Learning a Context-Dependent Semantic Parser for Temporal Expression Resolution

Jesse Dodge

Computer Science & Engineering
University of Washington

June 15, 2013

Abstract

We present a semantic parsing approach for temporal expression resolution. Previous work, including current state of the art approaches, primarily use pattern matching and regular expression-like rules for grounding temporal expressions. However, the compositionality of language suggests a more linguistically principled approach.

In our system, using a hand-built lexicon and CCG parser, we first construct a base CCG parse for each temporal phrase, then build a set of candidate context-dependent logical forms from each base parse. We learn to select one parse from this set using a perceptron and features based on the logic, the phrase itself, and, critically, the full context the in which the phrase appears. Finally, we execute this logical form to a standardized date and time representation. This approach builds on previous successes using CCG for semantic parsing, and using a small featureset of linguistically motivated features, we achieve moderate accuracy on task A of TempEval3.

1 Introduction

Temporal expression resolution is the task of mapping from a Temporal phrases are ubiquitous. They appear in all kinds of text, from e-mail to newswire to Wikipedia articles. Much of the information in print and Internet media consists of text that describes events occurring at a specific point in time. For a computer to be able to comprehend this text, it will need to be able to recognize events, and reason about how they are located temporally. Building a system that can understand temporal relations in text is a first step toward a system that can understand much of the information available in news articles, scientific journals and on the Internet. Improved reasoning about times and events will enable us to build more effective systems for question answering, information extraction, and summarization. For example, a system that had extracted the two relations CEO(Steve Jobs, Apple) and CEO(Tim Cook, Apple) wouldnt be able to answer the question Who is the CEO of Apple? It would also need to understand the temporal relationship between the two relations.

In section 2 we cover previous work on this problem. Then, in section 3, we go over our representation and approach to parsing. Section 4 reveals our learning. Section 5 shows our results. Finally, section 6 talks about future work.

2 previousWork

Previous work on this task has almost entirely been rule-based. The current state of the art approaches use a combination of regular-expression matching and hand-written interpretation functions to ground temporal expressions. These hand-built approaches are very domain-specific; they have a difficult time when applied to domains other than the one they were built for. Moreover, the complexities of natural language are often lost on these systems. Nested or hierarchical expressions are easy to handle when taking a parsing approach designed to deal with these phenomena, but these types of phrases can cause more rigid rule-based systems to fail. For example, phrases such as *the third Wednesday of each month* or *the second month of last year* are easier to ground using a parsing approach. While most of the work in this area has been on deterministic systems, one notable exception is ParsingTime [1]. This system learned to parse from temporal phrases to logical forms, but is limited in that it doesn't take context into account. Their system uses a PCFG to parse the phrases to a logical form, then executes the form to get a result. However, their approach only looks at the temporal phrase itsef; it doesn't have any signal from the context in which the phrase appeared, such as verb tense or the result of parsing previously uttered phrases. Previous work has shown that using CCG for semantic parsing can be successful. Zettlemoyer and Collins TODO: CITE used CCG to map from text queries to logical forms for a number of datasets, often achieving state-of-the-art results. Often in previous work the parse itself is used as a query to a database, which can be evaluated directly. In this work, however, the parse is latent; we execute the parse using a deterministic algorithm, as our signal comes from the fully ground date or duration. A number of approaches have also used a learning approach for the lexicon used to parse. In this work, however, we are using a hand-built lexicon, for greater coverage and to enable future work in building a joint model of detection and parsing.

This work builds upon previous work using CCG for semantic parsing.

3 Representation

In this section, we first define our type system, then discuss our lexicon, then describe how we parse. Our system takes a three-step parsing approach. First, we use a CCG parser to build a set base parses for each temporal phrase in isolation (where each base parse is referred to a a context-independent parse). Then, for each context-independent parse, we deterministically build five context-dependent parses. Once we select a single parse from the set of candidate context-depndent parses (which we discuss in the section titled Learning), we execute the parse to ground to a final representation.

3.1 Types within CCG

We define five

Definition 1 (Range)**.** A range is a period between two times. For example, *June 13th, 2013*, *today*, and *1987* all ground to ranges.

Definition 2 (Sequence)**.** A sequence represents an infinite sequence of ranges. For example, *each Thursday* grounds to a range. The duration between each range in a sequence is the same. In the example *each Thursday*, there is a week between each range. Sequences are represented as under-specified ranges. If we were interested in grounding the phrase *June 13th, 2013*, we could first ground *June 13th* to the sequence of all June 13ths, which we could represent as XXXX-6-13, i.e. June 13th of an unspecified year.

Definition 3 (Duration)**.** A duration is a period of time with no specified start or end dates. For example, *two months*, *three years*, and *one day* all ground to durations of time.

Definition 4 (Number)**.** Numbers only appear in logical forms, never in fully executed output. Numbers are necessary when grounding durations, such as *4 years*, *1 hour*, or *3 weeks*.

Figure 1: Above we can see three example parses. Each of these parses represents a base parse for the given temporal phrase. We can see how the base parse is built from the individual categories drawn from the lexicon. In the first example, we are parsing the phrase *third quarter*. The entries in the lexicon that are being used in this parse are $\lambda x.nth(x,3)$ and *quarter*, which correspond to *third* and *quarter*, respectively. These combine to create the base parse for the first phrase, $nth(quarter,3)$. The second example is very similar to the frist. However, the third example first does forward composition (combining $\lambda x * (x,1)$ with *year*) then does backward composition (combining $\lambda x. * (x,-1)$ with $*(year,1)$). The first context-independent parse, $nth(quarter,3)$ represents the sequence of all third quarters, which needs additional logic to be correctly ground to a single third quarter. The second parse, $*(weeks,5)$, can be fully ground to a duration of five weeks without needing an additional context step. The third parse, $*(*(year,1),-1)$, can't actually be ground to a date at all without additional contextual information.

Definition 5 (Functional Types)**.** There are a number of functional types, all of arity less than or equal to two. They are fully outlined in a table in section TODO PUT SECTION HERE

3.2 Lexicon

A CCG is defined by a lexicon and set of combinators. In this work, we use standard forward and backward application with a hand-built lexicon to build our base parses.

3.3 Pasing Using CCG

First, for each temporal phrase in our dataset, we use a CKY parser to get a set of CCG logical forms. These are built only from the phrase itself, and don't take any additional information into account. Some ranges can be fully ground without looking at the context in which they appear, such as *June 6th, 2013*. Similarly, durations such as *five weeks* or *a year* don't need to look at surrounding context to be executed to a final form. However, the majority of phrases do require some context to ground fully. Phrases such as *Friday* are parsed to logic that represents sequences, in this case the sequence of all Fridays. To ground phrases such as those to a single range, we need to look at context.

3.4 Building Context-Dependent Parse

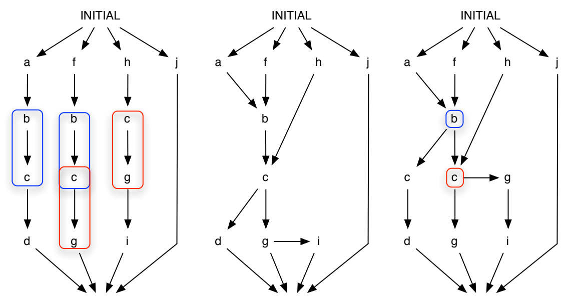
From each context-independent logical form we get from the base CCG parser, we use a deterministic process to build a set of five context-dependent logical forms. These represent the five possible groundings of each of our base parses. For fully-sepcified ranges and for durations, we don't actually need to change the logical form, so the frist of the the context-dependent logical forms is an unchanged version of the context-independent logical form. Sequences often represent an infinite number of ranges, such as $nth(quarter, 3)$. We can ground these sequences to one of three ranges, either the range before the document was published, the range after the document was published, or the range during which the document was published. Finally, for a class of temporal phrases that refer to a previously uttered temporal phrase, such as *a year earlier*, we build a logical form that grounds the current phrase based on the previous one. An example is shown in the figure above.

3.5 Building Context-Dependent Parse

4 Generalizing the Technique

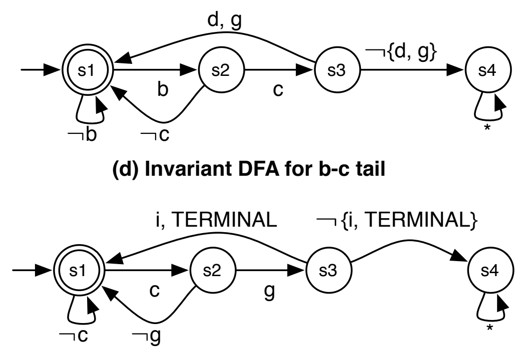
Synoptic is only one of the many algorithms used to infer models from executions. These techniques are difficult to compare and nearly impossible to combine. Some are slow, and some produce models that may not be minimal.

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 [3], 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.



(a) Potential trace graph with tails for k = 1 highlighted (b) Trace graph from (a) after applying kTails for k = 0 (c) Trace graph from (a) after applying kTails for k = 1

Figure 2: A sample trace graph and its corresponding models using InvariMint for kTails for k = 0 and k = 1.



(d) Invariant DFA for b-c tail

(e) Invariant DFA for c-g tail

Figure 3: Invariant DFAs for the tails in Figure 2a for $k = 1$.

4.1 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 2a.

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 after applying $k = 0$ and $k = 1$ to the model in Figure 2a are shown in 2b and 2c 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 $k = 0$.

4.2 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 2b.

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 3 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.

5 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 **??**.

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.

5.1 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.

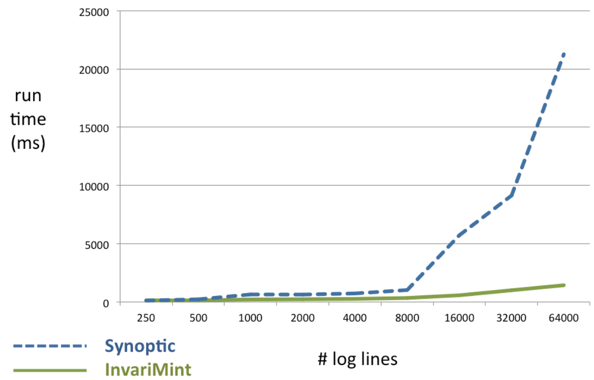


Figure 4: Run time of Synoptic versus InvariMint as the size of the input log increased.

5.2 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 [6] which determines return paths for packets on the Internet. Figure 4 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.

5.3 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.

5.3.1 Union of Synoptic models

Synoptic generates a final model that accepts all of the input traces as well as some synthetic traces satisfying all 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 5 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 5 which accepts both sets of synthetic traces generated by different Synoptic models.

Figure 1 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 ambiguous 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.

5.3.2 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 6 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 6, for example, is not followed by an event y even though y can immediately follow a because that would allow traces violating the $x\ AP\ y$ 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.

5.3.3 Removing spurious edges

To best approximate Synoptic, InvariMint models should not contain any spurious edges. However, we have not yet

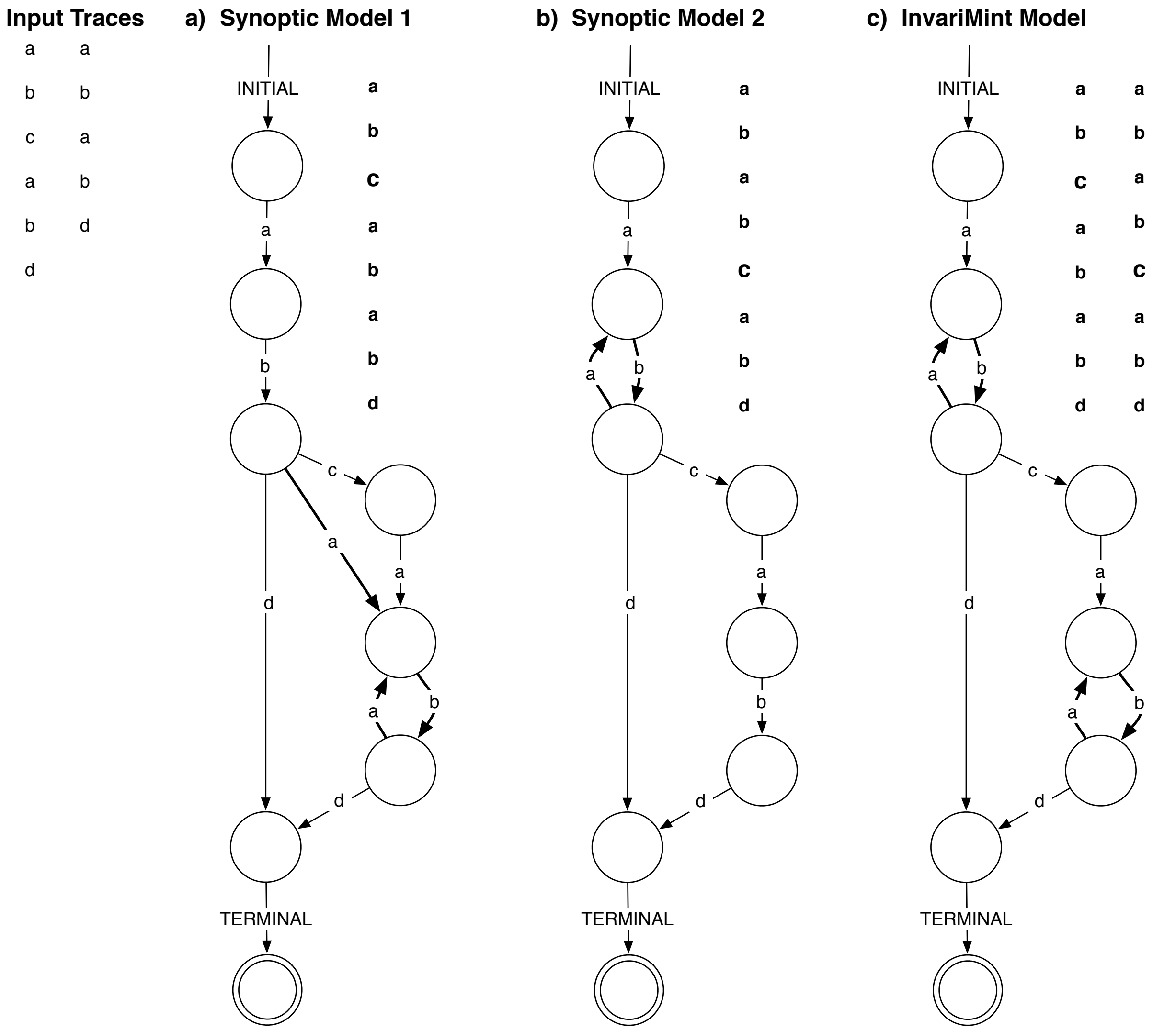


Figure 5: Models a and b are possible final Synoptic models given the provided input traces. Next to each model is an example of a synthetic trace accepted by that model but not the other. The corresponding InvariMint model, c, accepts the union of synthetic traces accepted by both Synoptic models.

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 5 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

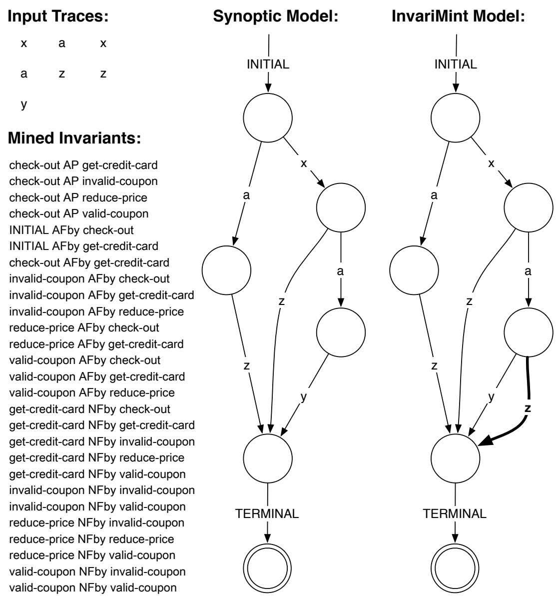


Figure 6: An abstract example of a spurious edge in an InvariMint model. The input traces and mined invariants are shown at left. Each edge in the Synoptic model (translated to an InvariMint-style DFA for easier comparison) corresponds directly to an event in some input trace. The InvariMint model is 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.

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.

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] Angeli, G., Manning, C. D., and Jurafsky, D. Parsing time: Learning to interpret time expressions. In *North American Chapter of the Association for Computational Linguistics (NAACL)* (2012).

[2] 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.

[3] 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.

[4] 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.

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

[6] 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).

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