Mining Behavioral Specifications for Distributed Systems

Sandeep Kumar
National University of Singapore
sandeep@comp.nus.edu.sg

ABSTRACT

The effort and coordination necessary for the creation and maintenance of software specifications result in their exclusion from real life software engineering processes. This exclusion comes at a high price as the absence of well documented specifications make the maintenance and reuse of software difficult and error-prone. As part of our research, we develop automatic approaches to infer high-level software specifications. Specifically, we study the specification inference problem in the context of distributed systems and advance mining techniques to tackle concurrency and system scalability. Our inferred specifications can uniquely capture interactions between several concurrent processes in the system thereby enabling better comprehension of overall system behavior. Our technique for inferring class level specifications make the mined specifications more readable, accurate and less susceptible to changes in system configurations.

1. PROBLEM AND MOTIVATION

The process of software maintenance is responsible for a large fraction of the costs and effort involved in the development and use of software systems. Software specifications have a central role in ensuring efficiency and productivity during this phase of the software life cycle. Various stakeholders in a software project (developers, testers, users etc.) turn to the specification for a mutual understanding of expected program behavior. In software, complexity managed by designing logically separate objects or processes that interact with each other in a manner that is mutually agreed upon. This separation is explicit in distributed software systems in which physically separate components interact over communication channels. A specification of how processes or objects should interact under various use case scenarios is essential for proper maintenance of such systems. This is especially true, when the interacting objects or processes are programmed and maintained by independent teams or businesses. The specification of interaction protocols helps to avoid ambiguity; making the addition of new features and the upgrade of components easier and less error prone. They also act as a record or contract capturing the terms of agreement between developers of the interacting components.

Although the importance of software specifications is constantly advocated, in reality, they are seldom granted priority and usually found to be out-of-date or entirely absent. Specifications created during early phases of the project become obsolete as they are not updated in the wake of requirement and design changes. In many distributed systems, communication standards are documented informally in natural language, whereby many aspects are left open to interpretation. In other cases, such standards are never documented but arrived at through email exchanges and discussion forums. This state of affairs motivates our research into techniques that automatically generate high level software specifications. In practice, correctness of software (including distributed software) is ensured through rigorous testing and multiple iterations of bug fixing. *We propose techniques that observe how "correct" distributed software execute test cases and infer a specification of execution scenarios, with each behavior expressed in terms of the interactions between components*. The observed executions may involve a large and varying number of concurrently executing active objects or processes. Summarizing observed behavior into a generic and readable specification is non-trivial and challenging. Certain actions from concurrent processes may interleave in a non-deterministic manner. The number of actual processes or objects in the system may have non-trivial dependencies to input and environmental parameters. Fortunately, in most cases, processes can be grouped into a few sets of behaviorally identical processes called process classes or simply classes and system behavior can be conveniently described as *class-level specifications* [21, 11, 22]. As class-level specifications hide specifics regarding the actual objects involved in the execution, they are more generic and less complex.

We note that "mined" specifications can enhance program comprehension and help with the subsequent evolutionary process of the software. Development teams can also apply these methods when faced with the task of integrating legacy systems or third-party software to get a quick appreciation of their behavior in the absence of other formal or informal documentation. We envision that our research contributions will enable the development of a framework that will assist software engineers with the automatic creation and continued maintenance of behavioral specifications. We also investigate how mining techniques can directly assist in comprehending the major changes that form part of software's evolution.

2. BACKGROUND AND RELATED WORK

In software systems, autonomous components collaborate with each other by direct invocation or, as in distributed computing models, through message passing to achieve the desired computational tasks. The manner in which various components collaborate is key to understanding the functioning of these systems. In telecommunications systems, interaction protocols are expressed in the form Message Se-

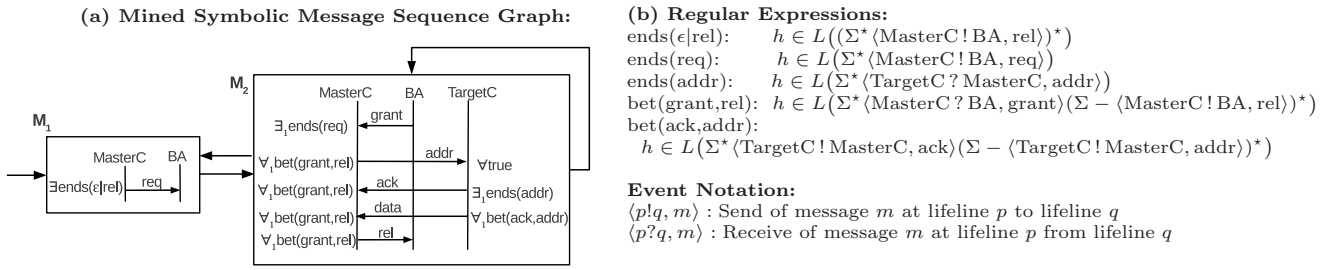

Figure 1: Class-level specification of centralized bus arbitration protocol

quence Charts (MSC) [1]. Sequence diagrams are part of UML standards and commonly used in the design of object oriented software. MSCs, like sequence diagrams, specify behavioral scenarios by depicting the order of communication between two or more processes. As these specifications carry a 'broad picture' and are visual in nature, they become a natural means by which developers and users can understand and reason about the functioning of the system as a whole. We select a high-level variant of the MSC standards called *Symbolic* Message Sequence Graphs (SMSG) [28] as the language of mined specifications, which are directed graphs whose vertices are labelled with *Symbolic* Message Sequence Charts (SMSCs) that represent class-level specifications of component interaction snippets.

2.1 Specification Language

Figure 1 shows the SMSG specification of a simple bus arbitration protocol that 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 two basic SMSCs: M_1 and M_2. Syntactically, SMSCs contain a set of vertical lines, or lifelines, to represent classes. Lifelines contain a sequence of internal or external events. Class level interactions are depicted by horizontal arrows connecting the send events of one lifeline to receive events in another. Events on a lifeline may contain *guards*, which are quantified formula regarding object variables or process execution history, specified to constrain the manner in which the class level interactions may be realized in a concrete way. In our example, guards in the SMSCs contain predicates of the form $bet(X, Y)$ or $ends(X)$, where X and Y range over action labels. Figure 1(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. Apart from these predicates, the guards also contain a quantifier — one of $\forall$, $\forall_k$, $\exists$ or $\exists_k$. Quantifiers specify how many of the objects satisfying the predicate may/should be involved in the interactions. The universal quantifier $\forall$ require all objects satisfying the predicate to be involved in the interaction. The most restrictive quantifier, $\forall_k$, requires not only that all objects satisfying the predicate are to be involved, but also that there be exactly k such objects. The existential quantifiers require at least one ($\exists$) or any k objects ($\exists_k$) that satisfy the predicate to participate.

The SMSC M_1 specifies the request phase of the protocol. The MasterC class contains a single event labelled with the guard $\exists\, ends\,(\epsilon \mid rel)$. The guard ensures that either the master device is making a request for the first time (*i.e* $h = \epsilon$, the empty string), 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 ($\exists_1 ends(addr)$). The master proceeds to read from the device that responded and relinquishes control by sending the *rel* message to the arbiter. The system proceeds to process pending requests (loop to M_2) or accepts new requests in basic SMSC M_1.

2.2 Related Work

Advances in the area of formal software verification have motivated a lot of recent work into techniques that automatically create specifications . Several approaches, broadly referred to as *specification mining*, have been shown to succeed in mining specifications in diverse forms — ranging from program invariants [9] to state machines [4] and temporal properties [18].

Many techniques are geared towards identifying patterns in the invocation order of Application Programming Interfaces (API) methods based on examples of usage by client programs[3, 16, 25, 30]. The patterns are then represented in the form of a finite state machine (FSM) which represents the set of correct behaviors the system can execute. In systems having concurrently executing processes, there can be an explosion in the number of states making the mined specifications imprecise and difficult to read. It is also difficult to understand system behavior when separate specifications are mined from each component as the context under which components perform certain actions is lost. *Our work represents a fundamental shift from existing approaches in that it analyses collective behavior of concurrent processes.* This enables us to mine specifications in the language of Symbolic Message Sequence Graphs (MSG).

Techniques have been developed to mine statistically significant patterns which can be interpreted as temporal rules [17, 29, 2, 32, 18, 10]. While temporal rules are useful for formal verification and creating test cases for other deployments of the API, it is difficult to gain a quick understanding of system behaviors without piecing together several rules. Efforts have been made in program visualization by con-

structing UML sequence diagrams from dynamic executions [6, 24]. 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.* [27], perform a static inter-procedural analysis to reverse engineer UML Sequence Diagrams from programs. We base our specification discovery on dynamic analysis of systems. This allows us to concentrate on the set of execution scenarios that are realized during program execution. Mining specifications of scenario based specifications from source code alone is challenging as the valid program paths and possible system configurations must be estimated through imprecise static analysis.

Lo et. al. in [19] mine for Live Sequence Charts (LSC) that contain symbolic lifelines representing a class. However, the mined LSCs have limited "symbolic power" in comparison to SMSGs – 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 [12, 31]. The invariants are conditions that characterize system load or relationships between variables from processes in the system. These invariants have to be universally satisfied by all states of the system. In contrast, our approach infers invariant properties, expressed as quantified formulas, of class-level interactions at specific states of the system.

Finally, property inferencing in a fixed logical language has been studied earlier, as evidenced by the work on Daikon [8]. Daikon, via dynamic analysis, attempts to infer potential invariants — properties that may hold in a certain control location of a program. In contrast with existing applications of Daikon, *we extend its basic principle to infer guards that reason about execution histories.*

3. APPROACH

Traces that are collected by executing instrumented versions of the subject program becomes the main input data for our mining process. An execution trace is a time ordered series of events which identify the action that is performed, the process performing it and the counterpart process (in external events). Figure 3 describes the main stages involved in mining class-level specifications. Traces are first converted to a partial order representation called *dependency graphs*. Events from concrete objects are then transformed to *class-level events*. Dependency graphs are broken down into sequences of *basic MSCs*. The MSC sequences is used by applying automaton learning techniques to construct an aggregating MSG. Finally, a guard inferencing approach is used to produce the output specification.

Capturing Partial Order: In the initial stage of mining, traces are transformed into a partial order representation called trace dependency graphs. Dependency graphs are acyclic directed graphs whose vertices represent events in the trace and edges their ordering. The chronological order of events within each process is maintained by a minimal set of directed edges. A set of edges from send events to their respective receive events capture the causal relationship between these events. All other event ordering in the trace is discarded and not considered to be important to specify the scenario represented by that trace. In effect, edges in the dependency graphs capture the "happened before" relationship defined by Lamport [15].

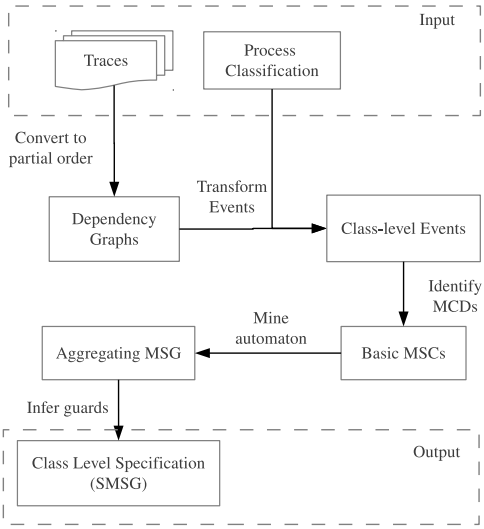

Figure 3: Overview of mining procedure

Transforming Events: Trace dependency graphs contain concrete events that have concrete process/object labels. We use the *process classification* input, which tells how objects should be grouped into classes, to transform events in the dependency graphs into class-level events. Class-level events record event related data as two separate portions: abstract and concrete. The abstract portion contains only the class names of the participating objects and the label of the action that is performed. In the concrete portion, class-level events record the *class-state* (vector containing state of every member object of the class) prior to event execution and also the precise set of participating objects, referred to as the *selection* of objects. Only the abstract portion of the class level events are used to infer the basic structure of the output SMSG model (by algorithms for identifying basic MSCs and automaton learning). The concrete information is accumulated in events of the mined abstract model and preserved for the final guard inference phase. If a group of concrete objects of the same class concurrently execute the same action, those concrete events can be grouped into a single class-level event having the set of participating concrete objects as the object selection. A concrete event which cannot be grouped with others is represented by a class-level event having a selection of just one object. Each dependency graph of concrete events is transformed, grouping similar events when possible, into a dependency graph consisting entirely of class-level events. Process states can be captured by recording the precise state of the memory and registers (or simply values of variables) in the trace events. In our implementation, we approximate the state of a process by its local *execution history* — the exact sequence of events executed by that process before executing the current action.

Identifying Basic MSCs: Our next step is to break down dependency graphs into smaller blocks that are likely to form basic MSCs. There are many possible ways to break down a graph into fundamental blocks. If the dependency graphs are broken down too finely, it loses the power to capture concurrent execution in the form of weak partial order relationships. If broken down coarsely, we lose the ability to identify recurring portions and thereby infer a minimal graph based model. We strike a balance between these con-

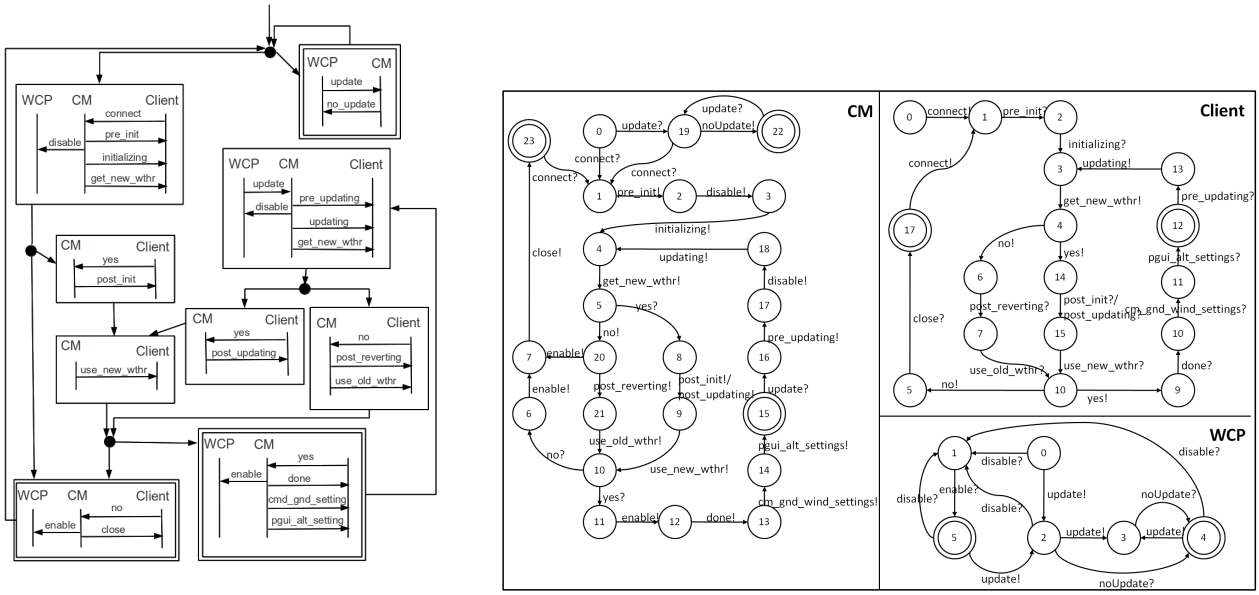

Figure 2: The Mined MSG for CTAS (top) and the learnt automata for individual processes

Table 1: 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

flicting requirements by introducing the notion of *Maximal Connected Dependency Graphs*(MCDs). MCDs are portions of the trace dependency graphs, of maximal size, that appear (as per input data) to function as indivisible basic blocks. We have developed an efficient algorithm to identify a set of MCDs for a given set of dependency graphs [14]. By identifying MCDs and breaking down dependency graphs into a concatenation of these MCDs, we, in essence, represent the trace set as strings of basic MSCs.

Automaton Learning: Several methods have been proposed to approximately identify a regular language based on a set of representative strings from that language [5, 7]. SMSGs can be viewed, at a high level, as a regular language of strings formed from an alphabet consisting of its basic SMSCs. As we have represented our traces in the form MCD strings, we can use automaton learning algorithms to construct an automaton that emits MCDs at its transitions. We used the *sk-string* algorithm for identifying probabilistic automata to mine for such an automaton [26]. This Mealy model machine is transformed into a Moore model, in which each state has a corresponding basic MSC. During the automaton learning phase, portions of the input strings are folded or merged with other strings based on certain matching heuristics. We ensure that concrete information in class level events contained within the strings is accumulated during these merge operations. At the end, class-level events of the basic MSCs will contain a set of accumulated class-state and selection information. The state machine, with basic MSCs as output at states, and accumulated concrete information in the MSCs' events is referred to as an aggregating Message Sequence Graph or aggregating MSG.

Inferring Guards: We devise a novel guard inference technique that operates on the aggregating MSG events to produce an SMSG. For inferencing, a set of templates for predicates, such as those based on regular expressions described in Figure 1(b), is used as the basis. These templates are instantiated using events/variables pertinent to the input trace set to form a set of predicates. From this, a set of candidate predicates is short-listed by eliminating any predicate that is not satisfied by the accumulated set of selections and class-states. A single predicate is selected from the candidates based on heuristics such as its simplicity and discriminating power. If the selected predicate exactly discriminates the selected objects from non-selected objects in the accumulated concrete data then a universal quantifier is inferred. Otherwise an existential quantifier is inferred. The quantifiers are augmented with a fixed integer k ($\exists_k$ or $\forall_k$) if such an inference is supported by the trace set.

In our implementation of this strategy, we find a small set of simple regular expression based templates to be sufficient for expressing behavior in the systems that we studied. We instantiate these templates based on the trace alphabet to form a set of basic predicates. The basic predicates are tested, through a single pass, against the execution histories in the accumulated class-states. We also combine pairs of basic predicates using logical operations such as negation, disjunction and conjunction to form a larger set of composite predicates. The details of the inferencing strategy, the templates and heuristics used is to appear in [13].

Evaluation: We evaluate our mining approach, like past techniques to mine finite-state-machines [16, 20], by comparing the language implied by the mined specification (MS) against that of a manually constructed "correct" specification (CS), quantifying their proximity in terms of precision

and recall in the following manner:

$$\text{precision} = \frac{\#\text{ of strings generated by } MS \text{ and accepted by } CS}{\text{Total } \#\text{ of strings generated by } MS}$$

$$\text{recall} = \frac{\#\text{ of strings generated by } CS \text{ and accepted by } MS}{\text{Total } \#\text{ of strings generated by } CS}$$

As behavioral models may generate infinitely many sequences, some of infinite length, only a finite set of test sequences are generated in practice. For comparisons involving automata, the generated strings are traces collected by execution of the models. The execution of a set of automata (one for each process) is performed by simulating their asynchronous communication over bounded message buffers. For comparisons between MSG based specifications, concrete MSCs, rather than traces are used. Concrete MSCs are obtained from SMSGs, by first concatenating basic SMSCs along accepting paths and then generating valid concrete realizations (with a bound on objects per class) from the concatenated SMSC.

Mining Evolving Specifications: Comprehending the changes between software versions is essential for debugging and testing. Traditionally, changes between versions is understood by tracking the bugs fixed or new features added. When such information is not available, one has to resort to analyzing files affected or code level differencing. High-level specifications such as SMSGs forms an alternative to visualize changes between software versions. As no single party can be expected to have in-depth understanding of all parts of the software, visualizing changes at the high-level specification can be more intuitive. Model-differencing techniques have been used to comprehend changes across versions. As we have argued, behavioral specifications such as SMSGs are not readily available for each software version (as opposed to models such as class diagrams which are easier to reverse engineer). Moreover, to the best of our knowledge, there are no differencing methods that can be directly applied to models such as SMSGs which are a combination of scenario based and state based specifications.

Our present research looks into methods to use specification mining for the purpose of program change comprehension. Existing techniques for mining takes traces from a single program version and produces a specification of that version. We look to extend these techniques to analyze two sets of traces collected from separate versions of the program. Using a differencing algorithm on dependency graphs, we detect likely changes within basic MSCs extracted from the trace sets. After identifying such low-level differences, we merge the trace sets by hiding program versions as concrete information within the trace events. Once the high-level models are mined, we detect model differences by analyzing accumulated version information in the aggregating model. Our preliminary experiments with this approach on traces collected from different versions of programs, executing identical test suites, show that major changes such as modifications in the invocation sequences, state transitions etc. can be intuitively visualized at the SMSG model level.

4. RESULTS AND CONTRIBUTIONS

We consider the following distributed systems as case studies for evaluation: (a) "Center TRACON Automation System" [23] 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. Details regarding these experiments are available in [13, 14]. We perform comparison of our scenario based model against mined automata. For this comparison, we considered specific instances of the systems to limit 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. The mined MSG specifications were found to have better accuracy when compared to the automata based specifications [14]. The MSG specifications were also found to be more concise and easier to comprehend. The mined concrete MSG and mined automata for the CTAS system with a single client is as shown in Figure 2.

Table 1 tabulates the precision, recall and F_1 measure of the mined concrete MSGs and the mined SMSGs. 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. The assessment clearly shows 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. In certain cases, the mined SMSGs have better precision. This can be attributed to the presence of guards which constrain process actions based on previous execution history and not just the present state as per the concrete model.

We have proposed a specification mining framework to mine class level specifications for distributed systems. Related research efforts in specification discovery, have either been limited to the analysis of state based specification of isolated components or have targeted universal rules or invariants in the system. We see that even in fairly simple distributed systems, comprehension of global behavior is extremely challenging from such specifications. Our research moves specification discovery to sequence diagram based specifications, from which interactions between components can be clearly visualized. As the system scenarios are broken down to a few basic MSCs that are composed in a graph form, complete system behavior can be appreciated by reading these basic MSCs. Such specifications have an informal touch as their structuring is similar to that used in informal documentation written in plain English.

Another important novelty of our work is its move towards mining class level specifications as opposed to object level specifications. This is particularly important for keeping the specifications of distributed systems, usually having large number of behaviorally similar processes, concise and readable. Since specification mining aims for behavior comprehension, arguably this makes for a strong case to mine succinct class level specifications.

At a technical level, we developed the notion of MCDs and provide an efficient method to detect them from trace dependency graphs. This has allowed us to apply existing automaton learning techniques to mine scenario based models. We have also provided a mechanism for inferring rich guards which serve as object selectors, corresponding to events. Our experiments demonstrate that such a rich language of guards allows us to mine accurate system specifications for distributed systems. These technical contributions can easily be extended to other kinds of software systems such as embedded systems or object oriented programs, where MSC or sequence diagram based specification is commonly used.

5. REFERENCES

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

[2] 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.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining Specification. In *POPL*, 2002.

[4] 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.

[5] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE TOC*, 21:591–597, 1972.

[6] L. C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE TSE*, 32(9):642–663, 2006.

[7] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *TOSEM*, 7(3), 1998.

[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE TSE*, 27(2):99–123, 2001.

[9] 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.

[10] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, 2008.

[11] A. Goel, A. Roychoudhury, and P. Thiagarajan. Interacting process classes. *TOSEM*, 18(4), 2009.

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

[13] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Inferring class level specifications for distributed systems (to appear). In *ICSE*, 2011.

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

[15] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

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

[17] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *KDD*, 2007.

[18] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *JSME*, 20(4):227–247, 2008.

[19] D. Lo and S. Maoz. Mining Symbolic Scenario-Based Specifications. In *PASTE*, 2008.

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

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

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

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

[24] 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*, pages 176–190, 2002.

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

[26] 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.

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

[28] A. Roychoudhury, A. Goel, and B. Sengupta. Symbolic message sequence charts. In *ESEC-FSE*, 2007.

[29] H. Safyallah and K. Sartipi. Dynamic Analysis of Software Systems using Execution Pattern Mining. In *ICPC*, 2006.

[30] 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.

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

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