Traceur: Inferring Variable Control Flow Using Synoptic With Multiple Relation Types

Timothy Vega

Computer Science & Engineering
University of Washington

December 17, 2012

1 Abstract

Debugging systems from logged behavior is time consuming and obtuse. Synoptic simplifies the task of inspecting a log by inferring and presenting the developer with an event-based model that represents the logged behavior. Unfortunately, Synoptic reasons about and represents the logged events exclusively in the time domain — two events are related iff one of the events preceded the other event. We developed a theoretical framework for extending Synoptic inference to models that capture non-temporal relations between events. We focused on adding a single, arbitrary dependent relation on top of the independent, temporal relation. The construction generalizes to a variable number of dependent relations. We motivate the problem, and explain the proposed solution.

2 Introduction

2.1 Synoptic

The purpose of Synoptic is to transform system logs into finite state machine models. System logs are generally difficult for humans to parse and understand, and they are even harder to use to identify the presence of bugs. On the other hand, finite state machine models are easy for humans to parse and understand. Given an expected model and an inferred model, a developer can identify either a flaw in the expected model or a bug in the system.

To accomplish this task, Synoptic does the following (at a high level):

- Logs are modeled as traces
- Invariants are mined from the traces
- Traces are transformed into a partition graph
- Invariants are used to refine the partition graph through model checking

Figure 1: Synoptic

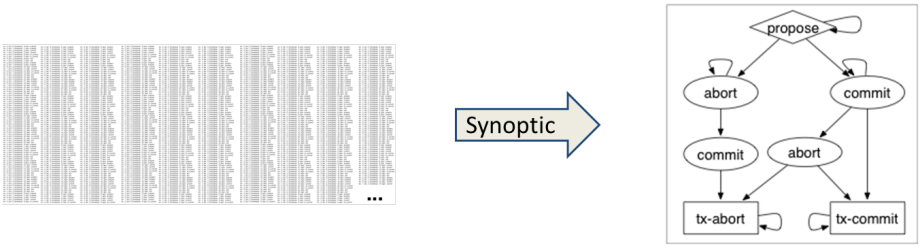


Figure 2: Modeling Logs As Traces

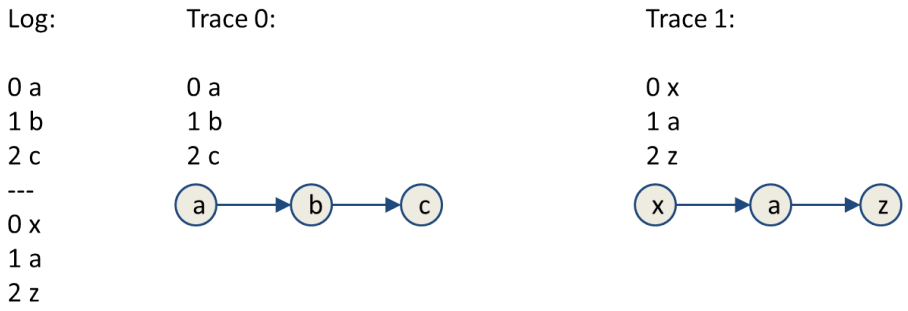
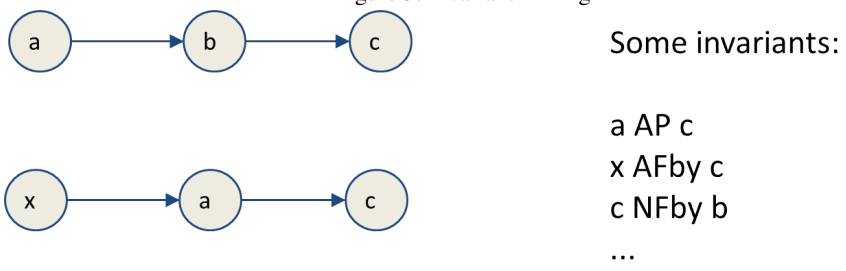


Figure 3: Invariant Mining



Unfortunately, Synoptic assumes that all logged events are only related to each other through time. With multiple relations, the system can classify arbitrary event realtionships as long as the information is available.

2.1.1 Modeling Logs As Traces

The first step in the process is to parse input logs. System logs consiste of one or more totally ordered sets of events. These sets are denoted as traces and are modeled in the system as chains. Chains are linear, directed acyclic graphs. Vertices in a chain are logged events and edges connect a logged event to its immediate temporal successor. The transitive closure of the edges in a chain is the toal ordering of the events in the chain.

We hereafter use traces and chains to refer to the same concept.

2.1.2 Invariant Mining

The second step in the process is to mine invariants from the traces. The system mines the following three invariant types:

- x **Always Followed by** y ($x \rightarrow y$). Whenever the event type x appears, the event type y always appears later in the same trace.
- x **Never Followed by** y ($x \not\rightarrow y$). Whenver the event type x appears, the event type y never appears later in the same trace.
- x **Always Precedes** y ($x \leftarrow y$). Whenver the event type y appears, the event type x always appears before y in the same trace.

2.1.3 Generating The Partition Graph

The third step in the process is to abstract the traces into a partition graph. Trace events are partitioned into sets, and each set is represented as anode in the partition graph. Edges are directly translated from the trace graph to the partition graph. In other words, if an edge exists in at race from a node of type x to a node of type y, then a corresponding edge exists in the partition graph from the x partition node to the y partition node.

Figure 4: Generating The Partition Graph

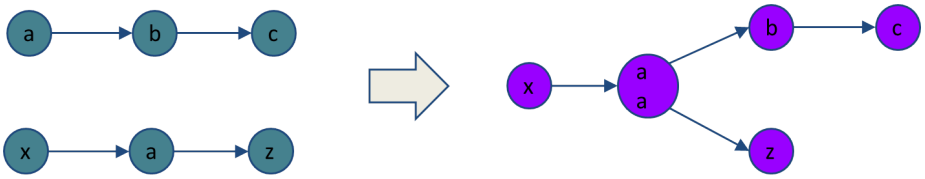
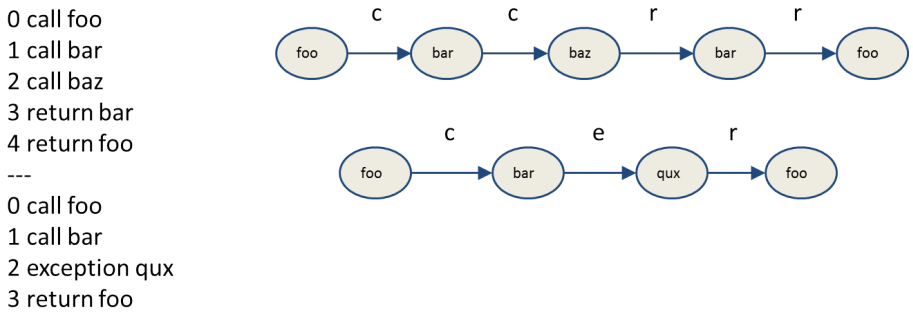


Figure 5: Motivating Example



2.1.4 Model Checking

The fourth step in the process is model checking. Upon construction, the partition graph is the most compact or abstract model that can be generated from the logged traces. The opposite end of the spectrum is the trace graph which is the least compact model. The trace graph overfist the input traces and makes no generalizations. Synoptic aims to present the user a model that strikes a balance between these two extremes. The initial partition grpah over-generalizes the logs, so by a process of iterative refinement (model checking), Synoptic produces a partition graph such that all paths through the graph violate no invariants.

2.2 Motivating Example

Control flow in Java consists of method calls, returns, and exceptions. Given a log containing this information, Synoptic overlooks it and only processes temporal structure.

Without multiple relations, Synoptic cannot mine invariants on non-temporal relation types.

3 Multiple Relations Model

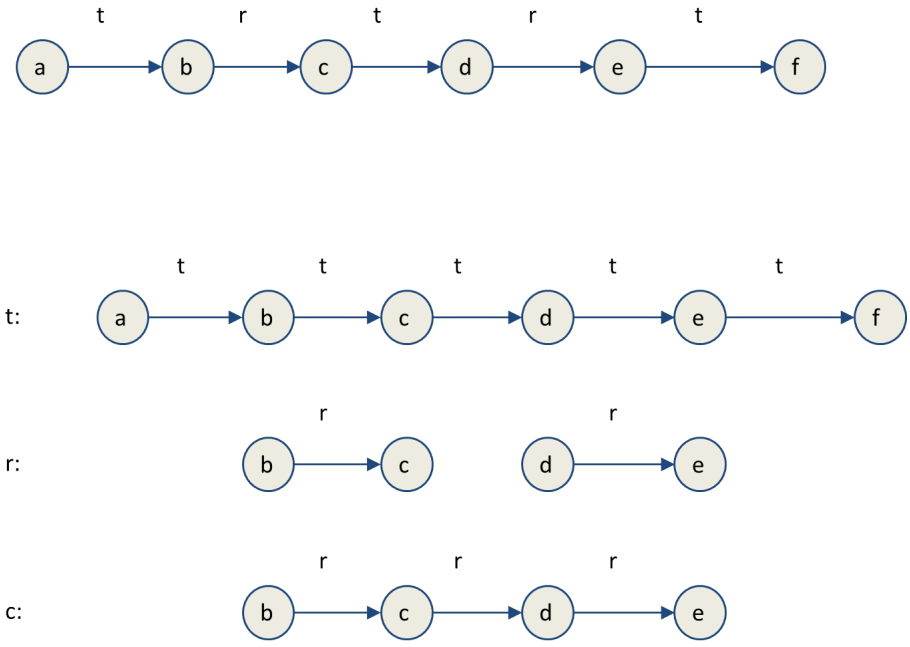
3.1 Traces

In Synoptic, all trace edges have an implicit relation type of time. In Traceur, each edge has at least one explicit relation type with a maximum of two. Each edge has at least one explicit relation type because Traceur chains are required to have a relation type that is on every edge. We hereafter denote that relation as the independent relation. Each edge can have a maximum of two edges because we allow an extra relation. These extra relations are constrained to have the same type. We hereafter denote the extra relation as the dependent relation. For all traces, independent relations have the same type, and dependent relations have the same type as well. Relation types are specified by the user input.

3.2 Invariants

Each trace contains a combination of independent and dependent type relation edges. We denote this as the base trace, and from the base trace, we filter three subtraces for multi-relation invariant mining.

Figure 6: Traces



- Independenent subtrace
  - The independent subtrace consists of every edge with an independent relation and all nodes incident to those edges.
- Dependent subtrace
  - The dependent subtrace consists of every edge with a dependent relation and all nodes incident to those edges.
- Composite subtrace
  - The composite subtrace consists of the dependent relation with additional edges, of the dependent relation type, connecting disconnected, neareast neighbors over the independent relation.

Given the subtraces, we mine indepenent, dependent, and composite invariants.

4 Formal Exposition

4.1 Definitions

Two special event types – *INITIAL* and *TERMINAL* are added internally by Synoptic to keep track of initial and terminal events in the traces.

Definition 1 (Event Types)**.** A set of event types is a finite set (alphabet) $E \supseteq \{INITIAL, TERMINAL\}$.

An event instance is a logged event and has an event type that is identified by a *tid* which is the trace that it occurred in and by an *index* which is its temporal position in that trace.

Definition 2 (Event Instance)**.** Let $tid, index \in \mathbb{N}$, and $e \in E$ be an event type. An event instance is a 3-tuple, $\hat{e}_{index} = (e, index, tid)$. We say that the event type of $\hat{e}_{index}$ is e.

Definition 3 (Trace)**.** A trace of length n with trace identifier *tid*, denoted as T_{tid}, is a set of n event instances such that the first event instance has a type *INITIAL* and the last has a type *TERMINAL*. $T_{tid} = \{(e, index, tid) | 1 < i < n\} \cup \{(INITIAL, 1, tid), (TERMINAL, n, tid)\}$

Each trace contains a set of event instances that is totally ordered by the transitive closure of the time relation, which is a strict total order.

Definition 4 (Time Relation)**.** Let $tid \in \mathbb{N}$. Let T_{tid} be a trace such that $|T_{tid}| = n$. A time relation t_{tid} is a relation over T_{tid} such that $t_{tid} = \{(\hat{e}_i, \hat{e}_{i+1}) | 1 \leq i < n, (\hat{e}_i, \hat{e}_{i+1} \in T_{tid})\}$

A chain is a totally ordered subset of a trace.

Definition 5 (Chain)**.** Let t_{tid} be a time relation. Let $j,k \in \mathbb{N}, 1 \leq j < k < n$. A chain, $c_{j,k}$, is a relation that is a subset of t_{tid} such that $c_{j,k} = \{(\hat{e}_i, \hat{e}_{i+1}) | j \leq i < k, (\hat{e}_i, \hat{e}_{i+1}) \in t_{tid}\}$

Event instance relations consist of the set of distinct and non-overlapping chains of a single relation in a trace. Event instance relations are partial orders.

Definition 6 (Event Instance Relation)**.** Let t_{tid} be a time relation. An event instance relation, r_{tid}, is a set of chains, such that $(c_{j,k} \in r_{tid}) \wedge (c_{j',k'} \in r_{tid}) \implies k < j' \vee k' < j$

The bridge pair takes two chains from an event instance relation and relates the maximum element of one with the minimum element of the other with respect to the relation. Equivalently, bridge pairs link together unordered chains in an event instance relation.

Definition 7 (Bridge Pair)**.** Let r_{tid} be an event instance relation. Let $c_{i,j}, c_{k,l} \in r_{tid}$. Let $s = supremum(c_{i,j}) \wedge t = infimum(c_{k,l})$. A bridge pair for $c_{i,j}, c_{k,l}$ is $b_{c_{i,j},c_{k,l}} = \langle s,t \rangle$.

The bridge set is the set of bridge pairs such that there is no chain that is ordered between the two elements of any bridge pair.

Definition 8 (Bridge Set)**.** Let r_{tid} be an event instance relation. The bridge set, b_{tid}, is a relation such that $b_{tid} = \{b_{c_{i,j},c_{k,l}} | (\nexists (c_{a,b} \in r_{tid}))$ such that $(j < a), (b < k)\}$

The composite relation is a totally ordered relation composed of an event instance relation and its bridge set.

Definition 9 (Composite Relation)**.** Let r_{tid} be an event instance relation. The composite relation is a strict total order, $c_{tid} = r_{tid} \cup b_{tid}$

The transitive closure of the composite relation is a strict total order.

Definition 10 (Log)**.** A log L of size k is a set of k traces. $L = \{T_1, \ldots, T_k\}$.

An event invariant is a property of a log that defines relationships between event types.

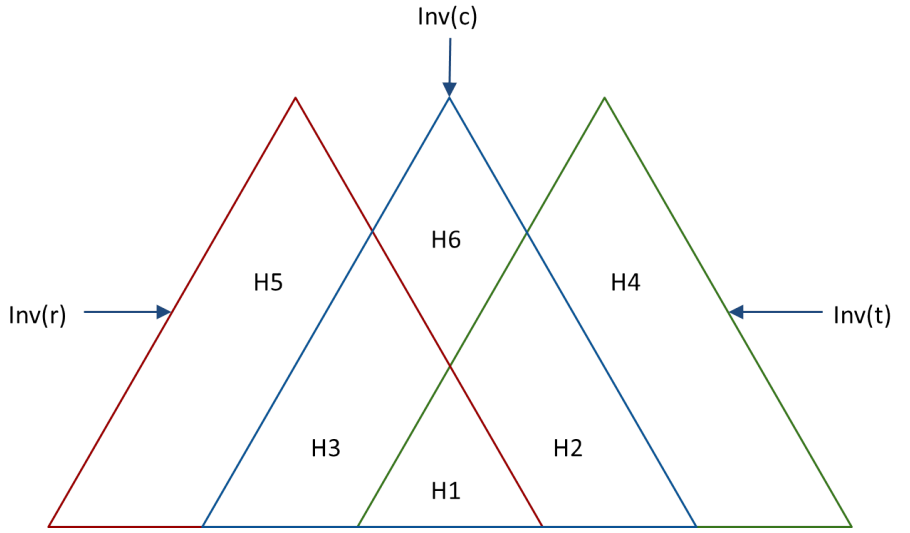
Let a and b be event types. There are three types of temporal event invariants, they are denoted as follows:

- $a \rightarrow b$: a is always followed by b
- $a \leftarrow b$: a is always preceded by b
- $a \not\rightarrow b$: a is never followed by b

Definition 11 (Event Invariant)**.** Let L be a log. Let $T_{tid} \in L$ be a trace. Let $i, j \in \mathbb{N}$. Let $\hat{a}_i, \hat{b}_j$ be event instances such that $\hat{a}_i, \hat{b}_j \in T_{tid}$. Let s be a strict total order over T_{tid}. We denote the strict total order, s, as $<_s$. The three types of invariants are as follows:

- $a \rightarrow_s b \iff (\forall T_{tid} \in L : \hat{e}_A \in T_{tid} \implies \exists \hat{e}_B \in T_{tid}$ such that $\hat{e}_A <_s \hat{e}_B)$
- $a \leftarrow_s b \iff (\forall T_{tid} \in L : \hat{e}_B \in T_{tid} \implies \exists \hat{e}_A \in T_{tid}$ such that $\hat{e}_A <_s \hat{e}_B)$
- $a \not\rightarrow_s b \iff (\forall T_{tid} \in L : \hat{e}_A \in T_{tid} \implies \nexists \hat{e}_B \in T_{tid}$ such that $\hat{e}_A <_s \hat{e}_B)$

Figure 7: Invariant Sets Venn Diagram



Each of the three event invariants may relate any pair of event types. Thus, for a set of event types E there can be at most $3|E|^2$ invariants.

The system mines the above invariants by collecting three kinds of counts across the log for relation s. Each relation is traversed once in the forward and once in the backward direction to count:

- $\forall a$, $Occurrences_s[a]$: the number of event instances of type a
- $\forall a,b$, $Follows_s[a][b]$: the number of event instances of type a that are followed by at least one event instance of type b
- $\forall a,b$, $Precedes_s[a][b]$: the number of event instances of type b that are preceded by at least one event instance of type a

The invariants are mined from the following equivalences:

- $a \rightarrow_s b \iff Follows_s[a][b] = Occurrences_p[a]$
- $a \not\rightarrow_s b \iff Follows_s[a][b] = 0$
- $a \leftarrow_s b \iff Follows_s[a][b] = Occurrences_p[b]$

4.2 Invariant Sets Intersection

Theorem:

We denote the set of invariants mined from relation s as $Inv(s)$.

Let t be the time relation, r be a variable relation, and c be the composite relation of r.

Consider $Inv(r)$, $Inv(t)$, and $Inv(c)$ on some log L. Then the following seven claims are true about any invariant, i, in these sets:

Claim 1 $i \in Inv(t) \cap Inv(c) \cap Inv(r)$
Claim 2 $i \in Inv(t) \cap Inv(c) \wedge i \notin Inv(r)$
Claim 3 $i \in Inv(c) \cap Inv(r) \wedge i \notin Inv(t)$
Claim 4 $i \in Inv(t) \wedge i \notin Inv(r) \cup Inv(c)$
Claim 5 $i \in Inv(r) \wedge i \notin Inv(t) \cup Inv(c)$
Claim 6 $i \in Inv(c) \wedge i \notin Inv(r) \cup Inv(t)$
Claim 7 $i \in Inv(t) \cap Inv(r) \implies i \in Inv(c)$

Proof: Claims 1-6

Two logs will be used to demonstrate the claims by construction.

Consider the log:
$Initial \rightarrow_t a \rightarrow_r b \rightarrow_t x \rightarrow_t b \rightarrow_r a \rightarrow_t Terminal$
$Initial \rightarrow_t b \rightarrow_t b \rightarrow_t Terminal$

	$Inv(t)$	$Inv(r)$	$Inv(c)$
$a \leftarrow b$			✓
$a \leftarrow x$	✓		
$Initial \rightarrow a$		✓	✓
$Initial \rightarrow b$	✓	✓	✓
$a \not\rightarrow a$		✓	

$Initial \rightarrow b \in Inv(t) \cap Inv(c) \cap Inv(r) \implies$ Claim 1
$Initial \rightarrow a \in Inv(c) \cap Inv(r) \wedge Initial \rightarrow a \notin Inv(t) \implies$ Claim 3
$a \leftarrow x \in Inv(t) \wedge a \leftarrow x \notin Inv(r) \cup Inv(c) \implies$ Claim 4
$a \not\rightarrow a \in Inv(r) \wedge a \not\rightarrow a \notin Inv(t) \cup Inv(c) \implies$ Claim 5
$a \leftarrow b \in Inv(c) \wedge a \leftarrow b \notin Inv(r) \cup Inv(t) \implies$ Claim 6

Consider the log:
$Initial \rightarrow_t a \rightarrow_r b \rightarrow_t x \rightarrow_t b \rightarrow_r a \rightarrow_t Terminal$

$a \leftarrow b$ satisfies Claim 2 because $a \leftarrow b \in Inv(c) \wedge a \leftarrow b \notin Inv(r) \cup Inv(t)$
■

Proof: Claim 7
Suppose $i \in Inv(t) \cap Inv(r)$

Lemma 1: $Occurrences_r[a] = Occurrences_c[a]$

1	$\hat{e} \in r \implies \hat{e} \in c$	def. composite relation
2	$\hat{e} \in c \implies \hat{e} \in r \vee b$	def. composite relation
3	$\hat{e} \in b \implies \hat{e} \in r$	def. bridge pair
4	$\hat{e} \in c \implies \hat{e} \in r$	3 and 4
5	$\hat{e} \in r \iff \hat{e} \in c$	1 and 4

Let $p = \langle a,b \rangle$ Let $q = \langle b,a \rangle$

Lemma 2: $Follows_r[a][b] \leq Follows_c[a][b]$

1	$p \in r \implies p \in c$	def. composite relation
1	$\lvert r \rvert \leq \lvert c \rvert$	def. composite relation
3	$p \in r \leq p \in c$	1 and 2

Lemma 3: $Precedes_r[a][b] \leq Precedes_c[a][b]$

1	$q \in r \implies q \in c$	def. composite relation
1	$\lvert r \rvert \leq \lvert c \rvert$	def. composite relation
3	$q \in r \leq q \in c$	1 and 2

Lemma 4: $Follows_c[a][b] \leq Follows_t[a][b]$

1	$p \in c \implies p \in t$	def. composite relation
1	$\lvert c \rvert \leq \lvert t \rvert$	def. composite relation
3	$p \in c \leq p \in t$	1 and 2

Lemma 5: $Precedes_c[a][b] \leq Precedes_t[a][b]$

1	$q \in c \implies q \in t$	def. composite relation
1	$\lvert c \rvert \leq \lvert t \rvert$	def. composite relation
3	$q \in c \leq q \in t$	1 and 2

Lemma 6: $a \rightarrow_r b \implies a \rightarrow_c b$

1	$Follows_r[a][b] = Occurrences_r[a]$	def. $a \rightarrow_r b$
2	$Follows_r[a][b] \leq Follows_c[a][b]$	lemma 2
3	$Occurences_r[a] \leq Follows_c[a][b]$	1 and 2
4	$Occurences_c[a] \leq Follows_c[a][b]$	3 and lemma 1
5	$Occurrences_c[a] \geq Follows_c[a][b]$	def. *Follows*
6	$Follows_c[a][b] = Occurrences_c[a]$	4 and 5
7	$a \rightarrow_c b$	6 and def $\rightarrow_c$

Case 1: i is of type **AFby**

1	There exists $a \rightarrow_r b$ for some a and b	$i \in Inv(t) \cap Inv(r)$
2	$a \rightarrow_c b$ exists	1 and lemma 6

Lemma 7: $a \leftarrow_r b \implies a \leftarrow_{r,t} b$

1	$Precedes_r[a][b] = Occurrences_r[a]$	def. $a \leftarrow_r b$
2	$Precedes_r[a][b] \leq Precedes_c[a][b]$	lemma 3
3	$Occurences_r[a] \leq Precedes_c[a][b]$	1 and 2
4	$Occurences_c[a] \leq Precedes_c[a][b]$	3 and lemma 1
5	$Occurrences_c[a] \geq Precedes_c[a][b]$	def. *Precedes*
6	$Precedes_c[a][b] = Occurrences_c[a]$	4 and 5
7	$a \leftarrow_c b$	6 and def $\leftarrow_c$

Case 2: i is of type **AP**

1	There exists $a \leftarrow_r b$ for some a and b	$i \in Inv(t) \cap Inv(r)$
2	$a \leftarrow_c b$ exists	1 and lemma 7

Lemma 3: $a \not\rightarrow_t b \implies a \not\rightarrow_{r,t} b$

1	$Follows_t[a][b] = 0$	def. $a \not\rightarrow_t b$
2	$Follows_t[a][b] \geq Follows_c[a][b]$	lemma 4
3	$0 \geq Follows_c[a][b]$	1 and 2
4	$0 \leq Follows_c[a][b]$	def. *Follows*
5	$Follows_c[a][b] = 0$	3 and 4
6	$a \not\rightarrow_c b$	5 and def $\not\rightarrow_c$

Case 3: i is of type **NFby**

1	There exists $a \not\rightarrow_t b$ for some a and b	$i \in Inv(t) \cap Inv(r)$
2	$a \not\rightarrow_c b$ exists	1 and lemma 8

$i \in Inv(c)$

■

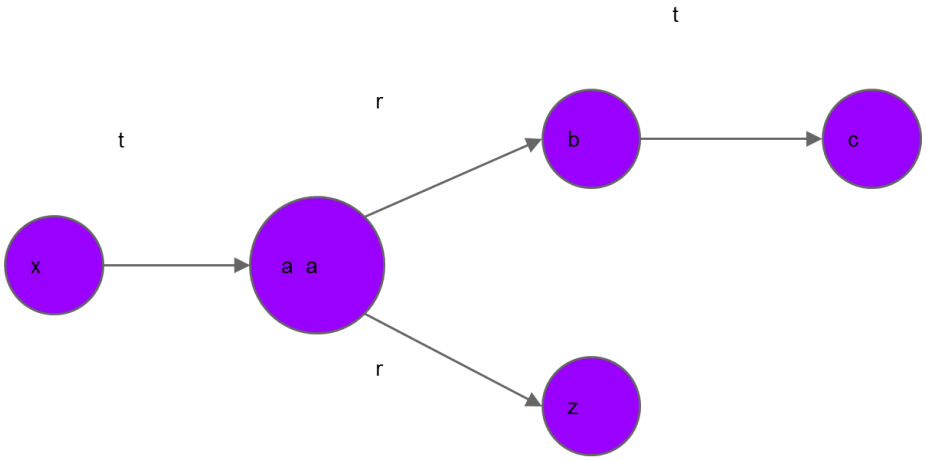
5 Implementation

5.1 Log Parsing

User input to Traceur consists of a log file and regular expressions for parsing. The regex is used to parse trace identifiers and event instances. Trace identifiers delimit separate traces within the log. Event instances make up the traces. Each event instance is required to have a timestamp and an event type. Event instances are also optionally allowed to specify an immediate, dependent relation and a closure, dependent relation. The immediate relates the instance, over the variable dependant realtion, as the successor to its temporal predecessor. The closure relates the instance, over the variable dependent relation, as the sucecssor to the last event instance identified as part of the dependent subgraph.

We set time as the independent relation.

Figure 8: Traceur Transition Function



	Synoptic	Traceur
Transition Function	delta(*target node*)	delta(*target node, traversed edge relation)*
On Traversal From a to b	delta(b)	delta(b, r)

5.2 Invariant Finite State Machines

Invariants in Synoptic and Traceur are implemented as finite state machines. The primary difference between the two is the transition function.

The partition graph is traversed during model checking. During traversal, input is fed to the invariant FSMs.

For Synoptic, the transition function takes the destination node as input.

For Traceur, the transition function takes the destination node and the relation typ eof the traversed edge.

In general, the Traceur FSMs have dual input transition functions, extra states to identify subgraphs, and deterministic behavior. This is particularly useful since the existing model checker implementation assumes deterministic FSMs. What this affords is FSM modularity and thus, the Traceur FSMs can be plugged into Synoptic's model checker.

6 Discussion

6.1 Future Work

The foundation outlined here theoretically generalized to a variable number of dependent relations. Implementing this requires simple extensions of the traces, partition graph, and invariant FSMs. Edges in the traces and partition grpah must be extended to specify a variable number of dependent relations. Additionally, the invariant FSMs should be extended to check over a variable number of dependent relations while maintaining the existing transition function.

The framework should also generalize to a variable number of indepenent relations. Any relation can be treated as independent so as long as it exists on every edge of the traces.

7 Related Work

All of Traceur is based on extending Synoptic to accomodate multiple-relations.

Ivan Beschastnikh and Yuriy Brun and Sigurd Schneider and Michael Sloan and Michael D. Ernst, Leveraging existing instrumentation to automatically infer invariant-constrained models, 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), 267-277, Szeged, Hungary, September 7–9, 2011.

Figure 9: Traceur AP

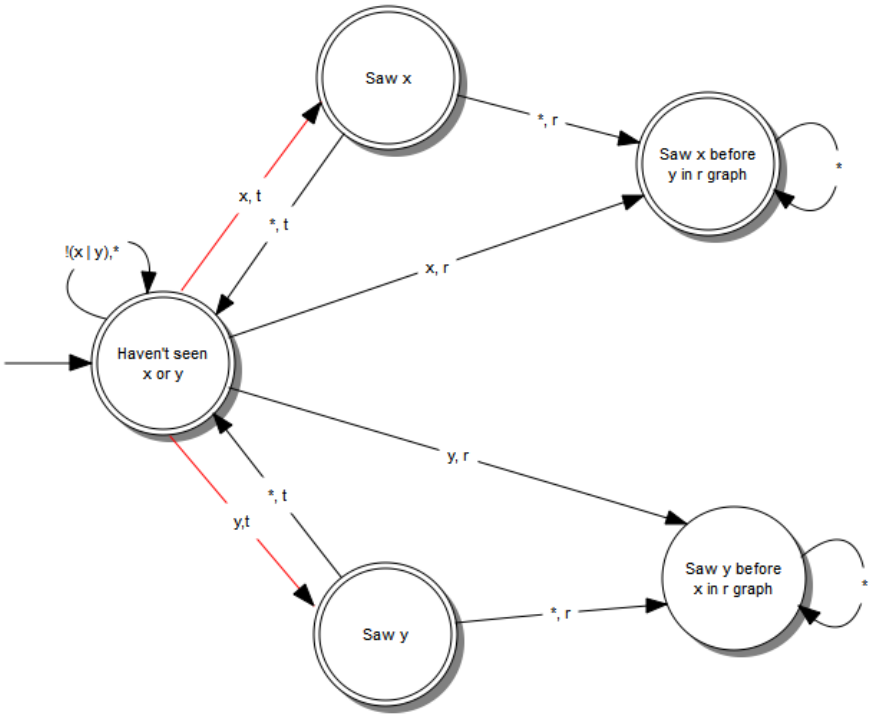


Figure 10: Traceur AFby

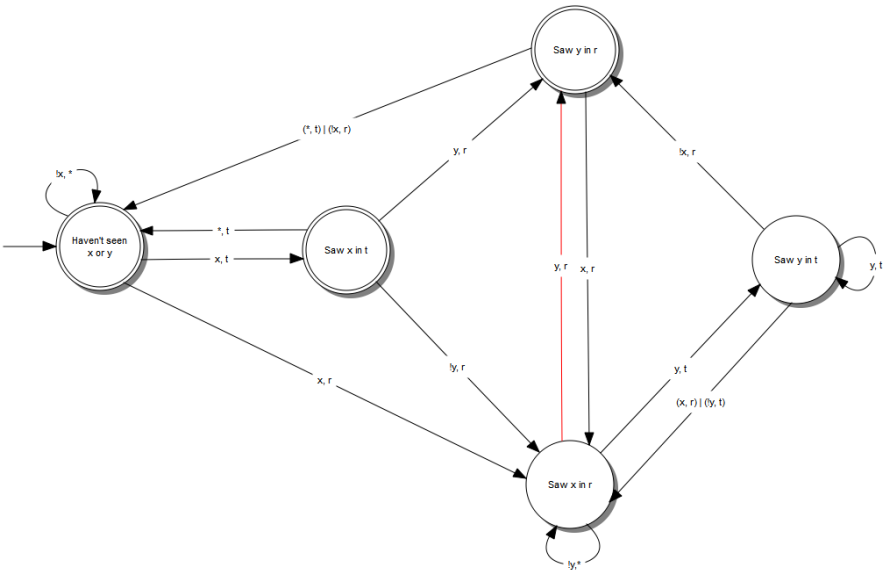


Figure 11: Traceur NFby