\![A_x]\!]$ such that either $L_x \supset SL(A_c)$, in which case $A_x \not\preceq A_c$, or $SL(A_c) \supset L_x$, in which case $A_c \not\preceq A_x$. Therefore, symbolic words are a possible source of non-minimality, but they cannot always be avoided.

Below we provide a condition which ensures that we remove from $[\![A]\!]$ only non-minimal completions. The intuition is that non-minimality of a completion can arise from a variable in A whose context matches the context of some known behavior. In this case, the minimal completion will be obtained by assigning to the variable the matching known behavior, whereas other assignments will result in supersets of the minimal completion. Or in other words, to keep only the minimal completion, one needs to remove the variable in this particular context.

Example 12 This intuition is demonstrated by Example 6, where the set of completions of the DSA from Fig. 11(b) contains non-minimal completions due to the symbolic word axb that co-exists with the word adb in the symbolic language of the DSA. Completions resulting by assigning d to x are strict subsets of completions assigning to x a different language, making the latter non-minimal. The DSA from Fig. 11(a) omits the symbolic word axb, keeping it equivalent to (b), while making its set of completions smaller (due to removal of non-minimal completions resulting from assignments that assign to x a language other than d).

Definition 11 Let A be a DSA. An accepting path π in A is *redundant* if there exists another accepting path π' in A such that $\pi \preceq \pi'$. A symbolic word $s \in SL(A)$ is *redundant* if its (unique) accepting path is redundant.

This means that a symbolic word is redundant if it is "less complete" than another symbolic word in $SL(A)$. In particular, symbolic words where one can be obtained from the other via renaming of variables are redundant. Such symbolic words are called *equivalent* since their corresponding accepting paths π and π' are symbolically-equivalent.

In Example 12, the path $\langle 0, 1, 6, 7\rangle$ of the DSA in Fig. 11(b) is redundant due to $\langle 0, 1, 2, 3\rangle$. Accordingly, the symbolic word axb labeling this path is also redundant.

An equivalent characterization of redundant paths is the following:

Definition 12 For a DSA A and a path π in A, we use $A \setminus \pi$ to denote a DSA such that $SL(A \setminus \pi) = SL(A) \setminus SL(\pi)$.

Lemma 3 *Let A be a DSA. An accepting path π in A is* redundant *iff $\pi \preceq A \setminus \pi$.*

Proof Let π, π' be two different accepting paths in A such that $\pi \preceq \pi'$. Let $SL(\pi) = \{s\}, SL(\pi') = \{s'\}$, where $s, s' \in SL(A)$. Since A is deterministic, $s \neq s'$. Therefore $SL(\pi) = \{s\} \subseteq SL(A) \setminus \{s'\} = SL(A) \setminus SL(\pi) = SL(A \setminus \pi)$. By Corollary 1, $\pi \preceq A \setminus \pi$.

Let π be an accepting path in A such that $\pi \preceq A \setminus \pi$. Then there exists a symbolic assignment σ to π such that $\sigma(SL(\pi)) \subseteq SL(A \setminus \pi) = SL(A) \setminus SL(\pi)$ (*). Furthermore, by Lemma 1, there exists such an assignment that maps each word in $SL(\pi)$ to a singleton language. Suppose $SL(\pi) = \{s\}$, and assume that $\sigma(s) = \{s'\}$, then $\sigma(SL(\pi)) = \sigma(s) = \{s'\}$. Thus by (*), $s' \in SL(A) \setminus \{s\}$. This ensures that s' is in $SL(A)$ and that $s \neq s'$. Therefore, there exists an accepting path $\pi' \neq \pi$ in A such that $SL(\pi') = \{s'\}$. The same symbolic assignment σ also witnesses structural inclusion of π in π' since $\sigma(SL(\pi)) = \{s'\} \subseteq SL(\pi')$. We conclude that $\pi \preceq \pi'$. □

Theorem 8 *If π is a redundant path, then $(A \setminus \pi) \equiv A$, and $[\![A \setminus \pi]\!] \subseteq [\![A]\!]$, i.e. $A \setminus \pi$ is at least as complete as A.*

Proof First, $A \setminus \pi \preceq A$ since $SL(A \setminus \pi) \subseteq SL(A)$ (see Corollary 1). For the other direction, recall that $\pi \preceq A \setminus \pi$ (by Lemma 3), and hence there exists a symbolic assignment σ such that $\sigma(SL(\pi)) \subseteq SL(A \setminus \pi)$. We define a symbolic assignment σ' that agrees with σ on the single word in $SL(\pi)$, and assigns to any other $x \in$ *Vars* in

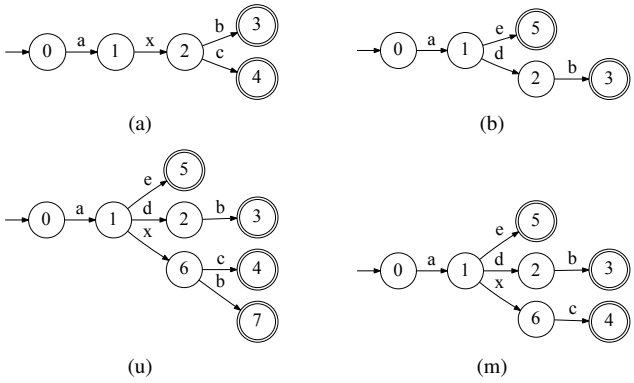


Fig. 13 Inputs (a) and (b), union (u) and minimized DSA (m).

any other context $\{x\}$. Therefore, $\sigma'(SL(A)) = \sigma(SL(\pi)) \cup SL(A \setminus \pi) = SL(A \setminus \pi)$, and σ' witnesses structural inclusion of A in $A \setminus \pi$. We conclude that $A \preceq A \setminus \pi$.

We show that $[\![A \setminus \pi]\!] \subseteq [\![A]\!]$. Let $L \in [\![A \setminus \pi]\!]$, and let σ_L be a concrete assignment such that $\sigma_L(SL(A \setminus \pi)) = L$. Moreover, let σ be such that $\sigma(SL(\pi)) \subseteq SL(A \setminus \pi)$ (as defined above), where we assume that σ assigns a singleton language to the single word in $SL(\pi)$. We use the composition of σ_L over σ to define a concrete assignment for the single word in $SL(\pi)$, and use σ_L for any other word in $SL(A)$. The result is a concrete assignment σ'_L such that $\sigma'_L(SL(A)) = \sigma'_L(SL(\pi)) \cup \sigma'_L(A \setminus \pi) = \sigma_L(\sigma(SL(\pi))) \cup \sigma_L(A \setminus \pi) = L$, where the last equality holds since $\sigma(SL(\pi)) \subseteq SL(A \setminus \pi)$, and hence $\sigma_L(\sigma(SL(\pi))) \subseteq SL(A \setminus \pi) = L$. Therefore, $L \in [\![A]\!]$ as well. □

Theorem 8 leads to a natural semi-algorithm for minimization by iteratively identifying and removing redundant paths. Several heuristics can be employed to identify such redundant paths.

In fact, when considering minimization of A into some A' such that $SL(A') \subseteq SL(A)$, it turns out that a DSA without redundant paths cannot be minimized further:

Theorem 9 *If $A \equiv (A \setminus \pi)$ for some accepting path π in A, then π is redundant in A.*

Proof Suppose $A \preceq A \setminus \pi$, and let $SL(\pi) = \{s\}$. Assume to the contrary that the path π is not redundant in A. This means that $\pi \not\preceq A \setminus \pi$. Thus, there is an assignment σ' to $A \setminus \pi$ such that for every assignment σ to π, $\sigma(SL(\pi)) = \sigma(s) \not\subseteq \sigma'(SL(A \setminus \pi))$ (*). This implies that for every assignment σ to A, $\sigma(SL(A)) \not\subseteq \sigma'(SL(A \setminus \pi))$ (otherwise $\sigma(s) \subseteq \sigma(SL(A)) \subseteq \sigma'(SL(A \setminus \pi))$ in contradiction to (*)). We conclude that $A \not\preceq A \setminus \pi$, in contradiction to the assumption. □

The theorem implies that for a DSA A without redundant paths there exists no DSA A' such that $SL(A') \subset SL(A)$ and $A' \equiv A$, thus it cannot be minimized further by removal of paths (or words).

Fig. 13 provides an example for consolidation via union (which corresponds to join in the lattice), followed by minimization.

8 Putting It All Together

Now that we have completed the description of symbolic automata, we describe how they can be used in a static analysis for specification mining. We return to the example in Section 2, and emulate an analysis using the new abstract domain. This analysis would combine a set of program snippets into a typestate for a given API or class, which can then be used for verification or for answering queries about API usage.

Firstly, the DSAs in Fig. 1 and Fig. 2 would be mined from user code using the analysis defined by Mishne et al [16]. In this process, code that may modify the object but is not available to the analysis becomes a variable transition.

Secondly, we generate a typestate specification from these individual DSAs. As shown in Section 2, this is done using the join operation, which consolidates the DSAs and generates the one in Fig. 3(b). This new typestate specification is now stored in our specification database. If we are uncertain that all the examples which we are using to create the typestate are correct, we can add weights to DSA transitions, and later prune low-weight paths, as suggested by Mishne et al.

Finally, a user can query against the specification database, asking for the correct sequence of operations between `open` and `close`, which translates to querying the symbolic word $open \cdot x \cdot close$. Unknown elimination will find an assignment such that $\sigma(x) = canRead \cdot read$, as well as the second possible assignment, $\sigma(x) = write$.

The precision/partialness ordering of the lattice captures the essence of query matching. A query will always have a $\preceq$ relationship with its results: the query will always be *more partial* than its result, allowing the result to contain the query's assignments, as well as *more precise*, which means a DSA describing a great number of behaviors can contain the completions for a very narrow query.

Acknowledgements The research was partially supported by The Israeli Science Foundation (grant no. 965/10) and EU's FP7 Program / ERC agreement no. 615688. Yang was partially supported by EPSRC. Peleg was partially supported by EU's FP7 Program / ERC agreement no. 321174. Shoham was partially supported by BSF grant no. 2012259.

References

1. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: TACAS, pp. 158–174 (2010)
2. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 25–34. ACM (2007)
3. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, pp. 98–109. ACM (2005)
4. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, pp. 4–16. ACM (2002)
5. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE TSE **33**(9), 577–591 (2007)
6. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**(3), 215–249 (1998). DOI http://doi.acm.org/10.1145/287000.287001

7. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA '06, pp. 17–24. ACM (2006)
8. David, Y., Yahav, E.: Tracelet-based code search in executables. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pp. 349–360. ACM, New York, NY, USA (2014). DOI 10.1145/2594291.2594343. URL http://doi.acm.org/10.1145/2594291.2594343
9. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: whats decidable? In: Haifa Verification Conference, HVC'12, *Lecture Notes in Computer Science*, vol. 7857, pp. 209–226. Springer (2012)
10. Gruska, N., Wasylkowski, A., Zeller, A.: Learning from 6,000 projects: Lightweight cross-project anomaly detection. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pp. 119–130. ACM (2010)
11. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: Proc. ACM Conf. on Programming Language Design and Implementation, pp. 234–245 (1990)
12. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI '88 (1988). DOI 10.1145/53990.53994. URL http://doi.acm.org/10.1145/53990.53994
13. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: P. Cousot (ed.) Static Analysis, *Lecture Notes in Computer Science*, vol. 2126, pp. 40–56. Springer Berlin Heidelberg (2001)
14. Lo, D., Khoo, S.C.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 265–275. ACM (2006)
15. Mariani, L., Pezzè, M.: Dynamic detection of COTS component incompatibility. IEEE Software **24**(5), 76–85 (2007)
16. Mishne, A., Shoham, S., Yahav, E.: Typestate-based semantic code search over partial programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pp. 997–1016. ACM (2012)
17. Monperrus, M., Bruch, M., Mezini, M.: Detecting missing method calls in object-oriented software. In: Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP'10, *LNCS*, vol. 6183, pp. 2–25 (2010)
18. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: F. Logozzo, M. Fhndrich (eds.) Static Analysis, *Lecture Notes in Computer Science*, vol. 7935, pp. 238–258. Springer Berlin Heidelberg (2013)
19. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14 (2014)
20. Peleg, H., Shoham, S., Yahav, E., Yang, H.: Symbolic automata for static specification mining. In: Proceedings of Static Analysis - 20th International Symposium, SAS 2013, *Lecture Notes in Computer Science*, vol. 7935, pp. 63–83. Springer (2013)
21. Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of the thirty-eighth annual ACM Symposium on Theory of Computing, STOC '06, pp. 467–476. ACM (2006)
22. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pp. 419–428. ACM, New York, NY, USA (2014). DOI 10.1145/2594291.2594321. URL http://doi.acm.org/10.1145/2594291.2594321
23. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, pp. 174–184. ACM (2007)
24. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986)
25. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 35–44. ACM (2007)
26. Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05, pp. 461–476 (2005)
27. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02, pp. 218–228. ACM (2002)

28. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: CAV, pp. 17–30 (2006)
29. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pp. 282–291. ACM (2006)