Inferring Class Level Specifications for Distributed Systems

Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury
National University of Singapore
{*sandeep,khoosc,abhik*}*@comp.nus.edu.sg*

David Lo
Singapore Management University
davidlo@smu.edu.sg

***Abstract*—Distributed systems often contain many behaviorally similar processes, which are conveniently grouped into classes. In system modeling, it is common to specify such systems by describing the class level behavior, instead of object level behavior. While there have been techniques that mine specifications of such distributed systems from their execution traces, these methods only mine object-level specifications involving concrete process objects. This leads to specifications which are large, hard to comprehend, and sensitive to simple changes in the system (such as the number of objects).**

In this paper, we develop a class level specification mining framework for distributed systems. A specification that describes interaction snippets between various processes in a distributed system forms a natural and intuitive way to document their behavior. Our mining method groups together such interactions between behaviorally similar processes, and presents a mined specification involving "symbolic" Message Sequence Charts. Our experiments indicate that our mined symbolic specifications are significantly smaller than mined concrete specifications, while at the same time achieving better precision and recall.

***Keywords*-Specification Mining; Distributed Systems**

I. Introduction

Classes of behaviorally similar processes are common in distributed systems, appearing in various application domains such as automotive, avionics, telecommunication and ground transportation. In particular, control systems in these application domains are of a distributed nature involving many behaviorally similar processes. Each of these processes has its independent flow of control, thereby being programmed as an active object. However, many such objects share similar behavior, and hence the system behavior can be more easily understood by describing behavior at the level of classes.

As an example, let us consider a telecommunication system with many phones and switches. Depending on the level of sophistication of the telecommunications software - the phone and switch objects could exhibit many snippets of behavior such as three way calling, call forwarding and call waiting. However, the exact identities of the phone/switch objects participating in such behavior are not important for comprehending the overall system behavior. Instead, the system behavior can be understood at the class level – the behavior of the *class* of phones, and the *class* of switches. Furthermore, due to the conventional usage of UML Sequence Diagrams or Message Sequence Charts (MSCs) [1] for describing sample interaction snippets in a distributed system and its wide usage in application domains such as telecommunications, behavior of such distributed systems with behaviorally similar processes can be conveniently described as class level specifications involving Sequence Diagrams [2], [3], [4].

In this paper, we tackle the problem of mining class-level specifications for distributed systems with behaviorally similar processes. The behavioral similarity of the processes is exploited in our mining, to infer a succinct system specification that is easier to manage, maintain and comprehend than specifications of concrete object behavior. We note that building formal models of systems is hard and time consuming. Furthermore, for legacy systems, such formal models may be missing, or even outdated (since they may not capture significant software updates that have taken place since the model was constructed). Thus, since the intended functionality of the program may never be formally captured, we can try to capture it indirectly by running the program and finding patterns in its execution traces. Such *dynamic specification mining* utilizes data mining techniques to observe patterns in the execution of software that can then be interpreted as its specification. Specifications produced from such techniques may be represented in various forms ranging from program invariants [5] to state machines [6] and temporal properties [7]. In this work, we propose the mining of class-level system specifications involving Sequence Diagrams from distributed system execution traces. Our mined specifications are called *Symbolic* Message Sequence Graphs, a graph of symbolic MSCs [8].

As an example, Figure 1(a) shows an MSC that describes interactions between multiple devices having a shared bus as is typical in a bus architecture like PCI (Peripheral Component Interconnect). The bus master broadcasts the address (*addr*) of the intended target device by placing it in the bus and all connected devices decode it. Only one of the devices responds by asserting a control signal (*ack*). This MSC description is specific in that it addresses a unique scenario in which there are exactly three connected devices and when the intended target device is *Target*$_2$. The specification in Figure 1(b) stands for a generic interaction between a Master device and the *class* of target devices and therefore is a parameterized version of the original specification. In this *Symbolic* Message Sequence Chart

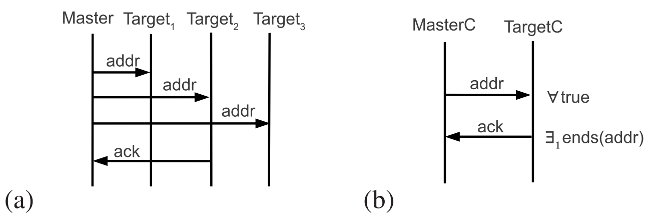


Figure 1. Concrete and Symbolic Message Sequence Charts describing interactions in a computer bus

(SMSC) [8], message communication events are symbolic actions that have annotations that specify the exact nature of the interaction. The guard $\forall true$ signifies that the message "addr" is intended for all target devices. The guard $\exists_1 ends(addr)$ makes it a requirement that exactly one of the devices that have received "addr" will respond with "ack".

The major technical difficulty in our mining process is to infer the *guards* of the events involving several behaviorally similar processes (or active objects) in a class. Such guards can be seen as "object selectors" – they select a subset of objects from the set of objects in a class of processes. Instead of specifying such an object selector (as one would do in distributed system modeling), we are automatically inferring such object selectors from the system execution traces. We mine for object selectors which are less general, easy to comprehend and accurately match the positive and negative observations as seen in the system execution traces.

In summary, we present a specification mining framework for inferring class level system specifications. Our method is particularly useful for distributed systems having many behaviorally similar processes, since a succinct easy-to-comprehend class-level system specification can be inferred – as borne out by our case studies and experiments. Our experiments show improvement in *both* precision and recall (leading to a higher F_1 score) in the mined class level (or symbolic) specifications, as opposed to mined object level (or concrete) specifications.

II. Background

A. Concrete Events

Let us assume that the real system being analyzed has a finite set of processes – P. We shall refer to these processes as concrete processes of the system. We use the term *Concrete Event* to refer to an atomic action executed by any concrete process in the system. A concrete event is either an internal action $(\langle p, m\rangle, s_p)$ or an external action $(\langle p \oplus q, m\rangle, s_p)$ where,

- $p \in P$ is the *main* concrete process to which this event is associated.
- $\oplus \in \{!, ?\}$ is an action type: *send* (!) or *receive* (?).
- m is an action label,
- $q \in P$ is a counterpart process.
- s_p, the current state of process p,

In a message passing context, $\langle p!q, m\rangle$ can be used to depict the event in which p sends message m to q and $\langle q?p, m\rangle$ for the event in which q receives message m from p. The current state s_p is captured using $\{\ldots, (p.x_j, v_j), \ldots\} \bigcup \{(h_p, v_h)\}$. Here, $p.x_j$ refers to a process variable and v_j refers to its corresponding value. The variable h_p refers to the local *execution history* or simply history of p when it is at state s_p. It is the sequence of actions executed by p prior to reaching s_p. For example, the sequence $\big((\langle p, m_1\rangle, s_p^1), (\langle p!q, m_2\rangle, s_p^2),\ (\langle p?q, m_3\rangle, s_p^3)\big)$ represents an execution history of the concrete process p at state s_p.

B. Process Classes

Our mining technique relies on a classification of processes in the system being analyzed. Formally, process classification is a surjective function $\Gamma : P \to \mathcal{P}$, from a set of concrete processes to $\mathcal{P}$ which is a set of process classes. For $\tilde{p} \in \mathcal{P}$, we shall use the notation $\Gamma^{-1}(\tilde{p})$ to refer to the set of concrete processes belonging to process class $\tilde{p}$. We assume that our analysis has prior knowledge of the classification of concrete processes. In practice, process classification can be obtained from either the source code of the respective processes or as input from user.

C. Symbolic Events

Class level specifications contain actions that are attached to a process class without being specific about the concrete processes that perform it. We refer to such class level actions as *symbolic events*, and these are the subjects which we shall infer.

A symbolic event is of the form $(\langle \tilde{p} \oplus \tilde{q}, m\rangle, \mathcal{Q}.g)$ ($(\langle \tilde{p}, m\rangle, \mathcal{Q}.g)$ for internal events). Here, $\tilde{p} \in \mathcal{P}$ is the process class to which the action is associated and $\tilde{q} \in \mathcal{P}$ its counterpart. A symbolic event signifies the abstraction of a set of concrete events at one or more concrete processes belonging to the same process class. The set of concrete events share the same action label m and (external) action type $\oplus$. The manner in which concrete processes may collectively participate in a symbolic event is specified through a quantified predicate $\mathcal{Q}.g$ that is referred to as its *guard*. For a process class $\tilde{p}$, $g : S_{\tilde{p}} \to \{true, false\}$ is a predicate on $S_{\tilde{p}}$, the set of all states of a process in $\tilde{p}$.

Definition 2.1 (Process Selection): For a process class $\tilde{p}$, a process selection is a predicate $\sigma : \Gamma^{-1}(\tilde{p}) \to \{true, false\}$ on the set of its concrete members.

Definition 2.2 (Process Class Context): The context of a process class $\tilde{p}$ is a function $\theta : \Gamma^{-1}(\tilde{p}) \to S_{\tilde{p}}$ that returns the state of each concrete member of that class.

The quantifier $\mathcal{Q}$ in $\mathcal{Q}.g$ specifies the number of concrete processes satisfying the predicate g that may participate in the symbolic action. It takes the form of one of the following: $\exists$, $\forall$, $\exists_k$ or $\forall_k$. The interpretation of the guard, denoted by $\mathcal{Q}.g(\sigma, \theta)$, is as follows.

1) $[\![\exists.g]\!](\sigma, \theta) \Leftrightarrow \forall p \in \Gamma^{-1}(\tilde{p}) : \sigma(p) \Rightarrow g(\theta(p)) \wedge |\sigma^{-1}(true)| \geq 1$
2) $[\![\forall.g]\!](\sigma, \theta) \Leftrightarrow \forall p \in \Gamma^{-1}(\tilde{p}) : \sigma(p) \Leftrightarrow g(\theta(p))$

3) $[\![\exists_k.g]\!](\sigma,\theta) \Leftrightarrow \forall p \in \Gamma^{-1}(\tilde{p}) : \sigma(p) \Rightarrow g(\theta(p)) \wedge |\sigma^{-1}(true)| = k$
4) $[\![\forall_k.g]\!](\sigma,\theta) \Leftrightarrow \forall p \in \Gamma^{-1}(\tilde{p}) : \sigma(p) \Leftrightarrow g(\theta(p)) \wedge |\sigma^{-1}(true)| = k$

The number of processes selected by σ is given by $|\sigma^{-1}(true)|$. Intuitively, the guard $\exists.g$ permits any combination of one or more processes in the class that satisfy g to participate in the action. The guard $\exists_k.g$ allows combinations of exactly k processes that satisfy g to participate. The guard $\forall.g$ requires the participation of all processes satisfying g. Finally, $\forall_k.g$, requires that exactly k processes satisfy g and that all of them be involved in the action. In this work, we opt for selecting processes for participation in the action based on the histories of the processes. The format of g thus characterizes the histories of the selected processes. We defer the details to Section II-F.

D. Process Class Constraints

In addition to using symbolic actions, class level specifications may assert how a process class should behave as a whole. We identify two such process class constraints: "$all(g)$" and "$count(g)\, op\, k$", where op can be either "=" or "≥". These predicates over the set of contexts of a process class $\tilde{p}$ are interpreted as follows as follows:

1) $[\![all(g)]\!](\theta) \Leftrightarrow \forall p \in \Gamma^{-1}(\tilde{p}) : \; g(\theta(p))$
2) $[\![count(g)\, op\, k]\!](\theta) \Leftrightarrow \exists_{P' \subseteq \Gamma^{-1}(\tilde{p})}, \forall_{p \in \Gamma^{-1}(\tilde{p})} : (|P'|\, op\, k \;\wedge g(\theta(p)) \Leftrightarrow p \in P')$

These constraints are further illuminated in Section II-F.

E. Symbolic Message Sequence Charts

Message Sequence Charts(MSCs)[1] are a visual formalism used to specify interactions between components in a system. The actions of each component or process are depicted as events along a vertical line in chronological order (top to bottom) that is referred to as its lifeline. Communication between processes are shown using horizontal arrows running between parallel lifelines that connect external events in one lifeline to its counterpart in an other lifeline. *Symbolic* Message Sequence Charts (SMSC's) are class level specifications that adopt basic MSC syntax, and introduce the concept of process classes. In an SMSC, a lifeline may either correspond to a concrete process or a process class that contains symbolic events. Semantically, an SMSC prescribes a partial order $\leq$ over the events from across lifelines. This partial order is a combination of the total ordering of events within each lifeline (denoted by $\leq_{\tilde{p}}$) and the ordering of send and receive counterparts (denoted by $\leq_{sm}$). Formally: $\leq \;\equiv\; \left((\bigcup_{\tilde{p}\in\mathcal{P}} \leq_{\tilde{p}}) \bigcup \leq_{sm}\right)^{\star}$.

A Symbolic Message Sequence Graphs (SMSG) is a high-level SMSC, which represents a collection of SMSCs in graph form. It is a directed graph with *basic SMSCs* as its vertices. Every path in the SMSG prescribes a valid scenario, which is specified by "concatenating" basic SMSCs located at vertices along the path. A *concatenation* of two basic MSCs M_1 and M_2 yields a bigger SMSC in which events from each process class $\tilde{p}$ in M_1 have to occur before the occurrence of any event of the same process class $\tilde{p}$ in M_2. The nature of such concatenation is 'asynchronous' because no ordering between events from across distinct process classes is explicitly enforced as a result of concatenation.

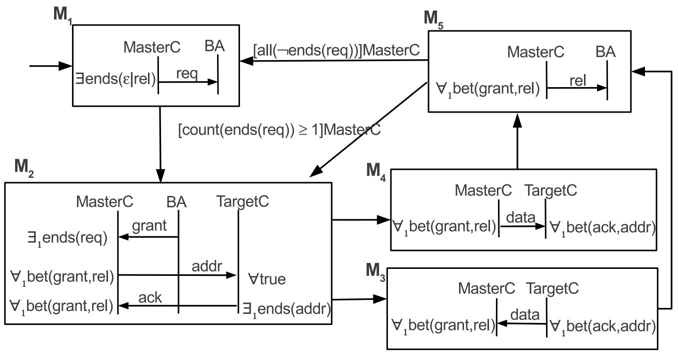
Furthermore, a process class constraint can be attached to an edge in an SMSG (referred to as *edge constraint*) to assert the condition of (the state of) the process class for the source SMSC to be concatenated to the target SMSC.

F. An Example of Class Level Specification

Figure 2 shows the SMSG specification of a simple bus arbitration protocol. The protocol specifies how a collection of target devices (process class TargetC) may be accessed by a class of bus masters (MasterC) through a shared bus that is arbitrated by a single bus arbiter: BA. The SMSG contains five basic SMSCs: $M_1, M_2 \ldots M_5$. Guards in the SMSG contain predicates of the form $bet(X,Y)$ or $ends(X)$, where X and Y range over action labels. Figure 2(b) shows how these predicates correspond to regular expressions and can be interpreted as constraints on the local execution history h of concrete processes. The predicate $ends(X)$ evaluates to *true* for a process iff its history ends with the action X and $bet(X,Y)$ evaluates to *true* when its history shows that Y has not been executed after the last execution of X.

The SMSC M_1 specifies the request phase of the protocol. The MasterC class contains a single event labelled with the guard $\exists\, ends\, (\epsilon\,|\, rel)$. The guard ensures that either the master device is making a request for the first time (*i.e* $h = \epsilon$), or it has released control over its previous request (it has sent the *rel* message to the bus arbiter). The quantifier $\exists$ suggests that one or more master devices may simultaneously make such a request. As defined in the SMSC M_2, the bus arbiter grants control over the bus to any one requesting master ($\exists_1\, ends\, (req)$) at a time. The master that has been granted control but is yet to relinquish it is subsequently referred to using the guard $\forall_1 bet(grant, rel)$. Once the master is granted control it is expected to broadcast the address of the intended target device over the bus. This broadcast is specified using the guard $\forall true$ at the TargetC lifeline. A unique device to which the address belongs responds to this broadcast from the master. The guard $\exists_1 ends(addr)$ accepts any selection in which exactly one of the processes that received the address responds with message *ack*.

In SMSCs M_3 and M_4 the master communicates data with the intended target device through the shared bus. In SMSC M_5 the master relinquishes control by sending the *rel* message. An edge constraint EC on process class $\tilde{p}$ is depicted by labelling the edge with $[EC]\tilde{p}$. The SMSC M_5 has two outgoing edges with process class constraints that apply to the process class MasterC. One of them, $count(ends(req)) \geq 1$, refers to the scenario when there are one or more master devices whose requests for bus have



(a) Mined Symbolic Message Sequence Graph:

(b) Regular Expressions:

ends(ϵ|rel): $h \in L\Big(\big(\Sigma^\star \langle \text{MasterC ! BA, rel}\rangle\big)^\star\Big)$

ends(req): $h \in L\Big(\Sigma^\star \langle \text{MasterC ! BA, req}\rangle\Big)$

ends(addr): $h \in L\Big(\Sigma^\star \langle \text{TargetC ? MasterC, addr}\rangle\Big)$

bet(grant,rel):

$h \in L\Big(\Sigma^\star \langle \text{MasterC ? BA, grant}\rangle \big(\Sigma - \langle \text{MasterC ! BA, rel}\rangle\big)^\star\Big)$

bet(ack,addr):

$h \in L\Big(\Sigma^\star \langle \text{TargetC ! MasterC, ack}\rangle \big(\Sigma - \langle \text{TargetC ! MasterC, addr}\rangle\big)^\star\Big)$

Explanation: Predicate ends(X) refers to the scenarios when the last event to be executed is X; similarly, predicate bet(X, Y) refers to scenarios in which the event Y has not occurred after the last execution of event X.

Figure 2. Class-level specification of centralized bus arbitration protocol

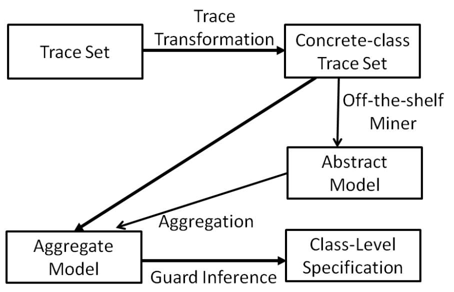


Figure 3. Overview of proposed mining procedure

not yet been granted. The other constraint, $all(\neg ends(req))$, refers to the complementary scenario when there is no process still waiting to be granted control of the bus. Together, these two constraints ensure that after M_5, M_2 is executed if there are more requests to be processed and M_1 is executed only after all requests have been processed.

III. Discovering Class-Level Specification

This section describes the proposed procedure (depicted in Figure 3) to mine class-level specifications. The first step involves incorporating class-level information into concrete events of the trace. The transformed traces will be used to mine an abstract model via a state-based model miner. The transformed traces, together with the abstract model are then used to generate an aggregate model. Lastly, guard inference is performed for each event in the aggregate model to generate the class level specification.

A. Transforming Traces

The trace transformation step prepares the traces for eventual extraction of class-level information by performing the following tasks:

1) It translates concrete process names to their corresponding process class names.
2) It combines similar consecutive concrete events of the same process class into a single *concrete-class* event.
3) For each concrete-class event thus created it records the process selection (σ) and the state information of the entire process class (θ).

Specifically, for each trace t consisting of concrete events of the form (e_p, s_p) where $e_p = \langle p \oplus q, m\rangle$[1], the trace transformer produces an embellished trace $\tilde{t}$ consisting of *concrete-class* events of the form $(\tilde{e}, \theta, \sigma)$, such that:

1) $e = \langle \tilde{p}_{(n_{\tilde{p}})} \oplus \tilde{q}_{(n_{\tilde{q}})}, m\rangle$, where $\tilde{p} = \Gamma(p)$, and $\tilde{q} = \Gamma(q)$ and $n_{\tilde{p}}, n_{\tilde{q}} \in \{1, \star\}$. [2]
2) σ identifies the process selection corresponding to this class-level action. (Definition 2.1)
3) θ returns the context of $\tilde{p}$ prior to this class-level action. (Definition 2.2)

For example, if there is a sequence of adjacent actions $(\langle p_1 \oplus q, m\rangle, s_{p1})$, $(\langle p_2 \oplus q, m\rangle, s_{p2})$, …, $(\langle p_l \oplus q, m\rangle, s_{pl})$, such that $p_1, p_2, \ldots, p_l \in \Gamma^{-1}(\tilde{p})$ are distinct concrete processes and $q \in \Gamma^{-1}(\tilde{q})$, these concrete events will be transformed into the following concrete-class event: $(\langle \tilde{p}_{(\star)} \oplus \tilde{q}_{(1)}, m\rangle, \theta, \sigma)$, where $\forall_{p \in \Gamma^{-1}(\tilde{p})} : \theta(p) = s_p$ (s_p is the state of p, just prior to execution of the concrete events being combined) and $\sigma^{-1}(true) = \{p_1, \ldots, p_l\}$.

B. Mining Abstract State-based Model

The concrete-class trace set $\{\tilde{t_1}, \ldots, \tilde{t_k}\}$ is then used by an off-the-shelf miner to produce a state-based model. Such a model describes the system behavior in terms of abstract, class-level events. In Section (IV-A) we discuss the specific mining approach used to generate the desired abstract MSC model.

Before presenting the traces to the state-based model miner, we strip off both the process selection σ and context information θ from the concrete-class events to form simplified *abstract* events of the form $\langle \tilde{p}_{(n_{\tilde{p}})} \oplus \tilde{q}_{(n_{\tilde{q}})}, m\rangle$. The resulting model will contain states and/or transitions attached with *abstract* events.

C. Generating Aggregate Model

Creation of a state-based model typically requires merging similar concrete-class events occurring at different traces and

[1]We ignore the presence of internal action for ease of presentation.

[2]Here, $\tilde{p}_{(1)}$ indicates that only one concrete process participates in this event and $\tilde{p}_{(\star)}$ indicates the participation of more than one concrete processes.

"folding" several concrete-class events occurring at different time stamps within a trace into one. Consequently, an action in the state-based model will correspond to multiple concrete-class events in the traces. The generation of an aggregate model first determines these correspondences. It then aggregates the process selections (σ_j) and process class context (θ_j) from all these concrete-class events. Lastly, it replaces the abstract events in the abstract model by the aggregated information. Specifically, an abstract event $\tilde{e}$ will be replaced by an *aggregate* event $(\tilde{e}, \mathcal{C})$ where $\mathcal{C} = \cup_i\{(\sigma_i, \theta_i)\}$ and σ_i and θ_i are retrieved from the corresponding concrete-class events. We refer to $\mathcal{C}$ as a *configuration* of the process class $\tilde{p}$.

Most techniques that mine for state based models can be easily adapted to record the mapping of abstract events in the model to corresponding concrete-class events in the trace that support it. Alternatively the correspondences may be recovered by "executing" the state based model according to the sequence of concrete-class events in the trace and thereafter mapping each instance in the trace to the corresponding abstract event executed by the model.

D. Inferring Symbolic Events

The last step of our discovery process is to derive symbolic events, of the form $(\langle\tilde{p} \oplus \tilde{q}, m\rangle, \mathcal{Q}.g)$, from the aggregate events, of the form $(\langle\tilde{p} \oplus \tilde{q}, m\rangle, \mathcal{C}_{\tilde{p}})$, of the aggregate model. We describe in this section how to infer the quantified guard $\mathcal{Q}.g$ from the configuration $\mathcal{C}_{\tilde{p}}$.

As we opt for guards that can characterize the history patterns of the participating processes, our inference algorithm aims to solve the following problem: "Find the most appropriate regular expression re that characterizes the history h_p of each participating process p belonging to the class $\tilde{p}$." The corresponding quantified guard is of the form $\mathcal{Q}.g_{re}$. For any concrete process $p \in \Gamma^{-1}(\tilde{p})$, the predicate obtained from re, denoted by $g_{re}(p)$, evaluates to *true* only when $h_p \in L(re)$. The inference algorithm requires several inputs: The configuration $\mathcal{C}_{\tilde{p}}$; a library of regular expression templates from which the candidate regular expressions are constructed; and a threshold min_sup defining the mandatory minimum number of histories required to support a candidate regular expression.

Algorithm 1 describes how a regular-expression based guard can be inferred from an aggregate event of class $\tilde{p}$. In line 1 the function *extractData* is used to extract relevant information from the configuration $\mathcal{C}_{\tilde{p}}$. Specifically, given that $\mathcal{C}_{\tilde{p}} = \{(\sigma_1, \theta_1),\ (\sigma_2, \theta_2),\ \ldots(\sigma_l, \theta_l)\}$ and $\Gamma^{-1}(\tilde{p}) = \{p_1, p_2, \ldots p_n\}$,
we have $extractData(\mathcal{C}_{\tilde{p}}) = (H^+, H^-, k)$, where

$$H^+ = \{\theta_i(p_j)(h_{p_j}) | i \in \{1, \ldots, l\} \wedge j \in \{1, \ldots, n\} \wedge \sigma_i(p_j)\}$$
$$H^- = \{\theta_i(p_j)(h_{p_j}) | i \in \{1, \ldots, l\} \wedge j \in \{1, \ldots, n\} \wedge \neg\sigma_i(p_j)\}$$

$$k = \begin{cases} r & \text{if } \exists r\ \forall_{i \in \{1,\ldots,l\}}(r = |\sigma_i^{-1}(true)|), \\ -1 & \text{otherwise.} \end{cases}$$

H^+/H^- refers to the set of execution histories of concrete processes when they had participated/not-participated in the action captured by the aggregate event. Moreover, k stores a positive constant r only if for every $(\sigma, \theta) \in \mathcal{C}_{\tilde{p}}$, the number of concrete processes which σ selects is r.

Line 2 in Algorithm 1 obtains R^+ and R^-, which are the sets of regular expressions accepting all histories in H^+ and H^- respectively, by invoking the function *getAccREs*. If there is no regular expression obtained in R^+, the algorithm will return $\exists.true$ (line 14) as the default gaurd. This permits any combination of concrete processes within the class to participate.

Among the re's in R^+, we select the most suitable re, based on certain ranking heuristics (line 5) that will be discussed later. Lines 6 to 12 specify how the right quantifier to this re is chosen. The final output is the quantified expression $\mathcal{Q}.g_{re}$. We discuss the method used to obtain accepting regular expressions as well as the ranking heuristics below.

Algorithm 1 Guard Inference

```
Require: Configuration: $\mathcal{C}_{\tilde{p}}$ for aggregate event at process class $\tilde{p}$; Threshold:
    $min\_sup$
Ensure: $\mathcal{Q}.g$ - Inferred Guard
 1: $(H^+, H^-, k) \leftarrow extractData(\mathcal{C}_{\tilde{p}})$
 2: $R^+ \leftarrow getAccREs(H^+, \tilde{p})$
 3: $R^- \leftarrow getAccREs(H^-, \tilde{p})$
 4: if $R^+ \neq \emptyset$ then
 5:     Select $re \in R^+$ such that rank($re$) is maximal.
 6:     if $supp(re, H^+) > min\_sup \wedge \exists_{re' \in R^-} : L(re) = \overline{L(re')}$ then
 7:         if $k > 0$ then $\mathcal{Q} \leftarrow \forall_k$ else $\mathcal{Q} \leftarrow \forall$ end if
 8:     else
 9:         if $k > 0$ then $\mathcal{Q} \leftarrow \exists_k$ else $\mathcal{Q} \leftarrow \exists$ end if
10:     end if
11:     Let $g_{re}(p)$ represent the proposition $h_p \in L(re)$ for any $p \in \Gamma^{-1}(p)$
12:     return $\mathcal{Q}.g_{re}$
13: else
14:     return $\exists.true$
15: end if
```

1) Finding Candidate Regular Expressions: For guard inference we choose a finite set of regular expression templates. This set of templates may be modified as neither the inference technique nor the proposed algorithm rely on any feature of these templates. For all $p \in \Gamma^{-1}(\tilde{p})$, let Σ denote the alphabet from which execution histories for a process class $\tilde{p}$ are formed. We select the following regular expression templates, as they are found to be intuitive and sufficiently expressive.

1) ϵ: The process is in its initial state
2) Σ^*A: The last action taken by the process is A.
3) Σ^*AB: The last two actions taken are A and B.
4) $\Sigma^*A(\Sigma - B)^*$: The action B has not occurred after the last execution of action A.

By substituting all pairs of events $e_1, e_2 \in \Sigma$ for variables A and B in these templates we generate a library of basic regular expressions referred to as $RELib_{\tilde{p}}$.

The method, $getAccREs(H, \tilde{p})$ in Algorithm 2 returns a set of regular expressions accepting all elements of H. Here, every element of H has to be tested against each expression in $RELib_{\tilde{p}}$. In addition, we also consider more complex regular expressions obtained by combining the basic ones.

Algorithm 2 is designed to efficiently find all the accepting regular expressions through a single pass of each string $h \in H$. The approach used is adapted from the one used by Yang et. al. to detect temporal rules based on some fixed templates [9]. Here, we describe a method where the process histories are tested for acceptance against an arbitrary library of regular expressions. We represent each regular expression by an automaton A_i (line 5). The current state of each A_i is maintained in $state[i]$. We traverse through (in line 10) each event e_j in the history h and apply the corresponding transition e_j to the current state of all automata.

On reaching the end of string h, those regular expressions that are at their accepting states are added to the set R_h in line 23. The set of expressions is expanded to the set R'_h, which contains regular expressions formed by combining one or more regular expressions (line 25). For example for a pair $re_1, re_2 \in RELib_{\tilde{p}}$, if $re_1 \in R_h$ but $re_2 \notin R_h$, the function *combineREs* will include the regular expressions re', re'' and re''' in the expanded set R'_h, for which $L(re') = \overline{L(re_2)}$, $L(re'') = L(re_1) \cup \overline{L(re_2)}$ and $L(re''') = L(re_1) \cap \overline{L(re_2)}$.

Algorithm 2 $getAccREs(H, \tilde{p})$

```
Require: $\tilde{p}$: Process class
Require: $H$: Set of execution histories of concrete processes from $\tilde{p}$
Require: $RELib_{\tilde{p}} \leftarrow \{re_1, re_2 \dots re_i \dots re_n\}$: Regular expression library
Ensure: $R$: Set of regular expressions accepting history $h$, $\forall h \in H$
 1: Let $\Sigma$ represent set of all events of the form $\langle \tilde{p} \oplus \_ , \_ \rangle$
 2: for all $h \in H$ do
 3:     Let $h = (e_1, e_2, \dots e_j \dots e_m)$
 4:     //Create automaton for each regular expression
 5:     Let $A_i \leftarrow (Q_i, \Sigma, \delta_i, q_{i0}, Q_{if})$ s.t. $L(A_i) = L(re_i)$
 6:     // $state[i]$ stores current state of $A_i$ (initialized to $q_{i0}$)
 7:     $state[i] \leftarrow q_{i0}$ (for $i = 1 \dots n$)
 8:     // $modeList(e)$: set of regexes for which event $e$ effects a state change
 9:     $modList(e) \leftarrow RELib_{\tilde{p}}$ (for all $e \in \Sigma$)
10:     for $j = 1 \dots m$ do
11:         for all $re_i \in modList(e_j)$ do
12:             $state[i] \leftarrow \delta_i(state[i], e_j)$
13:             for all $e \in \Sigma$ do
14:                 if $\delta_i(state[i], e) \neq state[i]$ then
15:                     $modList(e) \leftarrow modList(e) \cup \{re_i\}$
16:                 else
17:                     $modList(e) \leftarrow modList(e) - \{re_i\}$
18:                 end if
19:             end for
20:         end for
21:     end for
22:     //Form $R_h$, the set of basic regexes accepting $h$
23:     $R_h \leftarrow \{re_i | re_i \in RELib_{\tilde{p}} \wedge state[i] \in Q_{if}\}$
24:     //Form $R'_h$, the set of basic and complex regular expressions accepting $h$
25:     Let $R'_h \leftarrow combineREs(R_h)$
26: end for
27: $R \leftarrow \bigcap_{h \in H} R'_h$
28: return $R$
```

2) Ranking Guard Conditions: Given a set of regular expressions R^+ we have to select the most appropriate re that can be applied as a guard for the symbolic event. For trace set T, let $H^T_{\tilde{p}}$ denote the set of all execution histories of class $\tilde{p}$ which the entire trace set T has witnessed. We employ the following heuristics to rank the list of potential guards.

1) **Ranking based on rejection:** If re_1 and re_2 accept all positive samples, but re_1 rejects more negative samples than re_2 then rank$_{rej}(re_1) >$ rank$_{rej}(re_2)$.
2) **Ranking based on implication:** If re_1 and re_2 are candidate guards for an event belonging to process class $\tilde{p}$, if $\forall_{h \in H^T_{\tilde{p}}} h \in L(re_1) \Rightarrow h \in L(re_2)$ then rank$_{impl}(re_2) >$ rank$_{impl}(re_1)$. If for every execution history h witnessed in the traces, if h is included in the language of re_1, then it is also included in the language of re_2 then re_2 is preferred.
3) **Ranking based on likelihood:** If $|L(re_1) \cap H^T_{\tilde{p}}| > |L(re_2) \cap H^T_{\tilde{p}}|$ then rank$_{lkl}(re_1) >$ rank$_{lkl}(re_2)$. Here $|L(re_1) \cap H^T_{\tilde{p}}|$ refers to the number of strings in $H^T_{\tilde{p}}$ that are also part of language $L(re_1)$. If strings in $H^T_{\tilde{p}}$, the set of all execution histories witnessed in the trace, are more likely to be accepted by guard re_1 than by guard re_2 then re_1 is preferred.
4) **Ranking based on simplicity of guards:** If re_1 is a guard formed directly from one of the templates and re_2 is a composite guard then, rank$_{spl}(re_1) >$ rank$_{spl}(re_2)$.

With these heuristics we aim for an accurate regular expression that is also simple and easy to understand. By considering traces that are beyond the current historical data, the ranking criteria rank$_{impl}$ and rank$_{lkl}$ encourage the reuse of regular expressions across multiple events in the mined specification. This in turn improves the overall comprehensibility of class level behavior. We select the highest overall ranking guard as the inferred guard (line 5 in Algorithm 1). From our empirical studies the best results are obtained by finding overall ranking after applying the ranking methods in the following precedence order: rank$_{rej}$ > rank$_{impl}$ > rank$_{lkl}$ > rank$_{spl}$, where a ranking criterion is used only to break a tie resulting from the application of higher ranking criteria. The rank$_{rej}$ is given highest preference as it selects the guard that has the most distinguishing power (rejects most number of negative samples) and therefore likely to impact precision of mining. Apart from such automatic methods to discover guards, user assistance may be sought at this point to determine ideal guards from a shortlist. The user may also be able to assist in narrowing down the alphabet used for obtaining the basic regular expression library.

IV. Mining SMSGs

We employ the proposed class level specification mining approach to mine SMSGs. The method described here is based on our previous work on the *MSGMiner* framework [10] to mine concrete models in the language of MSG. We use *MSGMiner* to mine abstract behavior and combine it with our novel method for inferring symbolic events.

A. Mining Abstract Behavior

MSGMiner converts each trace in the trace set into a dependency graph whose vertices contain the events in the trace. The dependency graph captures the partial order of events from across processes. The chronological order of events within each process is maintained by a minimal set of directed edges. There is also a set of edges from send events to their respective receive events. The edges in this dependency graph is similar to the "happened before" relationship defined by Lamport [11]. The trace transformation operation discussed in Section III-A, is performed on the dependency graphs (partially ordered set of events) instead of traces (fully ordered sequence of events). When a group of concrete events from different concrete processes can be combined into a concrete-class event, the corresponding vertices are merged and labelled with that concrete-class event.

The *MSGMiner* framework breaks down the dependency graphs obtained from the set of traces into sequences of smaller dependency graphs called basic MSCs. From these sequences it mines for an MSG using a variant of the *sk-string* algorithm [12]. We adopt the same approach to construct an abstract MSG model by suppressing concrete information in the events. We also modify this process so as to maintain associations from each event in the abstract model to the set of trace locations that support it. Each abstract event is converted to an aggregate event by attaching a configuration formed by combining concrete information at the associated trace locations.

B. Conversion to Symbolic MSG

Using the guard inference technique discussed in Section III-D we infer a symbolic event from every aggregate event.

We also attach process class constraints at edges of the output SMSG. An edge from a basic SMSC, say M_A, to another basic SMSC M_B is labelled with a set of contexts $\Theta_{\tilde{p}} = \{\theta_1, \theta_2, \dots \theta_n\}$ for each process class $\tilde{p}$. Here, $\Theta_{\tilde{p}}$ is the set of contexts of $\tilde{p}$ from the trace set, in which, it has just finished execution of events in M_A and is about to execute of events in M_B.

We infer $G_{\tilde{p}}$, a set of process class constraints on $\tilde{p}$ at edges of the SMSG by first initializing it to a set of potential constraints of the form $[all(g_{re})]$, $[count(g_{re}) \geq k]$ and $[count(g_{re}) = k]$ (defined in Section II-D) with each g_{re} from the set of regular expression templates ($RELib_{\tilde{p}}$). The value of k is initially chosen based on any one $\theta \in \Theta$. We then iterate over remaining $\theta_i \in \Theta_{\tilde{p}}$ and discard or modify the constraints in $G_{\tilde{p}}$ as necessary so that the final set of constraints are satisfied by all elements of $\Theta_{\tilde{p}}$.

Having identified a set of constraints $G = \cup_{\tilde{p} \in \mathcal{P}} G_{\tilde{p}}$ at edges in an SMSG, we now discuss how it can be further reduced to a smaller set of important constraints. If a basic SMSC has l outgoing edges, having set of constraints $G_1, G_2 \dots G_l$ respectively, let $\hat{G} = G_1 \cap G_2 \cap \dots G_l$ represent the set of conditions that are common to all edges. As constraints in $\hat{G}$ are valid at every outgoing edge and not critical to the choice of an edge, they are discarded from $G_1, G_2, \dots G_l$. A set of constraints G can be further reduced to contain only those constraints that are not implied by any other constraint in G. In the mined SMSG, the conjunction of all constraints in G is imposed at the respective edge. To reduce the risk of over-fitting data, an edge constraint is imposed only if it has been inferred from a set of contexts $\Theta_{\tilde{p}}$ such that its size ($|\Theta_{\tilde{p}}|$) is greater than a specified threshold. We refer to this threshold as ec_min_sup and make it a parameter to the mining technique.

V. Evaluation

We evaluate our approach by mining specifications out of traces collected from few different systems. Each specification thus produced is compared to a correct specification of the system. The proximity of the mined specification to the correct one is quantified in terms of precision and recall. We also calculate precision and recall of concrete MSGs obtained from the technique expounded in [10].

A. Evaluation Method

Mining of finite-state-machine based techniques have typically been evaluated by comparing the language implied by the mined specification against that of a manually constructed "correct" specification [13], [14]. The mined and correct specifications can be viewed as both a generating model (one that produces a list of all valid action sequences) and an accepting model (answers if a given sequence is valid). *Precision* is defined as the ratio of the number of sequences generated from the mined specification and accepted by the correct specification to the total number of sequences generated by the mined specification. *Recall* is the ratio of the number of sequences generated by the correct specification and accepted by the mined specification to the total number of sequences generated by the correct specification. As behavioral models may generate infinitely many sequences, and sequences of infinite length, only a finite set of test sequences having bounded length are generated in practice.

We compare the mined specifications (both concrete and symbolic) against the correct specification of the system expressed as an SMSG. In our previous work [10], we have adapted the definition of precision and recall to the context of (non-symbolic) MSGs. An MSG generates partially ordered sets (rather than totally ordered sequences) of events, each expressed as an MSC, which are tested against the accepting model. In order to test SMSCs (generated by an SMSG) against the accepting model in a similar fashion, we will have to quantify what portion of the behaviors expressed by each SMSC is valid according to the accepting model. The quantified guards in symbolic events may refer to an unbounded number of configurations. We overcome this challenge, by transforming each generated SMSC to a finite number of concrete MSCs. This is done by producing all concrete realizations of the SMSC with each process class mapped to a finite set of concrete processes. In practice, we use the input process classification (Γ) for this mapping to ensure fair comparison with the mined concrete model as the number of concrete processes and their labels are consistent with what is present in the trace set. The concrete realizations must also honor the edge constrains in the SMSG. For example consider the set of SMSCs formed by concatenation of basic SMSCs beginning with the following path (M_1, M_2, M_3, M_5, M_2, $\ldots$) from the SMSG in Figure 2. Any concrete MSC derived from these SMSCs, must be able to satisfy the constraint on edge from M_5 to M_2, *i.e.*, the concrete realizations of this SMSC must involve simultaneous requests from two or more master devices.

To test if a concrete MSC M_{conc} is accepted by an SMSG, we search the SMSG for a path that forms an SMSC M_{sym}, such that M_{conc} is one of the valid concrete realizations of M_{sym}. We redefine precision and recall as follows.

$$\begin{aligned} \text{precision} &= \frac{\text{\# of MSCs generated by MM and accepted by CM}}{\text{Total \# of MSCs generated by MM}} \\ \text{recall} &= \frac{\text{\# of MSCs generated by CM and accepted by MM}}{\text{Total \# of MSCs generated by CM}} \end{aligned}$$

Here CM refers to the $SMSG$ specification that is correct and MM the mined model is either the mined $SMSG$ (evaluating the proposed approach) or the mined concrete MSG (evaluation of existing mining approach). We also compute the F_1 score (the harmonic mean of precision and recall) that is typically used by the information retrieval community to measure accuracy.

B. Case Studies

We consider the following distributed systems: (a) "Center TRACON Automation System" [15] an air traffic control system from NASA, (b) a system of server and VoIP clients communicating based on the Session Initiation Protocol (SIP) and (c) a system of Server and Clients that follow the XMPP Instant messaging and Chat protocol.

1) CTAS: CTAS is an Air Traffic Control system from NASA. The CTAS weather control logic specification [16] was one of the case studies recommended at the 3rd International Workshop on Scenarios and State Machines (SCESM04). The specification describes how weather aware client processes connect to a central Communications Manager. As access to the CTAS system is limited, we procure execution traces by implementing and executing a simulation of this system in Java. Our implementation is based on a formal specification of the system in Promela and high level HMSC that was developed by a fellow researcher.

2) Session Initiation Protocol: SIP is a signalling protocol used to establish, manage and terminate VoIP calls and multimedia sessions in general [17]. SIP clients interact with servers that perform the necessary call routing and function as gateways to the Public Switched Telephone Network (PSTN). We attempt to specify how clients should interact with their proxy server to achieve some of the basic call features. For this, we set up a system having three SIP clients connected to a single server. We use instrumented versions of *KPhone* [18] – a SIP client implementation and the *Opensips* server [19] both of which are available with source code under a GPL license. A trace set is produced by executing common use cases involving features such as basic call setup, call screening and call forwarding.

3) XMPP: Extensible Message and Presence Protocol is an open Instant Messaging standard originally developed by the Jabber open source community. The core functionality of the protocol is specified in rfcs 3920 and 3921. XMPP is the protocol for exchange of instant chat messages and presence information between various entities in a network that are addressed by unique jabber ID. The clients communicate to the server through structured XML messages. We discover the client-server interaction from a system having multiple jabber clients that are brokered by a single server. In the specification, the server and client processes are the lifelines and the message arrows represent transfer of XML messages. The *Openfire* XMPP server [20] and Jeti [21]/Pidgin [22] client implementations were instrumented and executed for trace collection. For discovering the core specification, we only record stanzas used for authentication or those having a *message* or *presence* tag and ignore rest of the message exchanges.

In addition to the core specification, XMPP Standards Foundation (XSF) has standardized several additional chat features. We attempt to mine behavioral specification for the Multi User Chat (MUC) functionality [23]. For this we collect traces by executing a separate set of use cases involving features such as service discovery, multi-party chat and creation and administration of chat rooms. Only messages sent from or addressed to the MUC conference service are recorded.

In our previous work on MSG mining [10], we considered specific instances of these systems in which there is limited behaviorally similarity between processes. This was done either by having only one concrete process from each process class or assigning a fixed role or duty to each process. We ease such restrictions in the experiments described here.

C. Experimental Setup and Results

In each of these systems, multiple processes perform asynchronous communication over TCP socket connections. Timestamped traces were collected by inserting instrumentation code into the source code at points where messages are written to or read from a socket. The traces were filtered and the message names abstracted with the help of text processing scripts. The mining was performed on a JVM running on an Intel duo core CPU with 1GB of available memory. The correct specification (that reflects allotted process and message labels) was manually derived by the

Table I
ACCURACY OF MINED CONCRETE MSG AND SMSG

System	**# events in trace set**	**Mined Concrete MSG**					**Mined SMSG**				
		Prec	**Rec**	**F_1 Score**	**# events in MSG**	**Time (s)**	**Prec**	**Rec**	**F_1 Score**	**# events in SMSG**	**Time (s)**
SIP	3326	0.8	0.05	0.09	222	97.7	0.64	0.66	0.65	54	55.7
XMPP-Core	5522	1	0.19	0.32	288	94.6	1	0.66	0.79	46	44.3
XMPP-MUC	7938	0.61	0.36	0.45	186	131.7	0.67	0.63	0.65	82	83.6
CTAS	11814	0.25	0.43	0.31	752	466.15	0.88	0.9	0.89	134	338.5

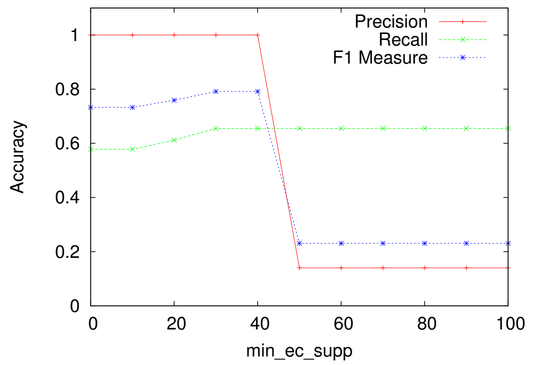


Figure 4. Plot showing impact of ec_min_sup on mining accuracy for the XMPP core protocol

authors for quantitative analysis and comparison based on available informal specifications.

Both symbolic and concrete methods employ a variant of the *sk-string* algorithm to mine finite state machines. The k parameter is set to 2 in both cases as is commonly used in most applications of the algorithm. It was observed that universally quantified guards can be accurately inferred even with a low minimum support (min_sup in Section III-D). The threshold ec_min_sup was found to play a critical role in determining whether the constraints on edges are correctly effected. The precision of mining for the core XMPP protocol example was found to be most sensitive with respect to the minimum support threshold and Figure 4 shows the impact of change in ec_min_sup to the accuracy measures. We use $ec\_min\_sup = 40$ and $min\_sup = 5$ for evaluation in all systems.

Table I tabulates the results from the case studies. It shows the precision, recall and F_1 measure of the mined concrete MSGs and the mined SMSGs for each of these systems that were studied. The column, "# events in trace set" indicates the total size of the trace set in each case. The columns "# events in MSG/SMSG" reflect the size of mined specifications in terms of the number of primitive actions they contain. It is seen that mined symbolic specifications have better accuracy when compared to the mined concrete specifications. The concrete specification mining approach gives poor recall as it does not consider similarity between processes. Concrete MSGs reflect only those process selections that were captured by the traces. In reality, the traces only capture a small fraction of all the possible configurations of the system. In certain cases, the mined SMSG has better precision. This can be attributed to the presence of guards and edge constraints by which the set of actions a process is allowed to perform is determined by its full execution history. The selected set of regular expression templates were found to be sufficient to infer meaningful guards and edge constraints.

D. Threats to Validity

A threat common to specification mining methods is the trace collection mechanism which may not ensure comprehensiveness of input traces. Such a threat can be countered by subjecting more input traces to mining. A threat to the validity of our assessment is that the correct specifications used for evaluation have been derived by hand based on informal specifications. We have partially addressed this threat by inspecting the set of behaviors specified by mined concrete and symbolic models but rejected by the correct model to ensure they are in fact incorrect behaviors.

VI. RELATED WORK

Several dynamic analysis approaches have been proposed to infer state machine specifications for Application Programming Interfaces (API) [24], [13], [25]. The inferred specifications typically describe correct invocation patterns of the API's methods by observing execution traces of "clients", *i.e.* programs that make use of the APIs. Methods that employ static analysis on client source code to synthesize similar API specifications have also been proposed [26], [27]. Static techniques do not require test suites or the actual execution of instrumented programs. On the other hand, dynamic analysis benefits from the ability to witness actual execution paths.

In a distributed setting, each process can be viewed as an API that is invoked, through message passing, by other processes in the system. As these methods do not explicitly handle interactions between multiple concurrent processes, they have to be applied at a per-process level by first slicing traces based on the association of events to processes. In our mining (dynamic) approach, concrete processes are grouped into classes and mined specifications describe collective class behavior. In many techniques to mine API usage patterns, after traces are sliced based on object identifiers, object level actions are lifted to class level actions. We note that the resulting specifications depict the behavior of individual objects of a class and are distinct from the class level symbolic specifications described in this paper. Class level symbolic actions allow us to construct a global behavioral model in the form of an SMSG. For distributed systems, an MSC-based mined specification (capturing the interaction snippets) is often easier to comprehend that an automata specification — as argued in our past research [10].

The invariant detection method of Daikon [28] has been combined with behavioral specification mining in [29], [30]. Boolean invariants regarding object variables or parameters are attached to transitions between states or to method calls to express guards. Our conception of guards is fundamentally different as it is a quantified expression that constrains the selection of processes from a process class. Yang et. al. in [9] propose a scalable algorithm to detect temporal rules in API usage by relying on predefined regular expression templates. Our use of regular expressions based templates is to identify quantified guards that can be applied for process selections or edge conditions within a behavioral specification. Gabel and Su in [31] improves the performance for templates having three or more characters using a BDD based symbolic mining technique. It should be noted that the word "symbolic" applies to the mining technique rather than the mined specification as in our case.

In [32], a library for automata learning is presented that exploits domain specific properties of distributed systems. Although behavioral similarity within process classes is used to improve the learning technique and output specifications, the output is a single concrete global state machine. Furthermore, the learning technique employed requires active experimentation during the specification inference process. In contrast we use an offline technique to build class level specifications that are smaller and therefore easier to understand.

Damas *et al.* [33] provide a technique to generate labelled transition system (LTS) for components in a system from an input of correct and incorrect scenarios. The LTS defines a set of internal states of the components and transitions between these states. Whereas their approach constructs an LTS for each concrete lifeline in the scenarios, we treat groups of behaviorally similar processes as a single component and specify actions using symbolic events. The constraints at states of the synthesized LTS characterize an internal state. In contrast, we use edge constraints and guards to specify global class level constraints or interaction properties. Unlike LTS, SMSGs provide both a scenario based and a state based description of system behavior.

Efforts have been made in program visualization by constructing UML sequence diagrams from dynamic executions [34], [35]. Such work constructs a sequence diagram from dynamic traces but does not produce graph-based models like MSG that include loops and branches. Rountev *et. al.* [36] proposes a static analysis based approach to reverse engineer UML sequence diagrams from source code. Lifelines in the sequence diagrams represent runtime objects. In contrast, our approach is based on the analysis of execution traces and mined specifications portray class level actions using symbolic events.

Lo et. al. in [37] mine for Live Sequence Charts (LSC) that contain symbolic lifelines representing a class. However, the mined LSCs have limited "symbolic power" – in rough terms, existential quantification of objects within a class can be mined, but universal quantification involving all objects in a certain class satisfying a certain guard is not mined for. Other dynamic techniques to infer invariants in distributed systems have been proposed [38], [39]. The invariants are conditions that characterize system load or relationships between variables from processes in the system. The invariants identified in these approaches have to be satisfied at all states of the system whereas invariants in our approach are used to enhance the accuracy and readability of a state based specification of the system.

Finally, property inferencing in a fixed logical language have been studied earlier, as evidenced by the work on Daikon [28]. Daikon, via dynamic analysis, attempts to infer potential invariants — properties that may hold in a certain control location of a program. In our work, we are inferring guards or logical formulae which capture the set of processes/objects (each with their own independent flow of control) which execute a common action. Furthermore, our inferred guards involve reasoning about execution histories, as opposed to Daikon which infers state-based potential invariants.

VII. Conclusion

In this paper, we have proposed a specification mining framework to mine class level specifications for distributed systems.

At a conceptual level, the main novelty of our work is to move towards mining class level specifications as opposed to object level specifications. This is particularly important for distributed systems with many behaviorally similar processes – as object-level specifications (with concrete processes/objects) are hard to comprehend. Since specification mining aims for behavior comprehension, arguably this makes for a strong case to mine succinct class level specifications.

At a technical level, our work provides a mechanism for inferring rich guards which serve as object selectors, corresponding to events. As shown in the paper, the guards allow for history based constraints and can be extended to handle state-based constraints. Universal/existential quantification captures whether all or any one process satisfying the guard executes the event in question. Our experiments demonstrate that such a rich language of guards allows us to mine highly accurate system specifications for distributed systems.

VIII. Acknowledgement

This work was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 and an NUS research grant R-252-000-403-112.

References

[1] "Message sequence charts," ITU-TS Recommendation Z.120, 1996.

[2] R. Marelly, D. Harel, and H. Kugler, "Multiple instances and symbolic variables in executable sequence charts," in *OOPSLA' 2001*, 2001, pp. 83–100.

[3] A. Goel, A. Roychoudhury, and P. Thiagarajan, "Interacting process classes," *TOSEM*, vol. 18, no. 4, 2009.

[4] R. Marelly, D. Harel, and H. Kugler, "Multiple Instances and Symbolic Variables in Executable Sequence Charts," in *OOPSLA*, 2002.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *IEEE TSE, 27(2):125*, 2001.

[6] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *In Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02), pages 4–16*, 2002.

[7] D. Lo, S.-C. Khoo, and C. Liu, "Mining temporal rules for software maintenance." *JSME*, vol. 20, no. 4, pp. 227–247, 2008.

[8] A. Roychoudhury, A. Goel, and B. Sengupta, "Symbolic message sequence charts," in *ESEC-FSE*, 2007.

[9] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das, "Perracotta: Mining temporal API rules from imperfect traces." in *ICSE*, 2006.

[10] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Mining message sequence graphs," in *ICSE*, 2011.

[11] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] A. V. Raman and J. D. Patrick, "The sk-strings method for inferring pfsa," in *In Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97)*, 1997.

[13] D. Lo and S.-C. Khoo, "SMArTIC: Towards building an accurate, robust and scalable specification miner." in *SIGSOFT FSE*, 2006.

[14] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *KDD*, 2010.

[15] NASA, "Center TRACON Automation System (CTAS)," //www.aviationsystemsdivision.arc.nasa.gov/research/foundations/sw_overview.shtml.

[16] ——, "CTAS Weather Control Requirements," //scesm04.upb.de/case-study-2/requirements.pdf.

[17] "RFC 3261 - Session Inititation Protocol." //www.ietf.org/rfc/rfc3261.txt/.

[18] "KPhone." //sourceforge.net/projects/kphone.

[19] "Opensips." //www.opensips.org/.

[20] "Jive software," //www.igniterealtime.org/projects/openfire/.

[21] "Jeti. Version 0.7.6 (Oct. 2006)." //jeti.sourceforge.net/.

[22] "Pidgin." //www.pidgin.im/.

[23] "XEP-0045: Multi-User Chat," //xmpp.org/extensions/xep-0045.html.

[24] G. Ammons, R. Bodik, and J. R. Larus, "Mining Specification," in *POPL*, 2002.

[25] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, 2009.

[26] S. F. Sharon Shoham, Eran Yahav and M. Pistoia, "Static specification mining using automata-based abstractions," in *In proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07), pages 174-184, London, United Kingdom*, 2007.

[27] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications." in *ESEC/SIGSOFT FSE*, 2007.

[28] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE TSE*, vol. 27, no. 2, pp. 99–123, 2001.

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

[30] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: better together," in *ASE '10*, 2010.

[31] M. Gabel and Z. Su, "Symbolic mining of temporal specifications." in *ICSE*, 2008.

[32] H. Raffelt, B. Steffen, and T. Berg, "Learnlib: a library for automata learning and experimentation," in *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, ser. FMICS '05, 2005, pp. 62–71.

[33] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE Trans. Software Eng.*, 2005.

[34] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE TSE*, vol. 32, no. 9, pp. 642–663, 2006.

[35] R. Oechsle and T. Schmitt, "Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi)," in *Revised Lectures on Software Visualization, International Seminar*, 2002, pp. 176–190.

[36] A. Rountev and B. Connell, "Object naming analysis for reverse-engineered sequence diagrams," in *ICSE*, 2005.

[37] D. Lo and S. Maoz, "Mining Symbolic Scenario-Based Specifications," in *PASTE*, 2008.

[38] G. Jiang, H. Chen, and K. Yoshihira, "Efficient and scalable algorithms for inferring likely invariants in distributed systems," *IEEE Trans. Knowl. Data Eng.*, 2007.

[39] M. Yabandeh, A. Anand, M. Canini, and D. Kostic, "Finding almost-invariants in distributed systems," in *SRDS*, 2011.