Synoptic : Unraveling Distributed Systems with Message-level Summarization and Inference Techniques

Slava Chernyak Josh Goodwin Sigurd Schneider Ivan Beschastnikh

University of Washington
Computer Science and Engineering
$\{$*chernyak, dravir, sigurd, ivan*$\}$*@cs.washington.edu*

1 Introduction

Distributed systems are often difficult to debug and to understand. Some of this difficulty can be ascribed to the complex behavior of the nodes comprising the system. For example, nodes may act concurrently with one another and may participate in multiple protocols simultaneously. Another difficulty is the distributed nature of state. No node can observe all the system state, and due to asynchrony in the networking environment, observations of remote state are stale unless protocols with significant overhead are used.

As a result, developers who design and implement distributed systems often analyze system behavior by collecting execution traces. However, a large system may generate thousands of messages – just a few nodes running a standard distributed protocol such as two-phase commit [6], or Paxos [9] can generate hundreds of messages. Manual inspection of traces scales poorly and by choosing to ignore certain parts of the trace it is easy for a developer to miss an important system behavior. It is therefore challenging to understand the behavior of a system from collected traces without tools to support this process. In this paper we describe the design and implementation of *Synoptic* – a tool to generate an Extended Finite State Machine (EFSM) representation of a distributed system execution. Synoptic analyzes the network-level traces generated by nodes in a distributed system and helps the developer to understand actual system behavior. It can be used to uncover potential system correctness issues and performance problems.

We approach the problem of understanding system behavior as a summarization and a data reduction challenge. Synoptic describes the behavior of a distributed system and reveals temporal and structural constraints relevant to this domain, such as those present in the payload of messages exchanged by nodes and in the ordering of messages in the traces. Although Synoptic reasons over the message traces, our representation of the distributed system is not a proof. We do not aim to generate a system specification, instead our goal is to generate a compact and *useful* summary representation of the observed system behavior. Additionally, we distinguish our problem from compression, which must retain *all* information. Our representation, on the other hand, can trade-off information content for conciseness. Additionally, unlike compressed content, our representation is intended to be inspected by a human. For this reason, the *form* of the representation is an important consideration.

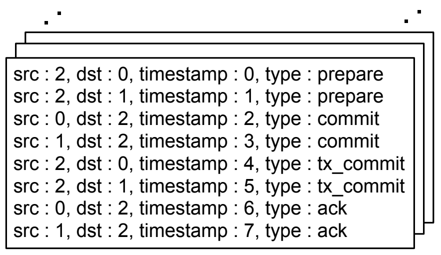
To reach our objective we develop a framework to search through the space of potential EFSM representations of the system trace. We develop two approaches of exploring this space – coarsening and refinement. Coarsening attempts to compact the representation without admitting a representation that may represent illegal traces. Refinement is the reverse process, but which a representation is expanded until it admits all the system traces.

To guide these two processes, we develop a decision engine and a set of policies relevant to our domain. Within this framework we implement two algorithms to demonstrate the trade-offs between coarsening and refinement. Our coarsening algorithm, *GK-Tail*, is based on the algorithm developed by Lorenzoli, et. al. [10] relying on the intuition that messages between nodes may be considered as remote method invocations. Our refinement algorithm, *Bikon* [1], is a bisimulation-inspired algorithm.

We wrap the framework and these two algorithms into a tool called *Synoptic*. We apply Synoptic to generate EFSM representation from real as well as synthetic traces of distributed systems. We evaluate these representations in a user study with a users from our target user population. In this study, systems researchers are asked to talk through their understanding of the represented system behavior. **TODO:** From this study we find that..

In summary, the contributions of this paper are the

[1]Bikon = [Bi]simulation + Dai[kon]



```
src : 2, dst : 0, timestamp : 0, type : prepare
src : 2, dst : 1, timestamp : 1, type : prepare
src : 0, dst : 2, timestamp : 2, type : commit
src : 1, dst : 2, timestamp : 3, type : commit
src : 2, dst : 0, timestamp : 4, type : tx_commit
src : 2, dst : 1, timestamp : 5, type : tx_commit
src : 0, dst : 2, timestamp : 6, type : ack
src : 1, dst : 2, timestamp : 7, type : ack
```

Figure 1: A single trace of a three node run of the two-phase commit protocol.

following. We develop a framework for finding a useful EFSM representation of a distributed system trace. Within this framework we implement and evaluate two algorithms on a variety of distributed system traces and in the context of a user study. This paper includes the representations found by both algorithms for a number of popular distributed protocols and systems, the results of our user study, and performance results for our coarsening and refinement algorithms.

We begin the rest of the paper with a motivation for our work. We describe why we developed Synoptic and give an example where it may be used to great effect. We then overview our assumptions and the design of our framework in section 3 and 4. In sections 5 and 6 we discuss refinement and coarsening in more detail. We then describe the interface to our decision engine and some of the policies it implements in section 7. We follow with section 8 and 9 in which we elaborate our two algorithms – the adapted GK-Tail and Bikon. Then, we present our evaluation in section 10. We finish with a discussion (section 11), and an overview of the related work in section 12.

2 Motivation

To motivate the problem more concretely we present an example scenario in which the Synoptic representation of a system trace might help the user understand system behavior. We focus on a popular atomic commitment protocol called two-phase commit [6], which is widely used to implement distributed atomic transactions. This protocol is often non-trivial to debug in large systems because it is inherently concurrent.

2.1 A Two-Phase Commit Example

The two-phase commit protocol must adhere to a number of temporal and structural invariants, which range over messages in the protocols – (P)repare, (C)ommit, (A)bort, (TXA)bort, (TXC)ommit. For a system of three

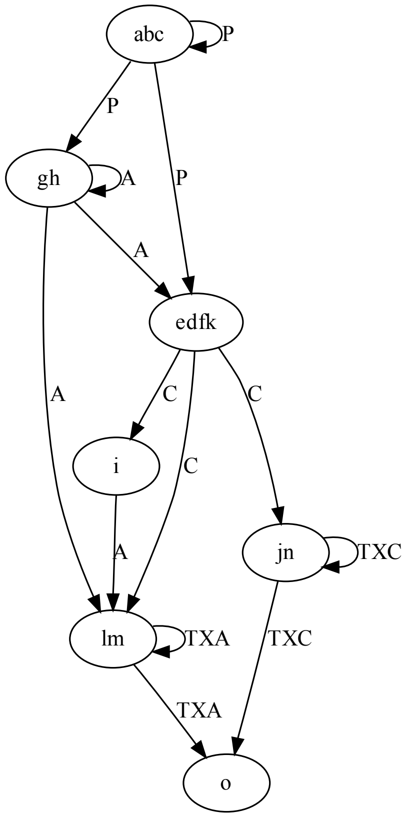


Figure 2: A Synoptic-generated representation of a buggy two phase commit protocol trace for two nodes, with one transaction manager.

nodes (with one node acting as the transaction manager) some of the important invariants are:

- $\Box(A \longrightarrow \Diamond TXA)$
- $\Box(TXA \longrightarrow \Diamond^{-} A)$
- $\Box(TXC \longrightarrow \Box\neg A)$
- $\Box(TXC \longrightarrow \Box\neg TXA)$
- $\Box(TXA \longrightarrow \Box\neg TXC)$
- $|C| + |A| = 2$

Given a trace that looks like Figure 1, how can a developer easily verify whether the invariants above hold? Figure 2 shows the representation generated by Synoptic for a buggy version of the protocol system, while Figure 9 shows the representation for a correct system. It is easy to verify by inspection that all of the above invariants are maintained in the correct representation. Moreover, the representation of the buggy system hints at where a problem might lie – the second invariant is violated since in Figure 2 the TXC message may preced an A message.

Note that the two-phase commit protocol also has non-temporal, or structural invariants. At the moment our system does not support these invariants, but we hope to add support for them soon.

3 Assumptions

We make a few assumptions concerning the environment or the distributed system whose traces we analyze.

- We assume a non-broadcast networking medium at the message capture layer. For example, we do not handle wireless environments in which the nodes in the system sniff and thereby receive all messages. However, we can operate in a wireless environment with all nodes restricted to using TCP for communication.
- We assume that the nodes are using a reliable communication protocol so that messages are never lost [2]. However, we do allow reordering of messages to occur. We also assume that transmission time is non-zero.
- We assume that the set of nodes in the system is constant and does not change throughout the trace. That is, nodes do not fail and new nodes do not join the system.
- If the system is composed of just one physical host the local clock can be used for timestamping all messages. To analyze the behavior of multiple network nodes, a consistent notion of time is necessary. To start with, we assume correctly synchronized clocks (e.g. nodes use NTP). Later, we hope to establish a global partial ordering on the messages in the system with vector clock algorithms [8].
- To allow Synoptic to capture messages the system must use protobufs [1]. However, this is not a fundamental assumption, and is an artifact of our implementation.

4 System Design

Synoptic is comprised of several components. Given an unmodified system as an input, we instrument the system to capture all network-level messages into a trace. For each message in this trace, we additionally associate useful meta information such as the source and destination node addresses. Because we assume that messages are never lost (see Section 3), we only capture outgoing messages. The traces are then collected from all the nodes, and are then merged into a single trace. We assume that the nodes use NTP or a similar protocol to synchronize their local time.

[2]The reliable communication assumption may be dropped if in addition to message transmission we are also able to observe message reception. In the absence of observable reception, we cannot distinguish between messaging behavior at the receiver node caused by a lost message and behavior caused by a message reception.

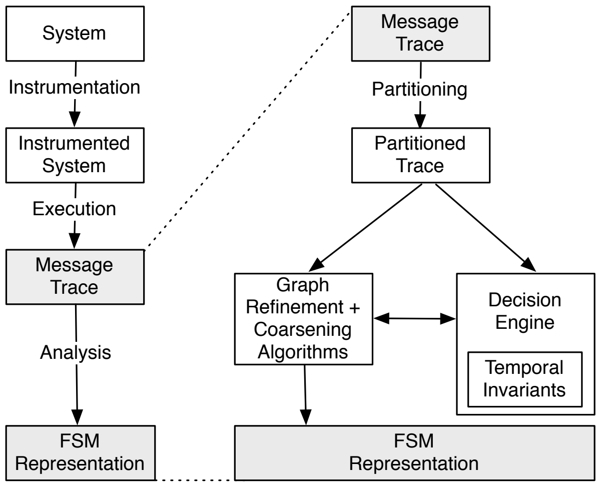


Figure 3: An overview of the system pipeline (left) and our two analysis pipelines implementing our analysis algorithms (right).

Next, Synoptic infers properties and relationships between messages in the trace that lead to a more concise representation. As part of this step, the trace may be partitioned and different representations may be generated for different partitions of the trace. Moreover, information content may be elided and excluded to make the representation more concise. Finally, the derived representation is presented to the user, along with all instances where information loss might have occurred. Figure 3 illustrates the components in the system's pipeline and the two types of analysis supported by Synoptic. In the rest of this section we elucidate those parts of our system that do not involve the analysis step.

4.1 Message Capture

We use the term *message* in a general sense to refer to a formatted message sent between two nodes in a distributed system. There are several ways to capture messages. For example, all traffic on a local machine could be sniffed (e.g. using `tcpdump`), or the system could be modified to use a custom network library that logs all messages sent and received to persistent storage.

We explicitly capture messages by modifying the distributed systems that we study. To simplify this task, we only support systems that use protobuf [1] specifications for their messages. This specification captures all

```
message SynopticMessage {
  required string src = 1;
  required string dst = 2;
  required int32 tstamp = 3;

  message SystemProtobufType {
    // Captured Message goes here
  }
  required SystemProfobufType = 4;
}
```

Figure 4: Protobuf format of a wrapped captured message.

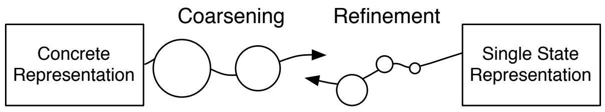


Figure 5: Coarsening and Refinement are two dual operations on the graph representation.

the message fields and their types. The availability of a protobuf specification enables structural analysis of the messages observed. These specifications also provide a natural way to distinguish messages according to their type.

Synoptic additionally assumes that the user specifies the fields of the protobuf specifications that should be used for structural invariant mining. We wrap all captured messages into our own message type, depicted in Figure 4. This way we can augment the captured messages with meta information such as the source and destination addresses, and a message timestamp.

4.2 Framework

After the trace is captured, the messages are annotated with *type*. Partitioning predicates are a set of predicates $p_1, ..., p_n$ such that for every message exactly one predicate is true. If predicate p_i is true for a message m, we say message m has type i. The partitioning predicates are assumed to be supplied by the user.

From the messages we form two different kinds of graphs:

- A state-based graph, in which messages are interpreted as transitions.
- A time-based graph, in which messages are interpreted as states, and are related by temporal properties.

These graph representations are duels of each other and one can be converted into the other. For the temporal graph, we define a partition as a set of messages. We can form graphs over partitions, and relate partitions according to properties of the states they contain. Section **??** will formulate the graph properties in more detail.

The Synoptic framework provides two dual operations to explore trace representations: refinement and coarsening as illustrated in Figure 5. Coarsening starts with a concrete trace of the system and merges nodes to derive smaller representations. Refinement starts with a single state representing the entire trace and enlarges the representations by splitting nodes.

5 Graph Refinement

The initial assumption of graph refinement is that all messages of a particular type are identical. Thus they are grouped into one partition. The algorithm then starts to find counter-examples to this hypothesis, and refines the partitions accordingly.

Partition refinement is based on a property that splitting the messages contained in the partition into two sets preserves the temporal properties of the messages observed in the original trace. For example, the behavioral predicates such as *appearing before* could be used, as well as structural predicates such as a including a certain data fields or having a field with a value in a certain value ranges. This is a possible area to apply Daikon [4] to refinement.

Partition refinement allows decisions to be made depending on the properties of the entire partition. For example, if only a small number of nodes can be split out, the algorithm can decide not to split so as to maintain a more compact representation. In general, statistical measures can be applied to the refinement decision process.

If certain refinements should be guaranteed, the initial graph can start with a more fine grained initial partitioning. At the moment we pre-partition using the user-supplied message type predicates.

6 Graph Coarsening

Graph coarsening is the dual operation of graph refinement. The initial assumption is that all messages are different. When similarities are detected between some set of nodes, the nodes are merged into a single node while preserving the transitions of the initial nodes. The notion of similarity can be again either behavioral, structural, or both. At the moment we consider mainly behavioral properties to decide whether to merge, and hope to use Daikon in the area of structural properties.

Graph coarsening takes the message type predicates into account and never tries to merge nodes with different

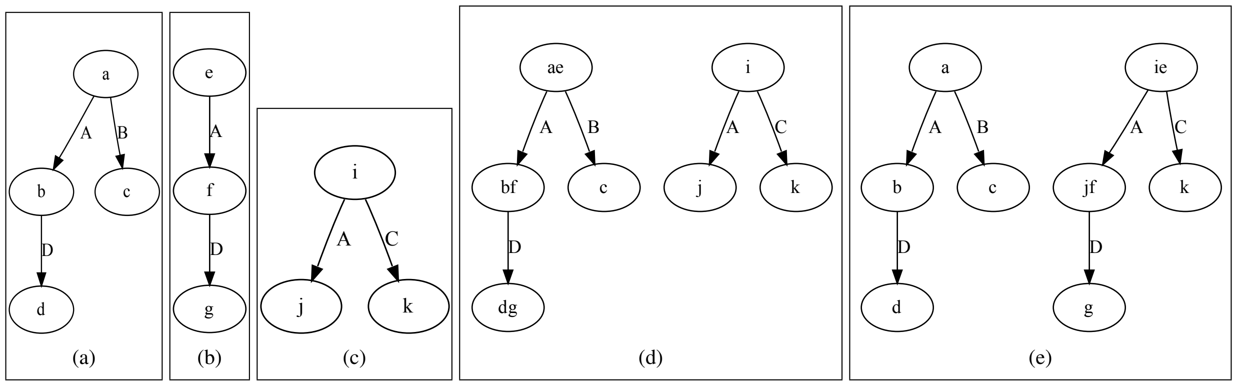


Figure 6: Above, (b) is 1-similar to (a) and (b) is 1-similar to (c). Merging (a) and (b) results in (d). Merging (b) and (c) results in (e).

types. In this way, we achieve the same restriction as graph refinement.

7 Decision Engine

All important decisions that the graph refinement and coarsening algorithms have to make are resolved by the decision engine. These decisions are encapsulated in the interfaced exposed by the decision engine to the algorithms.

7.1 Interface to Algorithms

The first important property is the decision whether the graph is considered a valid representation of the input traces. For this purpose the decision engine provides the method *check*, defined as

$$check \; : \; Graph \longrightarrow Boolean$$

The second method relates to how the graph can be refined. The *getSplit* method takes as input the graph and outputs a predicate to be applied on states that determines the split. This method can only be applied if there is at least some node that can be partitioned further. getSplit is defined as

$$getSplit \; : \; Graph \longrightarrow Predicate \times Partition$$

The final method exposed by the decision engine relates to how graph coarsening proceeds. The *getMerge* method takes as input a graph and a set of subsets of the nodes in the graph that are potential candidates for merging (i.e. which the algorithm is capable of merging). This method returns the element from the input set of subsets, which is the candidate the coarsening algorithm should merge. However, this method does not guarantee that the graph G' obtained after merging will satisfy $check(G')$.

$$getMerge \; : \; Graph \times \mathcal{P}(\mathcal{P}(S)) \longrightarrow \mathcal{P}(S)$$

7.2 *check* Policies

The decision engine can use different metrics to judge whether a representation is valid or not. Our only heuristic at the moment is to make sure that the representation satisfies a set of temporal invariants mined from the original input trace.

7.2.1 Temporal Invariants Satisfaction

The temporal relations between messages are potentially related to their causal dependencies. Because of this we intuit that strict temporal invariants should be preserved in the final representation of the trace. As an example, in the two phase commit protocol, the message `tx abort` is sent *because* a node sent `abort`, thus `tx abort` occurs *after* `abort`.

Mining invariants. We mine the patterns listed in Table 1 from the original input traces with the temporal relation between messages extrapolated from the timestamps. These patterns are partially based on the specification patterns and the classification scheme formulated by Dwyer et. al. [3]. Our algorithm first constructs the temporal graph with nodes representing messages, and then constructs the transitive closure of this graph. The invariants in Table 1 can then be deduced by considering all the outgoing and incoming transitions of a node. As an example, Appendix A lists all the temporal invariants mined for the two-phase commit protocol.

Checking an input graph. The mined invariants are checked against the input graph and True is returned only if the graph satisfies all of them.

Invariant	LTL$^-$ formula	Type
x AlwaysFollowedBy y	$\Box(x \longrightarrow \Diamond y)$	liveness
y AlwaysPrecededBy x	$\Box(y \longrightarrow \Diamond^- x)$	safety
y NeverAfter x	$\Box(x \longrightarrow \Box\neg y)$	safety

Table 1: Three types of detected invariants with corresponding LTL$^-$ formula and classification

7.3 *getMerge*Policies

The method *getMerge* gets a set of possible merges as argument, and has to return the best possible which then can be explored by the algorithm.

7.3.1 Random

Our current policy to decide which nodes to merge is to pick a node at random. For GK-Tail, this results in the behavior specified in the original paper [10]. However, the shape of the resulting graph depends on the random choice, and the algorithm is thus non-deterministic. To see this, consider Figure 6. In the figure, graph (b) is 1-similar to (a) and graph (b) is 1-similar to (c). Merging (a) and (b) results in graph (d), while merging (b) and (c) results in graph (e). Both (e) and (d) are different and the resulting sub-graphs in (d) and (e) are no longer 1-similar. Moreover, graph (d) is a more compact representation than (e). The non-deterministic choice of which nodes to merge, therefore, has important implications – the algorithm is not guaranteed to find the global minimum, and its choices are not symmetrical. These properties can be directly tracked to the fact that k-similarity is not an equivalence relation, since it is based on subsumption, and not equivalence, of graphs

7.4 *getSplit* Policies

The method *getSplit* takes a graph as argument, and returns a node, and a predicate. The predicate partitions the nodes in two sets, and this is the intended split. There are several possibilities for deciding on both, the node and the predicate to use for the split. The predicates can be grouped into two major categories: behavioral properties such as the presence of a certain transition, and structural properties, such as having a certain value in a data field.

7.4.1 Random

We can use the node to split at random. While this is guaranteed to eventually lead to a graph that satisfied the invariants, it may not result in the smallest such graph. Since the algorithm may stop splitting as soon as a call to *check* returns true, the final result is non-deterministic in this case, too.

7.4.2 Counter-example guided

A way to make a more informed decision is to use counter-examples from the invariants that were initially mined. For every unsatisfied invariant, there is a violating trace, From which a node can be chosen to be split. If this node is chosen well, the trace will not be present in the resulting graph any more. There are several possible ways to choose the node from inside the trace. In Bikon we pick the first violating node from the violating trace. We do so because it provides us with an intuitive measure of progress – splitting the first violating node leads us to hope that the resulting paths have no violations.

The returned predicate to use for splitting can be either a behavioral property, such as the presence of a certain transition in the node, or a structural property, such as certain values in data fields.

7.4.3 Daikon guided

In the case of structural properties, Daikon can be used to find a sensible splitting of a set of nodes according to the data-fields associated with them. A split could be evaluated by cross-validating the nodes from one candidate partition against the invariants from the other candidate partition, and vice versa. The more nodes do not satisfy the invariants, the better the split is.

8 The GK-Tail Algorithm

We now describe our adaptation of the GK-Tail algorithm. This algorithm consists of two steps, a partitioning stage and coarsening stage. Intuitively, the algorithm starts with the most refined representation of the trace and coarsens the graph at each stage by merging some two nodes. It terminates when it cannot find two nodes to merge, either because they are not similar enough, or because a merge would violate a temporal trace invariant.

8.1 Partitioning the trace

Partitioning of the trace first merges input-equivalent traces. This is an initial data-reduction step. Input-equivalent traces are those that represent a sequence of the same messages but with potentially different fields [3]

Next the trace is partitioned into a set of traces using a partition function. The choice of partition function is crucial to derive a proper representation for the traces. For example, the two-phase commit protocol traces used in the evaluation are partitioned by run of the system – each complete execution of the protocol is a unique trace. Another partitioning scheme might partition the global

[3] We plan to use Daikon to detect likely invariants over these parameters in the future.

trace by node, so that the resulting representation captures a single node's perspective of the system.

8.2 Coarsening

Let $G = (S, M)$ be the graph obtained by treating every trace as a sequence of states with edges between the states representing consecutive messages in the trace. We define the following two helper functions:

- `k-equiv(S,M)` to return $T \subseteq S \times S$ such that for all $(s,t) \in T$ we have that s is k-similar to t in G. By this we mean that the directed graph rooted at s, which is a subgraph of G with maximum path length of k is a subgraph of a similarly constructed graph rooted at t. There is a stronger version of k-similarity which mandates that the subgraphs be equivalent, however, we ignore this form in this paper.
- `k-merge(`G, s_1, s_2`)` to return G' that is obtained from the graph G by merging the nodes s_1 and s_2 up to depth k.

Then we can formulate our modified GK-Tail algorithm as follows:

```
Input: G
Let $S$ := k-equiv($G$)
While $S \neq \emptyset$
   Let ($s_1$, $s_2$) = DE.getMerge($G$, $S$)
   Let $G'$ = k-merge($G$, $s_1$, $s_2$)
   If DE.check($G'$)
      $G$ := $G'$
      $S$ := k-equiv($G$)
   Else
      $S$ := $S - \{(s_1, s_2)\}$
Output: G
```

9 The Bikon Algorithm

We modified a partition refinement algorithm [11] to build a bisimulation-inspired minimization that we call *Bikon*. This algorithm is novel because of a key property – the algorithm creates EFSMs that preserve all temporal properties of messages – e.g. message orderings. In addition, we use Daikon [4] to guide partition refinement towards representations that group together structurally interesting messages. In particular, we apply Daikon when temporal properties are no longer sufficient to discriminate bisimulation states.

The basic idea is to interpret the observed messages as the set of states S in a transition system $(S, A, \{\xrightarrow{a} \mid a \in A\})$, where the last component is a set of relations on S for every label from the set A. Between two states s, s' edges $s \rightarrow s'$ can be inserted to capture temporal notions such as *happens after*, or *is in response to*, which can be derived from timestamps, source/destination fields, etc. At this point, the graph is simply a different representation of the trace.

Using the partitioning predicates $p_1, \ldots, p_n$, we group states into partitions $\hat{S}$ and obtain a new transition system $\hat{G} = (\hat{S}, A, \{\xrightarrow{\hat{a}}_{\hat{S}} \mid a \in A\})$ where

$$\hat{S} = \{\{s \in S \,|\, p_i(s)\} \mid 1 \leq i \leq n\}$$
$$\xrightarrow{\hat{a}}_{\hat{S}} = \{(\hat{s_1}, \hat{s_2}) \in \hat{S} \times \hat{S} \mid \exists s \in \hat{s_1} : \exists s' \in \hat{s_2} : s \xrightarrow{a} s'\}$$

The transitions are inserted through existential abstraction, that means a transition between two partitions will be present if there is a transition from a state in one of the partitions to a state in the other partition. In the following, we will use the hat $\hat{}$ as an operator on graphs that returns the abstraction of a graph according to the above definition.

We then refine the graph until the *check* call to the decision engine returns true. If *check* returns false, we let the decision engine chooses a predicate p and a partition Δ to split according to the predicate.

The pseudo-code for Bikon is as follows:

```
Input: G
Let $(S, A, \rightarrow)$ = $\hat{G}$
While not DE.check($(S, A, \rightarrow)$)
   Let ($p$, $\Delta$) = DE.getSplit($G$)
   Let $\Delta'$ = $\{s \in \Delta \mid ps\}$
   Let $\Delta''$ = $\{s \in \Delta \mid \neg ps\}$
   $S$ := $S \cup \{\Delta', \Delta''\} - \{\Delta\}$
   $\rightarrow$ := $\{\xrightarrow{\hat{a}}_S \mid a \in A\}$
Output: (S,A,\goesto{})
```

10 Evaluation

10.1 GK-Tail Algorithm

We evaluated our framework on traces of a three-node two-phase commit protocol (with one node acting as the transaction manager). The resulting graph for $k = 1$ and disabled invariant detection is depicted in Figure 7. Considering that this representation can generate an invalid trace containing both, (A)bort and (TX-C)ommit, it is clear that this is not the right representation for the the protocol. The representation also contains self-loops, and thus allows for more behavior than observed from the input traces.

The same input was also processed with $k = 1$ and with invariant detection enabled. The resulting graph is depicted in Figure 8. The spurious trace is gone, since

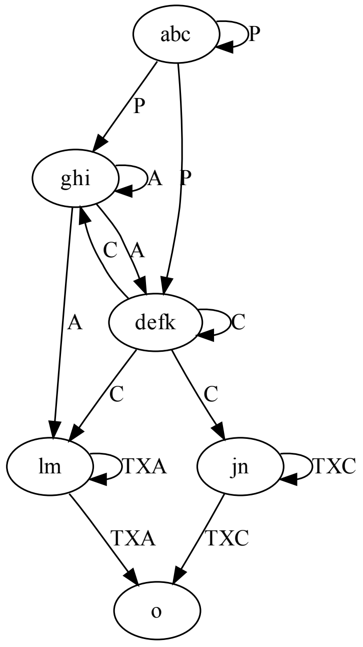


Figure 7: A representation derived using GK-Tail with k=1 and with temporal invariant checking disabled.

the crucial invariant was enforced by the decision engine. However, the automaton contains self-loops, and thus allows for an unbounded number of (A)borts and (C)ommits to occur before the transaction is eventually finished. This does not capture the important property that only two nodes participate in the protocol.

The most concise representation of the graph is obtained for $k = 2$. The resulting graph is depicted in Figure 9. In this case, the invariant detection does *not* alter the output. This is because the temporal invariant span is two, and $k = 2$ guarantees that traces up to this length are preserved. The significant improvement over Figure 8 is the absence of self-loops. Although the graph clearly shows that no more than two commits can occur, it misses the opportunity to merge states g, h, e, d.

Although checking temporal invariants does not alter the final representation for $k = 2$, it is important to note that choosing the right value of k is difficult without deep understanding of the system. On the one hand we have shown that the choice of k is crucial even in the presence of invariants. On the other hand we have show that small values of k in combination with invariant detection lead to smaller and - more importantly - correct, graph representation.

10.2 Bikon Algorithm

Bikon is able to produce the trace in Figure 8 as well. The trace in Figure 9 is out of reach, and we experienced

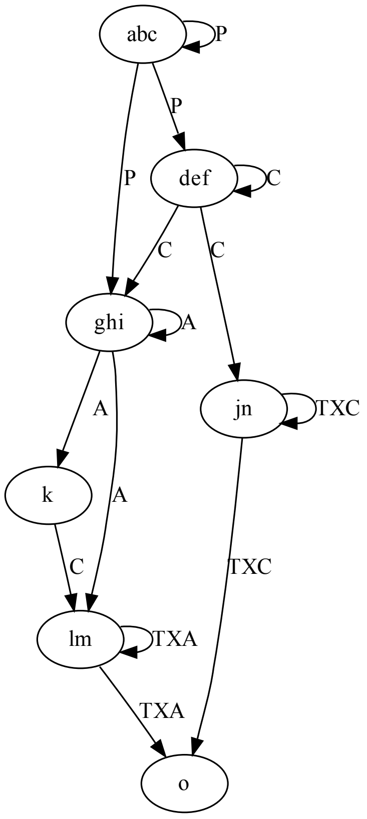


Figure 8: A representation derived using GK-Tail with k=1 and with temporal invariant checking enabled.

the same problems with self-loops.

We feel that the strength of Bikon lies in the ability to consider statistics, i.e. whether a node should be split could potentially made dependent on the properties of the other nodes in that partition. This is a direction we want to pursue for the final report.

10.3 User Study

TODO: As part of evaluation we will use people to evaluate our generated EFSMs. For instance, we will ask a person to consider the raw trace and to describe what the system does. We would then compare their description with what another person would describe when considering the generated EFSMs. We believe that the EFSMs would make it much easier to investigate system behavior and that it would greatly reduce the amount of time a person needs to spend to understand the various protocols inside a distributed system.

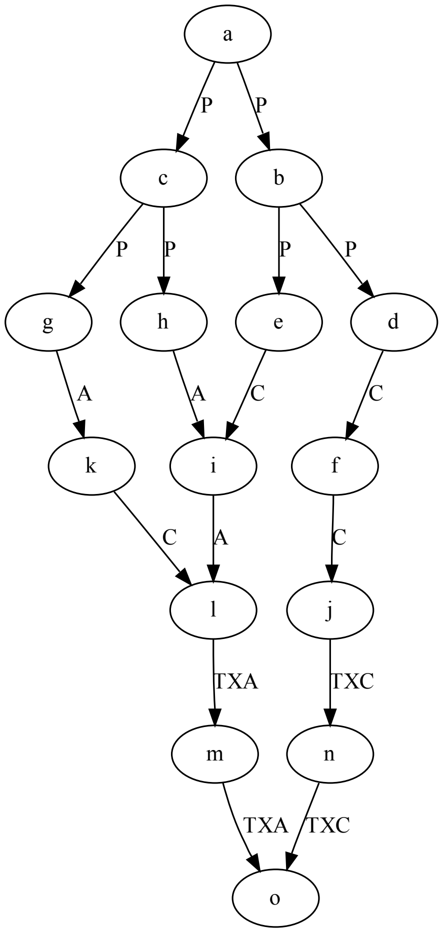


Figure 9: A representation derived using GK-Tail with k=2 and with temporal invariant checking enabled.

11 Discussion

There may be other potential trace partitioning strategies. For example, let the "type" of a message refer to the message signature defined by all the fields in the message. Then we can map the tuple $\langle$ type,src,dest $\rangle$ to a single transition. In other words, instead of grouping messages between nodes, we can separate them out based on src and dest pairs. We also augmented the fields of the message a timestamp, and hope to include latency values. We can use this augmented set of fields for a different partitioning strategy

It may be possible to include statistical methods in our framework. Specifically, it might be necessary to ignore parts of the trace to come up with a concise and representative model. This may correspond well to accounting for failure or other unexpected events in the distributed system. It is unclear, however, what specific statistical models or machine learning approaches are best to use since prior work on detecting such events and partitioning traces by protocol indicates that this is a challenging problem [2].

To evaluate the system in [10], the authors use a test coverage metric. By ensuring that the set of tests covers every transition arc in the EFSM representation, the tests can be considered to have a high degree of coverage. Our work may also be evaluated using this metric. If we are able to generate events in the system, then we can drive execution towards those transitions and states that were readily observed.

As part of future work we will evaluate our system on communication traces of the protobuf-based java interface to twitter.

12 Related Work

GK-Tail. The work by Lorenzoli, et. al. [10] is most closely related to ours, and inspired our use of the GK-Tail algorithm. In this work, multiple interaction traces are used to automatically generate an Extended Finite State Machine (EFSM). Each trace consists of a sequence of method invocations annotated with values for parameters and variables, and is generated by a single execution run of the program being analyzed. The resulting transition arcs of the EFSM are annotated with the relevant constraints on the data to transition from one state to the next state, providing a behavioral view of the program that includes how the input data affects the behavior. This algorithm is beneficial in understanding program behavior and in generating tests with better coverage. Our work generalizes GK-Tail to consider multiple k-future equivalent nodes, adapts the algorithm to the domain of distributed systems, and further modifies it to respect temporal constraints between nodes.

Bisimulation. Bisimulations are simulation relations that provide a strong notion of similarity for relational structures. They emerged in different fields [12], and their key property is preserving certain properties of the relational structure, for example, two strongly bisimilar transition systems are guaranteed to satisfy the same set of LTL formulas. Building on this property, an important applications in model checking is model minimization [5]. Our Bikon algorithm is a modification of a partition refinement algorithm [11]. Bikon uses invariants to determine which state to split next, and we stop splitting much earlier (once all invariants are satisfied), which results in a coarser representation.

Daikon. The Daikon [4] tool detects likely program invariants from observed program executions. During analysis of the execution traces, at marked program points Daikon instantiates a set of template invariants and tracks those invariants that hold true for the observed sequence of variable values. We plan to employ this technique to relate data fields of messages, and to find data invariants that hold between messages. Dynamically inferred program invariants have also been used by DIDUCE [7] to report invariant violations that might indicate software bugs. DIDUCE is an online tool that can be run to learn program invariants, as well as to check previously learned invariants. To infer invariants, DIDUCE instruments Java programs to output value of tracked expressions and uses a relaxation method by which strict invariants that match initial values are relaxed over time to accommodate new information.

Debugging distributed systems. Distributed systems are notoriously difficult to get right. Recent efforts by the systems community target bug finding in distributed systems with model checking. MODIST [14] explores the inter-leavings of concurrent events in an unmodified distributed system, thereby model checking the live system. CrystallBall [13] explores the state space of an actively executing distributed system, and when inconsistencies in possible future states are found, CrystallBall can be used to steer the distributed system away from buggy states. These tools, however, do not target the extraction of system properties that may be used to understand the behavior of the system they check. Property representations produced by Synoptic may be leveraged by these tools to guide model checking, and Synoptic may use these tools to target system execution towards states for which Synoptic lacks information to derive a concise representation.

Inferring temporal properties of programs. Perracotta [16] is a tool to mine interesting temporal properties from program event traces. It has been used to study traces of function calls for program evolution [15]. It has also been made scalable and robust to analysis of imprecise event traces to understand behavior and uncover bugs in very large code-bases such as the Windows Kernel and the JBoss application server [17]. Perracotta first instruments the program to output event information – all prior work uses method entrance and exit points. Second, the instrumented program is run – Perracotta relies on extensive test suites and random exploration of method calls to generate a broad range of event traces. Finally, the generated traces are used to infer program properties. The system generates candidate temporal patterns and then attempts to gain evidence that indicate the pattern holds by scanning through the trace. Patterns are expressed in terms of quantified regular expressions which are similar to regular expressions. The system uses a partial order hierarchy of properties that are built on the response pattern – the simple cause effect relationship between some two events.

Unlike Synoptic, Perracotta considers a totally ordered trace of events and does not consider properties that might be of interest in the domain of distributed systems. For instance, events may be concurrent and cannot be ordered with respect to one another. Additionally, Perracotta does not make use of any relationships between event data (e.g. method arguments/message payload).

Hidden Markov Models. A classic approach to modeling a system with unobserved state but with observations that may imply certain system state and state transitions are Hidden Markov Models (HMMs). The HMM model is powerful but it relies on knowing the set of states, as well as the transition and emission probabilities for transitioning between hidden states and emitting an observation in a state respectively. In our setting, the system state is not observable so the HMM model is applicable. However, from message traces alone it is not even clear what is the right number of system states to use for the HMM model. As well, the transition and emission probabilities are not known. Using HMMs in our setting is therefore impractical because of the large number of inputs that must be additionally supplied by the user.

References

[1] Protocol Buffers - Google's data interchange format, http://code.google.com/p/protobuf/. Accessed January 13, 2010.

[2] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma. Constellation: automated discovery of service and host dependencies in networked systems, no. msr-tr-2008-67, april 2008.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[5] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[6] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[9] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[10] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[11] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[12] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):1–41, 2009.

[13] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[14] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: transparent model checking of unmodified distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[15] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 340–351, Washington, DC, USA, 2004. IEEE Computer Society.

[16] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 23–28, New York, NY, USA, 2004. ACM.

[17] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM.

x AlwaysPrecedes y	x NeverFollowedBy y	x AlwaysFollowedBy y
P, P	C, P	A, TXA
P, C	A, P	
P, A	A, TXC	
P, TXC	TXC, TXA	
P, TXA	TXC, P	
C, TXC	TXC, C	
A, TXA	TXC, A	
	TXA, C	
	TXA, P	
	TXA, TXC	
	TXA, A	

Table 2: Temporal invariants mined for the two-phase commit protocol.

A Temporal Invariants Mined for Two-Phase Commit Protocol

Table 12 all the temporal invariants mined from valid two-phase commit protocol traces.