Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms

Ivan Beschastnikh, Yuriy Brun, *Member, IEEE*, Jenny Abrahamson, Michael D. Ernst, *Senior Member, IEEE*, and Arvind Krishnamurthy

Abstract—It is a staple development practice to log system behavior. Numerous powerful model-inference algorithms have been proposed to aid developers in log analysis and system understanding. Unfortunately, existing algorithms are typically declared *procedurally*, making them difficult to understand, extend, and compare. This paper presents InvariMint, an approach to specify model-inference algorithms *declaratively*. We applied the InvariMint declarative approach to two model-inference algorithms. The evaluation results illustrate that InvariMint (1) leads to new fundamental insights and better understanding of existing algorithms, (2) simplifies creation of new algorithms, including hybrids that combine or extend existing algorithms, and (3) makes it easy to compare and contrast previously published algorithms. InvariMint's declarative approach can outperform procedural implementations. For example, on a log of 50,000 events, InvariMint's declarative implementation of the kTails algorithm completes in 12 seconds, while a procedural implementation completes in 18 minutes. We also found that InvariMint's declarative version of the Synoptic algorithm can be over 170 times faster than the procedural implementation.

Index Terms—Model inference, API mining, specification mining, process mining, declarative specification, inference understanding, inference extensibility, inference comparison, InvariMint, kTails, synoptic

1 Introduction

Understanding a system's behavior is a difficult software engineering task that is required when a system behaves in an unexpected manner or when a developer must make changes to legacy code. Logging and log analysis of captured system behavior is one of the most ubiquitous, simple, and effective tools for system understanding. Unfortunately, the size and complexity of logs often exceed a developer's ability to navigate and make sense of the captured data. For example, production systems at Google log *billions* of events each day; these are stored for weeks to help diagnose errant behavior [40].

Model inference is one promising approach to help users make sense of large and complex executions. The goal of a model-inference algorithm is to produce a model, typically a finite state machine (FSM), that accurately and concisely represents the system that produced the log.

- *I. Beschastnikh is with the Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, Canada. E-mail: bestchai@cs.ubc.ca.*
- *Y. Brun is with the School of Computer Science, University of Massachusetts, Amherst, MA 01003. E-mail: brun@cs.umass.edu.*
- *J. Abrahamson is with Facebook Inc., Seattle, WA 98101. E-mail: jennya@fb.com.*
- *M. D. Ernst and A. Krishnamurthy are with Computer Science & Engineering, University of Washington, Seattle, WA 98195. E-mail: {mernst, arvind}@cs.washington.edu.*

Manuscript received 26 Aug. 2013; revised 23 Oct. 2014; accepted 28 Oct. 2014. Date of publication 9 Nov. 2014; date of current version 17 Apr. 2015. Recommended for acceptance by P. Inverardi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2369047

A model-inference algorithm outputs a model that accepts a formal language. (For an example model, see Fig. 2b.) The model's language is smaller than Σ^*: it is limited by certain temporal property instances that the algorithm mined from the log. Some of the types of mined properties may be explicit in the algorithm definition, whereas others may be implicit and deeply hidden in a procedural definition.

Numerous model-inference algorithms and corresponding tools already exist to help debug, verify, and validate systems [1], [6], [7], [8], [18], [19], [21], [25], [26], [28], [29], [31], [34], [38], [41]. Unfortunately, it is challenging to apply and build on top of this rich body of work. This is because model-inference algorithms are primarily expressed procedurally—as algorithms that iteratively modify a representation of the log (e.g., a graph) to infer a model that can be shown to a user. Such procedural specification obfuscates the key qualities of the algorithm, which makes procedural model-inference algorithms difficult to understand, extend, and compare.

This paper proposes **InvariMint**, an approach for specifying model-inference algorithms *declaratively*. InvariMint has two key features: (1) it explicitly specifies the types of properties that will be enforced in the final model, and (2) it decouples property type *specification* from the mechanism of property instance *mining*. A *property type* is a pattern-level description of possible temporal relationships between events, whereas a *property instance* is a concrete example of those relationships between specific types of events. An example of a property type is an FSM that only admits traces that start with events that come from some set X. An instance of this property for an input log of TCP packet

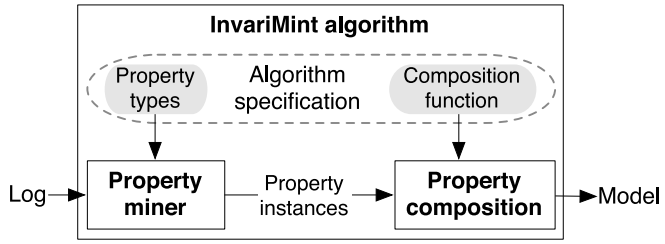


Fig. 1. An overview of the InvariMint approach. An InvariMint *algorithm* is instantiated by a specification, which consists of a set of property types and a composition function. The resulting InvariMint algorithm is a model-inference algorithm—it takes a log of traces as input, and it outputs an inferred model that describes the process that generated the input log. (See Fig. 2 for example input and output.) Internally, the algorithm uses property types to mine property instances, and then it applies the composition function to the property instances to derive the model.

types might be an FSM that admits traces that start with events that come from the set {syn, syn-ack}.

Fig. 1 overviews the InvariMint approach. With InvariMint, a model-inference algorithm is specified by a set of property types that the user wants to include in the inferred model. InvariMint mines instances of these properties from the log, then composes these instances to produce the final model. InvariMint represents each property instance as an FSM, and composes the FSMs using standard FSM operations (such as FSM union and intersection). Well-understood work on formal languages allows InvariMint to perform these operations efficiently and to produce minimal models [20].

The InvariMint approach supports model-inference algorithm understanding, extension, and comparison, as described in Sections 1.1-1.3.

1.1 Improving Algorithm Understanding

An inferred model represents a combination of certain property instances. For most algorithms, it is difficult for a developer to understand which property types and instances are true of the input log and which are artifacts of the algorithm.

For example, suppose that in an inferred model of an e-mail client, each `login` event is immediately followed by a `check mail` event—that is, the model satisfies the property instance "`login` must be immediately followed by `check mail`". Does that imply that the property instance was true for all traces in the log? Alternately, it is possible that the inferred model satisfies the property instance due to the model-inference process (e.g., because `login` was immediately followed by `check mail` most of the time).

As another example, suppose that an inferred model allows for events other than `check mail` to follow some `login` events. Does this mean that at least one trace in the log exhibited such behavior? Alternately, it is possible that the log traces always satisfied "`login` must be immediately followed by `check mail`", but the model-inference algorithm did not preserve that property instance (e.g., because the algorithm reasons about missing traces and generalizes the model to accept traces that were not observed).

InvariMint expresses an algorithm in terms of property types: the inferred model must satisfy instances of the given property types. With this formulation, algorithms become more clear, concise, and comprehensible. Further, this formulation makes evident certain complexities that may otherwise be hidden, such as non-determinism of the inference algorithm.

1.2 Supporting Algorithm Extensibility

It is difficult to modify or to compose existing model-inference algorithms to create interesting hybrids.

For example, suppose that a developer uses two different inference algorithms: one to model exceptional executions and another to model executions with sequences of calls into a specific library. The developer may want to compose these two algorithms to generate a single model, but combining the existing algorithms may require a complete algorithm redesign.

Further, it is difficult or impossible to exclude a specific log property instance from a specific invocation of the algorithm. Suppose that in a particular log, every `login` event is immediately followed by a `check mail` event. The developer may know that other events can follow `login` events: the property instance "`login` must be immediately followed by `check mail`" is an artifact of the log, which records only a subset of all possible executions. The developer may want the model-inference algorithm to ignore this observed, but inaccurate, property instance. However, because a procedural algorithm definition explicitly specifies neither property types nor instances, such exclusions may be difficult.

The declarative specification enables algorithm users to customize the algorithm to suit their needs. With InvariMint, it is easy to add, remove, and modify both (1) the instances of properties in a specific inference execution (e.g., each `login` event must be followed by a `check mail` event), and (2) the types of properties the algorithm preserves (e.g., an event may only follow another event if it did so at least once in the log). A user can tweak the algorithm by removing or adding a particular property instance or type to improve the inferred model, without having to modify the algorithm implementation. For example, the Synoptic [7] algorithm uses the kTails algorithm as a final (coarsening) step to derive a more compact final model. InvariMint can express this by simply merging the kTails property types into the Synoptic specification.

1.3 Simplifying Algorithm Comparison

Previously-published algorithms lack a common form to aid comparison. Instead, researchers must reason about pseudocode and work out complex proofs. A declarative approach specifies a model-inference algorithm in terms of log property types that the inferred model will satisfy.

InvariMint makes it easier to compare and improve model-inference algorithms. For example, two algorithms with incomparable procedural definitions may enforce overlapping sets of types of properties on their inferred models. Expressing the algorithms with InvariMint specifications makes this overlap evident.

1.4 Contributions

This paper makes four contributions.

Contribution 1. We describe InvariMint, a declarative approach for specifying model-inference algorithms.

InvariMint provides a common language for expressing model-inference algorithms. This declarative approach promotes algorithm understanding, extension, and comparison.

Contribution 2. We use InvariMint to develop a declarative specification of kTails [8], which leads to improved understanding of this previously-published, important model-inference algorithm.

From our past experiences with kTails, we know that this algorithm behaves non-obviously on large log inputs. For instance, it is neither apparent which states will be merged, nor what unobserved traces the final kTails-inferred model will accept. The InvariMint specification represents the kTails algorithm as a composition of a set of property types, each of which is easy to inspect to better understand the characteristics of the final kTails-inferred model.

Contribution 3. We use InvariMint to develop a declarative specification that approximates the Synoptic [7] model-inference algorithm. This leads us to identify hidden properties in the Synoptic algorithm and to find overlap in properties between Synoptic and kTails.

Synoptic is an algorithm constructed with explicit log property types in mind. Although Synoptic makes certain property types explicit, we found that its procedural declaration in fact preserves an additional property type. This property type does not appear in Synoptic's list of property types, and Synoptic's procedural declaration does not allow this property instance to be removed, altered, or relaxed. In contrast, a declarative specification of Synoptic makes this property instance explicit and allows a user to remove all properties of this type or to select individual instances of this property for specific log event types to enforce. More importantly, InvariMint makes the algorithm's user and developer explicitly aware of the full set of property types and instances it enforces. Our declarative specification of Synoptic is an over-approximation of the procedural Synoptic algorithm[1].

Contribution 4. We empirically demonstrate that algorithms expressed declaratively with InvariMint significantly outperform the equivalent procedural variants.

As an added benefit, the declarative versions of kTails and Synoptic with efficient property instance mining greatly outperform their procedural counterparts and scale linearly with log size. In benchmark testing on logs with 25K events, we found that declarative kTails was at least 10× faster than procedural kTails, and declarative Synoptic was 10× faster than procedural Synoptic. For larger logs, InvariMint implementations provide even greater efficiency benefits. For example, on a log of 50K events, declarative kTails completes in 12 seconds, while procedural kTails runs in 18 minutes. On a log of the same size, declarative Synoptic completes in under 1 second, while procedural Synoptic runs in 170 seconds.

1. That is, the InvariMint version of Synoptic outputs a model that accepts traces that are accepted by all possible models that Synoptic would return on the same input log, plus additional traces that Synoptic models may not include.

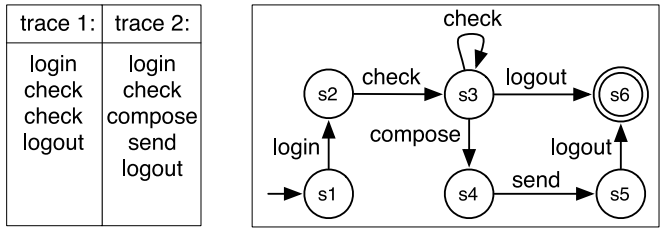


Fig. 2. (a) An example log of an email client with two traces. (b) The model inferred by SimpleAlg (Section 2) for the input log in (a).

A previous paper introduced InvariMint [5]. This paper extends the prior work in the following ways: (1) it gives a formal description of the InvariMint approach (Section 3); (2) it proves equivalence between the declarative kTails and procedural kTails algorithms (Section 4.3); (3) it corrects the specification of the SimpleAlg algorithm (Section 2); (4) it provides examples of more complex InvariMint specifications and a more detailed description of InvariMint limitations (Section 7); and (5) it improves the exposition throughout the paper.

The rest of this paper is structured as follows. Section 2 motivates the declarative approach, and Section 3 formalizes the InvariMint specification language (Contribution 1). Sections 4 and 5 present InvariMint specifications of kTails and Synoptic, respectively (Contributions 2 and 3). Section 6 empirically evaluates procedural and declarative algorithm specifications (Contribution 4). Section 7 discusses implications of our work. Section 8 places our work in the context of related research, and Section 9 concludes.

2 SimpleAlg: A Motivating Example

We will use SimpleAlg, a simple model-inference algorithm, to introduce and explain InvariMint concepts.

A model-inference algorithm's input is a *log*—a set of traces of a system's execution. Each *trace* is an ordered sequence of *events* (elements of a finite alphabet) that occur during execution. The output of a model-inference algorithm is a model. Algorithms in this paper generate (possibly nondeterministic) FSM models. Fig. 2 shows example input and output for SimpleAlg.

A model-inference algorithm aims to create a model that admits all traces in the input log, that is concise, and that generalizes to traces that are legal but were not observed in the log. In some cases, these measurements are subjective, and different model-inference algorithms infer models that are good for different purposes. This paper does not propose new model-inference algorithms nor evaluate existing ones. Instead, our focus is on the specification of the model-inference algorithms: the description of the process by which the models are inferred. Our declarative mechanism for specifying model-inference algorithms will enable people to better understand existing algorithms and to create new ones.

Fig. 2a shows an email client log with two traces. The event alphabet is {`login`, `check`, `compose`, `send`, `logout`}. `check` stands for `check mail`. Fig. 2b shows the model SimpleAlg infers from this input log.

SimpleAlg creates a single "initial" state (labeled s1) that is the start of every execution, and then generates a unique state (labeled s2, s3, s4, ...) for each event type in the log. Each of these states corresponds to an event and indicates that the event has occurred immediately prior to the state.

The example model in Fig. 2b has six states: the initial state (s1), and one state for each of the five event types — `login` (s2), `check` (s3), `compose` (s4), `send` (s5), and `logout` (s6).

The ordering of the events in the observed traces defines the transitions (edges) of a SimpleAlg-generated model. There is a transition from the state corresponding to the event type e_1 to the state associated with the event type e_2 iff there exists at least once, in some trace in the input log, an event of type e_1 immediately followed by an event of type e_2. That transition is labeled e_2. In the email client example, in trace 2 of the log in Fig. 2a, a `compose` event is immediately followed by a `send` event. Therefore, the SimpleAlg-generated model in Fig. 2b has a transition labeled `send` from state s4 (which corresponds to `compose`) to state s5 (which corresponds to `send`).

The language of a model inferred with SimpleAlg always contains every trace in the input log. SimpleAlg generalizes in the following way: if SimpleAlg ever observes an event of type e_1 to be immediately followed by an event of type e_2 in the input log, then whenever the system being modeled produces or consumes an e_1-type event, SimpleAlg assumes that it is legal for the system to then produce or consume an e_2-type event.

Because SimpleAlg merges all log events of the same type into a single state, the models it generates are compact: There is exactly one state for each event type, plus the initial state s1. Thus, the model size is independent of the total number of events in the log. The running time of SimpleAlg is asymptotically linear in the size of the log.

Fig. 3 shows the SimpleAlg pseudocode. While the pseudocode may help someone implement SimpleAlg, it does not convey the insights we presented above, and it lacks many of the desirable qualities of an algorithm description, such as ease of understanding, extensibility, and the ability to compare to other algorithms. In contrast to this procedural form, Sections 2.1 and will now capture the SimpleAlg algorithm declaratively, by specifying it in terms of temporal properties that relate event instances in the log. After that, Section 2.3 discusses the benefits of this declarative specification.

```
1  Input: Log L
2  let M = new, empty FSM model
3
4  // Create states
5  M.addState(init)
6  foreach (Trace t in L):
7    foreach (Event e in t):
8      let y = Event type of e
9      if (¬M.hasState(s_y)) : M.addState(s_y)
10
11 // Add transitions among the states.
12 foreach (Trace t in L):
13   // Add transition from init state to first event.
14   let f = Event type of first event in t
15   if (¬M.hasTransition(init, s_f)):
16     M.addTransition(src=init, dst=s_f, label=f)
17
18   // For each pair of adjacent events, add a transition
19   // between states of corresponding event types.
20   foreach (Event e in t):
21     if (∃f, f follows e in t):
22       let y = Event type of e
23       let z = Event type of f
24       if (¬M.hasTransition(s_y, s_z)):
25         M.addTransition(src=s_y, dst=s_z, label=z)
26
27 Output: M
```

Fig. 3. Procedural pseudocode for the SimpleAlg algorithm. Fig. 5 declaratively specifies SimpleAlg.

2.1 Decomposing SimpleAlg into Property Instances

One way of thinking about SimpleAlg is via the set of property instances—temporal relationships between events in the input log—that SimpleAlg implicitly mines from the log and then uses to build a final FSM model. We propose to express these individual properties as FSMs, then combine those FSMs to form the final model.

Fig. 4 shows six FSMs, each of which is a property instance of the input log in Fig. 2a. Each property instance FSM accepts each trace in the log; equivalently, the trace is in the language of each of the FSMs.

We make two important observations about these property instances:

- The four property instances in Fig. 4a have a similar shape—all of them represent the same, more general, type of property: "event x must be immediately followed by an event $y \in Y$". For example, π_1 is an instantiation of this property with the binding b, where $b(x) =$ `login` and $b(Y) = \{$`check`$\}$.
- The intersection of the six property instances in Fig. 4 is the model of Fig. 2b. These property instances can be thought of as a *factoring* of the SimpleAlg-generated model.

We can generalize the above observations as follows: (1) collections of property instances can be described more abstractly with *property types*, and (2) FSM operations, such as intersection, can *compose* property instance like the ones in Fig. 4 into models.

The above generalizations allow us to declaratively specify a model-inference algorithm by specifying (1) the property types and (2) the composition process by which instances of these property types are composed into the final model. Next, Section uses this declarative approach to specify SimpleAlg.

2.2 A declarative Specification of SimpleAlg

Using the SimpleAlg property instances from Fig. 4, this section constructs a declarative InvariMint specification

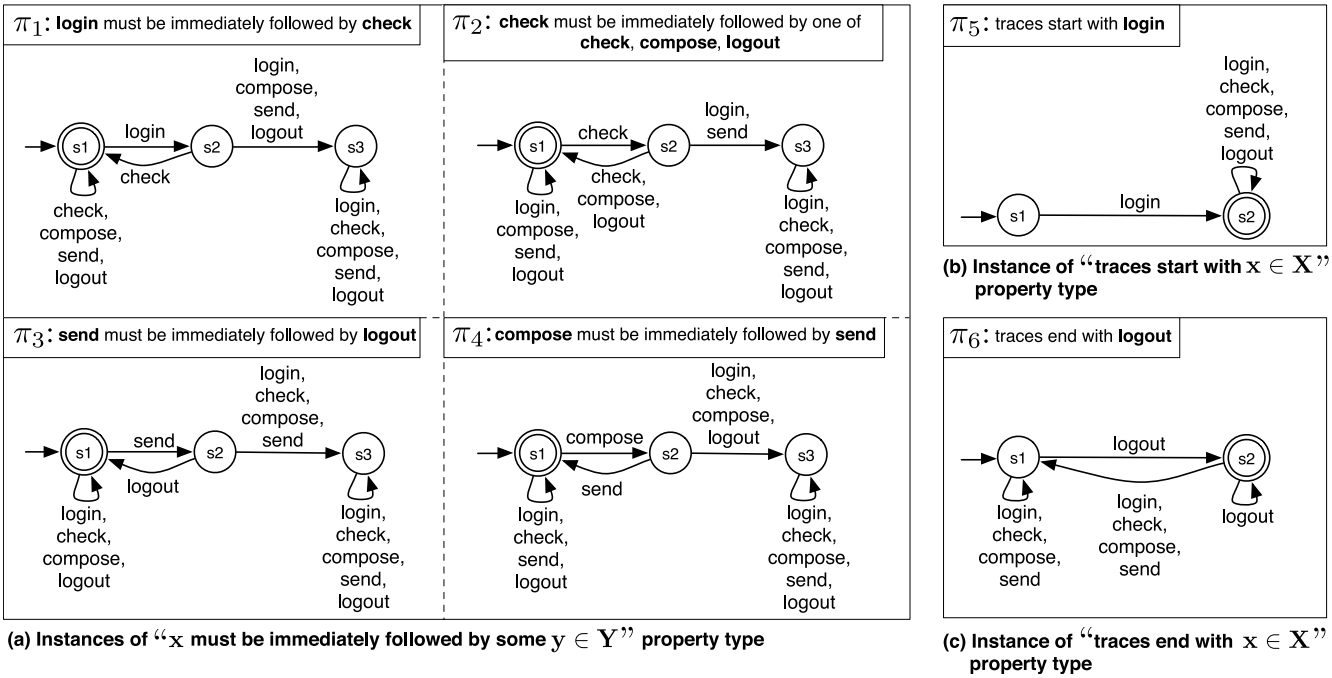


Fig. 4. Property instances mined by the InvariMint implementation of SimpleAlg from the log in Fig. 2a, based on property types in Figs. 5a, 5b and 5c.

of SimpleAlg. The first half of the specification describes the *types* of property instances that should be mined from an input log. The second half of the specification describes how to *compose* the mined property instances into the final model. Fig. 1 illustrates how property types and the composition function make up an InvariMint specification.

2.2.1 Property Types

We represent a *property type* as (1) a parameterized FSM (PFSM) and (2) an *IncludeBinding* binding evaluator function. Informally, a PFSM is a FSM with variable-labeled transitions. The transitions in a PFSM can be instantiated to event types (denoted with lowercase variables, e.g., x), or sets of event types (denoted with uppercase variables, e.g., Y). For example, π_2 in Fig. 4a is an instantiation of the PFSM in Fig. 5a with the binding b, where $b(x) = \texttt{check}$ and $b(Y) = \{\texttt{check}, \texttt{logout}, \texttt{compose}\}$. The function $IncludeBinding(\text{Log } L, \text{Binding } b)$ determines which bindings of transition variables to event types in a PFSM are valid. This function is used to mine property instances from an input log.

A PFSM can be instantiated as many different FSMs, or *property instances*, with the variables bound to different events and sets of events. Given a log and a property type, InvariMint mines property instances whose corresponding bindings—the assignment of PFSM variables to event types and sets of event types—cause *IncludeBinding* to evaluate to true. In addition, InvariMint instantiates a property type at most once for every event type variable in the PFSM, and in every such instantiation the event type *set* variables are instantiated to be maximal.

For example, InvariMint instantiates the PFSM in Fig. 5a for every choice of x with a maximal value for Y, such that the resulting property instances must accept every trace in the log. Section 3.1 describes the binding process more formally.

For SimpleAlg, there are three property types, shown as PFSMs in Fig. 5 and *IncludeBinding* functions in Fig. 6. In this paper we use linear temporal logic (LTL) [35] to compactly specify *IncludeBinding* (see Section 7.2 for examples of other variants). LTL statements use the operators *always* ($\Box$), *eventually* ($\Diamond$), *until* (U), and *next* ($\bigcirc$). The SimpleAlg property types can be described as "event x must be immediately followed by an event from set Y", "traces start with $x \in X$", and "traces end with $x \in X$". These properties simply specify that every trace in the log should be accepted by the inferred model with an additional generalization that if some x was followed by a y somewhere in the log, then every instance of x can be followed by a y. These three property types are sufficient for a complete declarative specification of SimpleAlg.

Each of the six property instances in Fig. 4 is an instantiation of one of these three property types. For a different log, the specific instantiations would differ from those in Fig. 4, but the property types would be the same.

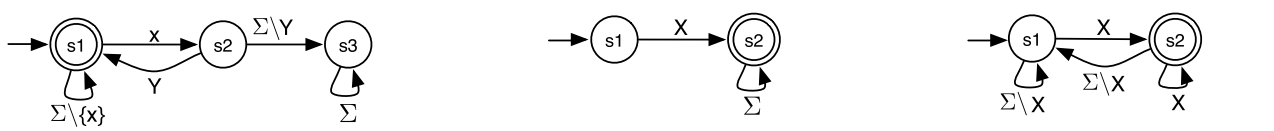


Fig. 5. PFSMs used in the InvariMint specification of SimpleAlg. The PFSMs correspond to the property instances in Fig. 4.

Id	Description	PFSM	$IncludeBinding(\text{Log } L, \text{Binding } b)$
a	x immediately followed by $y \in Y$	Fig. 5(a)	true : $\forall t \in L, \forall e_i \in b(Y),\ \Box(b(x) \rightarrow (\bigcirc e_1 \vee \ldots \vee \bigcirc e_i \vee \ldots \vee \bigcirc e_n))$ in $t \wedge$ $\forall e \in b(Y), \exists t \in L,\ \Diamond(b(x) \wedge \bigcirc e)$ in t false : otherwise
b	traces start with $x \in X$	Fig. 5(b)	true : $\forall t \in L, \forall e \in t, \exists f \in b(X),\ ((\Diamond e) \rightarrow \neg e\ \mathrm{U}\ f)$ in $t \wedge$ $\forall f \in b(X), \exists t \in L, \forall e \in t,\ ((\Diamond e) \rightarrow \neg e\ \mathrm{U}\ f)$ in t false : otherwise
c	traces end with $x \in X$	Fig. 5(c)	true : $\forall t \in L, \exists f \in b(X), \forall e,\ e \neq f,\ \Box(e \rightarrow \Diamond f)$ in $t \wedge$ $\forall f \in b(X), \exists t \in L, \forall e,\ e \neq f,\ \Box(e \rightarrow \Diamond f)$ in t false : otherwise

Fig. 6. The InvariMint specification of SimpleAlg. We use LTL to compactly specify $IncludeBinding$. LTL statements use the operators *always* (□), *eventually* (◇), *until* (U), and *next* (◯).

2.2.2 Composition Function

An InvariMint specification of an algorithm includes not only property types, but also a *composition function* for composing the property instances mined from an input log into a model. For SimpleAlg, the composition function is

$$Compose(\pi_1, \ldots, \pi_n) = Minimize(\cap \pi_i)$$

InvariMint uses existing FSM intersection and minimization algorithms [20] to compose property instances by intersecting them and minimizing the result to produce the final model. The resulting model is compact and accepts exactly those event sequences that satisfy all of the mined property instances.

This completes the InvariMint specification of SimpleAlg. Given the SimpleAlg property types and composition function the InvariMint declarative formulation of SimpleAlg produces exactly the same model as the procedural implementation (e.g., for the log in Fig. 2a the model in Fig. 2b is produced).

2.2.3 Other Formulations of SimpleAlg

Note that other InvariMint specifications of SimpleAlg are possible. For example, Fig. 7 shows an alternative set of PFSMs to specify SimpleAlg. In particular, the PFSM in Fig. 7b is an intersection of the PFSMs in Figs. 5b and 5c. These PFSMs, in combination with a new $IncludeBinding$ function,[2] is a different specification of SimpleAlg. Although this new specification is different, it describes the same SimpleAlg algorithm.

2.3 InvariMint Benefits

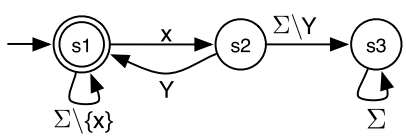
The declarative specification of SimpleAlg (Figs. 5 and 6) provides three benefits over the SimpleAlg procedural pseudocode (Fig. 3):

1) The declarative specification makes clear the key property types of the final model by decoupling these property types from the mining and composition procedures, while the pseudocode mixes all three. InvariMint's declarative approach makes it easier for a person to understand which properties of the log are preserved in the model, and which are not.
2) The declarative specification eases (1) adding new constraints to the model via defining new property types, and (2) eliminating constraints from the model by omitting property instances. For example, if we do not want `login` to only be immediately followed by `check`, we can simply omit the property instance π_1 in Fig. 4.
3) The declarative specification allows extending SimpleAlg to construct InvariMint specifications for kTails and Synoptic (as we will show in Sections 4 and 5.1). While the pseudocode for these algorithms looks completely different from SimpleAlg's pseudocode, the InvariMint declarative specifications reveal that both kTails and Synoptic share a key property type (Fig. 5a and its corresponding $IncludeBinding$ function) with SimpleAlg. The fact that all three algorithms share this property type is one of the insights gained from specifying these algorithms with InvariMint.

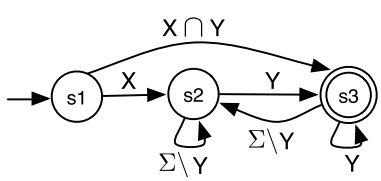
Before extending the SimpleAlg specification to kTails and Synoptic, Section 3 presents the InvariMint declarative approach formally.

3 The InvariMint Approach

InvariMint is an approach—or a common language—for describing model-inference algorithms, such as SimpleAlg,



(a) PFSM for: $\mathbf{x}$ immediately followed by $\mathbf{y} \in \mathbf{Y}$



(b) PFSM for: traces start with $\mathbf{x} \in \mathbf{X}$ and end with $\mathbf{y} \in \mathbf{Y}$

Fig. 7. Alternative PFSMs to specify SimpleAlg. (a) Identical to the PFSM in Fig. 5a. (b) A PFSM that captures the "traces start with $x \in X$" property of Fig. 5b and the "traces end with $y \in Y$" property of Fig. 5c in a single PFSM.

2. The new $IncludeBinding$ function for PFSM in Fig. 7b will merge the lower two $IncludeBinding$ functions in Fig. 6 by linking their "true" conditions with a conjunction.

kTails, and Synoptic. Fig. 1 overviews the InvariMint approach. Like other model-inference algorithms, an *InvariMint algorithm* takes as input a *log* of traces to be modeled, and outputs a *model*. We differentiate the concept of an InvariMint algorithm from an *InvariMint specification*, which specifies how an InvariMint algorithm behaves for a particular log input. An InvariMint specification has two parts: (1) a set of *property types* that describe properties to be mined from the log to derive *property instances*; and (2) a *composition function* that combines the mined property instances into a final model.

3.1 Property Types

A property type is represented as a parameterized FSM (PFSM)—an FSM with variable-labeled transitions (e.g., Fig. 5a)—and an $IncludeBinding$ binding evaluator function (e.g., Fig. 6a). Before we formalize property types, we first define some basic concepts, such as log, trace, event, and variable.

Definition 1 (Log, trace, and event). *The alphabet of* events *is a finite alphabet of logging statements a system can produce. An ordered sequence of events is a* trace*, and a set of traces is a* log.

Definition 2 (Event variable and event set variable). *An* event variable *is a variable, or a placeholder, for an event. An* event set variable *is a variable for a set of events.*

We use lowercase letters to denote event variables and uppercase letters to denote event set variables. Thus x could represent any single event, whereas Y could represent any set of events. Further, when an entity can be either an event variable or an event set variable, we simply use *variable*.

A parameterized FSM (PFSM) is an FSM whose finite alphabet of transition labels are log events (Σ) and variables (Σ_v)—both event variables and event set variables. The special ϵ symbol allows for transitions between PFSM states without consuming an alphabet symbol.

Definition 3 (Parameterized FSM (PFSM)). *A* PFSM P *is an FSM,* $P = \langle Q, Q_I, \Sigma \cup \Sigma_v, \Delta, Q_T \rangle$*, where:*

- Q *is a set of states.*
- $Q_I \subseteq Q$ *is a set of initial states.*
- Σ *is a set of events,* $\epsilon \in \Sigma$*.*
- Σ_v *is a finite set of* variables*.*
- $\Delta : Q \times \Sigma \cup \Sigma_v \rightarrow 2^Q$ *is a transition relation. If* $q' \in \Delta(q, \sigma)$*, we say that* P *transitions from* q *to* q' *on* σ*.*
- $Q_T \subseteq Q$ *is a set of terminal states.*

A PFSM can be instantiated as many different FSMs, with the variables bound to different events and sets of events. Further, if an event set variable is bound to the empty set, the transition labeled with this empty set cannot take place.

Definition 4 (Binding). *Let* E *be an alphabet of events such that* $\epsilon \notin E$*. Let* Σ_v *be a finite set of* variables*. Then, a function* $b : \Sigma_v \rightarrow \Sigma'_v$ *is a binding if for all variables* $\sigma_v \in \Sigma_v$*, if* σ_v *is an event variable, then* $b(\sigma_v) \in E$*, and if* σ_v *is an event set variable, then* $b(\sigma_v) \subseteq E$*.*

Note that a binding may bind multiple variables in Σ_v to the same element, or set of elements, in E.

Given an input log, the $IncludeBinding$ function determines whether a property instance, with a specific binding of variables to event types or sets of event types in the corresponding PFSM, is valid and should be included in the composition.

Definition 5 (Binding evaluator function). *Let* $\mathcal{L}$ *be the set of all possible logs, and let* $\mathcal{B}$ *be the set of all possible bindings for a PFSM* P*. Then the binding evaluator function* $IncludeBinding_P : \mathcal{L} \times \mathcal{B} \rightarrow \{\text{true}, \text{false}\}$*.*

We drop the P *when the PFSM corresponding to the binding evaluator function is clear from the context.*

For example, the version of $IncludeBinding$ in Fig. 6a returns true for event a and event set B if a was observed to only be immediately followed by the events from B across all traces in the log—that is, there is a trace for every $b \in B$ and there is a $b \in B$ for every trace such that eventually ($\Diamond$), if we observe an a event, then we will observe a b as the next ($\bigcirc$) event. This function is the binding evaluator for the PFSM in Fig. 5a.

Fig. 4a lists the property instances for which the $IncludeBinding$ function in Fig. 6a returns true on the log in Fig. 2a (with event set variables assigned to maximal sets). For instance, the property instance π_1 in Fig. 4a is included because there is a binding b, where $b(x) = \texttt{login}$ and $b(Y) = \{\texttt{check}\}$, for which $IncludeBinding$ returns true.

A *property type* is a PFSM and a corresponding $IncludeBinding$ function. The PFSM captures the "shape" of the property while the $IncludeBinding$ function determines when and how this shape should be instantiated for a given log input. The explicit separation of the PFSM from the $IncludeBinding$ function makes the specification language more expressive. For example, the shape may describe a temporal constraint "an event x must be immediately followed by an event $y \in Y$" (PFSM in Fig. 5a). The $IncludeBinding$ function can then be used to specify which x and Y can be used in an instantiation. One possibility is that $IncludeBinding$ chooses a Y that is a subset of the events that immediately follow event x in the observed traces (e.g., selecting a Y that contains the events that most frequently and immediately follow x). Another possibility is that the $IncludeBinding$ function allow a Y that includes events that *never* immediately follow x in the observed traces, building an inverse model of the input observations (i.e., model that accepts those traces that did not appear in the log). Yet another possibility is that the $IncludeBinding$ function is probabilistic and returns true for a log and a binding if a condition is satisfied by most (e.g., 99 percent) of the traces. Section 7.2 discusses this example in more detail.

A property instance is an FSM over an alphabet of events. For a property type T and an input log, a property instance for T is an FSM that is an instantiation of T's PFSM with a binding that is accepted by T's $IncludeBinding$ function.

Definition 6 (Property instance for a property type). *Let* E *be an alphabet of events such that* $\epsilon \notin E$*, and let* L *be a log over* E*. Let* T *be a property type with a PFSM* $P = \langle Q, Q_I, \Sigma \cup \Sigma_v, \Delta, Q_T \rangle$ *and an* $IncludeBinding$ *binding evaluator*

function. Then the FSM $\pi = \langle Q, Q_I, \Sigma \cup \Sigma'_v, \Delta', Q_T \rangle$ *is a* property instance *of property type* T *iff all of the following hold:*

- $\exists$ *a binding* $b : \Sigma_v \to \Sigma'_v$, *such that* $Include\ Binding(L, b)$ *evaluates to true.*
- *For all* $q \in Q, \sigma \in \Sigma$, $\hat{q} \in \Delta(q, \sigma) \Leftrightarrow \hat{q} \in \Delta'(q, \sigma)$
- *For all* $q \in Q, \sigma_v \in \Sigma_v$,
  - *if* σ_v *is an event variable, then:*
    $\hat{q} \in \Delta(q, \sigma_v) \Leftrightarrow \hat{q} \in \Delta'(q, b(\sigma_v))$
  - *if* σ_v *is an event set variable, then:*
    for all $e \in b(\sigma_v), \hat{q} \in \Delta(q, \sigma_v) \Leftrightarrow \hat{q} \in \Delta'(q, e)$

Given a property type T and a binding b, we will refer to the property instance generated by T on b as $T(b)$.

3.2 Composition Functions

InvariMint combines the derived property instances into a model using the composition function.

Definition 7 (Composition function). *Let* Π *be the set of all possible property instances and let* F *be the set of all possible FSMs. Then the* composition function $c : 2^{\Pi} \to F$ *composes a set of property instances into a single FSM.*

This paper's examples use composition functions that involve only FSM intersections and minimizations, but this limitation is not inherent to InvariMint. More complex functions may include unions, set differences, and other set operations. For example, an algorithm that supplements positive examples of traces with negative examples that the models needs to exclude, may subtract the model of the negative traces from one of the positive traces. In this case, a composition function could union some property instances, intersect other property instances, and then subtract one from the other (see Section 7.3 for more).

The composition function also determines what happens when an InvariMint algorithm mines zero property instances (e.g., if InvariMint is executed on an empty input log). For example, the composition function could produce a model that accepts all possible traces, or one that rejects them all.

3.3 Using Declarative Specifications to Specify Model-Inference Algorithms

Given a set of property types and a composition function, the InvariMint approach produces a model-inference algorithm implementation, as shown in Fig. 1. (The implementation takes an input log and produces a model.) The implementation uses the property types to mine property instances from the input log, and then uses the composition function to compose the mined property instances. Figs. 8 and 9 list unoptimized pseudocode for the property instance mining and property instance composition procedures. For example, the mining algorithm evaluates the *IncludeBinding* function on the log and every possible property instance of the input PFSM to determine which property instances are valid. Both of these general mining and composition algorithms can be further optimized and tailored to the specific PFSMs.

```
1  Input: Log L,
         Property types ⟨PFSM_1, IncludeBinding_1⟩, ...,
2                       ⟨PFSM_n, IncludeBinding_n⟩
3  let Props = {}
4  foreach (Property type ⟨PFSM_i, IncludeBinding_i⟩)
5    foreach (Binding b for PFSM_i)
6      if (IncludeBinding_i(L, b) and b is maximal):
7        Props = Props ∪ {π}
8  Output: Props
```

Fig. 8. The generic property instance miner algorithm.

kTails [8] and Synoptic [7] are two previously-published model-inference algorithms. To reinforce the InvariMint approach for declaratively specifying algorithms, Sections 4 and 5 present the InvariMint declarative specifications of kTails and Synoptic, respectively.

4 Expressing kTails with InvariMint

kTails [8] is a popular algorithm that has served as the basis for many modern model-inference algorithms [10], [12], [25], [26], [28], [29], [38], [39]. This section defines the kTails algorithm (Section 4.1), declaratively specifies it using InvariMint (Section 4.2), and discusses the insights that InvariMint reveals about kTails (Section 4.3). Later, Section 6.3 will empirically compare the procedural and declarative implementations of kTails.

4.1 kTails

kTails is a model-inference algorithm that works via state merging. kTails's inputs are a log and a parameter k. We refer to kTails with a specific k as kTails(k); e.g., we refer to kTails with $k = 2$ as kTails(2). kTails initially represents the log as an FSM composed of linear sub-FSMs, one per trace. These linear sub-FSMs are joined in a parallel fashion, with a single initial state transitioning to the start of each trace and all traces finishing by transitioning to a single terminal state. kTails then iteratively merges states in the FSM that are *k-equivalent*. Two states are k-equivalent if their kTails are identical.

A state's kTail is the set of strings of length k or shorter that map to valid paths starting from that state. The algorithm terminates and outputs the model when no two remaining states are k-equivalent. Fig. 10 lists the kTails pseudocode. Fig. 11 shows the output model produced by kTails(2) on the input log from Fig. 2(a).

The intuition behind kTails is that if two execution points have identical, k-long sequences of observed events following them, then those points likely represent the same program state. To infer a concise model, kTails merges execution points that it considers to represent the same program state. The process stops once all points deemed equivalent are merged. The parameter k determines the size and generality of the inferred model—a smaller k leads to more merges and produces more compact (and more general) models, while a greater k restricts state equivalence.

4.2 InvariMint Declarative kTails

We refer to InvariMint formulation of kTails as *declarative kTails* to distinguish it from *procedural kTails*. Declarative

```
1  Input: Property instances π_1, ..., π_n,
       Composition function c
2  let Model = c(π_1, ..., π_n)
3  Output: Model
```

Fig. 9. The generic property-instance-composition algorithm.

kTails includes a pre- and a post-processing step. The pre-processing step prepends an α symbol to the start of each trace and postpends ω to the end of each trace. These symbols are selected so that they do not already appear in any of the traces. The post-processing step modifies the final model to remove the α and ω; it makes states with incoming α transition be initial states, makes states with outgoing ω transition be accept states, and removes all α and ω transitions from the model. The extra α and ω symbols are necessary to properly identify the initial and terminal states (see Section 7.5 for more discussion of this requirement).

Declarative kTails uses property types to capture tail-equivalence. Its specification includes the PFSM in Fig. 12a, which mandates that all traces start with α and terminate on the first ω. Figs. 12a and 12b list the two PFSMs in the declarative kTails(1) specification and Figs. 13a and 13b list the corresponding *IncludeBinding* functions.[3] The composition function for declarative kTails(1), and in all declarative kTails specifications, is identical to the one used for SimpleAlg (Section 2.2.2)—FSM intersection followed by FSM minimization.

Specifying declarative kTails(2) requires three PFSMs listed in Figs. 12a, 12b, 12c and the corresponding *IncludeBinding* functions in Figs. 13a, 13b, and 13c.

Note that the property type with the PFSM in Fig. 12b and the *IncludeBinding* function in Fig. 13b is identical to the "immediately followed by" property type described earlier, in Section 2.2. This equality is not a coincidence—the k parameter generalizes the "immediately followed by" property type to k steps into the future.

The greater k is, the finer the granularity[4] of the property types that declarative kTails(k) enforces. For example, the property type in Figs. 12c and 13c states that an event x, followed by an event y, must be followed by one—any one—of the events in the set Z. This corresponds to merging all x, y tails together. This merging, expressed as a property type, captures the key state-merging quality of the algorithm. To see this, consider a log of three traces $\{abc, abd, acd\}$. One binding b such that $IncludeBinding(L, b)$ in Fig. 13c will evaluate to true is $b(x) = a$, $b(y) = b$, $b(Z) = \{c, d\}$. This binding is maximal because the set $\{c, d\}$ cannot be made larger without *IncludeBinding* returning false. The mined property instance for this binding corresponds to an execution of the state-merging operation in procedural kTails on the two states preceding the a events in the first two input traces.

3. Since the PFSM in Fig. 12a has no variable transitions (the PFSM is an FSM) and we need to include at least one copy of this FSM in the composition, the *IncludeBinding* function for this PFSM simply evaluates to true, regardless of the input log or binding.

4. Section 7 discusses in more detail the granularity of property types and how the wrong granularity may cause the algorithm to overfit to the input log.

```
1  Input: Log L, int k
2  let M = initial FSM model of traces in L
3
4  let merged be a Boolean
5  do:
6    merged = false
7    foreach (States s_1, s_2 in M):
8      if (s_1, s_2 are k-equivalent):
9        M.merge(s_1, s_2)
10       merged = true
11 while (merged)
12
13 Output: M
```

Fig. 10. The procedural kTails algorithm. Section 4.1 defines k-equivalence.

Fig. 12d shows the general PFSM used to specify declarative kTails(i) (of course, the complete specification also includes the property types from declarative kTails($i-1$)). In this case, a tail of length i—composed of a binding to $x_0, \ldots, x_{i-1}$—is constrained to be immediately followed by an event from the event set bound to Y.

Both the procedural and declarative kTails algorithms are parameterized by k, the size of the tail. The InvariMint specification requires more property types for larger k. Fig. 12d shows an informal template for a family of PFSMs (for each value of k there is exactly one PFSM represented in the Figure). Note that the InvariMint approach does not currently support such a parametric PFSM. However, our current implementation (see Section 6) of the declarative kTails is parametric: it takes a parameter k and computes the property types internally.

An important feature of the declarative kTails is that it is deterministic. This feature helped us better understand the kTails algorithm and helped reveal a bug in our initial procedural kTails implementation, which happened to be non-deterministic.

4.3 Comparing Procedural and Declarative Specifications of kTails

The model produced by the procedural kTails is identical to the model produced by declarative kTails. Next, we define the kTails algorithm by building on [12]. Then, we provide a proof of equivalence between the two types of kTails.

Let $\Sigma_{\leq k}$ denote the set of all strings of length k or less, including the empty string ϵ. Let a trace be a string over alphabet $\Sigma \cup \{\alpha, \omega\}$, and let a (pre-processed) log L be a set of traces, each of which starts with an α symbol and terminates with the ω symbol. Let PF_L be the set of all prefixes of strings in L. For example, consider the log $L = \{\alpha abc\omega, \alpha ab\omega, \alpha cd\omega\}$. The corresponding $PF_L = \{\alpha, \alpha a, \alpha ab, \alpha abc, \alpha abc\omega, \alpha ab\omega, \alpha c, \alpha cd, \alpha cd\omega\}$.

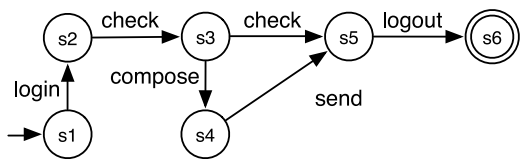


Fig. 11. Example output of kTails(2) on the input log in Fig. 2.

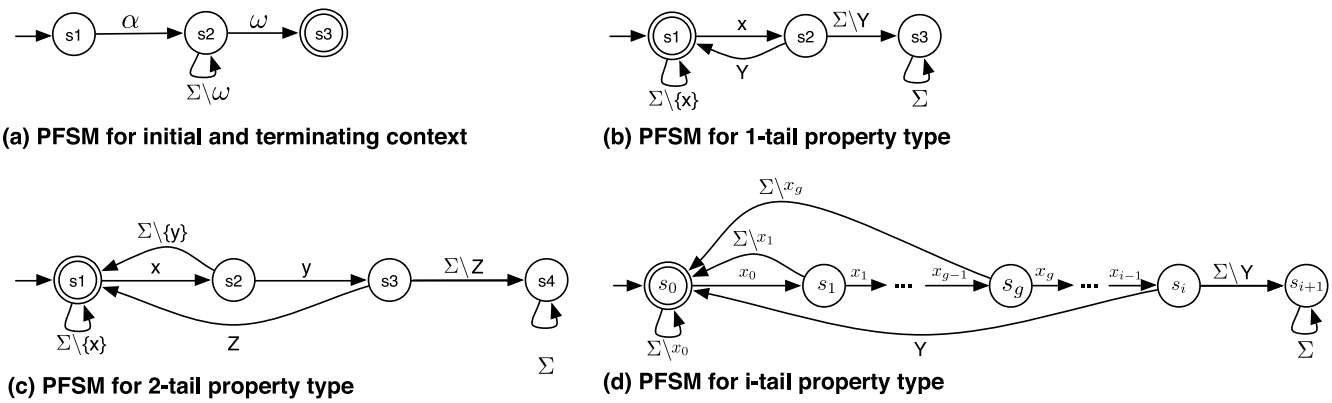


Fig. 12. (a) PFSM that mandates that traces start with α and terminate on the first ω. This PFSM is used by all declarative kTails specifications. This PFSM has no variable-labeled transitions (i.e., it is an FSM). (a+b) PFSMs necessary to specify kTails(1). (a+b+c) PFSMs necessary to specify kTails(2). (d) PFSM representing the i-tail property type. Labels $x_0 \ldots x_{i-1}$ are event variables and Y is an event set variable.

We use $p \cdot t$ to denote concatenation of string t to p, and refer to t as a *tail* of $p \cdot t$. One tail of the string $\alpha abc\omega = \alpha a \cdot bc\omega$ is $bc\omega$.

Definition 8 (Procedural kTails FSM $F_{pkTails}$). *The procedural kTails algorithm takes a log L and an integer k as inputs and generates a FSM $F_{pkTails}$. The* states *of $F_{pkTails}$ correspond to equivalence classes of prefixes from PF_L. One state corresponds to exactly one equivalence class. Every prefix must be assigned to some equivalence class and an equivalence class E is a maximal set of prefixes such that:*

$$\forall (p, p') \in E, \forall t \in \Sigma_{\leq k}, ((p \cdot t) \in PF_L \Leftrightarrow (p' \cdot t) \in PF_L).$$

The transition relation Δ *for equivalence classes, or states, in $F_{pkTails}$ is defined as follows. Given a state E_i and a symbol $a \in \Sigma$,*

$$\Delta(E_i, a) = \bigcup \{E[p \cdot a]\}, \forall p \in E_i,$$

where $E[p \cdot a]$ is the equivalence class of $p \cdot a$.

There is a single initial state *in $F_{pkTails}$—the state corresponding to $E[\alpha]$*[5].

An equivalence class E_i is an accept state *of $F_{pkTails}$ if $\exists\, s \in L$, such that $s \in E_i$.*

Definition 9 (Declarative kTails FSM $F_{dkTails}$). *For a log L and an integer k, let $F_{dkTails}$ be the FSM derived using the InvariMint kTails algorithm on L. This algorithm is specified by the 1-tail,..., k-tail property types (Figs. 12 and 13) and the composition function*[6]*:*

$$F_{dkTails} = \bigcap \left(\pi^1_1, \ldots, \pi^1_{n_1}, \ldots, \pi^k_1, \ldots, \pi^k_{n_k}\right) \cap \pi_{\alpha,\omega},$$

where $\pi^i_1, \ldots, \pi^i_{n_i}$ are the property instances for the i-tail property type, and $\pi_{\alpha,\omega}$ is an instance of the PFSM in Fig. 12a.

Theorem 1 (Declarative specification of kTails is exact). *For an input log L and an integer k, let $F_{pkTails}$ be the corresponding procedural kTails FSM and let $F_{dkTails}$ be the declarative kTails FSM. Then, the languages of the two FSMs are equivalent, or:*

$$\mathcal{L}(F_{pkTails}) = \mathcal{L}(F_{dkTails}).$$

Proof. We prove the two directions of equality in Theorem 1 separately.

(1) $\mathcal{L}(F_{pkTails}) \subseteq \mathcal{L}(F_{dkTails})$

Proof by contradiction:

Assume that $\exists\, s \in \mathcal{L}(F_{pkTails})$ and $s \notin \mathcal{L}(F_{dkTails})$.

Because $s \notin \mathcal{L}(F_{dkTails})$ and $F_{dkTails}$ is composed by intersecting property instances, there is a non-empty set of rejecting (non-accepting) property instances Π that make up $F_{dkTails}$. That is, $\forall \pi \in \Pi, s \notin \mathcal{L}(\pi)$.

The property instance $\pi_{\alpha,\omega} \notin \Pi$ because s must start with α and terminate with ω, so $s \in \pi_{\alpha,\omega}$.

Consider a specific $\pi^i_j \in \Pi$, where $i \leq k$ because of Definition 9. This property instance (corresponding to the i-tail PFSM in Fig. 12d) can reject s in two ways:

(1a) π^i_j rejects s by terminating in state s_{i+1}.

Note that once π^i_j is in state s_{i+1}, it cannot transition out of this state. Let r be the *shortest* prefix of s that causes π^i_j to enter state s_{i+1}.

In this case, r must be at least $i+1$ symbols long, and can be expressed as $r = v \cdot a_0 \cdots a_{i-1} \cdot a$. Since r causes π^i_j to enter state s_{i+1}, it must be the case that $a \notin Y$ (otherwise r would not lead to state s_{i+1}).

Now, consider the equivalence class $E[v]$. This class must be non-empty because $v \in PF_L$ (since $s \in \mathcal{L}(kTails)$ and v is a prefix of s). Because $E[v]$ contains v, $i \leq k$, and $a_0 \cdots a_i \cdot a$ is a tail of v, by definition of equivalence classes, $v \cdot a_0 \cdots a_i \cdot a \in PF_L$. However, this means that $a_0 \cdot \ldots \cdot a_i \cdot a$ should be accepted by π^i_j, which means that $a \in Y$. This contradicts the assumption that implied $a \notin Y$.

(1b) π^i_j rejects s by terminating in s_h, $1 \leq h \leq i$.

The i-tail PFSM in Fig. 12d mandates that each a_m bound to x_m must be followed by some a_{m+1} in some trace. Since ω is the last symbol in any trace, it cannot be bound to any x_m.

The above implies that $\forall g$, $1 \leq g \leq i$, $\forall \pi \in \Pi$, there is no transition on ω *into* s_g. Because all traces in L terminate with ω, $s \in \mathcal{L}(F_{pkTails})$ only if s terminates with ω.

But, this means that π^i_j must have a transition in which x_m is bound to ω. Contradiction.

5. After α is removed from $F_{pkTails}$ in the post-processing step, the final model may have multiple initial states.

6. We omit FSM minimization as it does not change the FSM's language.

Id	Description	PFSM	*IncludeBinding*(Log L, Binding b)
a	always include	Fig. 12(a)	true
b	1-tail	Fig. 12(b)	true : $\forall t \in L, \forall\, e_i \in b(Y),\ \square\big[b(x) \rightarrow (\bigcirc e_1 \vee \ldots \vee \bigcirc e_i \vee \ldots \vee \bigcirc e_n)\big]$ in $t \wedge$ $\forall e \in b(Y), \exists t \in L,\ \Diamond(b(x) \wedge \bigcirc e)$ in t false : otherwise
c	2-tail	Fig. 12(c)	true : $\forall t \in L, \forall\, e_i \in b(Y),\ \square\big[b(x) \wedge \bigcirc b(y) \rightarrow (\bigcirc^2 e_1 \vee \ldots \vee \bigcirc^2 e_i \vee \ldots \vee \bigcirc^2 e_n)\big]$ in $t \wedge$ $\forall e \in b(Z), \exists t \in L,\ \Diamond(b(x) \wedge \bigcirc b(y) \wedge \bigcirc^2 e)$ in t false : otherwise
d	*i*-tail	Fig. 12(d)	true : $\forall t \in L, \forall\, e_i \in b(Y),$ $\square\big[b(x_0) \wedge \bigcirc b(x_1) \cdots \wedge \bigcirc^{i-1} b(x_{i-1}) \rightarrow (\bigcirc^i e_1 \vee \ldots \vee \bigcirc^i e_i \vee \ldots \vee \bigcirc^i e_n)\big]$ in $t \wedge$ $\forall e \in b(Y), \exists t \in L,$ $\square\big[b(x_0) \wedge \bigcirc b(x_1) \cdots \wedge \bigcirc^{i-1} b(x_{i-1}) \wedge \bigcirc^i e\big]$ in t false : otherwise

Fig. 13. *IncludeBinding* functions corresponding to the PFSMs in Fig. 12. We use $\bigcirc^i$ to represent a sequence of i instances of $\bigcirc$.

We have shown that π_j^i cannot reject s since it cannot terminate on s in any non-accepting states. Therefore, by contradiction, $s \in \mathcal{L}(F_{dkTails})$ and $\mathcal{L}(F_{pkTails}) \subseteq \mathcal{L}(F_{dkTails})$.

(2) $\mathcal{L}(F_{dkTails}) \subseteq \mathcal{L}(F_{pkTails})$.

Note that $s \in \mathcal{L}(F_{dkTails})$ implies that s is accepted by all property instances that make up $F_{dkTails}$. Let $s = a_0 \cdots a_n$. Since s is accepted by $\pi_{\alpha,\omega}$, $a_0 = \alpha$ and $a_n = \omega$.

By induction on k, we will show that if $s \in \mathcal{L}(F_{dkTails})$ then there exists an accepting path of equivalence classes, $[E_0, \ldots, E_n]$, that corresponds to s, and thus $s \in F_{pkTails}$.

Base case ($k = 1$). We prove this base case by induction on n, assuming $k = 1$.

Base case ($n = 2$). Show that $s = a_0 \cdot a_1 \cdot a_2 = \alpha \cdot a_1 \cdot \omega$ maps to an accepting path E_0, E_1, E_2 in $F_{pkTails}$.

Let $E_0 = E[\alpha]$.

Since α is the first symbol in all traces in L, there must be a property instance π_j^1 corresponding to the 2-tail PFSM in Fig. 12b, that binds x to α. Since $s \in \mathcal{L}(F_{dkTails})$, π_j^1 must accept $\alpha \cdot a_1 \cdot \omega$. Therefore, π_j^1 must bind Y to a set B, such that $a_1 \in B$. Because π_j^1 was mined, $\alpha \cdot a_1$ is a prefix for some trace $t \in L$. So, there exists a non-empty equivalence class $E_1 = E[\alpha \cdot a_1]$ and E_0 has a transition on a_1 to E_1.

Now, consider the string $a_1 \cdot \omega$. Since a_1 appears in some trace there must be a corresponding property instance π_m^1 (based on the PFSM in Fig. 12b) that binds x to a_1 and binds Y to a set B' such that $\omega \in B'$. For π_m^1 to be mined, $\exists t' \in L$ such that $t' = p \cdot a_1 \cdot \omega$. Let $E_2 = E[t']$. Note that $E[s] = E[t']$. To see this, consider Definition 8. Since both s and t' end with ω, there is only one symbol $e \in \Sigma_{\leq k}$ that satisfies the constraint from the Definition that $(s \cdot e) \in PF_L \Leftrightarrow (t' \cdot e) \in PF_L$, namely $e = \epsilon$. This is because all strings in PF_L that contain ω contain ω as the last symbol. Also by Definition 8, Δ, the transition relation on equivalence classes allows a transition on ω from E_1 to E_2 because $\Delta(E_1, \omega)$ contains $E[\alpha \cdot a_1 \cdot \omega] = E[s]$ and $E_2 = E[s]$. As a result E_0, E_1, E_2 is an accepting path for s in $F_{pkTails}$.

Inductive case ($n = i$). Assume that for all $s \in L$ where $|s| = i$, s maps to an accepting path $E_0, E_1, \ldots, E_i$ in $F_{pkTails}$. Show that for $s' \in L$ such that $|s'| = i + 1$, s' maps to an accepting path $E_0, E_1, \ldots, E_{i+1}$ in $F_{pkTails}$.

Consider a subset of L, L', that contains traces from L that are $i + 1$ long but are truncated by ω to be i long. That is, $L' = \{s | (s \cdot \omega) \in L, |s| = i\}$.

We can apply the inductive case to L'. So, for all $s \in L'$, s maps to an accepting path $E_0, E_1, \ldots, E_i$ in $F'_{pkTails}$. Note that since $L \subseteq L'$, the states in $F'_{pkTails}$ are a strict subset of the states in $F_{pkTails}$. Now, it suffices to show that for an $s \in L'$ we can extend its path in $F'_{pkTails}$ by E_{i+1} so that $E_0, E_1, \ldots, E_i, E_{i+1}$ would be an accepting path in $F_{pkTails}$.

For an $s \in L'$ its $E_{i+1} = E[s \cdot \omega]$. So we need to show that for the Δ in $F_{pkTails}$, $E_{i+1} \in \Delta(E_i, \omega)$. This follows from Definition 8.

Inductive case ($k = j$). We prove this by induction on n. We assume that the proof statement is true for $k = j$ and perform induction on n to show that the statement is true for $k = j + 1$.

Base case ($n = 2$). Show that $s = a_0 \cdot a_1 \cdot a_2 = \alpha \cdot a_1 \cdot \omega$ maps to an accepting path E_0, E_1, E_2, E_3 in $F_{pkTails}$.

Since $k = j > 1$, $F_{dkTails}$ includes property instances corresponding to the 1-tail PFSM (Definition 9). This means that we can re-use the base case for $k = 1$ above and construct the path E_0, E_1, E_2, E_3 corresponding to s in $F_{pkTails}$ in the same manner. This construction also holds for $k = j + 1$.

Inductive case ($n = i$). Assume that $a_0 \cdots a_i$ maps to a valid path $E_0, \ldots, E_i$. Show that $a_0 \cdots a_{i+1}$ maps to a valid path $E_0, \ldots, E_{i+1}$.

Consider the string $t = a_{i-j} \cdots a_i$. Each symbol in t corresponds to a property instance P, for a particular k value, that makes up $F_{dkTails}$ and which accepts all of the symbols at the tail of t in front of the symbol.

For example, a_{i-j} corresponds to some property instance π^j, which accepts the tail $a_{i-j+1} \cdots a_i$ of t. Using the base case construction of overlapping prefixes, we construct a path $E_0, \ldots, E_{i+1}$ that corresponds to $a_0 \cdots a_{i+1}$. □

Theorem 1 states that the languages of the FSMs produced by the declarative and procedural kTails algorithms are equivalent; however, the models themselves may differ in the number of states and transitions. For example, InvariMint is guaranteed to produce the minimal

FSM for the language, whereas the procedural algorithm is not. While the models may look different, they are the same semantically. Smaller models may be easier for a human to understand, but otherwise, it is not clear that either model has benefits over the other.

Next, we use InvariMint to specify Synoptic, another model-inference algorithm.

5 Expressing Synoptic with InvariMint

This section describes the Synoptic model-inference algorithm and formulates it declaratively with InvariMint. We refer to InvariMint formulation of Synoptic as *declarative Synoptic* to distinguish it from *procedural Synoptic*. Later, Section 6.4 will empirically compare the procedural and declarative implementations of Synoptic.

5.1 Procedural Synoptic and Its Shortcomings

Procedural Synoptic is a model-inference algorithm that explicitly infers property instances from the log, then constructs a model that satisfies them.[7] Procedural Synoptic first infers an overly-general model of the log, which accepts too many traces. Then, it progressively refines the model until every trace in the language of the model satisfies specific property instances mined from the log. Because procedural Synoptic models enforce these observed property instances, the models accurately describe the underlying system and can improve understanding and aid debugging [7], can help to automate the generation of test oracles [33], and can be extended to model program performance [34].

The procedural Synoptic algorithm has four steps: (1) Mine three types of properties from the log: "x always followed by y" (whenever event x occurs in a trace, event y also occurs later in the same trace), "x always precedes y" (whenever event y occurs in a trace, event x also occurs earlier in the same trace), and "x never followed by y" (whenever event x occurs in a trace, event y never occurs later in the same trace). (2) Build an initial model by merging all states with the same outgoing event into a single state.[8] (3) Iteratively apply counterexample-guided abstraction refinement (CEGAR) [11] to derive a model that satisfies all of the mined property instances. Procedural Synoptic does this by model-checking the current (e.g., initial) model against the mined property instances to find a counterexample trace in the model's language, which falsifies one or more of the property instances. Procedural Synoptic then traces the found counterexample in the model to find the first state responsible for falsifying the property instance, and refines (splits) that state to remove the counterexample path. Procedural Synoptic repeatedly refines the model to eliminate counterexamples until it reaches a model that satisfies all of the property instances. (4) Finally, to compact the model, procedural Synoptic applies procedural kTails($k = 1$) to the refined model, but only performs a merge if it does not unsatisfy any of the property instances.[9]

As an example of a procedural Synoptic execution, consider the log in Fig. 2a. When run with this input log procedural Synoptic produces the same output as SimpleAlg—the model in Fig. 2b. This model satisfies all property instances (31 in total) of the three kinds of property types used by procedural Synoptic. For instance, this model satisfies the mined property instance "`login` always followed by `logout`".

While procedural Synoptic has been empirically shown to help developers improve their system understanding and find bugs [7], it has two features that may cause its users difficulty.

First, procedural Synoptic is externally non-deterministic. The order in which it resolves the counterexamples may affect the language of the final model it produces. In contrast, procedural kTails (Fig. 10) is internally but not externally deterministic (i.e., the order in which states are merged is non-deterministic, but the final model is always the same). (More generally, the problem procedural Synoptic tries to solve is NP-complete [2], [11], [17], so the non-deterministic algorithm attempts to balance running time against the size of the final model.) If a user makes a change to the input log and procedural Synoptic produces a different model, the user does not know if the input log difference explains the change in the returned model. This makes it difficult to apply procedural Synoptic to verify a bug fix or to check how a new feature impacts the model.

Second, while procedural Synoptic is significantly more efficient on large traces than kTail-based model inference, it may still be slow. This is because procedural Synoptic must maintain all of the parsed log traces in memory, and it makes repeated model-checking invocations and repeatedly traverses the model.

Next, we present a declarative specification that approximates procedural Synoptic. We show that the declarative Synoptic algorithm resolves the above two issues of non-determinism and efficiency, and discuss insights that we gained about Synoptic through this formulation.

5.2 Modeling Synoptic with InvariMint

Procedural Synoptic's use of well-defined property types simplifies the task of declaratively specifying it with InvariMint—each of the three mined property types in procedural Synoptic (always followed by, always precedes, and never followed by) has a corresponding InvariMint property type. Fig. 14 lists the PFSMs for these three property types and Fig. 15 lists the corresponding *IncludeBinding* functions.

However, while procedural Synoptic explicitly specifies some of the log property types that the inferred models will enforce, the procedural definition imposed a property type that was unknown both to Synoptic users and to us, the researchers who developed the algorithm. The process of specifying Synoptic declaratively with InvariMint revealed this property type. We found that the initial procedural Synoptic model is not captured by the three explicit property types; rather, the declarative

7. For simplicity, and despite minor differences, we use "property" where the Synoptic literature uses the term "invariant".

8. Procedural Synoptic uses an event-based graph model with nodes representing event types and unlabeled edges representing observed event orderings in the log. This model is equivalent to an FSM with anonymous states, which is the model type we use in this paper.

9. In an event-based model, procedural Synoptic uses kTails($k = 0$) to merge nodes with identical event labels. This is equivalent to kTails ($k = 1$) in a state-based model.

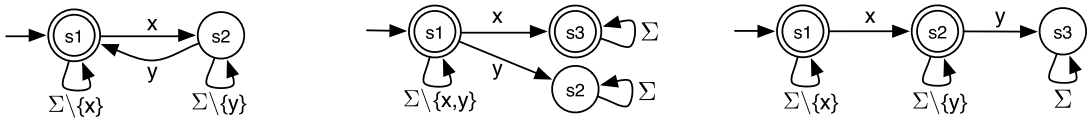


Fig. 14. Three of the five PFSMs used by InvariMint to specify declarative Synoptic. Figs. 5a and 12a show the remaining two PFSMs—the first captures the initial model, while the second identifies initial and terminal states.

specification requires the additional "immediately followed by" property type, which is exactly SimpleAlg's property type (Figs. 5a and 6a).

Finally, declarative Synoptic uses the same pre- and post-processing steps as the declarative kTails formulation (first paragraph of Section 4.2). The pre-processing step adds two fresh symbols—α and ω—to the each trace, and the post-processing step removes these from the final model. As in declarative kTails, these extra symbols are necessary to properly identify the initial and terminal states. Therefore, the last property type used by declarative Synoptic is the one in Figs. 12a and 13a, which mandates that all traces start with α and terminate on the first ω.

To compose declarative Synoptic property instances, InvariMint uses a composition function that is similar to SimpleAlg:

$$Compose(\pi_1, \ldots, \pi_n) = Min(\cdots(Min(\pi_1 \cap \pi_2) \cap \cdots) \cap \pi_n).$$

In this composition Min is FSM minimization, which minimizes intermediate models so as to maintain a small model in memory at run time. For a large number of property instances, this composition yields a faster algorithm.

Next, we evaluate declarative Synoptic.

5.3 Theoretical Evaluation

We were already intimately familiar with procedural Synoptic. Nonetheless, when we modeled Synoptic with InvariMint, we discovered a new feature, demonstrating how InvariMint can improve algorithm understanding. The InvariMint specification of Synoptic is, in fact, an *approximation* of the procedural Synoptic algorithm. A key feature of procedural Synoptic models is that every transition in the model is associated with some event in the log. This is because procedural Synoptic models are defined in terms of traces—a transition between two states in the model exists only if there are two observed states in the log that map to the model states and have this transition.

InvariMint models, on the other hand, are specified in terms of event types, so the particular trace-specific constraints are absent from an InvariMint model unless they are explicitly specified with property types. Therefore, for a given log, a declarative Synoptic model generalizes and includes the language of any procedural Synoptic model. Fig. 16 summarizes this relationship between the language of the model derived using declarative Synoptic, the languages of possible non-deterministically-derived procedural Synoptic models, and the input log. The InvariMint formulation is more permissive than procedural Synoptic and includes the language of all possible non-deterministically-derived procedural Synoptic models. Here, we prove that the language of a procedural Synoptic model is a subset of the model derived using declarative Synoptic (Section 6.2 empirically evaluates this containment). We also show that the InvariMint model does not satisfy any property instances that are not true of the input log. This result is analogous to Theorem 3 in [7].

Theorem 2 (Declarative Synoptic encompasses procedural Synoptic). *Let L be a log. Let $F_{\text{pSynoptic}}$ be an FSM produced with procedural Synoptic on L and let $F_{\text{dSynoptic}}$ be the FSM produced with declarative Synoptic on L. Let $\mathcal{L}(F_{\text{pSynoptic}})$ and $\mathcal{L}(F_{\text{dSynoptic}})$ be the languages of those models. Then $\mathcal{L}(F_{\text{pSynoptic}}) \subseteq \mathcal{L}(F_{\text{dSynoptic}})$.*

Proof. Let t be a trace in $\mathcal{L}(F_{\text{pSynoptic}})$. By construction, when procedural Synoptic terminates all traces accepted by its inferred model satisfy all instances of the always followed by, always precedes, and never followed by property instances mined from L. Therefore, t must satisfy all such property instances.

Consider each of the property instances intersected to form $F_{\text{dSynoptic}}$. First, each property instance of the three types described in Fig. 14 is mined from L, and therefore must be true in each trace in L. Since t satisfies all such property instances, the language of each of these instance FSMs must contain t. Second, each property instance of the type described in Fig. 5a accepts all traces whose transitions are pairs of consecutive events observed in L. Since each transition in $F_{\text{dSynoptic}}$ maps to at least one pair of consecutive events in at least one trace in L, a property instance FSM must accept t.

Id	Description	PFSM	*IncludeBinding*(Log L, Binding b)
a	x always followed by y	Fig. 14(a)	true : $\forall t \in L$, $\Box(b(x) \rightarrow \Diamond b(y))$ in t false : otherwise
b	x always precedes y	Fig. 14(b)	true : $\forall t \in L$, $(\Diamond b(y) \rightarrow \neg b(y)\ \mathrm{U}\ b(x))$ in t false : otherwise
c	x never followed by y	Fig. 14(c)	true : $\forall t \in L$, $\Box(b(x) \rightarrow \Box\neg b(y))$ in t false : otherwise

Fig. 15. $IncludeBinding$ functions corresponding to the PFSMs in Fig. 14.

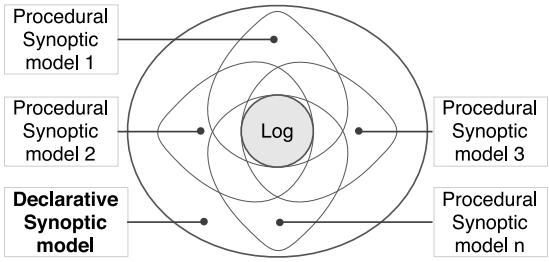


Fig. 16. The inclusion relationships between an input log, the language of the model derived from the log with declarative Synoptic, and the languages of all potential non-deterministically-derived procedural Synoptic models (numbered $1, \ldots, n$) for the same log.

Finally, $F_{\text{dSynoptic}}$ includes the property instance of the property type in Fig. 12a. This property instance accepts t because procedural Synoptic algorithm never refines the initial and terminal states associated with α and ω.

Since every property instance intersected to form $F_{\text{dSynoptic}}$ accepts t, $t \in \mathcal{L}(F_{\text{dSynoptic}})$. Therefore, $\mathcal{L}(F_{\text{pSynoptic}}) \subseteq \mathcal{L}(F_{\text{dSynoptic}})$. □

Theorem 3 (Models produced by declarative Synoptic do not include false property instances). *Let L be a log and let $F_{\text{dSynoptic}} = Compose(\pi_1, \ldots, \pi_n)$ be the FSM produced by declarative Synoptic on L.*

Let Π_{false} be the set of all property instances, such that $\forall \pi \in \Pi_{false}$, π is an instantiation of some declarative Synoptic property type that is not true in L.

Then $\forall i, \pi_i \notin \Pi_{false}$.

Proof. We present a proof by contradiction. Assume the opposite: $\exists \pi_i, \pi_i \in \Pi_{false}$.

Since π_i is used to construct $F_{\text{dSynoptic}}$, it must correspond to some property type, and by definition, to be included in $F_{\text{dSynoptic}}$, π_i must accept all traces in the input log L. Contradiction. □

As discussed in Section 5.1, procedural Synoptic is nondeterministic and executing it on two similar logs may produce different models, even when using identical random number generator seeds. Declarative Synoptic removes this non-determinism because FSM intersection and minimization are commutative. This, in turn, makes it possible to use the algorithm to assist in other development tasks, such as to check how a bug fix or new feature impacts the model.

6 InvariMint Empirical Evaluation

This section describes our open-source InvariMint implementation, and it empirically evaluates the declarative specifications of kTails and Synoptic against their procedural counterparts.

6.1 InvariMint Implementation

We have implemented InvariMint in Java, using the dk.brics [32] library for FSM operations. The InvariMint implementation is available as open source: http://synoptic.googlecode.com/.

Our InvariMint implementation exposes a command-line interface through which the user specifies the input log, property types, and the composition function. A user may write custom property types and composition functions by extending simple Java classes. Alternatively, a user may use one of the built-in property types and composition functions (these include all of the property types and composition functions described in this paper).

The property types and composition function form the specification of a model-inference algorithm. Meanwhile, the input log and a set of regular expressions for parsing this log are the input to the model-inference algorithm.

InvariMint produces models in the Graphviz format,[10] which can be visualized and manipulated using various open-source software.[11] Our most common use, however, has been to convert the Graphviz file into an image of the FSM model.

6.2 Overlap in Declarative and Procedural Synoptic Models

Recall that the InvariMint Synoptic specification is an approximation of the procedural Synoptic algorithm (Section 5.3). In this section, we evaluate how the models produced by these two algorithms compare in practice by measuring the overlap between models produced using the InvariMint Synoptic and the procedural Synoptic algorithms. We do this in two ways: First, we use a small log example to explain our model comparison approach. Second, we report on the overlap in models derived using the two algorithms in an experiment with randomly generated logs.

Fig. 17a shows a small log with two traces. Figs. 17b and 17c show the models generated for this log by procedural and declarative Synoptic algorithms, respectively. Note that the two models are different: declarative Synoptic adds a self-loop on event a to state s4. It does this because of the property instance "a immediately followed by $\{a, b\}$", corresponding to the property type in Fig. 5a. (Note that there is no outgoing transition from state s4 on event b because b is not allowed to immediately follow another b (i.e., the b on transition from state s3).)

We use a metric of model language similarity to compare models inferred by the two algorithms. A model's language is the set of strings it accepts. Because models with loops, such as the two models in the example, have infinite languages, we compare finite subsets of the languages consisting of all the strings in the language up to a constant length. For example, we can compare the two models in Fig. 17 by using a string length bound of 2. In this case, the languages are identical: both models accept the set of strings $\{ab, ba\}$. However, this bound does not even exercise all of the transitions in the models. A string length bound of 3 reveals that the procedural Synoptic model in Fig. 17b accepts the set $\{ab, ba, aab\}$, while the declarative Synoptic model in Fig. 17c accepts the set $\{ab, ba, aab, aba, baa\}$. This bound reveals that the two models accept different languages.

Let $\mathcal{L}_b(F_{\text{pSynoptic}})$ and $\mathcal{L}_b(F_{\text{dSynoptic}})$ be the length-bounded language subsets with bound b of the procedural and declarative Synoptic algorithms, respectively. Recall Theorem 2 in Section 5.3: for the same log, the language of the declarative Synoptic model always contains the entire

10. http://www.graphviz.org
11. e.g., Gephi: https://gephi.org/

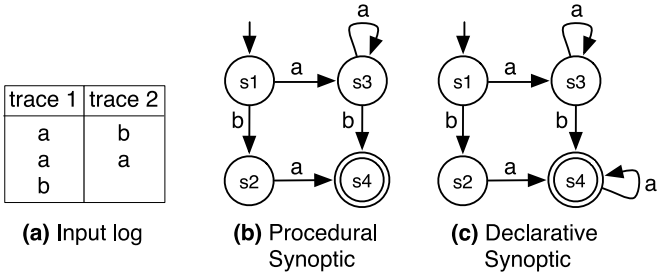


Fig. 17. An example log (a) with two traces for which procedural Synoptic (b) and declarative Synoptic (c) produce different models.

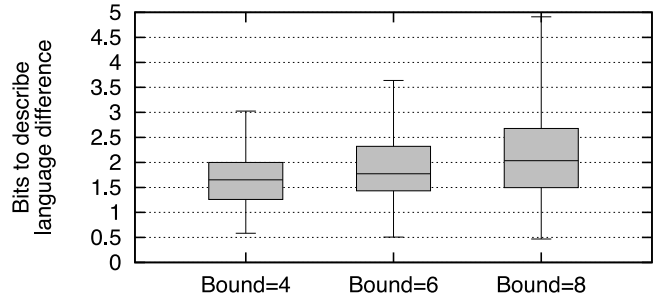


Fig. 18. Number of bits necessary to describe the difference in languages between declarative and procedural Synoptic models, in the 100-log experiment described in Section 6.2. Each box plot corresponds to a string length bound (4, 6, or 8) used to approximate the infinite language sets of pairs of models.

language of the procedural Synoptic model. This containment also holds for the length-bounded subsets of these languages, that is, $\mathcal{L}_b(F_{\text{pSynoptic}}) \subseteq \mathcal{L}_b(F_{\text{dSynoptic}})$. In the example, $\{ab, ba, aab\} \subset \{ab, ba, aab, aba, baa\}$. We can characterize the similarity of these two sets with the fraction of the declarative model's language that is covered by the procedural model's language. This coverage measure is monotonic—larger differences in the models result in larger differences in the measure—but not robust—small differences in models can result in large differences in the measure. (For example, the two models in Fig. 17 are quite similar—they have the same node sets (of size n) and differ by only one out of the $n \cdot n = 16$ possible edges among that node set—but that single edge difference results in a coverage measure of $\frac{3}{5} = 60\%$.) A more robust related measure is the number of bits necessary to describe the differences between the two models' languages. We thus use the following log-based, model-similarity, *coverage* metric:

$$log_2 \frac{|\mathcal{L}_b(F_{\text{dSynoptic}})|}{|\mathcal{L}_b(F_{\text{pSynoptic}})|}.$$

This coverage metric evaluates to 0 for two identical models (no bits necessary to describe the difference between two identical languages), and to 1 if the declarative model's language is twice as large as the procedural model's language. For the example in Fig. 17 with string length bound of 3, the coverage is $\log_2\left(\frac{5}{3}\right) = 0.73$ bits.

We next compare the two algorithms in an empirical study using 100 randomly-generated text logs. Each log contains six traces, and each trace has four events uniformly selected from an alphabet of eight possible event types. We ran declarative Synoptic (Section 5.2) on each log. Because procedural Synoptic is a non-deterministic algorithm and may produce different models for the same log, we ran procedural Synoptic ten times on each log, and used the model that is the union of the ten produced models for the comparison. (We found that ten Synoptic runs were sufficient to cover the different models procedural Synoptic can produce; rerunning the experiment with 1,000 Synoptic executions produced nearly identical results.) First, we empirically confirmed that Theorem 2 holds for the 100 models in the experiment. Second, for each of the 100 pairs of models, we computed the log-based coverage metric described above. We report results for three different string length bounds: bound of 4, a conservative bound that is the same as the length of traces in the experiment; and bounds of 6 and 8, which are 1.5 and 2.0 times the length of traces in the log. Fig. 18 shows the results as a box-plot, illustrating the median, 25th and 75th percentiles for metric for each of the string length bounds. For example, for the bound of 4, the minimum difference was 0.6 bits, the maximum difference was 3 bits, the median difference was 1.7 bits, and the average difference was 1.7 bits. This quantifies the differences between the models produced by the procedural and declarative Synoptic algorithms.

While declaratively inferred models always generalize the procedural models (which, in turn, generalize the input log), the declaratively inferred model introduce additional behavior (captured by our metric). This behavior may be removed by introducing additional property types that capture more trace context, which would make the declarative and procedural models more similar and decrease the metric. For example, the property type described in Figs. 12a and 13a excludes traces that do not start/end with events that were the initial/terminal events in the input traces. We leave a more detailed comparison of the two algorithms on more realistic program logs to future work.

6.3 Performance of Declarative and Procedural kTails

We performed two experiments to compare the performance of declarative kTails and procedural kTails. We found that the declarative kTails implementation outperforms procedural kTails on large logs with few property instances, while procedural kTails scales better with increasing number of property instances.

Our experiments were executed on an OS X 10.8 machine with a 2.8 GHz Intel i7 processor and 8 GB of RAM. In all experiments the bottleneck resource was the CPU. Our experiments used logs with tens of thousands of events. From our previous studies [7] we consider this to be a representative log size for logs generated by developers during debugging sessions. We used a script to generate synthetic logs with the desired number of event, event types, and other log characteristics. For all of the kTails experiments we used $k = 1$.

To evaluate scalability with respect to *log size*, we ran both algorithms on logs that ranged in size from 5 to 50 K events, but maintained a constant number of property instances per log. Every log in the experiment ranged over an alphabet of 5 event types, and each log was partitioned

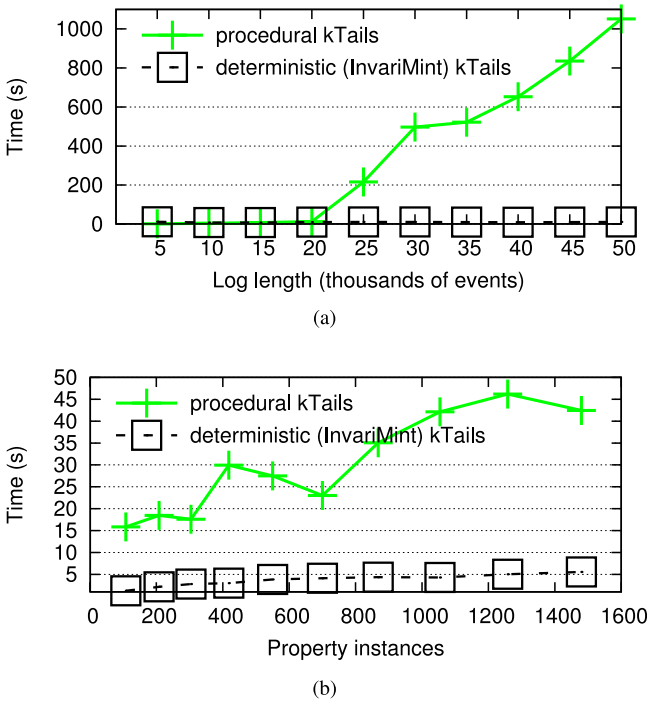


Fig. 19. (a) The running time of procedural kTails and the declarative InvariMint version of kTails for different log input sizes. The number of property instances true of the log was held constant at 182. (b) The running time of procedural kTails and the declarative InvariMint version of kTails for logs with different number of property instances. The size of the log was held constant at 25 K events.

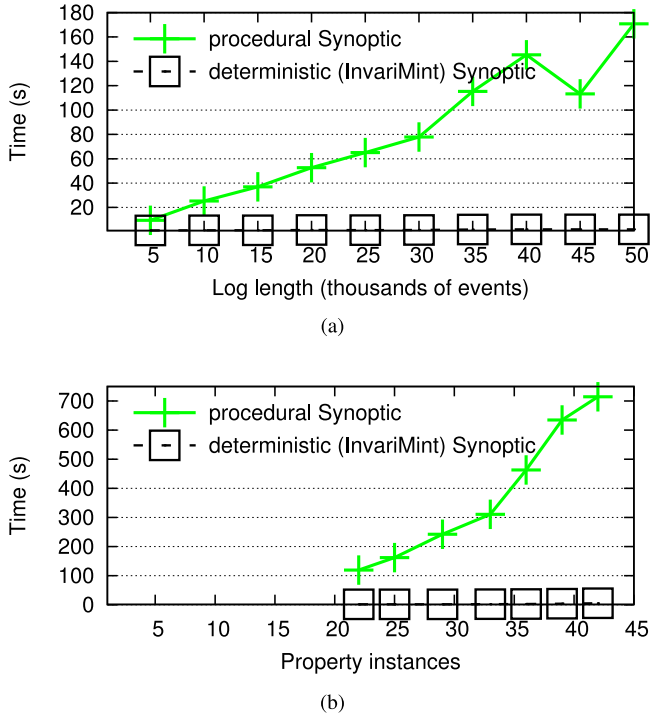


Fig. 20. (a) The running time of procedural Synoptic and the declarative InvariMint version of Synoptic for different log input sizes. The number of property instances true of the log was held constant at 19. (b) The running time of procedural Synoptic and the declarative InvariMint version of Synoptic for logs with different number of property instances. The size of the log was held constant at 25 K events.

into 20 traces of equal length. The number of property instances true for each log was held constant at 182. Fig. 19a plots the average runtime of three runs for each log size.

In the figure, as the log size increases the standard kTails algorithm scales poorly because it needs to perform more merges. The declarative kTails algorithm maintains an almost constant running time. This is because the algorithm composes a constant number of property instances in constant time—composing 182 property instances used in the experiment took about 10 seconds. Although the time to mine property instances does increase linearly with log size, it remains insignificant (for a 50 K event log, all property instances are mined in under one second).

To evaluate scalability with respect to *number of property instances*, we varied the number of property instances for the log from 108 to 1,480, but maintained a constant log size of 25 K events. Logs were drawn from an alphabet that had between 9 and 37 event types. Fig. 19b plots the average runtime of three runs. Overall declarative kTails had a lower running time than procedural kTails.

6.4 Performance of Declarative and Procedural Synoptic

We compared the performance of procedural Synoptic against the declarative Synoptic implementation. Both algorithms are implemented in Java and we use the same experimental setting as in the kTails experiments (Section 6.3).

We carried out two experiments to compare algorithm performance across different log sizes (Fig. 20a), and across logs with varying number of property instances (Fig. 20b). As with the kTails algorithm, Fig. 20a indicates that the declarative version of Synoptic outperforms procedural Synoptic on large logs. As the number of property instances increases (in Fig. 20b), declarative Synoptic continues to outperform procedural Synoptic. The difference in performance might be due to the fact that models generated with declarative Synoptic may include behaviors not captured by models generated with procedural Synoptic (Section 6.2).

6.5 InvariMint Performance

The InvariMint implementation run time can be broken up into the time it takes to mine the property instances, and the time it takes to compose those instances. For the property types we have considered in this paper, the mining time was negligible. For the InvariMint implementations of kTails and Synoptic, the composition time, and in particular the intersection of property instances, dominated the overall run time. The time to minimize the FSMs was negligible.

In general, we expect InvariMint implementations of algorithms to outperform procedural implementations. This was true for Synoptic and kTails in our experiments (Figs. 19 and 20), but it might not be true for all algorithms.

There are numerous factors that affect how the procedural and InvariMint implementations of algorithms perform, including:

Memory footprint. Our procedural implementations of kTails and Synoptic keep the entire input log in memory as both algorithms require continued access to the input log throughout the execution. For example, our Synoptic procedural implementations requires looking at the input log to infer the edges between its refined partitions at every

refinement step. For input logs that are larger than the available memory, these algorithms likely experience significant overhead. In contrast, after the InvariMint implementation mines the property instances from the input log, it no longer needs to access the log and deals exclusively with the property instances.

Slow dependencies. Many procedural implementations rely on slow subroutines. For example, the procedural Synoptic implementation performs model checking at every refinement stage. This both increases the run time and bounds the implementation's scalability, as model checking does not scale well. In contrast, declarative Synoptic does not have this dependency.

Algorithm uniqueness. While procedural definitions of algorithms can rely on obscure or one-off algorithms, many FSM manipulations necessary for the composition function are well-known and have been optimized through years of research. For example, declarative kTails and Synoptic benefit from the fact that FSM intersection and minimization are well studied, optimized procedures [20].

Coarsening versus refinement. Some procedural algorithms start with a very large FSM, and work to coarsen that graph, (e.g., kTails starts with an FSM that depicts every input trace as a separate path, and then must merge many states before arriving at a compact model). This process can be quite slow for large input logs, even if those logs have very few relevant property instances. By contrast, other procedural algorithms such as Synoptic start with a minimal FSM and refine (enlarge) it to more accurately describe the traces, so it performs fewer FSM transformations and operates on smaller FSMs.

In our experiments, we observed that declarative kTails and Synoptic were particularly beneficial over their procedural counterparts when the input logs were large and there were relatively few types of events. On very small logs (ones that can fit entirely in memory), procedural implementations may perform better than InvariMint implementations. Similarly, for logs with a very large number of types of events, the increased number of property instances slows down composition, and may benefit procedural implementations.

7 Discussion

This section describes several other benefits of declarative specifications, the power of binding evaluator and composition functions, tips for creating declarative specifications, and the limitations of our approach.

7.1 Benefits of Declarative Specification

Sections 4 and 5 presented insights derived from expressing existing model-inference algorithms with InvariMint. This section describes other benefits to declaratively specifying model-inference algorithms.

Declarative specifications can improve the efficiency of model checking and run-time verification through, for example, parallelization. Declarative specifications enable identifying violated property instances, which can be more helpful than a path counterexample in understanding a behavior or why a trace fails to conform to the model.

InvariMint can be robust to specifications with redundant, overlapping, or conflicting property types. For example, a composition function that intersects property instances will ignore redundant and overlapping property instances, and will immediately reveal conflicting property instances as their intersection would be the empty set. The user can use this to better understand the structure of algorithm.

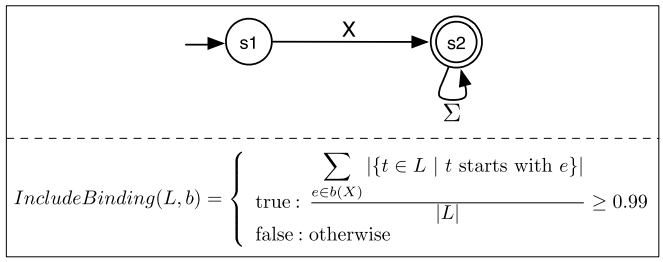


Fig. 21. A property type with a probabilistic $IncludeBinding$.

7.2 Complex Binding Evaluator Functions

The property types we presented so far use $IncludeBinding$ functions that are expressed as LTL formulae and closely mirror the corresponding PFSMs. For example, the property type in the SimpleAlg specification uses an $IncludeBinding$ that tests if "x immediately followed by $y \in Y$" for a particular binding of x and Y over all traces in the log (Fig. 6a). This function corresponds to the PFSM in Fig. 5a, which resembles the Büchi automaton of the true case in the $IncludeBinding$ function. More complex binding evaluator functions are possible, including a probabilistic function.

Probabilistic $IncludeBinding$. An algorithm designer can use an $IncludeBinding$ that returns true for a log and a binding if the condition is satisfied by *most* (e.g., 99 percent) of the traces. This is useful when the log contains errors or incomplete traces, or if the model is intended to represent only most common behavior. For example, for a live, online email server, a log may come from a period in the middle of the server's operation, rather than from its start to its termination. As a result, some traces may be missing the `login` event at the start, while others may be missing the `logout` event at the end. InvariMint can still mine the property instances that all traces start with `login` and end with `logout`, as long as an overwhelming fraction of the traces satisfy this property instance. Fig. 21 shows an example property type with a probabilistic $IncludeBinding$. This function corresponds to a PFSM that captures the initial events in the model. This $IncludeBinding$ evaluates to true if the candidate events have appeared as initial events in at least 99 percent of the traces in the log.

This property type illustrates that $IncludeBinding$ does not need to mirror the PFSM. Furthermore, this shows the power and flexibility of separating the PFSM specification of the shape of the event type from the $IncludeBinding$ specification of when the event type is instantiated.

7.3 Complex Composition Functions

All of the composition functions described so far in this paper have intersected property instances to derive the final model. However, more complex composition functions are possible. The intersection composition ensured that the model satisfied *all* of the mined property instances in the presented algorithms. Property instances, however, can

describe more than just the *must* rules. Next, we present a composition function that takes into account negative examples, in which property instances capture both the *must* and *must not* rules.

Composing with positive and negative examples. Consider a model inference algorithm for which some of the property types describe what must occur and the remaining property types describe what must not occur. Given two input logs: L^+ containing positive examples (e.g., actual executions of a system), and L^- containing negative examples (e.g., executions manually constructed to be invalid), the algorithm mines the must property instances, π_i^+, from L^+ and the must not property instances, π_j^-, from L^+. These property instances are then composed into the final model with the following composition function:

$$Compose(\pi_1^+, \ldots, \pi_n^+, \pi_1^-, \ldots, \pi_m^-) = \bigcap \pi_i^+ \setminus \bigcap \pi_j^-.$$

In this composition, two separate models are first composed, one for the positive examples and one for the negative examples. Then, the composition function subtracts the negative-example model from the positive-example model. This composition function illustrates how the InvariMint approach can specify more advanced model inference algorithms, such as those that use positive and negative examples.

7.4 Tips for Declaratively Expressing Algorithms with InvariMint

Model-inference algorithms use different approaches to generalize from the traces in the input log. While typical model-inference algorithms "bake in" the generalization logic into their procedural definitions, InvariMint uses property types and the composition function to make the generalization of the inference process explicit. In designing an InvariMint specification, it is critical to identify the types of properties that should be true and false of models inferred by the algorithm. For example, the Synoptic algorithm explicitly preserves properties of the form "event x is never followed by event y" in the final model. Sometimes, an algorithm's properties are less explicit and must be derived from a procedural specification, like the one for SimpleAlg in Fig. 3. In this case, we found it useful to focus on answering why the algorithm adds or retains a particular edge in the final model. The "immediately followed by" property type in the SimpleAlg InvariMint specification captures exactly this rationale.

It is also important to identify the right property-type granularity. Property types that are too fine-grained and too close to the input traces (e.g., union of the linear positive example trace FSMs) lead to models that overfit the log, rather than describe the algorithm.

Property types can describe algorithm operations. For example, Section 4 showed how a single property type describes merging of all states with the same k-tail. However, it is important to not simply simulate the procedural algorithm with the property types. Instead, we found that property types that work well are those that capture what the algorithm enforces.

If the procedural algorithm deals with positive examples of traces (as both kTails and Synoptic do), we found it helpful to start with a formulation that produces a model that is a generalization of the desired model. This model may enforce fewer property instances, which makes it easier to reason about the composition. Then, this initial formulation can be refined towards the desired specification by introducing new property types.

7.5 InvariMint Limitations

Although InvariMint can declaratively specify the Synoptic and kTails algorithms, it has some limitations.

Loss of trace context during mining. By construction, the declarative property types abstract input traces into property instances. This process of mining of property instances is lossy—potentially useful trace context is lost. An example of this is the initial and terminal states in the trace, which may be necessary to identify the initial and terminal states in the inferred model. To retain this information, the InvariMint kTails (Section 4.2) and InvariMint Synoptic (Section 5.2) algorithms add α and ω symbols during the pre-processing step and removed them during the post-processing step. The InvariMint approach could be generalized to allow for such pre- and post-processing steps by design. However, our goal is to describe the core approach and leave the details of what traces (pre-processed or not) are expected as input and how the output is post-processed up to the InvariMint user.

Limited expressiveness: property instances as LTL properties. The InvariMint approach uses the PFSM formalism for specifying just those properties of traces that are relevant to the algorithm. However, a PFSM models traces as independent sequences of events and cannot capture more complex properties, such as the statistical properties used in sk-strings [37], or performance-based properties used in Perfume [34].

Usability of declarative specifications. An InvariMint specification may include multiple PFSMs, a set of corresponding $IncludeBinding$ binding evaluator functions, and a composition function. Naturally, an algorithm designer who works with such a declarative specifications must be mathematically sophisticated. To a more general audience, the declarative specification may be less accessible than a procedural implementation in a popular programming language. Further, certain kinds of parameters may be more evident in procedural specifications than in declarative ones. For example, the k parameter for the kTails algorithm is simple to understand in the procedural version, but requires the introduction of new property types, albeit each created automatically from a general property type template shown in Fig. 12d. This can reduce the usability of declarative specifications.

Incremental computation. The current InvariMint implementation assumes that the entire log is available as input and does not work incrementally on new input traces. This is not a fundamental limitation and InvariMint can be extended to support incremental computation. Particularly, the event instances can be cached and checked against the new traces, new event instances can be mined from the new input traces, and the resulting set of event instances can be re-composed. This approach is not as fine-grained as in [30], but it can be extended to include some of the ideas in that work (e.g., by associating a timestamp with every transition in an event instance to support expiration).

8 Related Work

We have previously introduced InvariMint [5]. We add to this prior work by formally describing InvariMint (Section 3), giving a complete proof of the equivalence between the InvariMint declarative kTails algorithm specification and the procedural kTails specification (Section 4.3), and by improving exposition by correcting the specification of SimpleAlg and by improving the formalization of the binding evaluator functions.

Li et al. introduce an approach that resembles InvariMint, but targets reactive systems [23], [24]. Their approach is similar to InvariMint in that it uses a set of property templates and mines LTL specifications based on these templates. Unlike InvariMint, the mined properties are not composed into a model.

Walkinshaw and Bogdanov [39] propose a model-inference technique in which the user provides a model-inference algorithm with LTL formulae, which are then checked by a model checker and are used as constraints on feasible state merges in the inference algorithm. InvariMint uses LTL differently. Our intent is generalize the specification of model-inference algorithms. To this end, LTL formulae encode valid bindings of variables in a parameterized FSM to event types for a particular log input.

The kTails algorithm [8] is the basis for numerous model-inference algorithms [10], [12], [25], [26], [28], [29], [38], [39]. Many of these algorithms can be modeled with InvariMint to better understand, extend, combine, and compare them. At least two of the techniques require richer models than the standard FSM models we use in this paper. GK-Tails [29] requires EFSMs, and RPNI [10] requires Probabilistic FSMs. For example, the Alergia algorithm in [10] cannot be easily specified using InvariMint because the similarity of two states (during a merge in Alergia) is based on the transition probabilities, which are updated after each merge. It would be highly challenging to express this procedural merging algorithm with an InvariMint specification.

There are numerous algorithms to mine temporal property instances, like the ones we have used in this paper [3]. Javert [15] is a temporal specification mining tool that infers specifications by composing simpler "micropatterns" into larger ones. Javert's focus is on implementing this composition efficiently. InvariMint also uses composition to derive larger models from property instances, but the focus is on expressiveness of the declarative specification. We can leverage Javert's insights to improve our InvariMint implementation.

Structural, data-value property types that relate internal program variables are often described with variable values and can encode method pre- and post-conditions, as well as class-level property types. An example of a structural property type is two co-existing integer variables in a method are always equal. Automatically inferring property instances of these types from program executions [13] can improve model inference [29]. Combining structural and temporal property types can improve scenario-based specification mining [27].

Model-inference frameworks can facilitate algorithm comparison [36]. However, to date, these frameworks have been used to compare model performance and accuracy, not property types enforced by model inference. Further, much of the kTail-based model-inference work compares the recall and precision of inferred models against manually-specified ground-truth models. This process is manual, error-prone, and, again, compares model quality, as opposed to model-inference property types. Model quality is a notoriously challenging aspect of model inference [25]. QUARK, a comparison framework, allows for comparing the quality of models generated by algorithms such as kTails [8] and sk-strings [37]. InvariMint is complementary to these frameworks, as it aims to unify model-inference algorithms with a declarative specification language, facilitating algorithm comparison, and model property type comparison. Recent work by Gabel and Su tackles the question of false specifications in a mining algorithm through targeted program transformation. Their approach can be used to validate property instances instantiated during an InvariMint execution [16].

Non-FSM model inference (e.g., of UML sequence diagrams [42], communicating automata [9], and symbolic message sequence graphs [22]) can also aid developer tasks. Some of this work is similar to kTails, and we believe InvariMint can be extended to accommodate such algorithms. Similarly, InvariMint may be extendable to other types of property types, such as those used to infer behavioral models of web-services [4], [14].

9 Conclusion

Model-inference algorithms can automatically mine models of complex systems. Such models aid numerous development tasks, such as program understanding and debugging. Unfortunately, existing model-inference algorithms are defined procedurally, making them difficult to understand, extend, and compare to one another. We have presented InvariMint, a declarative specification approach for model-inference algorithms. InvariMint enables specification of algorithms in terms of the types of properties they enforce in the models they infer. InvariMint's declarative specifications (1) provide insight into how inference algorithms work and how the model relates to the underlying system, (2) allow for easy extension of existing algorithms to construct hybrid alternatives, and (3) provide a common language for comparing and contrasting the essential aspects of model-inference algorithms. We demonstrated the benefits of InvariMint by declaratively specifying the kTails algorithm and declaratively approximating the Synoptic algorithm. In addition, the InvariMint versions of these algorithms outperform their procedural analogs. We look forward to applying InvariMint's declarative approach more broadly and bringing these benefits to additional algorithms. InvariMint is available as an open-source tool: http://synoptic.googlecode.com

Acknowledgments

We would like to acknowledge Joseph Devietti, who proposed an early version of the InvariMint idea in a conversation. InvariMint is supported by NSERC, Google, Microsoft Research via a SEIF award, DARPA grant FA8750-12-2-0107, and NSF grants CNS-0963754 and CCF-1016701.

REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2007, pp. 25–34.

[2] D. Angluin, "Finding patterns common to a set of strings," *J. Comput. Syst. Sci.*, vol. 21, no. 1, pp. 46–62, 1980.

[3] C. M. Antunes and A. L. Oliveira, "Temporal data mining: An overview," in *Proc. Workshop Temporal Data Mining*, 2001.

[4] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2009, pp. 141–150.

[5] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Unifying FSM-inference algorithms through declarative specification," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2013, pp. 252–261.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2014, pp. 468–479.

[7] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2011, pp. 267–277.

[8] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Comput.*, vol. 21, no. 6, pp. 592–597, Jun. 1972.

[9] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker, "Learning communicating automata from MSCs," *IEEE Trans. Softw. Eng.*, vol. 36, no. 3, pp. 390–408, 2010.

[10] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Proc. Int. Colloquium Grammatical Inference Appl.*, 1994, pp. 139–152.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. Comput. Aided Verification*, 2000, pp. 154–169.

[12] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Trans. Softw. Eng. Methodology*, vol. 7, no. 3, pp. 215–249, 1998.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[14] M. D. Ernst, R. Lencevicius, and J. F. H. Perkins, "Detection of web service substitutability and composability," in *Proc. Int. Workshop Web Serv.—Modeling Testing*, 2006, pp. 123–135.

[15] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 339–349.

[16] M. Gabel and Z. Su, "Testing mined specifications," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, pp. 4:1–4:11.

[17] E. M. Gold, "Language identification in the limit," *Inform. Control*, vol. 10, no. 5, pp. 447–474, 1967.

[18] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2002, pp. 291–301.

[19] J. Hatcliff and M. B. Dwyer, "Using the Bandera tool set to model-check properties of concurrent Java software," in *Proc. Int. Conf. Concurrency Theory*, 2001, pp. 39–58.

[20] J. E. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," Stanford Univ., Stanford, CA, USA, Tech. Rep. STAN-CS-71-190, 1971.

[21] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2014, pp. 178–189.

[22] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Inferring class level specifications for distributed systems," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2012, pp. 914–924.

[23] W. Li, "Specification mining: New formalisms, algorithms and applications," Ph.D. thesis, EECS Dept., Univ. of California, Berkeley, Berkeley, CA, USA, Mar. 2014.

[24] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Proc. ACM/IEEE Int. Conf. Formal Methods Models for Codes.*, 2011, pp. 43–50.

[25] D. Lo and S.-C. Khoo, "QUARK: Empirical assessment of automaton-based specification miners," in *Proc. Working Conf. Reverse Eng.*, 2006, pp. 51–60.

[26] D. Lo and S.-C. Khoo, "SMArTIC: Towards building an accurate, robust and scalable specification miner," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 265–275.

[27] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2010, pp. 387–396.

[28] D. Lo, L. Mariani and M. Pezzè, "Automatic steering of behavioral model inference," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 345–354.

[29] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2008, pp. 501–510.

[30] L. Mariani, A. Marchetto, C. D. Nguyen, P. Tonella, and A. Baars, "Revolution: Automatic evolution of mined specifications," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, 2012, pp. 241–250.

[31] L. Mariani and M. Pezzè, "Dynamic detection of COTS component incompatibility," *IEEE Softw.*, vol. 24, no. 5, pp. 76–85, Sep./Oct. 2007.

[32] A. Møller. (2010). dk.brics.automaton—Finite-state automata and regular expressions for Java. [Online]. Available: http://www.brics.dk/automaton/

[33] C. D. Nguyen, A. Marchetto, and P. Tonella, "Automated oracles: An empirical study on cost and effectiveness," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2013, pp. 136–146.

[34] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral resource-aware model inference," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2014, pp. 19–30.

[35] A. Pnueli, "The temporal logic of programs," in *Proc. Symp. Found. Comput. Sci.*, 1977, pp. 46–57.

[36] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.

[37] A. V. Raman and J. D. Patrick, "The sk-strings method for inferring PFSA," in *Proc. Workshop Automata Induction, Grammatical Inference, Lang. Acquisition*, 1997, http://www.cs.iastate.edu/~honavar/mlworkshop.html#papers

[38] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2001, pp. 221–230.

[39] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 248–257.

[40] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Experience mining Google's production console logs," in *Proc. Workshop Manag. Syst. Log Anal. Mach. Learn. Techn.*, 2010, pp. 5:1–5:9.

[41] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2006, pp. 282–291.

[42] T. Ziadi, M. A. A. da Silva, L. M. Hillah, and M. Ziane, "A fully dynamic approach to the reverse engineering of UML sequence diagrams," in *Proc. IEEE Int. Conf. Eng. Complex Comput. Syst.*, 2011, pp. 107–116.

Ivan Beschastnikh received the PhD degree from the University of Washington in 2013. He is an assistant professor in the Department of Computer Science at the University of British Columbia. He has broad research interests that touch on systems and software engineering. His recent projects span distributed systems, formal methods, modeling, and program analysis. More information is available on his homepage: http://www.cs.ubc.ca/~bestchai/.

Yuriy Brun received the MEng degree from the Massachusetts Institute of Technology in 2003 and the PhD degree from the University of Southern California in 2008. He is an assistant professor in the School of Computer Science at the University of Massachusetts, Amherst. He completed his postdoctoral work in 2012 at the University of Washington, as a CI fellow. His research focuses on software engineering, distributed systems, and self-adaptation. He received a 2013 IEEE TCSC Young Achiever in Scalable Computing Award and a 2014 Microsoft Research Software Engineering Innovation Foundation Award. He is a member of the IEEE, the ACM, and the ACM SIGSOFT. More information is available on his homepage: http://www.cs.umass.edu/~brun/.

Jenny Abrahamson received the BS and MS degrees in 2012 and 2013, respectively, from the University of Washington, both in computer science & engineering. She is a software engineer working at Facebook Inc. on the platform team. Her research interests include the practical aspects of building complex software systems and helping developers to design, implement, and debug software.

Michael D. Ernst is a professor of computer science & engineering at the University of Washington. His research aims to make software more reliable, more secure, and easier (and more **fun**!) to produce. His primary technical interests include software engineering and related areas, including programming languages, type theory, security, program analysis, bug prediction, testing, and verification. His research combines strong theoretical foundations with realistic experimentation, with an eye to changing the way that software developers work. He was previously a tenured professor at MIT, and before that a researcher at Microsoft Research. He a senior member of the IEEE.

Arvind Krishnamurthy is an associate professor of computer science & engineering at the University of Washington. His research interests include all aspects of building practical and robust computer systems. His recent work is aimed at making improvements to the robustness, security, and performance of Internet-scale systems.