THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, Singapore 117590

TRC8/07

Mining Modal Scenarios from Program Execution Traces

David LO, Shahar Maoz and Siau Cheng, KHOO

August 2007

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Mining Modal Scenarios from Program Execution Traces

(Technical Report Version)

David Lo[†], Shahar Maoz[‡] and Siau-Cheng Khoo[†]

[†]Department of Computer Science, National University of Singapore,
[‡]Department of Computer Science and Applied Mathematics,
The Weizmann Institute of Science,
dlo@comp.nus.edu.sg,shahar.maoz@weizmann.ac.il,khoosc@comp.nus.edu.sg

Abstract. Specification mining is a dynamic analysis process aimed at automatically inferring suggested specifications of a program from its execution traces. We describe a novel method, framework, and tool, for mining inter-object scenario-based specifications in the form of a UML2-compliant variant of Damm and Harel's Live Sequence Charts (LSC). LSC extends the classical partial order semantics of sequence diagrams with temporal liveness and symbolic class level lifelines, in order to generate compact and expressive specifications. The output of our algorithm is a sound and complete set of statistically significant LSCs (i.e., satisfying given thresholds of support and confidence), mined from an input execution trace. We locate statistically significant LSCs by exploring the search space of possible LSCs and checking for their statistical significance. In addition, we use an effective search space pruning strategy, specifically adapted to LSCs, which enables efficient mining of scenarios of arbitrary size. We demonstrate and evaluate the utility of our work in mining informative specifications using a case study on Jeti, a popular, full featured messaging application.

Categories and Subject Descriptors: D.2.1[Software Engineering]: Requirements/Specifications–Tools; D.2.7[Software Engineering]: Distribution, Maintenance,and Enhancement–Restructuring, reverse engineering and reengineering
General Terms: Algorithms, Design, Experimentation
Keywords: Specification Mining, UML Sequence Diagrams, Live Sequence Charts

1 Introduction

Analyzing the behavior of software systems, in order to aid program comprehension, reduce their maintenance costs, and improve their quality, is a complex and challenging task. Having incorrect, incomplete, or outdated documented specifications, as a result of short time-to-market constraints, changing requirements, and poorly managed product evolution, reduces comprehension of the code base, increases maintenance costs, and adds challenges towards verification of their correctness. One approach to address this challenge is to automatically

infer specifications of a system from its execution traces by a dynamic analysis process referred to as *specification mining* (see, e.g., [8, 28]).

In this work, we focus on mining specifications of reactive systems, discrete event systems which maintain ongoing interaction with their environment, and on their behavioral specification using inter-object scenarios. Scenarios, depicted using variants of sequence diagrams, are a popular means to specify the inter-object behavior of systems (see, e.g., [16]), are included in the UML standard, and are supported by many modeling tools. In particular, we are interested in modal scenarios presented using a UML2 compliant variant of Damm and Harel's Live Sequence Charts (LSC) [11, 18], which extends the partial order semantics of sequence diagrams with universal and existential modalities and allows symbolic class level lifelines, resulting in compact and expressive specifications. The popularity and intuitive nature of sequence diagrams as a specification language in general, together with the additional unique features of LSC, motivate our choice for the target formalism of our miner. Moreover, the choice is supported by previous work on LSC (see, e.g., [21, 23, 29]), which can be practically used to visualize, analyze, manipulate, test, and verify the specifications we mine.

Our algorithm mines statistically significant LSCs of arbitrary size from program traces. Statistical significance is based on satisfaction of minimum thresholds of support and confidence (metrics adopted from data mining [15]). The algorithm leverages research in the pattern mining domain where mining is modeled as a search space exploration and efficiency is improved greatly by performing effective search space pruning strategy. The following sections describe our target specification language (i.e., live sequence charts), our mining algorithm, a case study, and a short discussion of related work. We conclude with a summary of our contributions and directions for future work.

2 Modal Scenarios

We use a restricted subset of the LSC language. An LSC includes a set of instance lifelines, representing system's objects, and is divided into two parts, the *pre-chart* ('cold' fragment) and the *main-chart* ('hot' fragment), each specifying an ordered set of method calls between the objects represented by the instance lifelines. A universal LSC specifies a *universal liveness requirement*: for all runs of the system, and for every point during such a run, whenever the sequence of events defined by the pre-chart occurs (in the specified order), eventually the sequence of events defined by the main-chart must occur (in the specified order). Events not explicitly mentioned in the diagram are not restricted in any way to appear or not to appear during the run (including between the events that are mentioned in the diagram). The semantics of LSC is comparable to that of various Temporal Logics [22]. For a thorough description of the language and its semantics see [11].

Syntactically, instance lifelines are drawn as vertical lines, pre-chart (main-chart) events are colored in blue (red) and drawn using a dashed (solid) line. LSCs can be visualized and edited within standard UML2 compliant modeling tools (e.g., IBM Rational Software Architect [3]) using the *modal* profile [18].

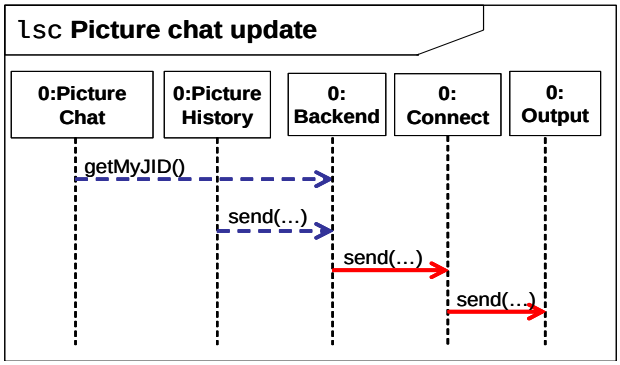

Fig. 1. Mined LSC: `Picture chat update`

Fig. 1 shows an example LSC (adopted from the case study described in Sec. 4). This LSC specifies that "whenever `PictureChat` calls the `Backend` method `getMyJID()`, and sometime in the future the `PictureHistory` calls the `Backend` method `send()`, eventually the latter must call the `send()` method of `Connect` and `Connect` must call the `send()` method of `Output`". Note that if the pre-chart begins but never completes (or the order of events violates it), the main-chart does not have to occur and there are no other restrictions on the order of the events appearing in it. In the example, if the first method `getMyJID()` is never called by the `PictureChat`, or if it is called but the next method `send()` never occurs, the subsequent hot methods may occur or not occur unrestrictedly.

3 Mining Framework

The input for the mining algorithm are finite traces consisting of events, where each event corresponds to a triplet: caller object identifier, callee object identifier, and method signature. The output of our algorithm are statistically significant LSCs satisfying user-defined thresholds mined from the traces. We first define the statistical significance metrics considered, namely support and confidence – adopted from data mining [15]. We then describe the mining algorithm.

3.1 Witnesses,Support and Confidence

To relate between LSCs and execution traces we first introduce notions of positive- and negative- witnesses. These are then used to compute the support and confidence values of an LSC with respect to a trace.

We consider traces to be finite words over a finite alphabet of events $\Sigma = \{a, b, c...\}$, where a unique letter corresponds to a unique triplet. We use the symbol $+\!\!+$ to represent the concatenation operator between finite words. An

LSC $L(pre, main)$ defines a word m built from the concatenation of its pre-chart and main-chart finite words, i.e., $m = pre{+}{+}main$. For two words w, u we denote the projection of w onto the alphabet of events appearing in u by w_u. As an example, $abcadb_{ab} = abab$.

A *positive-witness* of a word w with respect to a trace T is defined as a *minimal* subword s of T such that $s_w = w$. Positive-witnesses of an LSC $L(pre, main)$ are the positive-witnesses of the word $pre{+}{+}main$. The set of positive-witnesses of a word w (an LSC L) with respect to a trace T is denoted by $pos(w, T)$ ($pos(L, T)$). Consider the trace $T_1 = eaeebabcedacbccdaaadabe$ as a running example. It includes two positive-witnesses of $L_1 = (ab, d)$, which are the sub-strings *abced* and *acbccd* starting at the 6th and 11th positions of the trace, respectively.

A *negative-witness* of an LSC $L(pre, main)$ with regard to a trace T, is a positive-witness of the word pre that cannot be extended to a positive-witness of L. The set of negative-witnesses of an LSC L with respect to a trace T is denoted by $neg(L, T)$. Using the previous example, T_1 includes two negative-witnesses of L_1, corresponding to the sub-strings *aeeb* and *ab* starting at the 2nd and 21st positions of the trace, respectively.

The semantics of LSC (like most formal specification languages used for reactive systems, e.g., LTL [20]) is originally defined over infinite paths. The traces we consider, however, are, of course, finite, and we do not want the arbitrary truncation of the trace to affect our confidence of the suggested universal liveness requirement specified by the LSC. We therefore need to adapt the semantics of LSC, and specifically, the definition of negative-witnesses, to finite (so called 'truncated') paths using a notion of *strong-negative-witness*. Roughly, a strong-negative-witness is negative because it explicitly violates the order specified by the main part of the LSC and not because it reaches the end of the trace. Formally, a strong-negative-witness of an LSC $L(pre, main)$ with regard to a trace T, is a positive-witness of pre, p, such that for any word w, p cannot be extended to a positive-witness of L over $T{+}{+}w$. The set of strong-negative-witnesses of an LSC L with respect to a trace T is denoted by $strong\_neg(L, T)$. Using the example above, note that the second negative-witness of L_1 in T_1 ends at the end of the trace and is not a strong-negative-witness.

We use the above notions of witnesses to define the statistical *support* and *confidence* metrics for LSC. Given a trace T, the *support* of an LSC $L(pre, main)$, denoted by $\mathrm{sup}(L)$, is simply defined as the number of positive-witnesses of L found in T. The *confidence* of an LSC L, denoted by $conf(L)$, measures the likelihood of a subword in T satisfying pre to be followed by a subword satisfying $main$ or the end of the trace is reached. Hence, confidence is expressed as the ratio between the number of non-strong-negative-witnesses of the LSC and the number of positive-witnesses of the LSC's pre-chart. Formally:

$$\mathrm{sup}(L, T) \equiv_{def} |pos(L, T)|$$
$$conf(L, T) \equiv_{def} \frac{|pos(L,T)| + (|neg(L,T)| - |strong\_neg(L,T)|)}{|pos(pre,T)|}$$

Notation-wise, when T is understood from the context, it can be omitted. Using the previous example, we have $\sup(L_1, T_1) = 2$, $conf(L_1, T_1) = (2 + 2 - 1)/4 = 0.75$.

The support metric is used to limit the extraction of commonly observed interactions. The confidence metric restricts mining of such pre-chart that is followed by a particular main-chart with high likelihood. Note that LSCs with high but imperfect confidence, i.e., less than 1, are also interesting to mine (see, e.g., the notion of imperfect traces [38]), since, in general, these may reveal errors in the program or in the trace generation process (see, e.g., [8, 12, 38]). The support and confidence values for each mined LSC are included in the algorithms output.

3.2 Algorithm

We are now set to describe the basic LSC mining algorithm and sketch its soundness and completeness.

Many previous algorithms used for specification mining, e.g., [38], need to explicitly check *all possible* specifications obeying a certain template. These do not scale for specifications of an arbitrary size since the number of possible specifications is *arbitrarily large*. Rather than checking for *all possible LSCs*, we immediately *prune search spaces* containing statistically insignificant LSCs using the following property.

Property 1 (**Monotonicity of Support**). For a trace T, an LSC $L(pre, main)$, and a word w: $|pos(pre{+}{+}main, T)| \geq |pos(pre{+}{+}main{+}{+}w, T)|$.

Intuitively, the above property means that if a certain LSC does not meet the minimum support threshold, all its extensions will not meet the minimum support threshold.

An outline of the algorithm is given in Fig. 2. Its input includes a trace and thresholds for support and confidence, and its output is a set of LSCs. The algorithm starts by mining a complete set of words, each having the number of positive instances greater than or equal to the support threshold. Next, it continues to compose these words into LSCs meeting the confidence threshold.

The main algorithm is given in procedure `MineLSC`, which calls the procedure `MineSupportedWords` to mine a complete set of words that meet the support threshold (line 1). `MineSupportedWords` calls `MineRecursive` to recursively add events to the current set of words in a depth first fashion. Once an extended word does not meet the support threshold (line 16), we know all its extensions will not meet the support threshold either (from Property 1), and thus we can stop recursing. After the set of words meeting the support threshold is mined, the algorithm continues to compose these words into LSCs meeting the confidence threshold (lines 3-8). A complexity analysis of our algorithm is available in Appendix A.

Our algorithm for LSC mining is sound and complete; i.e., not only all the output LSCs are statistically significant (i.e., meet the support and confidence thresholds), but also all the possible LSCs that are statistically significant are

indeed included in the output. Soundness follows immediately from the algorithm. Completeness follows from the monotonicity property. The formal proof is outside the scope of this paper.

```
Procedure MineLSC
Inputs: TR : Input Trace; min_sup: Min. Sup. Thresh.;
min_conf: Min. Conf. Thresh.
Output: A set of statistically significant LSCs
1:  Let WSet = MineSupportedWords(TR,min_sup)
2:  Let LSCResult = {}
3:  For every word w in WSet
4:    For every prefix pfx of w
5:      Let main = sfx, where pfx ++ sfx = w
6:      Let NewLSC = Create new LSC (pfx,main)
7:      If (conf(NewLSC) ≥ min_conf)
8:        Output NewLSC
```

```
Procedure MineSupportedWords
Inputs: TR : Input Trace; min_sup: Min. Sup. Thresh.
Output: A set of supported words
9:  Let EV = Single-events appearing ≥ min_sup in TR
10: Let WSet = {}
11: For every f_ev in EV
12:   Call MineRecurse(TR,min_sup,EV,f_ev,WSet)
Return WSet
```

```
Procedure MineRecurse
Inputs: TR : Input Trace; min_sup: Min. Sup. Thresh.;
EV: Frequent Events; curW: Current word considered;
WSet: Current set of supported words
Output: Updated supported words set (WSet)
13: Add curW to WSet
14: For every f_ev in EV
15:   Let nxtW = curW ++ f_ev
16:   If (|pos(nxtW,TR)| ≥ min_sup)
17:     Call MineRecurse(TR,min_sup,EV,nxtW,WSet)
```

Fig. 2. Outline of the mining algorithm

The mined set of LSCs is post-processed to identify class level LSCs (see LSC *symbolic instances* [30]). In addition, we provide an array of additional user-guided filters and abstractions to further refine the resulting set of mined scenarios and reduce the complexity of the mining process. The details is available at Appendix B.

4 Case Study

We demonstrate our approach using Jeti [4], a popular full featured open source instant messaging application. We used AspectJ to instrument the application and created trace files by recording interactions between several Jeti clients. Each of the files is approximately 1K events long (consisting of about 120 unique methods and 600 unique events). We also analyzed one long trace of 10k events. In general, the mining time for a 1K-long trace ranged between a few seconds and several minutes on a Pentium IV 3Ghz PC with 2GB memory.

Many statistically significant LSCs were mined, revealing informative scenarios. We refer interested readers to download the complete traces from [5], which also provides details on mining parameters used, runtime required, and mined results for each experiment on the traces. Two of the mined LSCs are highlighted below. Some other mined LSCs are highlighted at Appendix C.

First, a mined LSC involving sending of messages when one client starts communicating with another is shown in Fig. 3 (Top). The scenario starts whenever a user uses the roster tree to select a party to communicate with. Then, the roster tree will initiate the chat and set up the chat window. After several resources and identifiers of communicating parties are obtained, eventually, an initial message is sent via the `Backend/Connect/Output` channel. Second, from traces involving the use of Jeti's group whiteboard, the miner has captured a scenario of drawing a line and sending it to the other chat users (see Fig. 3 (Bottom)).

We have implemented a programmatic translation of the mined suggested LSCs (represented in simple textual format) into UML2 Sequence Diagrams, using the Eclipse UML2 APIs [2] and the *modal* profile [18]. Thus, we viewed selected results from Jeti visually inside IBM Rational Software Architect (RSA) [3] (see Fig. 4). In addition to the visual representation itself, which helped a lot in understanding the mined scenarios, we were able to use RSA to edit and manipulate the mined LSCs, group them into use cases, annotate them, print them, etc.

Finally, we used the S2A compiler [17], developed at the Weizmann Institute of Science, to programmatically compile selected LSCs into (monitoring) *scenario aspects* [29]. These served as scenario-based tests for Jeti and allowed us to 'validate' selected mined LSCs during subsequent executions. The details are available at Appendix C.

5 Related Work

Most specification miners produce an automaton (e.g., [8, 10, 28, 32]). Unlike these miners, we mine a set of LSCs from traces of program executions. Sequence diagrams in general and LSCs in particular specify inter-object behavior, where the different role of each participating object and the communications between the different objects are made explicit.

In [38], Yang et al. present an interesting work of mining two-event temporal logic rules (i.e., of the form $G(a \rightarrow XF(b))$, where G, X, and F are LTL operators [20]), which are statistically significant with regard to a user-defined

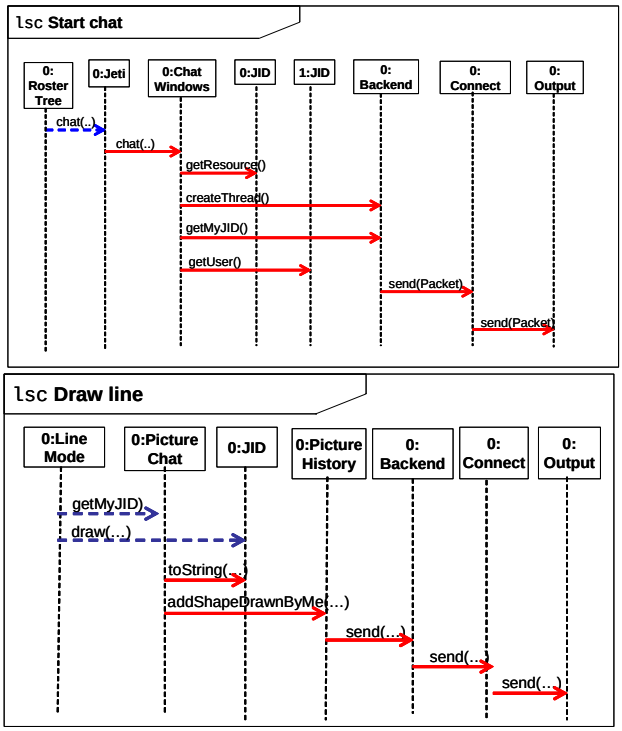

Fig. 3. Mined LSCs: `Start chat` & `Draw line`

'satisfaction rate'. The algorithm presented, however, does not scale to multi-event rules of arbitrary length. To handle longer rules, Yang et al. suggest a partial solution based on concatenation of mined two-event rules. Yet, the method proposed might miss some multi-event rules or introduce superfluous rules that are not statistically significant – it is neither sound nor complete. In contrast, we mine LSCs of arbitrary size; scalability is accomplished by utilizing our own search space pruning strategy adapted from data mining domain. The method is sound and complete as all mined LSCs are statistically significant and all statistically significant LSCs are mined. The semantics of LSC includes ordering constraints which are not considered in [38].

Some work consider mining frequent patterns of software behavior (e.g., [27]). In contrast to our work, the patterns mined do not express inter-object behavior depicted using sequence diagrams.

Many work suggest and implement different variants of reverse engineering of objects' interactions from program traces and their visualization using sequence diagrams (see, e.g., [1, 9]), which may seem similar to our work. Unlike our work, however, all consider and handle only concrete, continuous, non-interleaving, and complete object-level interactions and are not using aggregations and statistical methods to look for higher level recurring scenario patterns; the reverse engineered sequences are used as a means to describe single, concrete, and relatively short (sub) traces in full (and thus may be viewed as 'existential'). In contrast, we

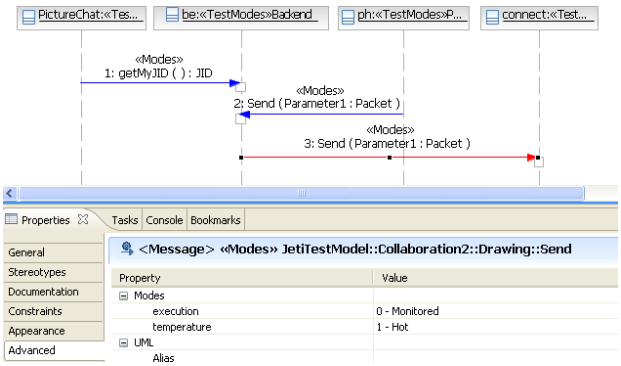

Fig. 4. A mined LSC shown inside IBM RSA.

are looking for universal (modal) sequence diagrams, which aim to abstract away from the concrete trace and reveal statistically significant recurring potentially universal scenario-based patterns, at the object-level as well as the class-level, ultimately suggesting scenario-based system requirements.

6 Conclusion & Future Work

In this paper we have proposed a novel method to mine a sound and complete set of statistically significant modal scenarios from program execution traces. Our work takes advantage of the unique features of LSC as a specification language and of the available tools to visualize and use the mined specifications. The presented case study shows the utility of our approach. Our current method is limited to mining of total order LSCs. In the future, we plan to mine for additional features of sequence diagrams in general, such as explicit partial order, various structural constructs (alternatives, loops, etc.), and functional state invariants.

Acknowledgement We would like to thank David Harel for his valuable comments and advice.

References

1. Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/.
2. Eclipse UML2. http://wiki.eclipse.org/index.php/MDT-UML2.
3. IBM Rational Software Architect. http://www-306.ibm.com/software/rational/.
4. Jeti. Version 0.7.6 (Oct. 2006). http://jeti.sourceforge.net/.
5. LSC Mining: supporting material. http://www.comp.nus.edu.sg/~dlo/lscminer/.
6. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Int. Conf. on Very Large Data Bases*, 1994.
7. R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.
8. G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *POPL*, 2002.
9. L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Trans. Software Eng.*, 32(9):642–663, 2006.
10. J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *TOSEM*, 7(3):215–249, July 1998.

11. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
12. D.Lo and S-C.Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.
13. M. El-Ramly, E. Stroulia, and P. Sorenson. Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces. In *SIGKDD*, 2002.
14. A. Hamid and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ACM SIGSOFT European Soft. Eng. Conference/International Symposium on Foundations of Soft. Eng.*, 2005.
15. J. Han and M. Kamber. *Data Mining Concepts and Techniques, 2nd Ed.* Morgan Kaufmann, 2006.
16. D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.
17. D. Harel, A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal UML sequence diagrams. In *FASE*, 2007.
18. D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 2007.
19. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
20. M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge, 2004.
21. J. Klose, T. Toben, B. Westphal, and H. Wittke. Check it out: On the efficient formal verification of Live Sequence Charts. In *CAV*, 2006.
22. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *TACAS*, 2005.
23. M. Lettrari and J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In *UML*, 2001.
24. J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI*, 2006.
25. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *TSE*, 32(3):176–192, 2006.
26. Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *SIGSOFT FSE*, 2006.
27. D. Lo, S. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *SIGKDD*, 2007.
28. D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.
29. S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.
30. R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *OOPSLA*, 2002.
31. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, 2001.
32. S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, 2001.
33. H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proc. of Int. Conf. on Program Comprehension*, 2006.
34. M. Spiliopoulou. Managing interesting rules in sequence mining. In *Proc. of Euro. Conf. on Principles and Practice of Knowledge Discovery in Databases*, 1999.
35. J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

36. T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.
37. X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. of SIAM Int'l Conf on Data Mining*, 2003.
38. J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.

A Algorithm Complexity

Our algorithm is a novel data mining algorithm belonging to the family of pattern-mining algorithms (*e.g.*, [6, 7, 31]). The worst-case complexity of pattern mining algorithms in general is combinatorial to the length of the dataset (*e.g.*, length of input gene sequence, length of customer purchase history, etc). However, in most cases, the runtime is much better than combinatorial. This is accomplished by employing search space pruning strategy. As proof of their utility, these algorithms have been widely referred to and utilized in the literature of various domains (a simple search via citeseer will show the fact) including software engineering [36, 33, 14, 26, 25, 12].

Our algorithm employs depth first search to the search space of possible LSCs and employs a monotonocity-based pruning strategy (often referred to as *apriori* property). At least at high level, the scalability of our algorithm is on par with techniques in [6, 7, 31]. The three works employ monotonicity-based pruning strategies; the last of the three is an improvement of the second and employs depth first search to the search space.

It should be clearly noted that our algorithm is different from existing pattern mining strategies. The output of our algorithm is statistically significant Live Sequence Charts which is different from those mined by existing work in the pattern mining domain (*e.g.*, association rules [6], sequential pattern [7, 31, 37, 35], etc.). Different outputs imply different ways to count their statistical significance, different search spaces to traverse, and in many cases, different pruning strategies.

The most expensive portion of our algorithm is the MineSupportedWords procedure. With hashing, the complexity of the composition of words to LSCs is only linear to the size of the mined words (compare with association rule mining [6] and sequential rule mining [34]). The worst case complexity of MineSupportedWords is similar to those of sequential pattern miners [7, 37, 35]. The latter two are improvements over the first one but still their worst case complexity are combinatorial to the length of the longest sequence.[1] They improve the earlier on typical runtime by improved search space pruning strategy. Given a trace of length n with k unique events, the worst case complexity of our algo-

[1] Our algorithm can also be generalized to mine from a set of traces rather than a single trace.

rithm is $O(k^n)$. Considering the maximum length of the mined LSCs is m, the worst case complexity of our algorithm is $O(k^m)$ – typically $m \ll n$.[2]

Our case study shows that in many cases the algorithm runs very fast (i.e., many experiments complete in a few seconds – see our website [5]) due to the effective pruning strategy. We do encounter cases where the algorithm runs too slow. If this is the case, one can employ one of the user-guided filters and abstractions (in Appendix B) to further improve the efficiency. By employing some of the filters and abstractions, the worst case complexity can be made linear to the length of the pattern (e.g., by specifying a predefined pre-chart and further restricting the minimum and maximum length of patterns to be mined).

As a side note, in general pattern mining algorithm utility is not measured by its worst case complexity. Indeed, most pattern mining papers do not discuss worst case complexity. Efficiency is measured by running the algorithm on real data or on benchmark datasets(*e.g.*, [6, 7, 37, 35, 27, 13]). This is the case since the benefit of search space pruning strategies is nullified if the worst possible case (which might not appear in many or most practical cases) is considered (e.g., mining when the support threshold is set to 0).

As a future work, we plan to adapt the work on closed pattern mining (*e.g.*, [37, 35]) and generators (*e.g.*, [24]) for LSC mining. The worst case complexity will still be the same, however the typical runtime will potentially be much improved by employing more effective pruning strategy.

B Filters and Abstractions

In this section we discuss the use of object information and present a series of extensions to the basic mining algorithm.

B.1 Using object information

Mining class-level LSCs Class-level LSCs, following LSC Symbolic Instances [30], allow to specify scenarios at the level of (abstract) classes, and thus enable expressive and succinct scenario-based specifications in an object-oriented context. Syntactically, this is done by labeling instance lifelines with class names rather than specific object names.

Roughly, a class-level LSC specifies a modal scenario that applies to all objects of the classes referenced on its lifelines. However, in a specific instance of the scenario (i.e., in a specific positive-witness), each lifeline binds to a single object throughout the scenario. The semantics of class-level LSCs (as LSCs with symbolic instances) was defined in [30], implemented in the Play-Engine tool [19], and was specifically adopted to support object-oriented inheritance and interface implementation in Java in [17, 29].

[2] Interestingly, considering the length of the longest mined pattern (or word) to be constrained by a *constant*, Li *et al.* state that the complexity of one of the pattern miners (*i.e.*, [37]) to be quadratic ($O(n^2)$) [25].

As the input of our specification mining algorithm is a concrete trace, the basic algorithm mines only concrete object-level LSCs. In order to find class-level LSCs, we employ a process of generalization and aggregation, using the caller and callee identifiers attached to each event on the trace.

Since any positive (negative) witness of a concrete object-level LSC is also a positive (negative) witness of the corresponding class-level LSC, the support and confidence metrics extend naturally to the class-level case as totals. In addition, we compute and present for each class-level LSC mined the maximum support and confidence values over its corresponding concrete LSCs, and also the number of unique concrete corresponding LSCs. We allow the user to set minimum thresholds for these metrics. The miner will filter out class-level LSCs not meeting the thresholds.

Connectivity criterion An LSC may be drawn as a graph whose nodes represent the participating instances and whose directed edges represent method calls (as in a UML2 *Communication Diagram*). We say that an LSC is connected if the resulting graph representation is connected (i.e., if all nodes are reachable from the initial node). While unconnected LSCs may be useful in general, in the context of mining they are probably of little value to the user. We thus check LSC connectivity using a simple depth-first search algorithm, and allow the user to filter out unconnected LSCs, despite their statistical significance. Note that connected sub-graphs of such LSCs, if statistically significant, are not filtered out in the process.

The ability to mine class-level LSCs and use connectivity as a criterion, rely on the fact that object information for caller and callee is *not* abstracted away in our traces. We believe these two features are novel and have important impact on the quality of the results and the usefulness of our approach.

B.2 User-guided filters and abstractions

Although the LSCs mined by the basic algorithm are statistically significant, many of them may still not be of high value to the user. The following lists a series of extensions to the basic algorithm that we have defined and implemented. Some extensions allow the user to use apriori knowledge (or otherwise, knowledge obtained from previous mining sessions) in order to define various abstractions and filters aiming at improving the quality and usefulness of the LSCs mined. Others are motivated by various properties of sequence diagrams in general and modal sequence diagrams in particular. Some of the presented extensions, when used, may also speed up the mining process significantly.

Ignore intra-object method calls. Some of the traced events may refer to method calls from an object to itself. Since LSC typically focus on inter-object scenarios, we allow the user the option to filter out these method calls from the trace, pre-mining. Note that this will *not* remove method calls between two different objects of the same class.

Ignore set. Based on previous knowledge about the system, the user may consider some of the method calls as not interesting. We thus allow the user to

specify a set of method signatures to be ignored. The mining algorithm will ignore the specified methods.

Consider equivalent sets. The user may have previous knowledge which hints that two or more methods correspond to the same abstract concept. We thus allow to specify these using (necessarily disjoint) sets of events. The miner will not distinguish between methods belonging to the same set.

Equivalent consecutive duplicates. In some cases, consecutive repetitions of the same event may not be interesting to the user. We thus allow to specify a list of methods whose consecutive repetitions, if found, should be abstracted away. For example, if method x is in the list, the miner will not distinguish between the words $x, xx, xxx \ldots$ etc.

Disjoint sets. In some cases, the user may know that two or more methods should not appear in the same LSC. We thus allow to provide this information to the miner and have extended the basic mining algorithm to support this requirement.

Predefined pre-charts. We allow the user to specify that one is only interested in mining LSCs whose pre-charts are included in a specific predefined set or use only a predefined subset of the events alphabet.

Removing logically redundant LSCs. Given any two LSCs $M_1(pre_1, main_1)$, $M_2(pre_2, main_2)$ such that
$pre_1{+\!+}main_1 = pre_2{+\!+}main_2$, the one with shorter pre-chart logically entails the other. Therefore, given such LSCs with equal support and confidence values, we keep the one with the minimal pre-chart length and filter out the rest.

Removing sub LSCs. Given any two LSCs $M_1(pre_1, main_1)$, $M_2(pre_2, main_2)$, we say that M_1 is a sub LSC of M_2 iff $pre_1{+\!+}main_1$ is a sub-word of $pre_2{+\!+}main_2$. We allow the user to choose to filter out sub LSCs (given equal support and confidence) and keep the ones with the richer alphabet. Note that if M_1 is a sub LSC of M_2, it is not always true that M_2 logically entails M_1.

Min/max length thresholds. In general, long LSCs convey more information than short ones and are more difficult to identify manually. On the other hand, the longer the LSCs mined, the longer it takes for the mining algorithm to run. We thus allow the user to set minimum and maximum length thresholds. The miner will filter out LSCs which are shorter (longer) than the minimum (maximum) length threshold.

Density. Given a trace containing many repetitions of lock (l) and unlock (u) (abstracting object information in this example), the following LSCs may be found statistically significant: $(l, u), (lu, l), (lul, u)$ etc. Only the first, however, is probably of interest to the user. To distinguish the first from the rest we introduce a notion of density; i.e., the ratio between the number of unique events and the total number of events in the LSC; and allow the user to set a corresponding minimum threshold.

Main-pre ratio. In general, LSCs with shorter pre-chart and longer main-chart are more restrictive and thus more informative. We define the main-pre ratio as the ratio between the length of the main-chart and the length of the pre-chart, and allow the user to set a corresponding minimum threshold.

Finally, the user may choose to sort the output LSCs by their support, confidence, length, or number of participating objects.

C Extended Case Study

In addition to the message sending behavior shown in Fig. 3 (Top), a more generic one specifying more common behavior of messages sent (i.e., "whenever the `Backend` sends a packet to the `Connect`, eventually the latter sends a packet to the `Output`"), was found in another mined LSC (see Fig. 5). The miner discovered both LSCs, and as expected, the more general LSC's support value was much higher than that of the other. This demonstrates that our method not only can extract recurring scenarios, but is also able to distinguish more frequently observed scenarios from relatively rare ones.

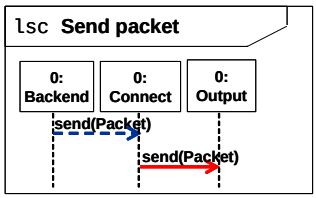

Fig. 5. Mined LSC: `Send packet`

Next, a scenario involving flashing icons, which occurs whenever a message is received by a chat window that is not in focus, is shown in Fig. 6. The scenario starts with messages being received and appended to the user interface (i.e., `ChatSplitPane` located within `ChatWindow`). A check is made to the identity of the incoming message and the icon starts flashing. Eventually, when the user clicks on the flashing window, flashing is stopped. We used this scenario to test the user-guided predefined pre-chart extension described earlier, by providing the miner the following series of events ending in flashing icons being started as additional input, and applying it to a long 10K events trace.

(Caller, Callee, Method Signature)
1. (ChatWindows;ChatWindow;appendMessage)
2. (ChatWindow;JID;equals)
3. (ChatWindow;ChatSplitPane;appendMessage)
4. (ChatSplitPane;titlescroller.Plugin;start)
5. (ChatSplitPane;titleflash.Plugin;start)
6. (titleflash.Plugin;Flash;start)

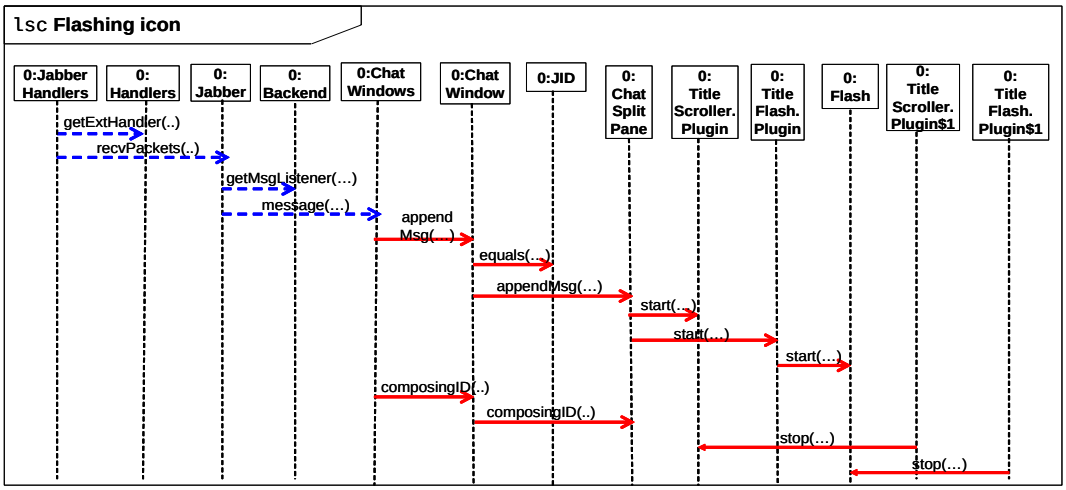

Fig. 6. Mined LSC: `Flashing icon`

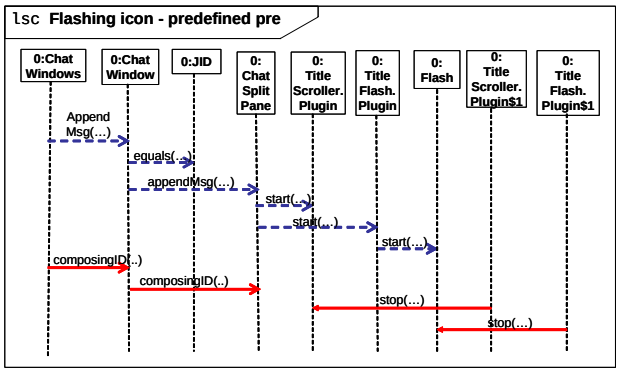

Fig. 7. Mined LSC: `Flashing icon` **(10K long trace)**

Without the additional input, mining this rather long trace took a few hours, however, using the predefined pre-chart, mining was completed within a few minutes. One of the resulting mined LSCs is shown in Fig. 7. The above illustrates the usefulness of the user-guided predefined pre-chart extension and suggests a methodology: the user may mine relatively short traces to find candidate LSCs of particular interest, and then evaluate these LSCs or related ones on much longer traces at a very low computational cost. Finally, we note that we obtained the long trace by tracing the interaction between a number of communicating entities; while we show here only the class-level LSC, many of its corresponding object-level LSCs, bound to different `ChatWindows` were 'active' simultaneously, i.e., one had started flashing before another one had stopped. This shows that our method is able to extract common class-level scenarios from complex traces

where the different corresponding object-level positive-witnesses interleave and thus overlap.

Also interestingly, similar to the result shown in Fig. 3 (Bottom) expressing the scenario of drawing a line, the mining results also included additional very similar LSCs corresponding to drawing of ellipse and rectangle. Indeed, the only difference between these LSCs was the participating classes of the first leftmost lifelines. We thus performed a super-class aggregation resulting in the LSC shown in Fig. 8. Note the abstract class `Mode` referenced on the leftmost lifeline. This mined LSC takes advantage of the semantics of LSC symbolic instances in defining compact and expressive scenarios.

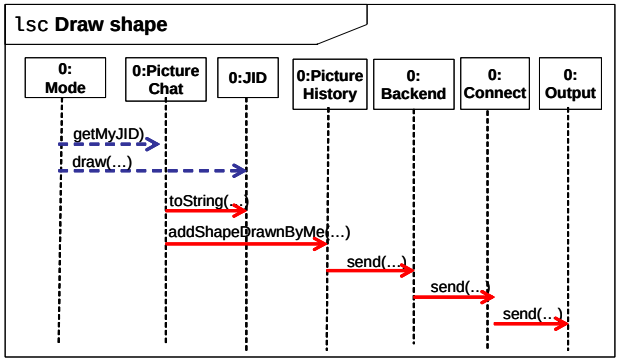

Fig. 8. Mined LSC: Drawing a general shape (`Mode`)

Finally, we also used the S2A compiler [17], developed at the Weizmann Institute of Science, to programmatically compile selected LSCs into (monitoring) Scenario Aspects [29]. The generated scenario aspects traced the application while simulating the progress of each of the previously mined scenarios, and thus served as scenario-based tests for Jeti. This allowed us to 'validate' selected mined LSCs during subsequent executions of Jeti. A log file generated by S2A includes a scenario-based trace where completions (occurrences of positive-witnesses) and violations (occurrences of negative-witnesses) are shown.

For example, we compiled the LSC shown in Fig. 1 discussed previously into a scenario aspect, ran Jeti, and checked that the scenario-based trace produced includes no violations. We then found in Jeti's code the call that corresponds to the first hot method call in the suggested LSC, and changed it so that it will not always occur. In other words, we embedded a bug into Jeti, so that although all updates to the chat picture will still be sent to the `Backend`, some of them will not be further sent to the `Connect` object, i.e., some will not be sent to the remote users on the chat. We were then able to view, in the scenario-based trace produced during subsequent executions (see Fig. 9), the hot violations that have

```
E: 1180527437140 75: void nu.fw.jeti.jabber.Backend.send(Packet)
B: jeti.msdaspects.MUSDAspectJetiTest01[57] lifeline 1 <- nu.fw.jeti.jabber.Backend@2bee2bee
B: jeti.msdaspects.MUSDAspectJetiTest01[57] lifeline 0 <- nu.fw.jeti.plugins.drawing.shapes.PictureChat@2bdc2bdc
C: jeti.msdaspects.MUSDAspectJetiTest01[57] (1,1,0,0) Cold
E: 1180527437140 76: void nu.fw.jeti.jabber.Backend.send(Packet)
B: jeti.msdaspects.MUSDAspectJetiTest01[57] lifeline 2 <- nu.fw.jeti.plugins.drawing.shapes.PictureHistory@76687668
C: jeti.msdaspects.MUSDAspectJetiTest01[57] (1,2,1,0) Hot
F: jeti.msdaspects.MUSDAspectJetiTest01[57] Violation
```

Fig. 9. Excerpt from the scenario-based monitor output generated by S2A. A violation detection in Jeti. Note the events that have occurred, the lifelines' bindings, and the 4-tuples representing cut state changes.

occurred when new packets were sent without sending earlier ones first through the connection.