Symbolic Automata for Static Specification Mining

Hila Peleg[1], Sharon Shoham[2], Eran Yahav[3], and Hongseok Yang[4]

[1] Tel Aviv University, Israel [2] Tel Aviv-Yaffo Academic College, Israel [3] Technion, Israel
[4] University of Oxford, UK

Abstract. We present a formal framework for static specification mining. 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 specification mining, 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.

1 Introduction

Programmers make extensive use of frameworks and libraries. To perform standard tasks such as parsing an XML file or communicating with a database, programmers use standard frameworks rather than writing code from scratch. Unfortunately, a typical framework API can involve hundreds of classes with dozens of methods each, and often requires sequences of operations to be invoked on specific objects to perform a single task (e.g., [14, 6, 12, 3, 13]). Even experienced programmers might spend hours trying to understand how to use a seemingly simple API [6].

Static specification mining techniques (e.g., [10, 7, 2, 15]) have emerged as a way to obtain a succinct description of usage scenarios when working with a library. However, although they demostrated great practical value, these techniques do not address many interesting and challenging technical questions.

In this paper, we present a formal framework for static specification mining. The main idea is to 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 static specification mining, and show interesting relationships between the *partialness* and the *precision* of a specification.

Representing Partial Specifications using Symbolic Automata We focus on generalized typestate specifications [11, 7]. 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.

To make specification mining more widely applicable, it is critical to allow mining from *code snippets*, i.e., code fragments with unknown parts. A natural approach for mining from code snippets is to capture gaps in the snippets using gaps in the specification. For example, when the code contains an invocation of an unknown method, this approach reflects this fact in the mined specification as well (we elaborate on this point later). Our *symbolic automaton* is conceived in order to represent such 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 languages 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 Mining 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, that 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. Treating one symbolic automaton as a query being matched against a database 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 formally define the notion of *partial typestate 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, and for completing certain partial parts of such specifications.

Related Work Mishne et. al [7] present a practical framework for static specification mining and query matching based on automata. Their framework imposes restrictions on the structure of automata and they could be viewed as a restricted 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.

Weimer and Necula [14] use a lightweight static analysis to infer simple specifications from a given codebase. Their insight is to use exceptional program paths as negative examples for correct API usage. They learn specifications consisting of pairs of events $\langle a, b\rangle$, where a and b are method calls, and do not consider larger automata.

Monperrus et. al [8] attempt to identify missing method calls when using an API by mining a codebase. They only compare objects with identical type and same containing method signature, which only works for inheritance-based APIs. Their approach deals with identical histories minus k method calls. Unlike our approach, it cannot handle incomplete programs, non-linear method call sequences, and general code queries.

Wasylkowski et. al [13] 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 [5] considers limited specifications that are only pairs of events. [1] also mines pairs of events in an attempt to mine partial order between events. [12] mine specifications (operational preconditions) of method parameters to detect problems in code. The mined specifications are CTL formulas that fit into several pre-defined templates of formulas. Thus, the user has to know what kind of specifications she is looking for.

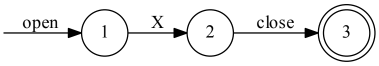
Shoham et. al [10] use a whole-program analysis to statically analyze clients using a library. Their approach is not applicable in the setting of partial programs and partial specification since they rely on the ability to analyze the complete program for complete alias analysis and for type information.

Plandowski [9] uses the field of word equations to identify assignments to variables within conditions on strings with variable portions and regular expression. Ganesh et. al [4] expand this work with quantifiers and limits on the assignment size. In both cases, the regular language that the assignments consist of does not allow variables, disallowing the concept of symbolic assignments of variables within the branching of the automata for the regular language. 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.

2 Overview

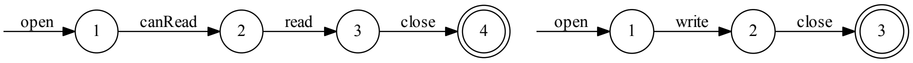
We start with an informal overview of our approach by using a simple `File` example.

```
void process(File f) {
  f.open();
  doSomething(f);
  f.close();
}
```



(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.



(a) (b)

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

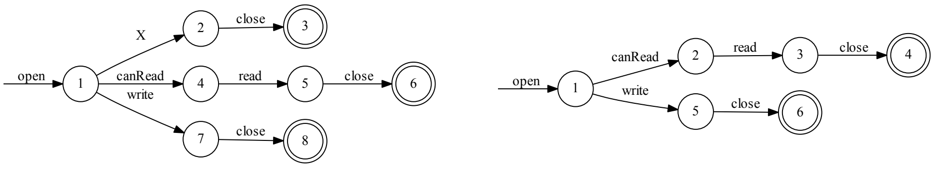
2.1 Illustrative Example

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 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.

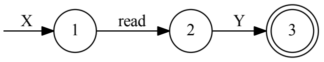
Our first step is to *consolidate* the specifications into a more comprehensive specification, describing as much of the API as possible, while losing no behavior represented by the original specifications.

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)).



(a) (b)

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.



$\texttt{X} \mapsto \texttt{open}$
$\texttt{Y} \mapsto \texttt{close}$

(a) (b)

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.

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).

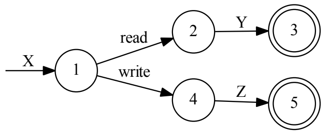
As this very partial specification is included in the API's specification, there will be a match. Furthermore, we can deduce what should replace the symbolic portions of the query. This means the user can get the reply to his query that `X` should be replaced by `open` and `Y` by `close`.

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



(a)

(*,X,`read`) $\mapsto$ `open`,`canRead`
(*,X,`write`) $\mapsto$ `open`
`Y` $\mapsto$ `close`
`Z` $\mapsto$ `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.

without a bound due to loops or recursion [10]. In Section 4.2, we show how to obtain a lattice based on symbolic automata.

In second role — used for consolidation and answering code-search queries — query matching can be understood in terms of *unknown elimination* in a 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:

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 alphabets 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.

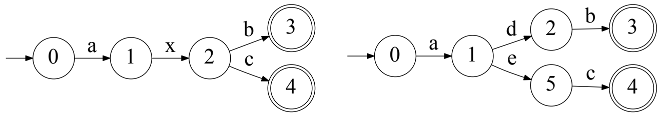
We write $(q, l, q') \in \delta$ for a transition $\delta(q, l) = q'$ where $q, q' \in Q$ and $l \in \Sigma \cup \mathit{Vars}$. If $l \in \mathit{Vars}$, the transition is called *symbolic*. We extend δ to words over $\Sigma \cup \mathit{Vars}$ in the usual way. Note that this extension of δ over words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

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.,

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



(a) (b)

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

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

When σ maps (sw_1, x, sw_2) to SL, we refer to (sw_1, sw_2) as the *context* of x. The meaning is that an occurrence of x in the context (sw_1, sw_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 $sw = l_1 l_2 \dots l_n$, where $l_i \in \Sigma \cup \mathit{Vars}$ for every $1 \leq i \leq n$,

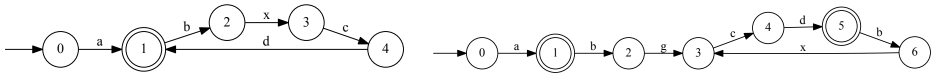
$$\sigma(sw) = SL_1 SL_2 \dots SL_n$$

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, $\sigma(SL) = \bigcup\{\sigma(sw) \mid sw \in SL\}$.

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.



(a) (b)

Fig. 7. DSA before and after assignment

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 $\sigma(sw_1, x, sw_2) = SL$ for every $(sw_1, sw_2) \in C_1 \times C_2$. 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:

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

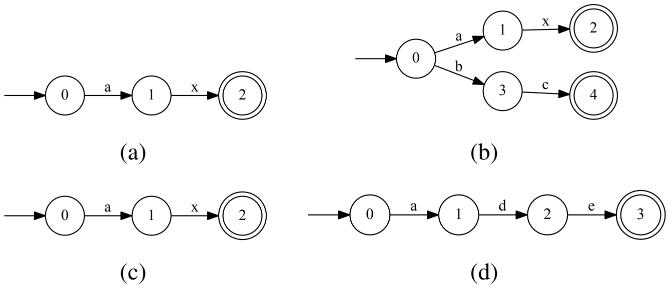
We call $[\![A]\!]$ the set of *completions* of A.

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 $[\![A]\!] = \{SL(A)\}$.

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.



(a) (b)

(c) (d)

Fig. 8. Dimensions of the preorder on DSAs

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.

Preorder on Symbolic Words

Definition 4. *Let sw_1, sw_2 be symbolic words. $sw_1 \leq sw_2$ if for every concrete assignment σ_2 to sw_2, there is a concrete assignment σ_1 to sw_1 such that $\sigma_1(sw_1) = \sigma_2(sw_2)$.*

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 sw_2 (using a concrete assignment), the same completion can also be obtained by filling in the unknowns of sw_1, ensures that every unknown of sw_2 is also unknown in sw_1 (which can be filled in the same way), but sw_1 can have additional unknowns. Thus, sw_2 has "no more" unknowns than sw_1. In particular, $\{\sigma(sw_1) \mid \sigma \text{ is a concrete assignment}\} \supseteq \{\sigma(sw_2) \mid \sigma \text{ is a concrete assignment}\}$. 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 replationship holds because $abxcd$ is more complete (carries more information) than ayd.

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. Equivalently: 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 that is 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.*

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.*

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

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

Example 4. The DSA depicted in Fig. 9(a) is not symbolically-included in the one depicted in Fig. 9(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.

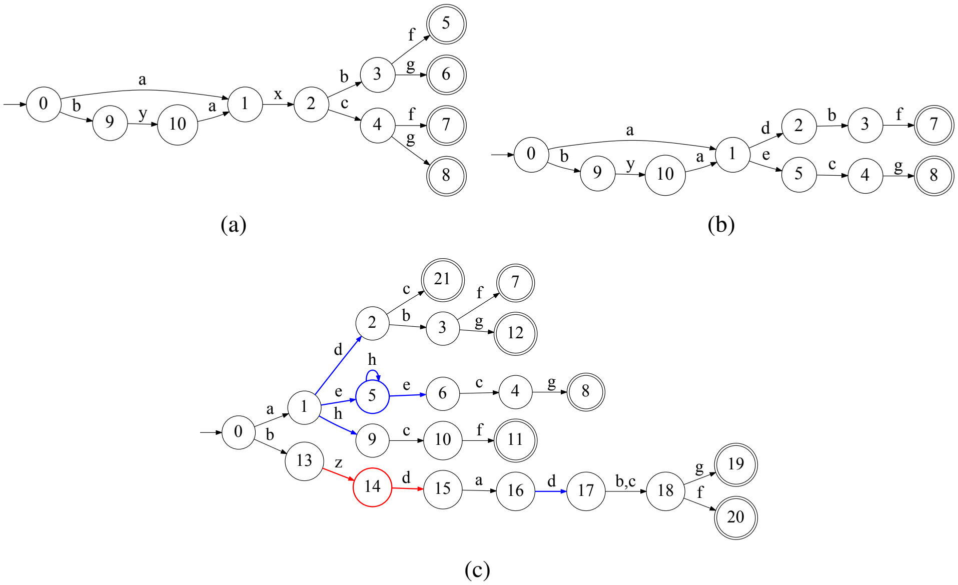


Fig. 9. Example for a case where there is no assignment to either (a) or (b) to show (a) $\preceq$ (b) or (b) $\preceq$ (a), and where there is such an assignment for (a) so that (a) $\preceq$ (c).

On the other hand, the DSA depicted in Fig. 9(a) is symbolically-included in the one depicted in Fig. 9(c), since there is a witnessing assignment that maintains all the contexts of x: $\sigma : (a, x, b*) \mapsto d, (a, x, cf*) \mapsto h, (a, x, cg*) \mapsto eh^{*}e, (bya, x, *) \mapsto d, (*, y, *) \mapsto zd$. 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 5. The DSAs in Fig. 10 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). Examining the example closely shows that the reason that symbolic-inclusion

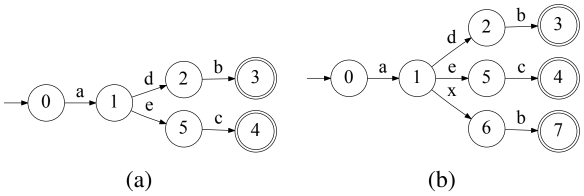


Fig. 10. Equivalent DSAs w.r.t. symbolic-inclusion

also holds in the other direction is the fact 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 $\mathbf{DSA}/\equiv$.*

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, 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. $\sqsubseteq$.*

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 $a \in \Sigma$,

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

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

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

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 $l \in \Sigma \cup$ 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 6. Consider the DSAs in Fig. 9(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.

Theorem 7 (Completeness). *For al 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.*

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. 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 is removed. 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) is removed. When a simulation pair is removed, any simulation pair for which it is a witness and no other witness exists is also removed (for efficiency, we also remove all its witnesses that are not witnesses for any other pairs). If at some point $(\iota^1, \{\iota^2\})$ is removed, then the algorithm concludes that A_1 is not symbolically simulated by A_2. If no more pairs are to be explored, the algorithm concludes that there is a symbolic simulation, and it is returned.

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). In both cases, if q_1' is a final state, we make sure that B_2' contains at least one final state as well. Otherwise, no witness exists.

In order to prevent checking the same simulation pair, or related pairs, over and over again, we keep all removed pairs. When a witness (q_1', B_2') is to be added as a simulation pair, we make sure that no simulation pair (q_1', B_2'') where $B_2' \subseteq B_2''$ was already removed. If such a pair was removed, then clearly, (q_1', B_2') will also be removed. Moreover, since B_2' was chosen as the maximal set that fulfills the requirement, any other possible witness will comprise of its subset and will therefore also be removed. Thus, in this case, no witness is obtained.

As an optimization, when for some simulation pair (q_1, B_2) we identify that all the witnesses reachable from it have been verified and remained as simulation pairs, we mark (q_1, B_2) as verified. If a simulation pair (q_1, B'_2) is to be added as a witness for some pair where $B'_2 \supseteq B_2$, we can automatically conclude that (q_1, B'_2) will also be verified. We therefore mark it immediately as verified, and consider the witnesses of (q_1, B_2) as its witnesses as well. 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, it is not used when maximal witnesses are desired (e.g., when looking for all possible unknown elimination results).

Example 7. Consider the DSAs depicted in Fig. 9(a) and (c). A simulation between these DSAs was presented in Example 6. We now present the simulation computed by the above algorithm, where "maximal" sets are used as the sets simulating a given state.

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

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

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, in this section we also wish to handle cases where the set of possible contexts in A_1 is infinite.

We choose a unique witness for every simulation pair (q_1, B_2) in H and every transition $l \in \Sigma \cup \mathit{Vars}$ 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 witness. The reason for making this choice will become clear later on.

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, where 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.

Formally, for all q_1, q_1', x with $\delta(q_1, x) = q_1'$, we do the following:

(a) For every simulation pair $(q_1, B_2) \in H$ we compute a set of incoming contexts, denoted $in(q_1, B_2)$ (see *computation of incoming contexts* in the next page). 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 an 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 $out(q_1', B_2', q_2')$ (see *computation of outgoing contexts*). 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 an x-witness, 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 *computation of outgoing contexts*), we compute the set of words leading from q_2 to q_2' in A_2. We denote this set by $lang(q_2, q_2')$. The "contribution" relation ensures that for every state $q_2 \in B_2$ there is at most one state $q_2' \in B_2'$ such that $lang(q_2, q_2') \neq \emptyset$.

(d) Finally, for every pair of simulation pairs $(q_1, B_2), (q_1', B_2') \in H$ where (q_1', B_2') is an x-witness of (q_1, B_2), and for every pair of states $q_2 \in B_2$ and $q_2' \in B_2'$, if $in(q_1, B_2) \neq \emptyset$ and $out(q_1', B_2', q_2') \neq \emptyset$ and $lang(q_2, q_2') \neq \emptyset$, 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, as 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, as 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, as 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.

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 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 a 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 particular choice of 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.

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 an 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 an 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 W_C, G_C is nondeterministic since multiple states $q_2 \in B_2$ can have outgoing l-transitions.

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. 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 $\delta_W((q_1, B_2), sw) = (q_1', B_2')$ then for every $q_2' \in B_2'$, $\{sw\} \cdot out(q_1', B_2', q_2') \subseteq \bigcup\{out(q_1, B_2, q_2) \mid q_2 \in B_2 \wedge (q_1', B_2', q_2') \in \delta_C((q_1, B_2, q_2), sw)\}$.

This means 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').

Note that this property holds trivially if $out(q_1', B_2', q_2') = SL(A_C(q_1', B_2', q_2'))$, 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. In many cases (including the case where A_1 contains no loops, and the case where no two symbolic transitions are reachable from each other) this can be achieved by simple heuristics. In addition, in many cases the simulation H can be manipulated such that the sets $SL(A_C(q_1', B_2', q_2'))$ themselves will become pairwise disjoint.

Lemma 1. *If for every* $(q_1', B_2', q_2') \in Q_C$*,* $out(q_1', B_2', q_2') \subseteq SL(A_C(q_1', B_2', q_2'))$*, and for every* $(q_1', B_2') \in Q_W$*,* $\bigcup_{q_2' \in B_2'} out(q_1', B_2', q_2') = \bigcup_{q_2' \in B_2'} SL(A_C(q_1', B_2', q_2'))$*, and the consistency property holds then the assignment* σ *defined as above satisfies* $\sigma(SL(A_1)) \subseteq SL(A_2)$*.*

Example 8. Consider the simulation H from Example 6, computed for the DSAs from Fig. 9(a) and (c). Unknown elimination based on H will yield the following assignment: $\sigma(a, x, b(f|g)) = d, \sigma(a, x, cg) = eh^*e, \sigma(a, x, cf) = h, \sigma(bya, x, (b|c)(f|g)) = d, \sigma(b, y, ax(b|c)(f|g)) = zd$. 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 $(1, \{1\}), (1, \{16\})$ and their x-witnesses $(2, \{2, 6, 9\}), (2, \{17\})$ respectively. Namely, 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 $out(2, \{2, 6, 9\}, 2) = b(f|g), out(2, \{2, 6, 9\}, 6) = cg, out(2, \{2, 6, 9\}, 9) = cf$ 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 7, where the x-witness for $(1, \{1\})$ is $(2, \{2, \ldots 12, 20\})$, we would still get the same assignment since for any $q \neq 2, 6, 9$, $out(2, \{2, \ldots 12, 20\}, q) = \emptyset$. 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 $(10, \{15\})$.

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 5). 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 there is no $L' \in [\![A]\!]$ such that $L' \subseteq L$, then 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 $sw \in (\Sigma \cup \mathit{Vars})^* \setminus \Sigma^*$, there are always comparable completions in $[\![A]\!]$. Namely, if assignments σ and σ' differ only on their assignment to some variable x in sw (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 $L = \sigma(SL(A)) = \sigma(SL(A) \setminus \{sw\}) \cup \sigma(sw) \supset \sigma'(SL(A) \setminus \{sw\}) \cup \sigma'(sw) = \sigma'(SL(A')) = L'$. 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-minimlaity, 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 9. This intuition is demonstrated by Example 5, where the set of completions of the DSA from Fig. 10(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. 10(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 $sw \in SL(A)$ 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 are redundant. Such symbolic words are called *equivalent* since their corresponding accepting paths π and π' are symbolically-equivalent.

In Example 9, the path $\langle 0, 1, 6, 7\rangle$ of the DSA in Fig. 10(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 2. *Let A be a DSA. An accepting path π in A is* redundant *iff $\pi \preceq A \setminus \pi$.*

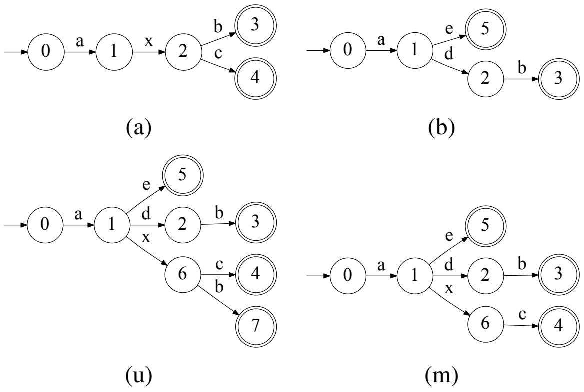


Fig. 11. Inputs (a) and (b), union (u) and minimized DSA (m).

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.*

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.*

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. 11 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 [7]. 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). Yang was partially supported by EPSRC. Peleg was partially supported by EU's FP7 Program / ERC agreement no. [321174-VSSC].

References

1. ACHARYA, M., XIE, T., PEI, J., AND XU, J. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07*, pp. 25–34.
2. ALUR, R., CERNY, P., MADHUSUDAN, P., AND NAM, W. Synthesis of interface specifications for Java classes. In *POPL* (2005).
3. BECKMAN, N., KIM, D., AND ALDRICH, J. An empirical study of object protocols in the wild. In *ECOOP'11*.
4. GANESH, V., MINNES, M., SOLAR-LEZAMA, A., AND RINARD, M. Word equations with length constraints: whats decidable? In *Haifa Verification Conference* (2012).
5. GRUSKA, N., WASYLKOWSKI, A., AND ZELLER, A. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *ISSTA '10*.
6. MANDELIN, D., XU, L., BODIK, R., AND KIMELMAN, D. Jungloid mining: helping to navigate the API jungle. In *PLDI '05*, pp. 48–61.
7. MISHNE, A., SHOHAM, S., AND YAHAV, E. Typestate-based semantic code search over partial programs. In *OOPSLA'12: Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications* (2012).
8. MONPERRUS, M., BRUCH, M., AND MEZINI, M. Detecting missing method calls in object-oriented software. In *ECOOP'10* (2010), vol. 6183 of *LNCS*, pp. 2–25.
9. PLANDOWSKI, W. An efficient algorithm for solving word equations. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing* (2006), STOC '06.
10. SHOHAM, S., YAHAV, E., FINK, S., AND PISTOIA, M. Static specification mining using automata-based abstractions. In *ISSTA '07*.
11. STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. 12*, 1 (1986), 157–171.
12. WASYLKOWSKI, A., AND ZELLER, A. Mining temporal specifications from object usage. In *Autom. Softw. Eng.* (2011), vol. 18.
13. WASYLKOWSKI, A., ZELLER, A., AND LINDIG, C. Detecting object usage anomalies. In *FSE'07*, pp. 35–44.
14. WEIMER, W., AND NECULA, G. Mining temporal specifications for error detection. In *TACAS* (2005).
15. WHALEY, J., MARTIN, M. C., AND LAM, M. S. Automatic extraction of object-oriented component interfaces. In *ISSTA'02*.