InvariMint: Modeling Logged Behavior with Invariant DFAs

Jenny Abrahamson

Computer Science & Engineering
University of Washington

March 16, 2012

Abstract

Developers often struggle to understand their systems. In prior work we created Synoptic, a tool to help developers by inferring a concise and accurate system model from execution logs. However, Synoptic has numerous limitations: (1) it is slow when run on large input logs, (2) it is non-deterministic and does not always generate optimal models, (3) it is difficult to extend, and (4) the Synoptic algorithm is complex and difficult for users to understand.

To remedy the above limitations we developed InvariMint, which reformulates the core Synoptic process in terms of formal languages. InvariMint generates a DFA for each log invariant, intersects the invariant DFAs, and minimizes the resulting model. This model inference approach is independent of the size of the log, deterministic, easy to extend, and uses standard techniques that are easy to understand. We have also found that the InvariMint approach generalizes to other algorithms used in model inference, such as the k-Tails algorithm. We think that InvariMint can act as a common denominator in model inference, using which researchers can easily compare a wide range of algorithms and which provides users with a simple process for deriving customized hybrid algorithms.

1 Introduction

Logging is commonly used by developers to record state and behavior information about their systems. Logged information is useful for, among other things, understanding and debugging programs. Manually inspecting logs is both tedious and challenging: logs can be enormous, spread across multiple files, and cluttered with extraneous details. Also, often errors can not be spotted without examining logs from multiple executions side-by-side. These issues are exacerbated as the scale of a system grows.

Figure 1 presents a log snippet from a hypothetical online shopping system. The site has a bug which is captured in the log. Unfortunately this snippet is plagued by some of the above challenges of log analysis, making the bug difficult to spot.

```
74.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:25:33] "GET HTTP/1.1 /invalid-coupon.php"
74.15.155.103 [06/Jan/2011:07:27:05] "GET HTTP/1.1 /valid-coupon.php"
74.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"
74.15.155.103 [06/Jan/2011:07:28:14] "GET HTTP/1.1 /reduce-price.php"
74.15.155.199 [06/Jan/2011:07:29:02] "GET HTTP/1.1 /get-credit-card.php"
13.15.232.201 [06/Jan/2011:07:30:22] "GET HTTP/1.1 /reduce-price.php"
74.15.155.103 [06/Jan/2011:07:30:55] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:31:17] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:31:20] "GET HTTP/1.1 /get-credit-card.php"
74.15.155.103 [06/Jan/2011:07:31:44] "GET HTTP/1.1 /get-credit-card.php"
```

Figure 1: Apache log snippet from an online shopping cart session - can you find the bug?[1]

In this section we first describe Synoptic, a tool designed to help developers by inferring models of systems from their execution logs. We then introduce InvariMint, a tool for constructing similar models using a new formal languages approach.

1.1 Synoptic

Synoptic [1] simplifies and improves the process of log analysis by generating a concise model that accurately captures important properties of the process that generated the log. The algorithm, illustrated in Figure 2, starts with a small initial model and performs stepwise refinement operations until the model satisfies some basic properties inferred from the input logs. Finally Synoptic performs coarsening operations to reduce the size of the model if possible without violating any log properties.

In addition to execution logs, Synoptic takes as input user-generated regular expressions that specify how to parse relevant behavior from the logs as well as how to partition the input logs into individual execution traces. The final model is an NFA that accepts traces satisfying each of the properties inferred from the log. This includes all of the input traces.

[1] The user with IP address 13.15.232.201 reduces the shopping cart price despite applying an invalid coupon.

1.1.1 Invariants

The mined properties are invariants that capture temporal relationships between user-specified events in the log. Synoptic mines three types of invariants [1]:

- **a Never Followed by b**: Whenever the event type *a* appears, the event type *b* never appears later in the same trace.
- **a Always Followed by b**: Whenever the event type *a* appears, the event type *b* always appears later in the same trace.
- **a Always Precedes b**: Whenever the event type *b* appears, the event type *a* always appears before *b* in the same trace.

These have been shown to capture the most commonly used patterns in formal specification [3].

1.1.2 Model Inference

Synoptic begins with an initial model in which every instance of event type *a* is merged into a single partition and there exists an edge between the partitions for *a* and *b* if an event *b* ever immediately followed an event *a* in any input trace. The model is then refined by splitting partitions until it satisfies each of the mined invariants.

Splitting a partition involves dividing that partition's event instances into two sets that become distinct partitions of the same event type. Refinement is a costly procedure because on each iteration Synoptic enumerates counter example paths through the model for each unsatisfied invariant. From these, one partition is selected to be divided such that at least one counter example is eliminated. To make this decision, Synotic refers to the input logs to track the source of event instances and to ensure that every input trace is accepted by the final model.

After the model satisfies each invariant, there is an additional coarsening step which seeks to minimize the model by merging partitions without un-satisfying any invariants.

The final model is an NFA state machine that accepts possible execution sequences of the system inferred from the input logs, including all of the input traces. Developers report finding it easy to find buggy behavior using these models.[1].

We next present InvariMint, which employs a novel technique for generating similar graphical log summaries.

1.2 A formal languages perspective

Another way to frame Synoptic is in terms of the language of traces accepted by the final model. All Synoptic models, including the intermediate ones, accept the input traces. Further, by construction, the final Synoptic model accepts only those traces that satisfy the mined invariants.

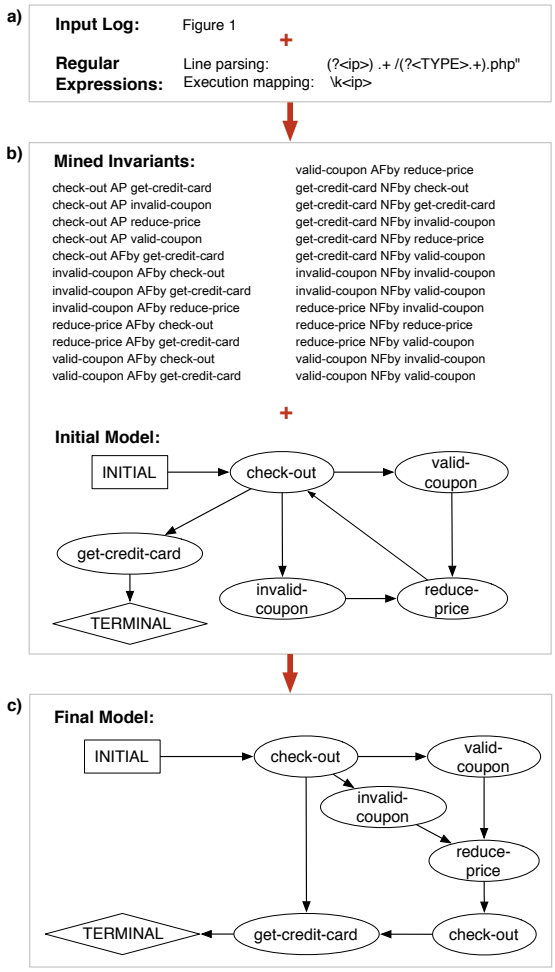


Figure 2: The Synoptic process for the log snippets in Figure 1. The user-provided regular expressions, (a), tell Synoptic how to parse interesting events from the input log and how to divide the log into individual traces of execution. Here traces are distinguished by IP address and Synoptic models client behavior during the shopping cart session. Starting from a compact initial model and the set of invariants mined from the input logs, (b), Synoptic infers a model of the system by iteratively satisfying each unsatisfied invariant (e.g. valid-coupon Never Followed by valid-coupon). The final model, (c), accepts only traces satisfying *all* of the invariants.

The language acceptance perspective motivates a new approach for deriving Synoptic–style models. In this approach, we generate a minimal deterministic finite automaton (DFA) that accepts the intersection of the languages accepted by DFAs corresponding to each log invariant.

To evaluate this technique, we built InvariMint. InvariMint, like Synoptic, accepts as input execution logs and regular expressions, and mines temporal invariants. Rather than a series of refinement and coarsening steps, however, InvariMint replaces the model inference engine in Synoptic with a series of DFA operations.

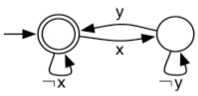
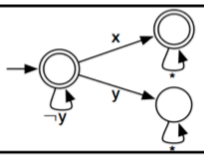
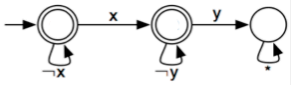
Invariant Regular Expression	Invariant DFA
X always followed by Y ([^X]|(X[^Y]*Y))*	
X always precedes Y [^Y]*(X.*)?	
X never followed by Y [^X]*(X[^Y]*)?	

Figure 3: The regular expressions used to translate each invariant type into an invariant DFA.

First InvariMint creates a DFA for each invariant. Then InvariMint intersects these DFAs to generate a model that accepts only traces satisfying all of the invariants. This model can be efficiently reduced to the most compact representation possible that preserves the language of the intersected model by applying DFA minimization.

InvariMint processes the input logs only once to mine invariants. By using an inference algorithm that avoids referencing the input logs, InvariMint generates minimal models more efficiently than Synoptic.

2 Motivation

In addition to efficiency, the formal languages perspective is poised to improve upon a number of other Synoptic shortcomings.

First, Synoptic itself is not very flexible. Built into many places of its implementation are assumptions about the three explicitly mined invariants described in Section 1.1. Adding a new invariant would require new invariant miner, model checker, and counter-example generator code. So while it is certainly possible to incorporate new invariants, there are a large number of potentially useful invariants and each individually would require substantial effort to implement.

Second, though intended for developers, Synoptic is challenging for users to understand as the aggregate process is complex. For the final model to be most helpful, we believe that users would benefit from understanding the process that generated the model.

Finally, there is not a known optimal way to choose the order in which invariants are satisfied during Synoptic's refinement phase. This problem introduces nondeterminism into the process, as the order in which invariants are satisfied affects the final model and can cause non-optimal refinement steps. Synoptic is thus not guaranteed to find a globally minimal model satisfying all invariants, and will generate an arbitrary locally minimal model. Because smaller models are easier to analyze, the global minimum is preferred.

As we will soon see, InvariMint addresses each of these deficiencies. But first, a more formal treatment of its approach.

3 Formalizing the technique

This section defines the formalisms used by InvariMint, explains the process used to construct a model equivalent to Synoptic's initial model, and provides implementation details describing how InvariMint generates a final model.

3.1 Definitions

Definition 1 (Deterministic Finite Automaton)**.** A finite state machine that accepts or rejects finite strings of symbols. The strings for DFAs described in this paper are traces of executions using log events as symbols. Each DFA is defined by a finite set of states S, a finite alphabet of log event types Σ, a transition function $\delta : S \times \Sigma \to S$, an initial state $I \in S$, and a set of accept states $F \subseteq S$.

Definition 2 (DFA Language)**.** For a DFA D let $L(D)$ denote the language accepted by D. This describes the set of traces accepted by D.

Definition 3 (Initial Model)**.** For a set of input traces T, consider the initial model M_i to be exactly the initial model used by Synoptic as described in Section 1.1. By construction, $L(M_i) \supseteq T$, and M_i is deterministic (as there is exactly one node per event type).

Definition 4 (Invariant DFA)**.** Let $i_1, \ldots, i_m$ be the set of invariants true over T. For each invariant i_j, let I_j be the corresponding DFA. Figure 3 illustrates how we translate each type of mined invariant into a corresponding DFA.

Definition 5 (DFA Intersection)**.** DFA intersection combines two DFAs A_1 and A_2 such that the resulting DFA A will accept a trace if and only if it is accepted by both A_1 and A_2.

Let $A_1 = (S_1, \Sigma, \delta_1, I_1, F_1)$ and $A_2 = (S_2, \Sigma, \delta_2, I_2, F_2)$ be two DFAs. We define the intersection A of A_1 and A_2, written $A_1 \cap A_2$, as the quituple

$$(S, \Sigma, \delta, I, F) := (S_1 \times S_2, \Sigma, \delta, I_1 \times I_2, F_1 \times F_2),$$

where δ is a function given by

$$\delta((s_1, s_2), \alpha) = \delta_1(s_1, \alpha) \times \delta_2(s_2, \alpha).$$

A can be thought of as a machine that runs A_1 and A_2 simultaneously. A symbol α being fed into A at start state $(q_1, q_2) \in I$ is the same as A_1 reading α at state q_1 and A_2 reading α at state q_2. The set of all possible next states for the configuration $((s_1, s_2), \alpha)$ in A is the same as the set of all possible combinations (t_1, t_2), where t_1 is a next state for the configuration (s_1, α) in A_1 and t_2 is a next state for the configuration (s_2, α) in A_2.

Definition 6 (DFA Minimization)**.** Hopcroft's DFA minimization [4] reduces a DFA, A, to the smallest possible DFA preserving the language of A. To do this, states in A that are behaviorally equivalent, or take the same action on all inputs, are grouped into a set that corresponds to a single state in the final minimized model.

3.2 Constructing the initial model

The Synoptic model is not equivalent to the intersection of all the mined invariants. Synoptic's initial model is constructed by merging the input traces in such a way that if an event x is ever immediately followed by an event y in some input trace then there is an edge between x and y in the model. Likewise if x is never immediately followed by y across all input traces then there is no edge between x and y in the initial model.

The initial model thus encodes a set of "can/cannot be immediately followed by" properties that are not explicitly expressed with Synoptic's mined invariants but are instead the result of using the Synoptic process. To capture those implicit properties, InvariMint expresses those properties as invariants as well. For two event types x and y, either x can be immediately followed by y, written $x\ CIFby\ y$, or x is never immediately followed by y, written $x\ NIFby\ y$.

$NIFby$ and the mined invariants provide truth values that apply to every trace whereas $CIFby$ is a special kind of invariant that describes a global property of *all* input traces. $x\ CIFby\ y$ means that in some traces it is possible for x to be immediately followed by y but says nothing about whether this will be the case in an arbitrary trace. These cannot be translated into useful DFA invariants: beginning with an overly general model, InvariMint generates a more descriptive model by intersecting the model with invariants that constrain the behavior of each individual trace.

However for two event types x and y, either $x\ CIFby\ y$ or $x\ NIFby\ y$ will be mined from the input traces. Because of this relationship, the intersection of all $NIFby$ invariants is sufficient for encoding all can/cannot be immediately followed invariants. $x\ NIFby\ y$ is translated to a DFA using the regular expression $([\hat{}x]|xx^*[\hat{}xy])^*x^*$.

We must also introduce an $Initial...Terminal$ invariant describing the property that every trace must begin with a synthetic initial event and terminate with a synthetic terminal event. This is necessary for InvariMint to fully express the notion of traces with clearly defined start and end points that is implicit in Synoptic models. The initial and terminal events are added to the input traces used by the invariant miner. This ensures that the temporal relationships between these synthetic events and all other log events are captured in the set of mined invariants.

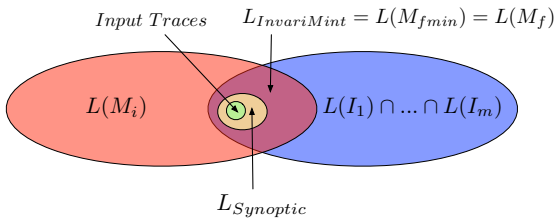


Figure 4: This diagram illustrates the relationships between languages accepted by various models of the input traces. $L(M_i)$ is the language of the initial model which includes all of the input traces and is constrained only by the $NIFby$ invariants. $L(I_1) \cap \ldots \cap L(I_m)$ is the language of the intersected mined invariants. By design, $L(I_1)$ accepts all of the input traces. $L_{Synoptic}$ also accepts all of the input traces and is constrained by both the mined invariants and M_i. Since Synoptic is non-deterministic, $L_{Synoptic}$ may vary given the same inputs but always respects the above constraints. $L_{InvariMint}$ is precisely the intersection of $L(M_i)$ and $L(I_1) \cap \ldots \cap L(I_m)$.

The initial InvariMint model is thus defined as the intersection of languages accepted by these invariants. That is:

$$M_i = \text{NIFby}_1 \cap ... \cap \text{NIFby}_n \cap Initial...Terminal$$

The language of InvariMint's initial model is equivalent to the language of Synoptic's corresponding initial model.

3.3 Constructing the final model

Define the DFA M_f as the intersection $M_i \cap I_1 \cap \ldots \cap I_m$ where M_i is the initial model and $I_1, \ldots, I_m$ correspond to the set of mined invariants defined in Section 1.1. By using the same invariants as Synoptic, we can both quantitatively and qualitatively evaluate differences between models produced by the two techniques.

The corresponding language $L(M_f)$ consists of all traces that satisfy all of the mined and immediate invariants. This language is generative — it includes T, the input traces, as well as all possible stitchings of traces that satisfy the invariants. These stitchings are possible traces allowed by the invariants but not yet observed in any input log. Such traces are called synthetic traces, and exist in both Synoptic and InvariMint models. These traces predict likely future system behavior.

Having formed M_f we can apply Hopcroft's classic DFA minimization algorithm to M_f to derive M_{fmin}. This algorithm guarantees that M_{fmin} is the smallest possible

DFA that accepts the same language as M_f. Figure 4 summarizes the above discussion, which captures the relationships between the languages accepted by M_{fmin}, M_f, and the other DFAs.

3.4 Implementation

InvariMint uses the same parsing and invariant mining code used by Synoptic as well as additional mining code to extract the immediate invariants.

The state machines manipulated by InvariMint are implemented as a layer above the dk brics automata library [6]. This library provides regular expression parsing for converting invariants to DFAs, as well as intersection and minimization implementations of the algorithms described in Section 3.1. Our implementation provides an abstraction between the unicode alphabet used by dk brics and the EventTypes mined from the input logs.

4 Evaluation

In practice, InvariMint not only infers useful summary models of the system responsible for the input logs, but does so while addressing each of the Synoptic limitations described in Section 2.

To evaluate the success of the formal languages approach, we also measure the relative efficiency of the two tools and compare the final models generated by InvariMint with those generated by Synoptic.

4.1 Addressing Synoptic limitations

First, it is easy to add new types of invariants to InvariMint as long as the invariants can be expressed as DFAs. This is how we recreated Synoptic's initial model. This feature can also be used to more closely model known features of systems. As one example, consider some program in which an event x must occur exactly twice before some other event y. This would be a trivial addition to InvariMint, requiring only some additional mining code and a regular expression for translating that temporal relationship to a DFA. The same would be time consuming to implement in Synoptic.

Second, InvariMint is much simpler to explain than Synoptic. Whereas Synoptic uses highly specialized algorithms, DFA intersection and minimization are both common, widely understood techniques for manipulating state machines.

Third, InvariMint deterministically generates a globally minimal model that satisfies all mined invariants. In contrast to Synoptic's refinement phase, the order in which InvariMint intersects invariants has no affect on the final model.

Finally, by referencing the input logs only once to mine invariants rather than continually while constructing the final model, InvariMint scales better than Synoptic on large logs.

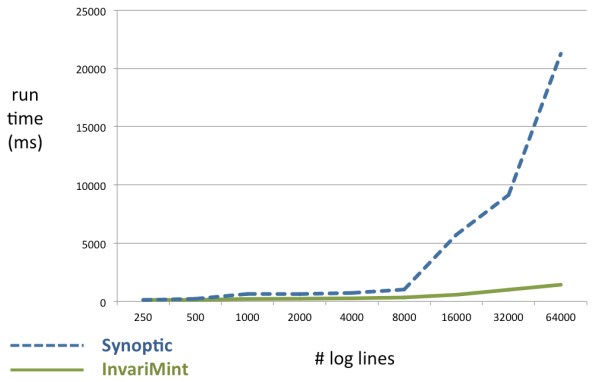


Figure 5: Run time of Synoptic versus InvariMint as the size of the input log increased.

4.2 Performance

InvariMint offers better scalability than Synoptic with respect to the size of the input log because the algorithm avoids repeated references to the log while inferring the final model. We measured scalability with respect to the size of the input log by timing both Synoptic and InvariMint on logs of varying size from the Reverse Traceroute system [5] which determines return paths for packets on the Internet. Figure 5 plots the run time in milliseconds of both Synoptic and InvariMint on input logs ranging from 250 to 64,000 lines. Recorded times are the average of 10 executions measured after 10 warm-up runs.

As the size of the log increases, InvariMint is far more efficient than Synoptic. Eventually parsing the input log and mining invariants dominates InvariMint's run time as opposed to the time taken to infer the final model.

These experiments only confirm that InvariMint is efficient with respect to the size of the input log. In the future we hope to measure InvariMint's scalability with respect to the number of log invariants. For all of the Reverse Traceroute logs, the number of invariants (both mined and implicit) is lower than 200.

4.3 Model comparisons

Despite satisfying the same invariants and accepting all of the input traces, InvariMint and Synoptic do not generate identical models. Exploring these differences is useful for thinking about discrepancies between the techniques and is important because the differences may make it harder or easier for developers to use the models.

4.3.1 Union of Synoptic models

Synoptic generates a final model that accepts all of the input traces as well as some synthetic traces satisfying all

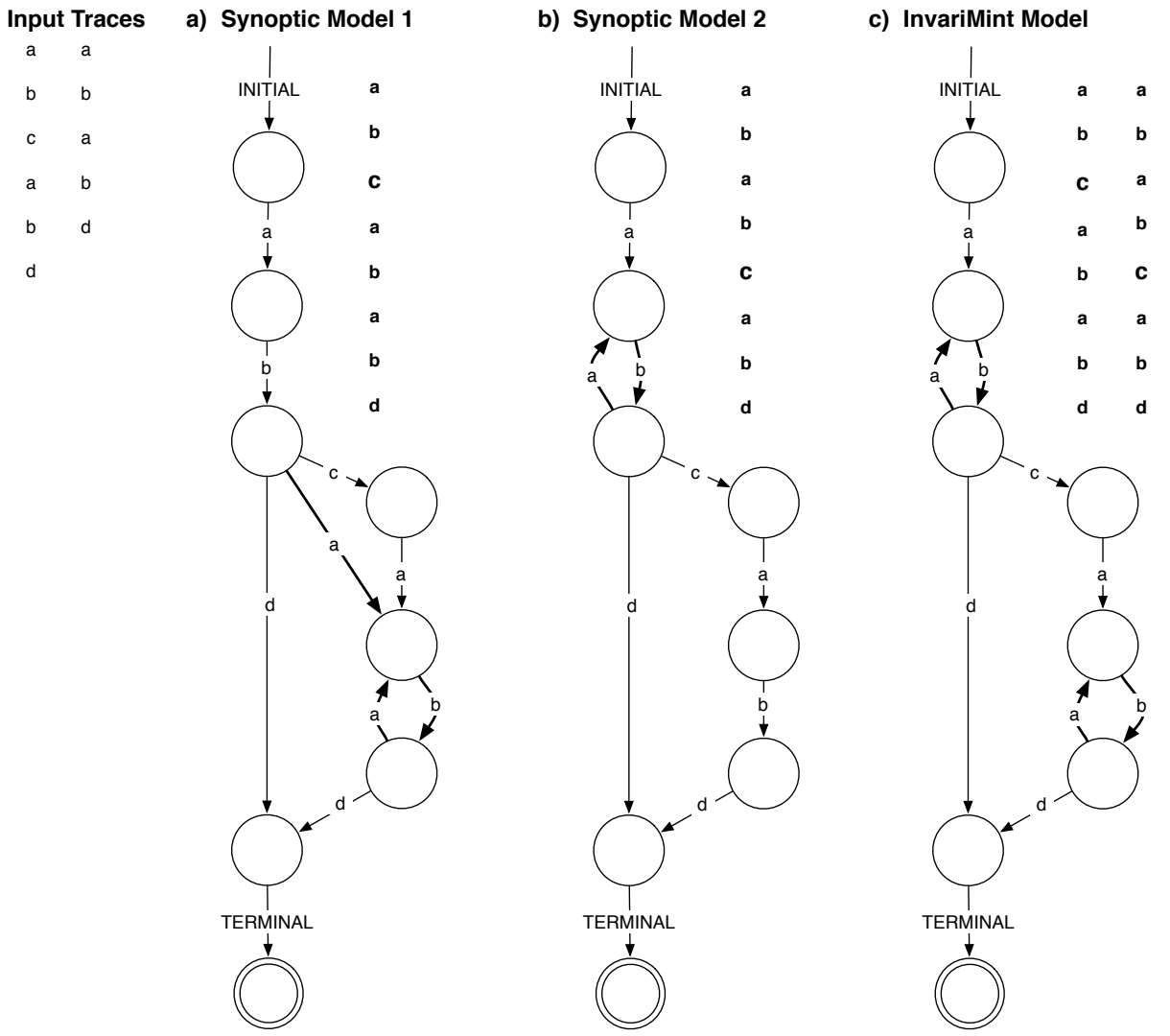


Figure 6: Models a and b are possible final Synoptic models given the provided input traces. Next to each model is an example of a synthetic trace accepted by that model but not the other. The corresponding InvariMint model, c, accepts the union of synthetic traces accepted by both Synoptic models.

of the mined and immediate invariants. Because Synoptic is non-deterministic, the set of synthetic traces accepted by the final model may vary depending on the order in which invariants were satisfied during refinement. Figure 6 illustrates an example of this: there are two possible final Synoptic models satisfying all log invariants for the given input traces, and these models accept a different set of synthetic traces.

This can be problematic for users. Consider the developer who finds a bug in their system by examining a Synoptic model. After modifying their system, the developer provides new input logs to Synoptic to verify that the bug is fixed. Though the buggy behavior is gone, it is possible that some unrelated part of the model has changed, leaving the developer to wonder whether they unintentionally introduced additional modifications to the system.

InvariMint solves this issue by generating models deterministically. Though not formally verified, we hypothesize that the final InvariMint model accepts the union of all possible synthetic traces accepted by any of the corresponding Synoptic models. This is the case for the InvariMint model in Figure 6 which accepts both sets of synthetic traces generated by different Synoptic models.

Figure 4 provides a visual illustration for the intuition behind this idea: the language accepted by InvariMint is precisely the intersection of the initial model and each of the mined invariants. Synoptic accepts a language that is some subset of that intersection.

The union property is poised to have a number of benefits. First, it provides a bound for the set of possible final Synoptic models which was previously an ambigu-

ous space. Second, the additional synthetic traces may be useful for users attempting to predict the range of possible system behaviors for purposes such as generating additional test cases. In future work we hope to conduct a usability study to evaluate the utility of additional synthetic traces.

4.3.2 Spurious Edges

Assuming that the languages accepted by Synoptic models are always a subset of the language accepted by a corresponding InvariMint model, the next question is whether there are any traces accepted by an InvariMint model that are not accepted by *any* corresponding Synoptic model.

In fact, InvariMint models can accept synthetic traces that do not exist in any corresponding Synoptic model. These are caused by **spurious edges** and highlight an important distinction between the two techniques: underlying the Synoptic models are specific event instances, whereas InvariMint deals only with event types.

During refinement and coarsening, Synoptic maintains partitions containing one or more concrete instances of that partition's event type from some input trace. For each event instance a in the partition, there exists an incoming edge from some partition containing the event instance that immediately preceded that instance a and there exists an outgoing edge that leads to some partition containing the event instance that immediately followed a. Synoptic models thus have the property that every edge in the final model corresponds to two successive events in at least one input trace. In other words, every edge respects the *edge-coverage property*.

The Synoptic model accepts some synthetic traces – traces not contained within the set of input traces – but these are limited by the edge-coverage property.

InvariMint models in contrast have no notion of input traces and are concerned only with the temporal relationships between event types. Because InvariMint models do not have the edge-coverage property, they are more permissive than Synoptic models, accepting a wider range of synthetic traces.

Figure 7 illustrates an example of this. The Synoptic model allows only the set of input traces in its final model. The InvariMint model allows an additional $INITIAL - x - a - y - TERMINAL$ synthetic trace because no mined invariant prohibit it.

More concretely, because an event z immediately followed an event a in at least one input trace, the $a\ NIFby\ z$ invariant was not mined during construction of the initial model and every instance of an event a in the InvariMint model can be immediately followed by an event z unless prohibited by some other invariant. The left-most event a in the InvariMint model in Figure 7, for example, is not followed by an event y even though y can immediately follow a because that would allow traces violating the

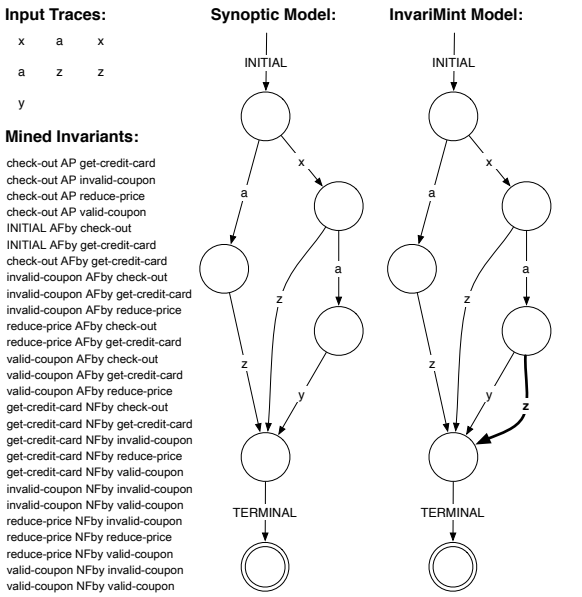


Figure 7: An abstract example of a spurious edge in an InvariMint model. The input traces and mined invariants are shown at left. Each edge in the Synoptic model (translated to an InvariMint-style DFA for easier comparison) corresponds directly to an event in some input trace. The InvariMint model is almost identical to the Synoptic model but includes an additional z edge from a to $TERMINAL$. This transition does not map to any event in the input logs so violates the edge-coverage property and is spurious. This spurious z edge is allowed by InvariMint because elsewhere an event z immediately followed an event a and none of the other invariants restrict this particular instance of z.

$x\ AP\ y$ invariant.

The z edge in the InvariMint synthetic trace would map to an a–z edge in the corresponding state-based model used by Synoptic. There does not exist any input trace that would traverse the z edge in this model, and thus the edge is an example of a spurious edge.

4.3.3 Removing spurious edges

To best approximate Synoptic, InvariMint models should not contain any spurious edges. However, we have not yet developed an adequate way to remove these edges from InvariMint models.

One plausible way to remove spurious edges is to traverse the input traces during one additional post-processing step and remove any edges in the final InvariMint model violating the edge-coverage property.

A problem with this strategy is that Synoptic generates NFAs with the edge-coverage property whereas InvariMint generates DFAs. Translating Synoptic models to DFAs generates models that accept the same language as the original model but creates a model that may violate the edge-coverage property.

This is due to additional edges and states introduced by the NFA-to-DFA translation. Thus InvariMint models can contain edges that both violate the edge-coverage property and are spurious, but can also contain edges that violate the edge coverage property but if removed would restrict the language of the model in ways that the corresponding Synoptic model does not.

For example, applying the post-processing operation to remove spurious edges on the InvariMint model in Figure 6 would remove the edges that allow the a-b-c-a-b-a-b-d synthetic trace. The resulting language would no longer be the union of languages accepted by all Synoptic models.

Spurious edges reveal a loss of context in InvariMint models. Synoptic models are more nuanced, preserving greater trace-specific information. Given a Synoptic model, it is possible to query specific edges to reveal precisely which input traces allowed that transition between events. This can be useful, for example, to pinpoint the precise executions in which a bug appeared. The mined invariants used by InvariMint do not capture any trace specific information.

Less context-sensitivity is not necessarily a bad thing: the InvariMint models are more general and ignore individual trace idiosyncrasies, leading to smaller models as more states can be merged. As the scale of the input grows, model brevity and efficiency may prove to be more valuable than trace-specific information.

5 Generalizing the Technique

Synoptic is only one of the many algorithms used to infer models from executions. These techniques are difficult to compare and nearly impossible to combine. Some are slow, and some produce models that may not be minimal.

One of the advantages of the InvariMint approach is that it provides a way to compare and combine different model inference approaches on the same terms, and does so in an efficient, deterministic way. In this section, we will first describe kTails [2], a widely used algorithm for FSM inference. We will then illustrate how kTails can be expressed by composing invariant DFAs. We argue that InvariMint offers a unifying framework for model inference methods by characterizing the existing techniques in terms of invariant DFAs.

5.1 kTails

kTails is a coarsening algorithm for inferring concise models. The algorithm begins with a fine-grained representation of a model in which each event instance (not event type) is mapped to its own partition as in Figure 8a.

kTails then iteratively merges pairs of k-equivalent partitions. These are partitions in the model that are roots of identical sub-graphs up to depth k. The resulting graphs

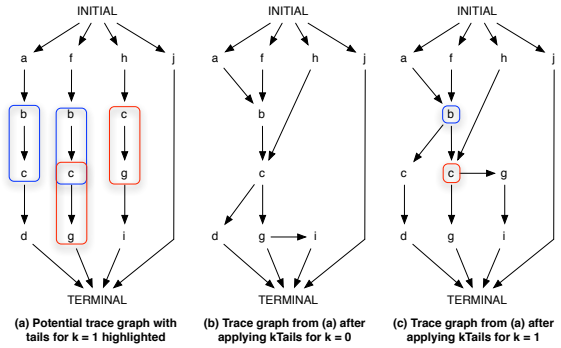


Figure 8: A sample trace graph and its corresponding models using InvariMint for kTails for k = 0 and k = 1.

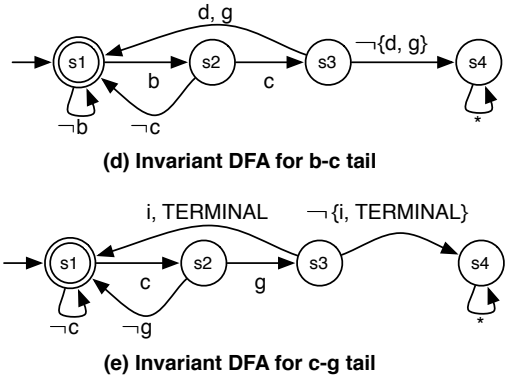


Figure 9: Invariant DFAs for the tails in Figure 8a for $k = 1$.

after applying $k = 0$ and $k = 1$ to the model in Figure 8a are shown in 8b and 8c respectively.

The initial kTails graph is analogous to a trace graph for Synoptic in which each execution trace from the input logs is a single path from the INITIAL to TERMINAL nodes. It is possible to infer Synoptic-style models by simply applying kTails to a trace graph for arbitrary values of k. Synoptic's initial model, in which all events of the same type are merged, is equivalent to running kTails on the trace graph with $k = 0$.

5.2 Representing kTails using InvariMint

When k = 0, kTails merges all event instances of the same type which is precisely equivalent to Synoptic's initial model and can be constructed with InvariMint using the *Never Immediately followed by* invariants. An example of this is shown in Figure 8b.

For $k > 0$, invariants can describe tails of length k in the model. To construct these, we mine all sub-graphs of depth $k+1$ in the initial graph that are shared by **at least two traces**. For all such tails, the set of events that can immediately follow the tail are also recorded. Each sub-graph is then translated into an invariant DFA which stipulates that if a tail is seen, it must be immediately followed by one of the next possible follow events. Figure 9

illustrates these DFAs for $k = 1$.

Using these translation mechanisms, InvariMint is able to express the key properties of kTails simply by taking the intersection of all immediate invariants and each of the tail invariants.

InvariMint offers a unifying framework for inferring models from executions by reducing other inference techniques to the set of invariants that describe the essential properties of their models. Because these invariants are all describable with formal languages, it is possible to generalize, combine, and compare existing techniques simply by adjusting the set of invariants used to construct the final model.

For example, using InvariMint a developer would be able to merge k-tails of varying length depending on the event type, and could intersect that model with whichever temporal Synoptic invariants deemed most appropriate. This flexibility provides a way for developers to better explore the space of formal specification and to quickly create customized hybrid inference algorithms.

6 Conclusion

Synoptic is a tool for turning complex executions logs into convenient summary models for analyzing systems. It infers these models by capturing key temporal properties of the input logs. InvariMint implements a new model inference engine for constructing similar models. The InvariMint technique uses a formal languages perspective to express log invariants which offers improvements over Synoptic's refinement and coarsening algorithms:

(1) Turning logs into models offers an easy-to-create and easy-to-use way to analyze systems. By using familiar DFA operations to infer these models, the InvariMint process is easy for users to understand.

(2) InvariMint is more scalable than Synoptic. The InvariMint model inference technique refers to the input traces once which makes the DFA intersection and minimization operations more efficient than refinement and coarsening.

(3) Whereas Synoptic generates models non-deterministically, InvariMint is both deterministic and captures all possible behaviors allowed by Synoptic models.

Additionally the formal language perspective is extensible: InvariMint makes it trivial to model any system behavior that can be expressed as a finite state machine. This generic approach allows InvariMint to be compatible with other inference techniques, and we believe that it is a unifying framework for generalizing, combining, and comparing those existing methods.

References

[1] BESCHASTNIKH, I., BRUN, Y., SCHNEIDER, S., SLOAN, M., AND ERNST, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Szeged, Hungary, September 7–9, 2011), pp. 267–277.

[2] BIERMANN, A. W., AND FELDMAN, J. A. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput. 21*, 6 (1972), 592–597.

[3] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 411–420.

[4] HOPCROFT, J. E. An n log n algorithm for minimizing states in a finite automaton. Tech. rep., Stanford, CA, USA, 1971.

[5] KATZ-BASSETT, E., MADHYASTHA, H. V., ADHIKARI, V. K., SCOTT, C., SHERRY, J., VAN WESEP, P., ANDERSON, T., AND KRISHNAMURTHY, A. Reverse Traceroute. In *Proc. of NSDI* (2010).

[6] MØLLER, A. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.