Unifying FSM-Inference Algorithms through Declarative Specification

Ivan Beschastnikh[†], Yuriy Brun[UM], Jenny Abrahamson[†], Michael D. Ernst[†], and Arvind Krishnamurthy[†]

[†]Computer Science and Engineering
University of Washington
Seattle, WA, USA
{ivan,jabrah,mernst,arvind}@cs.washington.edu

[UM]School of Computer Science
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

***Abstract*—Logging system behavior is a staple development practice. Numerous powerful model inference algorithms have been proposed to aid developers in log analysis and system understanding. Unfortunately, existing algorithms are difficult to understand, extend, and compare. This paper presents InvariMint, an approach to specify model inference algorithms declaratively. We apply InvariMint to two model inference algorithms and present evaluation results to illustrate that InvariMint (1) leads to new fundamental insights and better understanding of existing algorithms, (2) simplifies creation of new algorithms, including hybrids that extend existing algorithms, and (3) makes it easy to compare and contrast previously published algorithms. Finally, InvariMint's declarative approach can outperform equivalent procedural algorithms.**

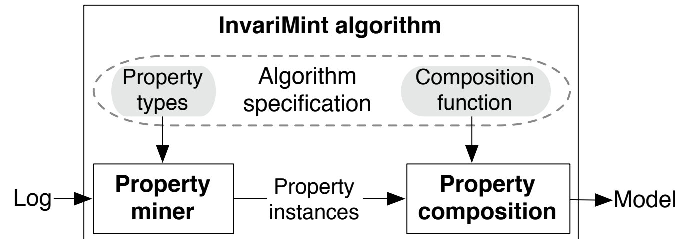


Fig. 1. An overview of the InvariMint approach. An InvariMint *algorithm* is parameterized by an algorithm specification, which consists of a set of property types and a composition function. The resulting InvariMint algorithm is a model-inference algorithm — it takes a log of traces as input and outputs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

I. Introduction

Understanding a system's behavior is a difficult development task that is required when a system behaves in an unexpected manner or when a developer must make changes to code they did not write. 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 future behavior [29].

Model inference is one promising approach to help users make sense of large and complex executions. The goal of model-inference algorithms is to produce a model, typically a finite state machine, that accurately and concisely represents the system that produced the log. Numerous such algorithms and corresponding tools already exist to help debug, verify, and validate systems [1], [6], [14], [15], [18], [19], [21], [22], [23], [26], [30].

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 and output a model that can be shown to a user. The procedural specification of these algorithms makes them difficult to understand, extend, and compare:

1. Understand. For most algorithms, it is difficult to understand which temporal and structural properties of the log are preserved in the inferred model. For example, if an inferred model of an email client requires each `login` event to be immediately followed by a `check mail` event, a developer may wish to know whether that property is true for all traces in the log, or is an artifact of the model-inference algorithm.

2. Extend. It is difficult to modify or to compose existing model-inference algorithms to create interesting hybrids. For example, a developer may use two inference algorithms: one to model exceptional executions and another that identifies executions with sequences of library calls not observed during testing. 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 instance of a log property from a specific invocation of the algorithm. If a log has every `login` event followed by a `check mail` event and if the developer decides that this property is an artifact of an incomplete log, the developer may want the model-inference algorithm to not preserve this property. However, because a procedural algorithm definition does not explicitly specify properties, such exclusions may be difficult.

3. Compare. Previously published algorithms lack a common form to aid comparison and juxtaposition. Instead, researchers must reason about pseudocode and work out complex proofs. A declarative approach, in which a model-inference algorithm is specified in terms of log properties that the inferred model will satisfy, allows researchers to, for example, identify

when two algorithms with vastly different procedural definitions produce models with identical, or overlapping, sets of properties.

This paper proposes InvariMint, a technique to specify 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 the mechanism of property *mining* from property *specification*. We illustrate the advantages of InvariMint by specifying three procedural algorithms declaratively. We find that InvariMint alleviates the above problems:

1. Understand. InvariMint expresses an algorithm *in terms* of the properties that the inferred model must satisfy. 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.

2. Extend. 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 in the log). New algorithms can be created, and multiple algorithms can be trivially composed to create hybrid approaches. For example, the Synoptic [5] algorithm uses the kTails algorithm as a final (coarsening) step to derive a more compact final model. Synoptic's InvariMint specification expresses this by simply merging the kTails property types into the Synoptic specification.

3. Compare. InvariMint makes it easier for those using and developing model-inference algorithms to compare and improve on those algorithms. For example, algorithms with incomparable procedural definitions may enforce overlapping sets of properties on their inferred models. InvariMint makes this overlap evident.

A model-inference algorithm outputs a model that accepts a formal language. The model's language is smaller than Σ^*: it is limited by certain temporal or structural properties that the algorithm mined from the log. Some of these properties may be explicit in the algorithm definition, whereas others may be implicit and deeply hidden in the procedural definitions.

InvariMint (Figure 1) represents a model-inference algorithm with the types of properties that are expected to be true of the inferred model. InvariMint mines instances of these properties from the log, represents each property as a DFA, and composes the DFAs using standard DFA operations (such as DFA union and intersection). Well-understood work on formal languages allows InvariMint to perform these operations efficiently and to produce minimal models [16].

To evaluate InvariMint, we applied it to two previously-published algorithms. First, we used InvariMint to declaratively and exactly specify the well-known kTails [6] algorithm. From our past experiences with kTails, we know that this algorithm behaves non-trivially on large log inputs. For instance, it is neither apparent which states will be merged, nor what synthetic traces the final kTails-inferred model will accept. The InvariMint formulation decomposes a kTails execution into a set of properties that are easy to inspect to better understand the characteristics of the final kTails-inferred model. The InvariMint kTails specification also provides the user with more fine-grained control over the execution of the algorithm — the user may remove a particular merge (by modifying a property instance) without having to modify the algorithm implementation.

Second, we used InvariMint to approximate Synoptic [5]. Synoptic is a more recent algorithm constructed with explicit log properties in mind. Although Synoptic attempts to make certain properties explicit, we found that it in fact preserves a set of implicit, or hidden, properties in its procedural declaration. Specifically, Synoptic allows a log event type to be immediately followed by another type only if such following occurred in the observed log. For example, Synoptic forbids a `login` event from being immediately followed by a `compose mail` event if, in the log, `login` was *always* immediately followed by a `check mail`. Synoptic's procedural declaration does not allow this property to be removed, altered, or relaxed, and hides this property from the user. In contrast, an InvariMint formulation of Synoptic makes this property 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 properties it enforces.

Finally, Synoptic is a non-deterministic algorithm. Depending on the order with which Synoptic satisfies the mined log properties, the algorithm might produce a different final model. Although our InvariMint Synoptic formulation is an approximation of Synoptic, its advantage is that it is deterministic and highly predictable. In particular, it is easier to check whether two different logs produce identical models.

As an added benefit, the InvariMint versions of kTails and Synoptic with efficient property mining scale linearly with log size and greatly outperform their procedural counterparts.

The rest of this paper is structured as follows: Section II uses an example model-inference algorithm to explain the InvariMint approach. Sections III and IV present InvariMint specifications of kTails and Synoptic, respectively. Section V discusses implications of our work. Section VI covers related prior research, and Section VII concludes.

II. The InvariMint approach

This section describes a model-inference algorithm named SimpleAlg and then overviews the InvariMint approach by outlining the InvariMint steps in specifying an example algorithm, called SimpleAlg. Sections III and IV extend SimpleAlg's specification to derive the kTails and Synoptic algorithms.

A. SimpleAlg

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. SimpleAlg's output is a model — a finite state machine whose language is a set of traces. (Figure 2 shows example input and

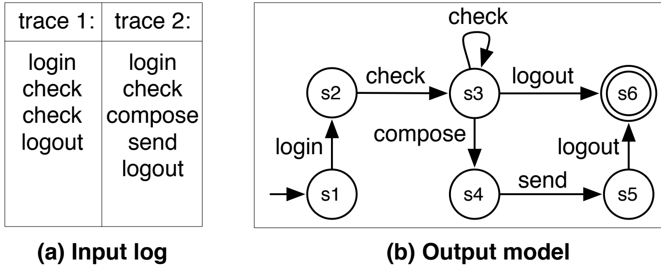


Fig. 2. **(a)** An example log of an email client with two traces. **(b)** The model inferred with SimpleAlg (Figure 3) for the input log in (a).

output.) The language corresponding to the model accepts all the traces in the log, as well as other traces. A model-inference algorithm's goal is to infer a model that accurately describes and generalizes the log: the extra accepted traces should be ones that are likely to be generated by the system that produced the log.

SimpleAlg is a model-inference algorithm. It generalizes in the following way: if SimpleAlg ever observes an event e_1 to be immediately followed by an event e_2 in the log, then whenever the system being modeled produces or consumes an e_1 event, SimpleAlg assumes that it is legal for the system to then produce or consume an e_2 event.

Pseudocode for SimpleAlg appears in Figure 3. In the generated model, each state represents an event that has just occurred.

```
1  Input: Log L
2  let M = new 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    M.addTransition(src=init, dst=s_f, label=f)
16
17    // For each pair of adjacent events, add a transition
18    // between states of corresponding event types.
19    foreach (Event e in t):
20       if (e.hasNext()):
21          let y = Event type of e
22          let z = Event type of e.next()
23          if (¬M.hasTransition(s_y, s_z)):
24             M.addTransition(src=s_y, dst=s_z, label=z)
25
26 Output: M
```

Fig. 3. Procedural pseudocode of the SimpleAlg algorithm.

The model contains one state for each unique event type that occurs in the log, plus one "initial" state. The model contains a transition from the state for event type e_1 to the state for event type e_2, with the label e_2, iff there exists a trace in the log in which an e_2 event immediately follows an e_1 event.

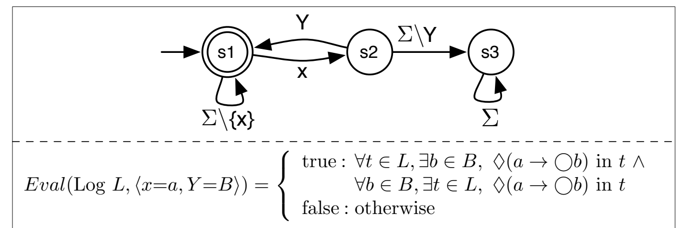
Figure 2(a) lists an email client log with two traces. The event alphabet is {`login`, `check mail` (shortened to `check`), `compose`, `send`, `logout`}. Figure 2(b) shows the model SimpleAlg infers for this input log. The model has six states, one for each event type (e.g., s4 corresponds to `compose`) plus the initial state (s1).

SimpleAlg's models are compact — the number of states is one more than the number of unique event types in the log, which is independent of the total number of events in the log. The running time is asymptotically linear in the size of the log. The inferred model's language always contains every trace in the input log, plus other traces SimpleAlg deemed likely.

B. *InvariMint overview*

InvariMint is an approach — or a common language — for describing model-inference algorithms, such as SimpleAlg. Figure 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**. The common language InvariMint uses to specify an algorithm is: a set of **property types** that describe properties to be mined from the log to derive *property instances*; and a **composition function** that combines the mined property instances into a final model.

Different model-inference algorithms take different approaches to generalizing the traces in the log to infer traces likely traces that are not in the log. What constitutes reasonable generalization is often subjective and depends on features of the system, its environment, and the specific development task that the model will be used for. While typical model-inference algorithms hard-code these features as assumptions in their procedural definitions, InvariMint uses property types and the composi-



(a) Property type (PFSM and Eval)

$$Compose(Prop_1, \ldots, Prop_n) = Minimize(\cap Prop_i)$$

(b) Composition function

Fig. 4. An InvariMint specification of SimpleAlg. This is equivalent to the pseudocode in Figure 3. **(a)** The property type "event x can be immediately followed by an event from set Y", represented as a parameterized FSM (PFSM) and a corresponding evaluation function (*Eval*). Given an input log, *Eval* determines the validity of bindings of parameters in the PFSM to event types. **(b)** The composition function, which InvariMint uses to compose a model from mined property instances.

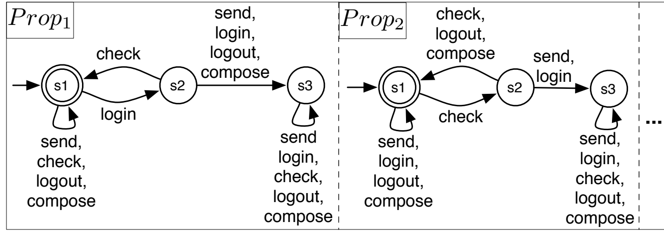


Fig. 5. Property instances mined by InvariMint from the log in Figure 2(a), based on property types in Figure 4(a). $Prop_1$ represents "event `login` can be immediately followed by an event from set {`check`}". $Prop_2$ represents "event `check` can be immediately followed by an event from set {`check`, `logout`, `compose`}".

tion function to generalize the model-inference process. Property types define desirable properties of the final model. For example, the SimpleAlg-inferred model preserves log properties, such as "event x can be immediately followed by an event from set Y". A property type consists of a parameterized FSM (PFSM) — an FSM with variable-labeled transitions (e.g., top portion of Figure 4(a)) — and an evaluation function to decide which bindings of PFSM variables to event types are valid in the log (e.g., bottom portion of Figure 4(a)). Together, the PFSM and evaluation function encode relationships between event types.

Using these evaluation functions, InvariMint mines the log for *property instances*, which are instantiations of the corresponding PFSM. InvariMint then combines the derived property instances into a model using the composition function (e.g., Figure 4(b)). The *Minimize* procedure referenced in this composition is the FSM minimization algorithm [16], which guarantees that the final model will be minimal.

We now illustrate InvariMint on the SimpleAlg example.

C. Specifying SimpleAlg with InvariMint

InvariMint's formulation of SimpleAlg has only a single property type: "event x can be immediately followed by an event from set Y". Figure 4 shows the InvariMint specification of SimpleAlg. Figure 4(a) shows the property type (a PFSM and an evaluation function). The PFSM is an FSM with variable labels that accepts all traces that relate event x and a set of events Y. The evaluation function defines which bindings of variables to log events result in valid property type instances. We use LTL to compactly specify evaluation functions. LTL statements use the operators *always* ($\Box$), *eventually* ($\Diamond$), *until* (U), and *next* ($\bigcirc$). For example, the evaluation function in Figure 4(a) returns true for event a and events set B whenever a can be immediately followed by only 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.

By indicating how to evaluate a binding of x and Y to event types, the evaluation function specifies how x and Y must relate: an event of type x must be immediately followed by one event from the set Y.

While all bindings can create property instances, the evaluation function determines which instances are valid for a given log. Figure 5 lists two of the property instances that are valid

```
1 Input: Log L,
        Property types ⟨PFSM_1, Eval_1⟩, ..., ⟨PFSM_n, Eval_n⟩
2 let Props = {}
3 foreach (Property type ⟨PFSM_i, Eval_i⟩)
4   foreach (Binding of variables in PFSM_i, B)
5     if (Eval_i(L, B)):
6       Props = Props ∪ {PFSM_i(B)}
7 Output: Props
```

Fig. 6. The generic property miner algorithm.

```
1 Input: Property instances Prop_1, ..., Prop_n,
        Composition function C
2 let Model = C(Prop_1, ..., Prop_n)
3 Output: Model
```

Fig. 7. The generic property composition algorithm.

for the log in Figure 2(a): $\langle x, Y\rangle = \{\langle \texttt{login}, \{\texttt{check}\}\rangle$, and $\langle x, Y\rangle = \langle \texttt{check}, \{\texttt{check}, \texttt{logout}, \texttt{compose}\}\rangle$. In addition to these two property instances, there are three others (one for each of `compose`, `send`, and `logout`). Note that $\langle \texttt{logout}, \emptyset\rangle$ is necessary to prevent allowing all events to follow `logout` in the inferred model.

Finally, InvariMint composes property instances using the composition function in Figure 4(b) to produce the final model. For SimpleAlg, the composition function returns the minimized version of the intersection of the property instances. Therefore, the resulting model is compact and includes only those traces that satisfy all of the mined property instances. This final model is identical to the one produced by SimpleAlg (Figure 2(b)). This paper mostly focuses on composition functions that involve only 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 uses positive and negative trace example may subtract the model of negative traces from one of positive traces.

D. InvariMint benefits

The InvariMint formulation of SimpleAlg provides three benefits over the SimpleAlg pseudocode: **(1)** The InvariMint formulation helps us understand the key properties of the final model derived with SimpleAlg by decoupling these properties from the mining and composition procedures, while the pseudocode mixes all three. **(2)** We can more easily add new constraints to the model by defining new property types, and eliminate behavior 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 $Prop_1$ in Figure 5. **(3)** We can, and will, extend the InvariMint formulation of SimpleAlg to construct InvariMint specifications for kTails and Synoptic. The pseudocode for these algorithms looks completely different from SimpleAlg's pseudocode, yet the InvariMint specification reveals that both kTails and Synoptic are based on the same property type (Figure 4(a)) used by SimpleAlg. The fact that all

```
1  Input: Log L, int k
2  let M = initial FSM model of traces in L
3
4  let merged = true
5  while (merged):
6    merged = false
7    foreach (States s1, s2 in M):
8      if (s1, s2 are k-equivalent):
9        M.merge(s1, s2)
10       merged = true
11
12 Output: M
```

Fig. 8. The kTails algorithm. Section III-A defines k-equivalence.

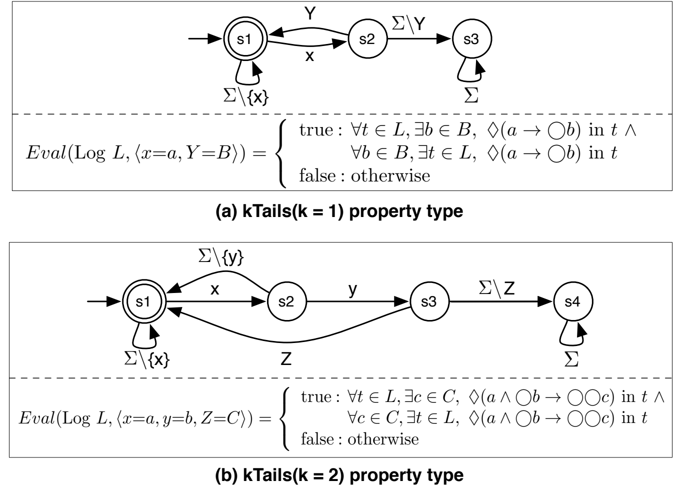


Fig. 9. **(a)** kTails(k=1) property type. **(a+b)** kTails(k=2) property types. Each of these is equivalent to the pseudocode in Figure 8 for the specific value of k.

three algorithms share this property type is one of the insights gained from specifying these algorithms with InvariMint.

InvariMint's goal is not to produce models, *per se*, but rather to provide a common language for expressing, or specifying, model-inference algorithms. Specifying different algorithms with the same language allows us to understand, combine, and compare the algorithms. InvariMint's common language is property types and composition functions. Once specified, the resulting property mining and property composition procedures (Figure 1) are straightforward. Figures 6 and 7 list the unoptimized pseudocode for these two procedures. Note that in practice, both of these algorithms can be further optimized and tailored to specific choices of property types and composition functions.

Next, we describe and evaluate the InvariMint specification of two previously-published model-inference algorithms — kTails and Synoptic.

III. Expressing kTails with InvariMint

kTails [6] is an extremely popular algorithm that has served as the basis for many modern model-inference algorithms. Unfortunately, there are many procedural descriptions of kTails, and it is difficult to tell if they produce identical or different models.

This section defines the kTails algorithm (Section III-A), demonstrates its InvariMint declarative specification (Section III-B), discusses the insights about kTails that InvariMint reveals (Section III-B), and reports on our empirical comparison of the procedural and declarative implementations of kTails (Section III-C).

A. kTails

kTails is a state-merging algorithm. kTails takes a log and a parameter k. It represents the log as a DFA composed of linear sub-DFAs, one per trace, that 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 DFA 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. Figure 8 lists the kTails pseudocode.

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. Therefore, 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.

In InvariMint kTails we introduce a pre- and a post-processing step. We modify each input trace to include an α and ω symbols at the start and end of each of the traces, respectively. After the property instances are composed into a final model we update states in the model with incoming α transition to be initial states, update states with outgoing ω transition to be accept states, and also remove all α and ω transitions from the model.

InvariMint uses property types to capture tail-equivalence and to specify kTails. Figure 9(a) lists the $k = 1$ property type for kTails. For $k = 2$, InvariMint requires two property types — the property type for $k = 1$ in Figure 9(a) and a new property type shown in Figure 9(b). Note that the property type for $k = 1$ kTails in Figure 9(a) is identical to the "can be immediately followed by" property type in Figure 4(a). This equality is not a coincidence — the k parameter generalizes the "can be immediately followed by" property type to k steps into the future.

The greater k is, the finer the granularity of the properties kTails enforces. For example, the property type in Figure 9(b) says that an event x, followed by an event y, must be followed by one — any one — of the events in the set Z. In other words, it corresponds to merging all x, y tails together. Section V discusses in more detail the granularity of properties and how the wrong granularity may cause the algorithm to overfit to the input log.

An important feature of the InvariMint kTails specification is

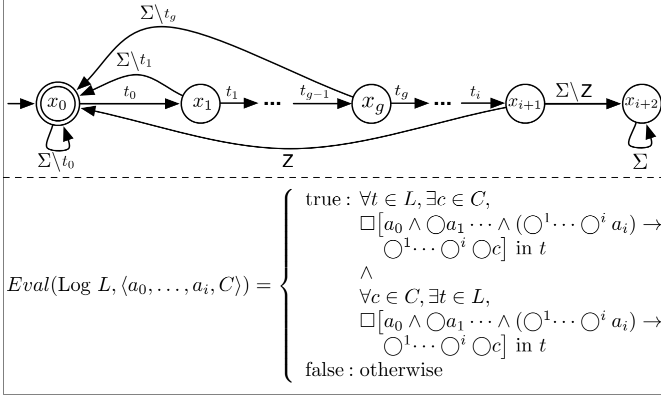


Fig. 10. The kTails($k = i$) property type.

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

B. Comparing procedural and InvariMint formulations of kTails

The model produced by the kTails algorithm behaves identically to the model produced by the InvariMint formulation of kTails. Next, we formally define the kTails algorithm based on the formulation in [10], and provide a proof of equivalence.

Let Σ_k denote the set of all strings of length k or less. Let a trace be a string over alphabet $\Sigma \cup \{\alpha, \omega\}$, and let a 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. We use $p \cdot t$ to denote concatenation of string t to p, and refer to t as the *tail*.

For example, consider the log $L = \{\alpha abc\omega, \alpha ab\omega, \alpha cd\omega\}$. Then, the corresponding $PF_L = \{\alpha\varepsilon\omega, \alpha a\omega, \alpha ab\omega, \alpha abc\omega, \alpha c\omega, \alpha cd\omega\}$. And, the string $\alpha abc\omega = \alpha a \cdot bc\omega$, in which $bc\omega$ is a tail.

Definition 1 (kTails FSM F_{kTails})**.** The kTails algorithm takes a log L and an integer k as inputs and generates a *kTails FSM* F_{kTails}. The **states** of F_{kTails} correspond to equivalence classes of prefixes from PF_L. An equivalence class E is a set of prefixes such that:

$$\forall (p, p') \in E, \forall t \in \Sigma_k, (p \cdot t) \in PF_L \Leftrightarrow (p' \cdot t) \in PF_L$$

That is, all prefixes in a class E have the same set of tails of length k or less, and every prefix in PF_L is assigned to some equivalence class.

The **transition function** Δ for equivalence classes, or states, in F_{kTails} 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 $p \cdot a$.

The **initial state** of F_{kTails} is $E[\varepsilon]$, and an equivalence class E_i is an **accept state** of F_{kTails} if $\exists\ s \in L$, such that $s \in E_i$.

Definition 2 (InvariMint kTails FSM $F_{InvMint}$)**.** For a log L and an integer k, let $F_{InvMint}$ be the FSM derived using the InvariMint algorithm specified by the kTails(k) property types and the input log L. We can express $F_{InvMint}$ as a composition of property instances[1]:

$$F_{InvMint} = \bigcap (P^1_1, \ldots, P^1_{n_1}, \ldots, P^k_1, \ldots, P^k_{n_k})$$

where $P^i_1, \ldots, P^j_{n_i}$ are the property instances for the PFSM corresponding to kTails($k = i$) property type. Figure 10 shows this generalized property type.

Definition 3 (Terminal rejection)**.** Let F be an FSM. F terminally rejects s if $\nexists\ t$ such that $s \cdot t$ is accepted by F.

Observation 1. *$F_{InvMint}$ does* not *terminally reject strings in PF_L.*

Proof: Consider a string $s \in PF_L$. Choose a tail t such that $s \cdot t$ is a trace in L. Such a tail must exist since s is a prefix for some trace in L. By construction, $F_{InvMint}$ accepts all strings in L. Therefore, $F_{InvMint}$ accepts $(s \cdot t) \in L$, and does not terminally reject s. □

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

$$\mathcal{L}(F_{kTails}) = \mathcal{L}(F_{InvMint})$$

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

1) $\mathcal{L}(F_{kTails}) \subseteq \mathcal{L}(F_{InvMint})$

Proof by contradiction:

Assume that $\exists\ s \in \mathcal{L}(F_{kTails})$ and $s \notin \mathcal{L}(F_{InvMint})$.

Because $s \notin \mathcal{L}(F_{InvMint})$ there is a non-empty set of rejecting (non-accepting) property instances $\Re$. That is, $\forall P \in \Re, s \notin \mathcal{L}(P)$. Let r be the *shortest* prefix of s to be rejected by some property instance $P^i_j \in \Re$, with $i \leq k$.

Now consider the prefix string r, which is rejected by P^i_j. We can express r as $r = u \cdot a$ for some $a \in \Sigma$. The property instance P^i_j (in Figure 10) can reject r in two ways:

(1a) P^i_j rejects r by terminating in state x_{i+2}, because $a \notin Z$.

In this case, r must be at least $i+1$ symbols long, and can be expressed as $r = v \cdot t_0 \cdots t_i \cdot a$. Consider the equivalence class $E_v = E[v]$. This class must be non-empty because there exists a transition on t_0 from E_v to $E[v \cdot t_0]$. Since E_v is non-empty, consider a prefix $p \in E_v$. Because $i < k$, and since $t_0 \cdots t_i \cdot a$ is a tail of v, by definition of equivalence classes, $p \cdot t_0 \cdots t_i \cdot a \in PF_L$. However, because $t_0, \ldots, t_i$ matches the tail corresponding to P^i_j, $a \in Z$. Contradiction ($a \notin Z$).

(1b) P^i_j rejects r by terminating in x_h, $0 < h \leq i+1$, and $s = r$.

Note that the LTL formula of the general kTails property type evaluation function (in Figure 10) mandates that each a_m bound to t_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 a_m in the evaluation function.

The above implies that $\forall g$, $0 < g \leq i+1$ there is no transition on ω *into* x_g. Since every trace terminates with ω, we can express

[1] We omit FSM minimization as it does not change the FSM's language.

r as $r = v \cdot \omega$. But, this contradicts P^i_j rejecting r in state x_h, since P^i_j can only terminate on $v \cdot \omega$ in states x_{i+2} or x_0.

We have shown that P^i_j cannot reject r since it cannot terminate on r in any non-accepting states. Therefore, by contradiction, $s \in \mathcal{L}(F_{InvMint})$ and $\mathcal{L}(F_{kTails}) \subseteq \mathcal{L}(F_{InvMint})$.

2) $\mathcal{L}(F_{InvMint}) \subseteq \mathcal{L}(F_{kTails})$

Note that $s \in \mathcal{L}(F_{InvMint})$ implies that s is accepted by all property instances that make up $F_{InvMint}$.

Let $s = a_0 \cdots a_n$. By induction on k and n, we will show that if $s \in \mathcal{L}(F_{InvMint})$ then there exists a valid and accepting path of equivalence classes, $[E_0, \ldots, E_n]$, that corresponds to s, and thus $s \in F_{kTails}$.

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, E_3 in F_{kTails}.

Let $E_0 = E[\varepsilon]$.

Since $\alpha \in \Sigma$, there must be a property instance P^1_j, of the property type in Figure 9(a), that binds t_0 to α. This P^1_j accepts $\alpha \cdot a_1$, and therefore P^1_j must bind Y to a set B, such that $a_1 \in B$. Next, the LTL formula corresponding to P^1_j tells us that $\exists\, t \in L$ such that $\Diamond(\alpha \rightarrow \bigcirc a_1)$. Since α is the first symbol for any trace, this means that $\alpha \cdot a_1$ is a prefix for this t. Since α is a valid prefix, there must exist a non-empty equivalence class $E_1 = E[\alpha]$. E_0 has a transition to E_1 on α, because α is a valid prefix for traces in L.

Now, consider the string $a_1 \cdot \omega$. Since a_1 appears in some trace there must be a corresponding property instance P^1_m. By the same reasoning as above, P^1_m binds t_0 to a_1 and binds Y to a set B' such that $\omega \in B'$. The LTL formula corresponding to P^1_m tells us that $\exists\, t' \in L$ such that $\Diamond(a_1 \rightarrow \bigcirc\omega)$. We can represent this t' as $t' = p \cdot a_1 \cdot \omega$.

Note that p (a prefix of t') and α have identical 1-tails, namely $\{a_1\}$. By construction of F_{kTails} this means that p and α belong to the same equivalence class E_1. Since, $p \cdot a_1$ is a valid prefix, there must exist an equivalence class $E_2 = E[p \cdot a_1]$, and there must be a transition from E_1 to E_2 on a_1.

Finally, we will use t' to construct E_3. Since $p \cdot a_1$ maps to E_2, there must be a transition on ω to E_3. This E_3 must be terminal because it contains the trace t'.

As a result, we have constructed an accepting path E_0, E_1, E_2, E_3 for the string s.

Inductive hypothesis (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 $a_i \cdot a_{i+1}$. Since a_i appears in some trace there must be a corresponding property instance P^1_j. By the reasoning in the base case, P^1_j binds t_0 to a_i and binds Y to a set B such that $a_{i+1} \in B$. The LTL formula corresponding to P^1_j tells us that $\exists\, t \in L$ such that $\Diamond(a_i \rightarrow \bigcirc a_{i+1})$. We can represent this t as $t = p \cdot a_i \cdot a_{i+1}$.

Based on our induction assumption, there exists an equivalence class E_{i-1} that corresponds to a_{i-1}. Since the prefix p is followed by a_i in t, p must also map to E_{i-1}. Therefore, we can extend $E_0, \ldots, E_{i-1}$ with E'_i and E_{i+1}, where $E'_i = E[p \cdot a_i]$ and $E_{i+1} = [p \cdot a_i \cdot a_{i+1}]$.

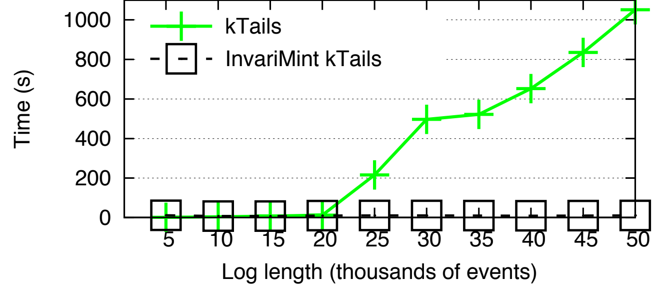


Fig. 11. The running time of procedural kTails and the declarative InvariMint version of kTails for different log input sizes. The number of property instances true of the log was held constant at 182.

Inductive hypothesis (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_{kTails}.

Since $k = j > 1$, $F_{InvMint}$ includes property instances corresponding to the kTails($k = 1$) property type (Definition 2). This means that we can re-use the base case for $\mathbf{k = 1}$ above and construct the path E_0, E_1, E_2, E_3 corresponding to s in F_{kTails} in the same manner. This construction also holds for $k = j + 1$.

Inductive hypothesis (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 P, for a particular k value, that makes up $F_{InvMint}$ 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 P^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 $a_0 \cdots a_{i+1}$. □

C. Empirical evaluation

We implemented InvariMint and the kTails algorithm in Java and evaluated their relative performance in two experiments. Both experiments were executed on an OS X 10.8 machine with a 2.8GHz Intel i7 processor and 8GB 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 [5] we consider this to be a representative log size for logs generated by developers during debugging sessions.

In the first experiment, we ran both algorithms on logs that ranged in size from 5K to 50K events, but maintained a constant number of property instances per log. Each log ranged over an alphabet of 5 event types, and each log was partitioned into 20 traces of equal length. The number of property instances true for each log was held constant at 182. We performed this experiment three times. Figure 11 plots the average runtime of the 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 InvariMint kTails algorithm maintains an almost constant running time. This is because for a constant number of property instances InvariMint kTails composes 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 50K event log, all property instances are mined in under one second).

In the second experiment, we varied the number of property instances for the log from 108 to 1,480, but maintained a constant log size of 25K events. Logs were drawn from an alphabet that had between 9 and 37 event types. As above, each run was repeated three times and Figure 12 plots the average for each set of three running times. Overall InvariMint kTails had a lower running time than procedural kTails. However, the relative ratio between the two running times indicates that InvariMint kTails scales worse than procedural kTails as the number of property instances increases.

Overall we found that our declarative InvariMint kTails implementation outperforms kTails on large logs with few property instances, while procedural kTails scales better with increasing number of property instances.

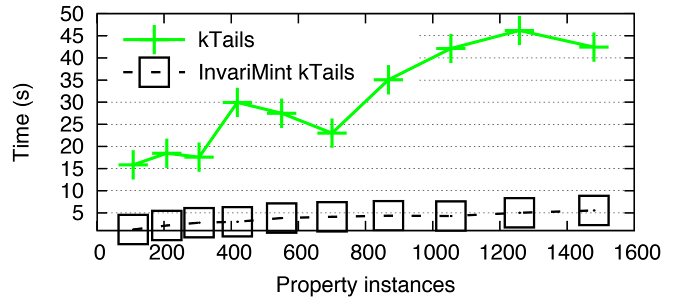


Fig. 12. The running time of procedural kTails and the declarative InvariMint version of kTails for logs with different number of property instances. The size of the log was held constant at 25K events.

IV. Expressing Synoptic with InvariMint

This section describes the Synoptic model-inference algorithm, formulates it with InvariMint, and evaluates the resulting formulation.

A. Synoptic and its shortcomings

Synoptic is a model-inference algorithm that explicitly infers properties from the log, then constructs a model that satisfies them.[2] Synoptic first infers an overly-general model of the log, which accepts too many traces. Then, Synoptic progressively refines the model until every trace in the language of the model satisfies specific properties mined from the log. Because Synoptic models enforce these observed properties, prior work has found that the models accurately describe the underlying system and can improve understanding and aid debugging [5].

The Synoptic algorithm has four steps: **(1)** Mine three kinds of properties from the log — "x AlwaysFollowedBy y" (whenever event x occurs in a trace, event y also occurs later in the same trace), "x AlwaysPrecedes y" (whenever event y occurs in a trace, event x also occurs earlier in the same trace), and "x NeverFollowedBy 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 anonymous[3] states with the same outgoing event into a single state. **(3)** Iteratively apply counterexample-guided abstraction refinement (CEGAR) [9] to derive a model that satisfies all of the mined properties. Synoptic does this by model checking the current (e.g., initial) model against the mined properties to find counterexample traces in the model's language, which falsify one or more of the properties. Synoptic then traces the found counterexample in the model to find the first state responsible for falsifying the property, and refines (splits) that state to remove the counterexample path. Synoptic repeatedly refines the model to eliminate counterexamples until it reaches a model that satisfies all of the properties. **(4)** Finally, to compact the model, Synoptic applies kTails(k=1) to the refined model, but only performs a merge if it does not un-satisfy any of the properties.[4]

While empirically shown to help developers improve their system understanding and find bugs [5], Synoptic has two features that may cause its users difficulty.

First, Synoptic is non-deterministic. The order in which it resolves the counterexamples may affect the language of the final model it produces. (More generally, the problem Synoptic tries to solve is NP-complete [9], [13], [2], 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 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 Synoptic to verify a bug fix or to check how a new feature impacts the model.

Second, while significantly more efficient on large traces than kTail-based model inference, Synoptic may still be slow. This is because 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 an InvariMint formulation that approximates Synoptic. We show that the InvariMint algorithm resolves the above two issues of non-determinism and performance, and discuss insights that we gained about Synoptic through this formulation.

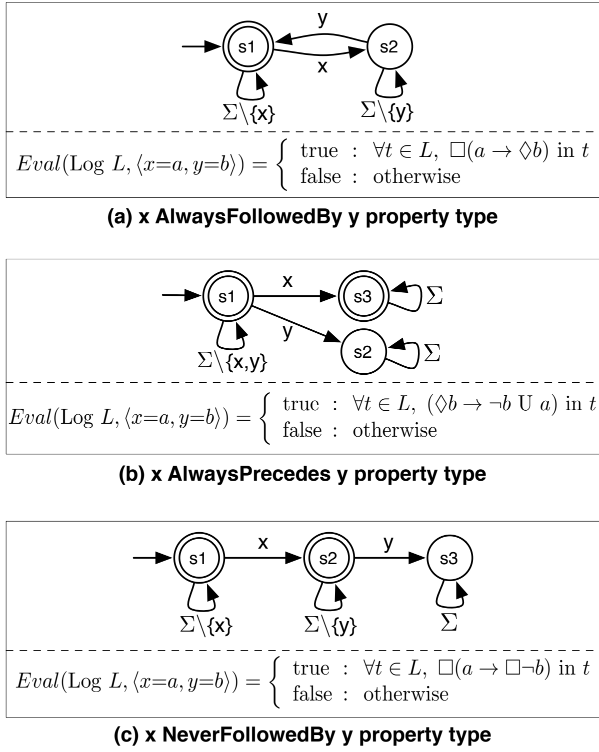
B. Modeling Synoptic with InvariMint

Synoptic's use of well-defined properties simplifies the task of declaratively specifying it with InvariMint — each of the three

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

[3] 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.

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



(a) x AlwaysFollowedBy y property type

(b) x AlwaysPrecedes y property type

(c) x NeverFollowedBy y property type

Fig. 13. Three of the four property types used by InvariMint to model the Synoptic algorithm. Figure 4(a) shows the fourth property type, which captures Synoptic's initial model.

mined properties in Synoptic (AlwaysFollowedBy, AlwaysPrecedes, and NeverFollowedBy) has a corresponding property type, shown in Figure 13.

However, while Synoptic explicitly specifies some of the log properties that the inferred models will enforce, its original procedural definition imposed a property 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. We found that the initial Synoptic model is not captured by the three explicit properties and the InvariMint formulation requires the additional "immediately followed by" property type, which is exactly SimpleAlg's property type (Figure 4(b)).

To compose Synoptic property instances, InvariMint uses a composition formula that is similar to SimpleAlg: $Compose(Prop_1, \ldots, Prop_n) = Minimize(\cdots (Minimize(Prop_1 \cap Prop_2) \cap \cdots) \cap Prop_n)$. This composition minimizes intermediate models so as to maintain a small model in memory at runtime. For a large number of property instances, this composition yields a faster algorithm.

Next, we evaluate this InvariMint formulation of Synoptic.

C. Theoretical evaluation

We were already intimately familiar with Synoptic. Nonetheless, when we modeled Synoptic with InvariMint, we discovered a new feature, demonstrating how InvariMint can improve algorithm understanding. The InvariMint formulation of Synoptic is, in fact, an *approximation* of the Synoptic algorithm. A key feature of Synoptic models is that they have no *spurious* transitions. That is, every transition in the model is associated with some event in the log — there are no uncovered, or *spurious*, transitions. The reason for this feature is that 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.

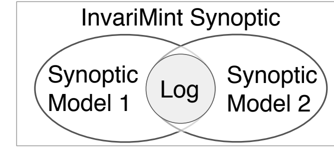


Fig. 14. The inclusion relationships between an input log, the language of the model derived from the log with InvariMint Synoptic, and the languages of two potential non-deterministically-derived Synoptic models for the log.

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, InvariMint models may contain spurious transitions. Figure 14 summarizes the relationships between the language of the model derived using an InvariMint formulation of Synoptic, the languages of possible non-deterministically-derived Synoptic models, and the input log. The InvariMint formulation is more permissive than Synoptic, and includes the language of all possible non-deterministically-derived Synoptic models. Here, we prove that a Synoptic model's language is a subset of the model derived using InvariMint Synoptic algorithm. We also show that the InvariMint model does not satisfy any Synoptic property instances that are not true of the input log. This result is analogous to Theorem 3 in [5].

Theorem 2 (InvariMint specification of Synoptic encompasses Synoptic)**.** *Let L be a log. Let $F_{Synoptic}$ and $F_{InvMint}$ be the FSMs produced by the Synoptic algorithm and the InvariMint Synoptic algorithm on L, respectively. Let $\mathcal{L}(F_{Synoptic})$ and $\mathcal{L}(F_{InvMint})$ be the languages of those models. Then $\mathcal{L}(F_{Synoptic}) \subseteq \mathcal{L}(F_{InvMint})$.*

Proof: Let t be a trace in $\mathcal{L}(F_{\text{Synoptic}})$. By construction, Synoptic terminates when all traces accepted by its inferred model satisfy all instances of the AlwaysFollowedBy, AlwaysPrecedes, and NeverFollowedBy property instances mined from L. Therefore, t must satisfy all such property instances.

Consider each of the property instances intersected to form F_{InvMint}. First, each property instance of the three types described in Figure 13 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 Figure 4(a) accepts all traces whose transitions are pairs of consecutive events observed in L. Since each transition in F_{InvMint} maps to at least one pair of consecutive events in at least one trace in L, a property instance FSM must accept t.

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

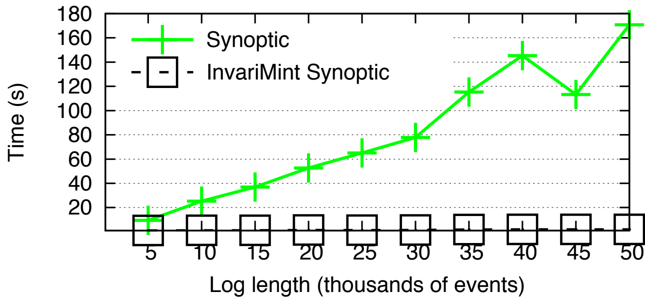


Fig. 15. The running time of procedural Synoptic and the declarative InvariMint version of Synoptic for different log input sizes. The number of property instances true of the log was held constant at 19.

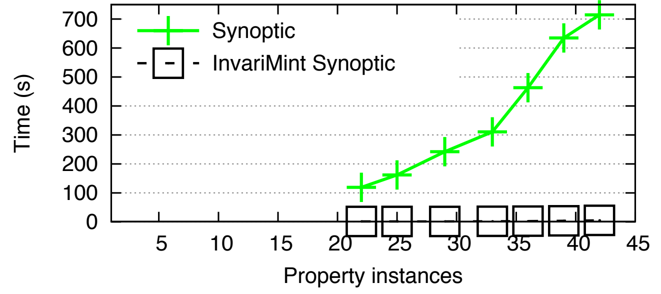


Fig. 16. The running time of procedural Synoptic and the declarative InvariMint version of Synoptic for logs with different number of property instances. The size of the log was held constant at 25K events.

□

Theorem 3 (Models produced by InvariMint Synoptic do not include false property instances)**.** *Let L be a log and let $F_{InvMint}$ be the FSM produced by the InvariMint Synoptic algorithm on L. More specifically, let $F_{InvMint} = Compose(P_1, ..., P_n)$.*

Let P_{false} be a set *of property instances, such that $\forall P_f \in P_{false}$, P_f is an instantiation of some Synoptic property type $\langle PFSM, Eval \rangle$, such that $\exists$ a binding B_f, $P_f = PFSM(B_f)$ and $Eval(L, B_f)$ is $false$. That is, P_{false} contains well formed property instances that are not true for the input log L.*

Then $\forall i, P_i \notin P_{false}$.

Proof:

We present a proof by contradiction. Assume the opposite — $\exists P_i$, $P_i \in P_{false}$.

Since P_i is used to construct F_{InvMint}, it must correspond to some property type $\langle PFSM, Eval \rangle$, and by the pseudocode in Figure 6, $\exists B$, $P_i = PFSM(L, B)$ and $Eval(L, B)$ is $true$.

However, by definition of the set P_{false}, $Eval(L, B)$ must be false. Contradiction.

□

As discussed in Section IV-A, Synoptic is non-deterministic and executing Synoptic on two similar logs may produce different models, even when using identical random number generator seeds. The InvariMint formulation of 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 verify a bug fix or to check how a new feature impacts the model.

D. Empirical evaluation

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

We carried out two experiments to compare algorithm performance across different log sizes (Figure 15), and across logs with varying number of property instances (Figure 16). As with the kTails algorithm, Figure 15 indicates that the declarative version of Synoptic outperforms procedural Synoptic on large logs. As the number of property instances increases (in Figure 16), InvariMint Synoptic continues to outperform Synoptic.

V. Discussion

Although this paper has presented insights derived from expressing existing model-inference algorithms with InvariMint, there are other benefits to the InvariMint formulation. If the model is used for model checking or runtime verification, a declarative specification can be more efficiently checked (e.g., in parallel) against a property, and can yield more efficient runtime conformance checking of a trace. A violated property instance can also be more helpful than a path counterexample in understanding why the property does not hold or why a trace does not conform to the model.

As an example of the generality and expressiveness of our approach, an evaluation function may deem a property valid if it is true in *most* of the traces. This can be useful when the properties are probabilistic or the log is incomplete, as when it is not feasible to capture a log from a live, online system's start of execution to its end. For example, some traces at the start may be missing the `login` event while others at the end may be missing the `logout` event. InvariMint can still mine the property that all traces start with `login` and end with `logout`, as long as an overwhelming fraction of the traces satisfy that property. Other kinds of property types include conditional properties (e.g., an event is present only if the username is `root`), properties on resource usage (e.g., time or space), and anomaly-detecting properties (e.g., two events co-occur rarely).

In this paper we use LTL to compactly specify evaluation functions. As a result, in all of the presented examples the PFSM could be automatically derived from the LTL — the PFSM is a parameterized version of the büchi automaton corresponding to the LTL formula. However, this is not possible for the alternative evaluation functions mentioned above, as these cannot be expressed with LTL.

InvariMint can be robust to specifications with overlapping or conflicting property types. For example, an evaluation function that intersects property instances will ignore overlapping property instances, and will immediately reveal conflicting property instances as their intersection would be the empty set.

A. Tips for declaratively expressing algorithms with InvariMint

First, identify the right property-type granularity. Do not simply simulate the procedural version of the algorithm with the

property types. Instead, consider the properties that the procedural algorithm enforces. Property types that are too fine-grained and too close to the input traces (e.g., union of positive example trace DFAs) lead to models that overfit the log, rather than describe the algorithm. Property types can describe algorithm operations. For example, Section III showed how a single property type describes merging of all states with the same k-tail.

If the procedural algorithm deals with positive examples of traces (as both kTails and Synoptic do), starting from a formulation that produces a model that is a generalization of the desired model may be easier, as this model may enforce fewer properties. Then, refine this model towards the desired model by introducing new property types or by refining the existing properties.

If the procedural algorithm deals with both positive and negative examples of traces (we have not shown such an algorithm in this paper), consider building separate models, one for the positive examples and one for the negative examples. Then, in the composition function, subtract the negative-example model from the positive-example model.

VI. Related work

The kTails algorithm [6] is the basis for numerous model-inference algorithms [10], [21], [18], [19], [22], [8], [26], [27]. 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 [22] requires EFSMs, and RPNI [8] requires Probabilistic FSMs.

There are numerous algorithms to mine temporal properties, like the ones we have used in this paper [3]. Data-value properties that relate internal program variables can encode method pre- and post-conditions, as well as class-level properties. Automatically inferring these properties from program executions [11] can improve model inference [22]. Combining data-value and temporal properties can improve scenario-based specification mining [20]. Recent work by Gabel and Zhendong can also be applied to validate property instances during an InvariMint execution [12].

Model-inference frameworks can facilitate algorithm comparison [24]. However, to date, these frameworks have been used to compare model performance and accuracy, not properties of 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 properties. Model quality is a notoriously challenging aspect of model inference [18]. QUARK, a comparison framework, allows for comparing the quality of models generated by algorithms such as kTails [6] and sk-strings [25]. 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 comparison.

Non-FSM model inference (e.g., of UML sequence diagrams [31], communicating automata [7], and symbolic message sequence graphs [17]) 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 properties, such as those used to infer behavioral models of web-services [4].

Walkinshaw et al. [28] 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.

VII. 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 two existing algorithms, kTails and Synoptic. For example, the InvariMint versions of these algorithms greatly 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

VIII. Acknowledgments

We would like to acknowledge Joseph Devietti, who proposed an early version of the InvariMint idea in a conversation. We also thank the anonymous reviewers for their helpful feedback. InvariMint is supported by Google, 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 *ESEC/FSE*, 2007.

[2] D. Angluin. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences*, 21(1):46–62, 1980.

[3] C. M. Antunes and A. L. Oliveira. Temporal Data Mining: An Overview. In *KDD 2001 Workshop on Temporal Data Mining*, 2001.

[4] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In *ESEC/FSE*, pages 141–150, 2009.

[5] I. Beschastnikh, Y. Brun, S. Schneider, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *ESEC/FSE*, pages 267–277, 2011.

[6] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE TC*, 21(6):592–597, 1972.

[7] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Learning Communicating Automata from MSCs. *IEEE TSE*, 36(3):390–408, 2010.

[8] R. C. Carrasco and J. Oncina. Learning Stochastic Regular Grammars by Means of a State Merging Method. In *ICGI*, 1994.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[10] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM ToSEM*, 7(3):215–249, 1998.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE*, pages 213–224, 1999.

[12] M. Gabel and Z. Su. Testing Mined Specifications. In *FSE*, 2012.

[13] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[14] S. Hangal and M. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE*, 2002.

[15] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *CONCUR*, 2001.

[16] J. E. Hopcroft. An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. Technical report, Stanford Univ., 1971.

[17] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Inferring Class Level Specifications for Distributed Systems. In *ICSE*, 2012.

[18] D. Lo and S.-C. Khoo. QUARK: Empirical Assessment of Automaton-based Specification Miners. In *WCRE*, 2006.

[19] D. Lo and S.-C. Khoo. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *FSE*, 2006.

[20] D. Lo and S. Maoz. Scenario-Based and Value-Based Specification Mining: Better Together. In *ASE*, pages 387–396, 2010.

[21] D. Lo, L. Mariani, and M. Pezzè. Automatic Steering of Behavioral Model Inference. In *ESEC/FSE*, 2009.

[22] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE*, 2008.

[23] L. Mariani and M. Pezzè. Dynamic Detection of COTS Component Incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[24] M. Pradel, P. Bichsel, and T. R. Gross. A Framework for the Evaluation of Specification Miners Based on Finite State Machines. In *ICSM*, 2010.

[25] A. V. Raman and J. D. Patrick. The sk-strings Method for Inferring PFSA. In *AIGILA*, 1997.

[26] S. P. Reiss and M. Renieris. Encoding Program Executions. In *ICSE*, 2001.

[27] N. Walkinshaw and K. Bogdanov. Inferring Finite-State Models with Temporal Constraints. In *ASE*, pages 248–257, 2008.

[28] N. Walkinshaw and K. Bogdanov. Inferring Finite-State models with temporal constraints. pages 248–257, Sept. 2008.

[29] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Experience Mining Google's Production Console Logs. In *SLAML*, 2010.

[30] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *ICSE*, 2006.

[31] T. Ziadi, M. da Silva, L. Hillah, and M. Ziane. A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams. In *ICECCS*, pages 107–116, 2011.