Specification Mining of Symbolic Scenario-Based Models

David Lo[1]
School of Information Systems
Singapore Management University
davidlo@smu.edu.sg

Shahar Maoz
Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
shahar.maoz@weizmann.ac.il

ABSTRACT

Many dynamic analysis approaches to specification mining that extract behavioral models from execution traces, do not consider object identities which limit their power when used to analyze traces of general object oriented programs. In this work we present a novel specification mining approach that considers object identities, and, moreover, generalizes from specifications involving concrete objects to their symbolic class-level abstractions. Our approach uses data mining methods to extract significant scenario-based specifications in the form of Damm and Harel's live sequence charts (LSC), a formal and expressive extension of classic sequence diagrams. We guarantee that all mined symbolic LSCs are significant (statistically sound) and all significant symbolic LSCs are mined (statistically complete). The technique can potentially be applied to general object oriented programs to reveal expressive and useful reverse-engineered candidate specifications.

1. INTRODUCTION

Software systems are often built with little documented specifications. Specifications are often incomplete and get outdated with time. To address these issues specification mining has been proposed [3]. Specification mining is concerned with extracting behavioral models from program execution traces, and is useful as an aid to program comprehension, bug detection, and maintenance tasks. Many approaches to mining behavioral specifications from execution traces of object oriented programs (e.g., [19, 21, 22, 29, 26, 10, 9, 32, 33]) do not consider objects' identity; object identity information is not recorded during tracing and is ignored during mining. This limits the power of these approaches to general object oriented programs where object instances of many classes, including multiple instances of the same class are created dynamically and interact in the execution.

In this paper we present a dynamic analysis behavioral specification mining approach that considers object identities, and, moreover, generalizes from the concrete objects to their classes while maintaining soundness and completeness. It can thus be applied to general object oriented programs and reveal most expressive and useful reverse-engineered candidate specifications. We do this in the context of *scenario-based specification mining*.

In scenario-based specification mining [25], data mining methods are applied to program execution traces in order to mine strongly observed inter-object universal sequence diagrams in the form of a UML2 compliant variant of Damm and Harel's *live sequence charts* (LSC) [6, 13]. LSC extends classical sequence diagrams with a universal interpretation and must/may modalities (we define the subset of LSC used in our work in Sec. 2). An LSC is composed of two parts: pre- and main-chart; a universal LSC specifies that whenever the pre-chart sequence of events occurs, eventually the main-chart sequence of events must occur.The popularity and intuitive nature of sequence diagrams as a specification language in general, together with the additional unique features of LSC, motivated our choice of the target formalism for our mining approach. Moreover, the choice is supported by previous work on LSC (see, e.g., [16, 18, 27]) which can be practically used to visualize, analyze, manipulate, test, and verify the specifications we mine.

An important characteristic of scenario-based specification mining in the context of the present paper is that object identifiers are *not* abstracted away during tracing and mining. Each event in the trace is a triple of caller, method signature, and callee identifiers. The mining algorithm then uses data mining methods to extract recurrent inter-object scenarios of the form pre/main-chart modulo user defined support and confidence thresholds (see [23, 25]).

In their most basic form, sequence diagram lifelines represent concrete objects. Thus, these diagrams apply only to the inter-object behavior between the specific concrete objects referenced on their lifelines, and their expressive power as specifications (e.g., for testing or as runtime monitors) is therefore limited in this sense.

Symbolic lifelines, representing classes rather than concrete objects, were introduced to LSC in [28]. These allow to present diagrams that apply to *all* objects of the represented classes, and thus enable the definition of most expressive and succinct specifications.

In the context of specification mining, indeed, our input is a trace made of concrete events, but we are not interested in extracting concrete scenarios; instead, we are looking for the more abstract symbolic class-level scenarios, which universally specify the behavior of *all* objects of the referenced classes, and whose concrete instantiations are frequently observed in the trace.

To introduce the problem of mining symbolic scenarios investigated in this paper consider an example, taken from a Space Invaders game application. The following universal scenario, involving the game control, a monster, and a laser beam, holds invariantly, on all runs of the application, and occurs very frequently in a typical execution of the game: "*Firing: whenever the game control calls a monster's fire method, the monster fires a laser beam, and the beam*

[1]The work was partly done while the author was full time at School of Computing, National University of Singapore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

is added to the game control list of entities. Eventually (after the beam hits another entity or moves out of screen), the game control removes the laser beam from the game". Note that many monsters and many laser beams are created dynamically and simultaneously interact during a typical run of the game. Why is it a challenge to extract this scenario from the game's execution traces? First, this scenario happens exactly once for each laser beam object involved; that is, each of the scenario's concrete instantiations, referencing concrete object identifiers, will occur only once (per set of objects per trace); hence, at the concrete objects level it will not be considered a frequently observed property. Second, a naïve abstraction mechanism, which will delete object identifiers from the trace while keeping their class names, would fail to support the extraction of this scenario, as nothing in the abstract trace would indicate that the laser beam that was fired is the *same* laser beam that was eventually removed (in the last sentence of the above description of the firing scenario, note the use of the definite "the beam" rather than the indefinite "a beam"). Fig. 1 shows a correct (left) and an incorrect (right) abstract representations of this scenario.

The main contribution of the present paper is a solution to the above, that is, a sound and complete method to extract class-level symbolic scenarios modulo user defined support and confidence thresholds, while maintaining object identification (i.e., lifeline binding) within each scenario, as required. During mining, suggested concrete scenarios are aggregated according to their class-level images in an isomorphic mapping, without loosing their topology. We call this a *binding preserving abstraction*.

Our work models mining as a search space traversal to identify significant class-level LSCs. A smart pruning strategy, based on an anti-monotonicity property, is employed to cut the search space of insignificant class-level LSCs. To mine class-level LSCs, we reason at the concrete and abstract level simultaneously. It is thus a novel example of a dynamic analysis specification mining work that considers object identities, and generalizes from concrete objects to their classes while maintaining soundness and completeness.

We organize the remainder of the paper as follows. The next section defines the subset of LSC that we consider, the semantics of concrete and abstract LSCs, and the notions of statistical significance, soundness, and completeness. Sec. 3 presents the challenge of mining symbolic scenarios and its solution. In Sec. 4 we illustrate the mining algorithm by means of a detailed example. Sec. 5 discusses additional issues. Related work is discussed in Sec. 6. Sec. 7 concludes and presents some future work directions.

2. CONCEPTS AND DEFINITIONS

We briefly recall the syntax and semantics of live sequence charts, specifically distinguish between concrete and abstract. We then define the statistical metrics of support and confidence used in our work as the basis for evaluating candidate LSC significance.

2.1 Live Sequence Charts

Live sequence charts (LSC) [6] is a formal and expressive extension of classical sequence diagrams. In this study, 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 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). For a thorough description of the language and its semantics see [6, 14]. A UML2-compliant variant of the language using the *modal* profile is defined in [13]. A translation of LSC into various Temporal Logics appears in [17].

Syntactically, instance lifelines are drawn as vertical lines; pre-chart (main-chart) events are drawn using dashed (solid) directed horizontal arrows colored in blue (red). Time goes top to bottom.

An additional important feature of LSC is its semantics of *symbolic instances* [28]. That is, rather than referring to concrete objects, instance lifelines may be labeled with a name of a class and defined as symbolic, i.e., formally representing any object of the referenced class. This allows a designer to take advantage of object-oriented inheritance and create more expressive and succinct specifications.

Note that in an instance of a symbolic LSC, i.e., when the scenario specified in the LSC 'occurs', each lifeline represents the unique single object that binds to it; that is, different symbolic lifelines of the same class necessarily represent different objects of that class. Once an object binds to a symbolic lifeline, it remains bound until the completion of that instance of the LSC. Many instances of the same symbolic LSC where lifelines are bound to different concrete objects may 'exist' simultaneously (see [28]).

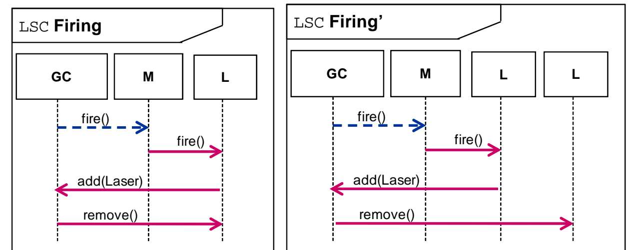

Figure 1: Correct (left) and Incorrect (right) Abstract Representations of `Firing`

We denote an LSC by $L(pre, full)$, where pre denotes the pre-chart and *full* denotes the full chart containing pre and main chart. For concrete LSCs, the pre and *full* correspond to concrete charts; for symbolic LSCs, they correspond to abstract charts.

EXAMPLE 1. *Fig. 1 (left) shows an example LSC taken from the Space Invaders application. The participants in this scenario are a game controller, a monster, and a laser beam. Roughly, this symbolic LSC specifies that whenever a* `gamecontroller` *(of type* `GameController`*) calls the* `fire()` *method of a* `monster` *(of type* `Monster`*), the* `monster` *should call the* `fire()` *method of a* `laser` *(of type* `Laser`*), the* `laser` *should call the* `gamecontroller` `add(Laser)` *method, and eventually, the* `gamecontroller` *will call the* `laser` `remove()` *method.*

2.2 Concrete and Abstract Charts

We consider two domains: concrete and abstract. The concrete refers to the actual low-level runtime information in which we are able to see object information. The abstract domain refers to class names (other abstractions may be possible, see Sec. 7). Hence, we consider two types of events: *concrete* and *abstract*.

A concrete event is a triplet: caller object identifier, callee object identifier, and method signature. An object is uniquely identified by its hash key (by calling System.identityHashCode()) and the class it is instantiated from. The input trace is simply a series of concrete events. An abstract event groups several related concrete events together: it replaces object identity with a higher level of abstraction,

namely, in our case, an object's class name. A simple map from a concrete event to its abstract event is defined as a projection: given a concrete event e, $proj(e)$ returns the abstract event of e, where the caller and callee objects identifiers were replaced by the names of their classes.

A chart (concrete or abstract) is composed of a list of (concrete or abstract) events $\langle e_1, e_2, \ldots, e_n \rangle$, a set of (concrete or abstract) lifelines $\{l_1, l_2, \ldots, l_k\}$ (for simplicity, we draw the lifelines ordered from left to right although the order of lifelines has no semantic meaning), and a binding function *bdg* mapping each event to a pair of lifelines (we omit obvious syntactic well-formedness rules, e.g., that *bdg* binds an object to a lifeline only if the two are of the same class, etc.). More formally:

DEFINITION 2.1 (**Concrete/Abstract Chart**). *A concrete/abstract chart C is a triplet $\langle E, L, bdg \rangle$ where E is a list of concrete/abstract events, L is a set of concrete/abstract lifelines, and bdg is a binding map $bdg : E \rightarrow L \times L$.*

Given a list of concrete events, a *single* set of concrete lifelines is determined by the names of concrete objects involved in the events. Thus, the binding map *bdg* of a concrete chart containing a list of concrete events (and a set of concrete lifelines, each corresponding to a unique concrete object) is trivial.

While a list of concrete events uniquely determines a set of lifelines (up to lifelines order, which has no meaning), a trivial binding function, and thus, a concrete chart, this is *not* the case for abstract events. Rather, given a list of abstract events, one may possibly define more than one (non-isomorphic) sets of abstract lifelines with corresponding bindings (as demonstrated earlier in Fig. 1). This is so because a class corresponding to an abstract caller or callee does not uniquely identify an abstract lifeline in the set, as it may include a number of abstract lifelines corresponding to the same class.

As an example, the binding of a concrete event $e = (oid1, oid2, m())$ in a concrete chart C, may be represented by a pair $\langle l_i, l_j \rangle$ corresponding to the ith and jth lifelines of C. $bdg(e,C)$ is the mapping returning the binding of a concrete event e from a concrete chart C. For example, in Fig. 2, $bdg((A\&12, A\&28, m1())) = \langle 1, 2 \rangle$.

To relate concrete and abstract charts, we propose the notion of *binding preserving abstraction*, essentially an isomorphic mapping between the two.

DEFINITION 2.2 (**Binding Preserving Abstraction**). *Consider a concrete chart $CC = \langle \langle e_1, \ldots, e_n \rangle, \{l_1, \ldots, l_k\}, bdg \rangle$ and an abstract chart $AC = \langle \langle E_1, \ldots, E_n \rangle, \{L_1, \ldots, L_k\}, BDG \rangle$. AC is a binding preserving abstraction of CC iff there exists a mapping abs from the lifelines of CC to the lifelines of AC s.t. $\forall_{1 \leq i,j \leq k}$ and $\forall_{1 \leq v \leq n}$, $bdg(e_v) = \langle l_i, l_j \rangle$ iff $BDG(E_v) = \langle abs(l_i), abs(l_j) \rangle$.*

We denote the binding preserving abstraction of CC by $abs(CC)$. It returns the abstract chart that is 'isomorphic' to CC.

To illustrate the concept of binding preserving abstraction consider Fig. 2. In the figure there are a concrete chart (extreme left, each lifeline corresponds to a separate object instance of class A identifiable via the object hash code) and several abstract charts. Two of the abstract charts are not isomorphic to the concrete chart, and thus do not consist of a binding preserving abstraction.

The above concept is important as there can be more than one symbolic lifeline corresponding to a particular class C in an abstract LSC L. When this is the case this denotes that more than one object instance of the class C participates in L.

Roughly, as explained above, an LSC specifies a universal temporal rule: "Whenever the pre-chart is satisfied, eventually the main-chart of the LSC must be satisfied". Satisfaction of the chart follows the semantics of LSC. We refer to a sub-trace (or a segment of consecutive events in the trace) satisfying the chart C (either concrete or abstract) as an instance of C (as mentioned above, events not appearing in C may appear or not appear in an instance of C, including in between events that do appear in C). A more operational definition is given in our previous work [23].

2.3 Statistical Significance and Guarantee

Our goal is to identify strongly observed properties in the trace. To formalize, we use two statistical metrics commonly used in data mining, namely support and confidence [11]. The support of the chart corresponds to the number of times instances of the pre- and main-chart appear in the trace, and the confidence of the chart corresponds to the likelihood of the pre-chart instance being followed by a main-chart instance in the traces. The support metric is used to limit the extraction to commonly observed interactions, while the confidence metric restricts mining to such pre-charts that are followed by a particular main-chart with high likelihood. Note that LSCs with high but imperfect confidence, i.e., less than 100%, may be interesting to mine too (see, e.g., the notion of imperfect traces [37]), since, in general, these may reveal errors in the program or in the trace generation process (see, e.g., [3, 19, 37]). We refer to charts satisfying the minimum support threshold (min_sup) as being *frequent*. Similarly, we refer to charts satisfying the minimum confidence threshold (min_conf) as being *confident*. A chart satisfying both thresholds is referred to as *significant*.

In addition, we provide a measure of guarantee to the mined LSCs. We refer to these measures of guarantee as statistical soundness (i.e., correctness) and completeness, frequently used concepts in data mining: all mined specification are statistically significant (soundness), and all statistically significant specifications are mined (completeness). Note that these are guaranteed also by systems like Daikon [8] and many data mining systems (e.g., [21, 34, 36]).

3. MINING SYMBOLIC SCENARIOS

A naïve way to mine significant LSCs is to consider all possible combinations of events for their statistics and report those that are significant. Assuming n distinct events, the number of all possible events combinations up to length k is n^k. Clearly, an algorithm that checks the significance of each and every one of these is not feasible (typical traces we consider can have more than 50 distinct events and the LSCs we are looking for are typically made of 4 to 8 events). Instead, we use the user-defined given support threshold, to cut out the sub-search spaces containing insignificant charts. Also, our interest in mining abstract LSCs complicates things further as an abstract LSC may correspond to a set of concrete LSCs. Hence, we need a better mining strategy.

In our study, we model mining as a search space traversal of significant LSCs. We start by finding frequent charts satisfying the minimum support threshold and then compose these charts into LSCs comprising of pre- and main- charts that satisfy the minimum confidence threshold. During the traversal, we employ a smart pruning property to identify sub-search spaces of insignificant LSCs. During mining, we need to reason both in concrete and abstract level to ensure correct identification of significant charts. Below we describe the process in more detail.

We employ the following anti-monotone property on support values, shown in Property 1, to cut sub-search spaces of insignificant LSCs. The property is an extension of the property used in [25] from concrete to abstract charts. It states that a longer chart M' having another shorter chart M as a prefix will have an equal or lower support as that of M. A proof sketch is available in the tech-

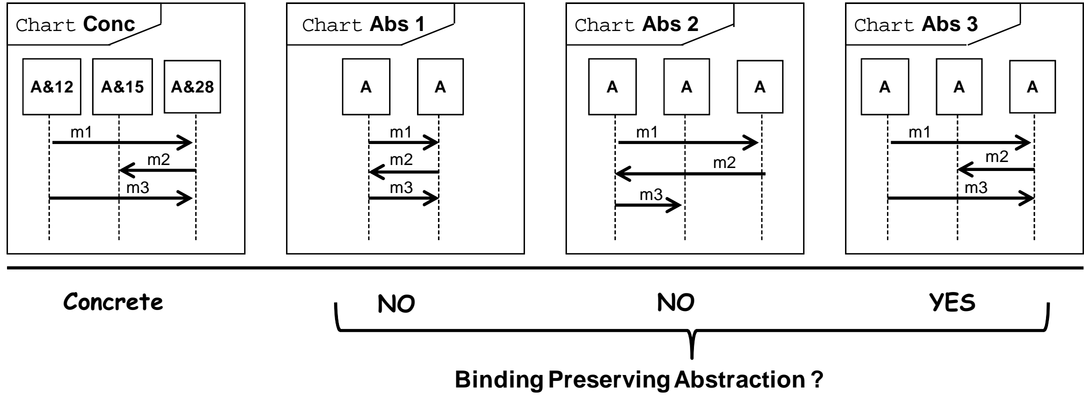

Figure 2: A Concrete Chart And Its Binding Preserving Abstraction

nical report [24]. We will see later that this property can be used to efficiently cut out the search space of infrequent LSCs.

PROPERTY 1 **(Anti-Monotone - Abstract)**. *Given* abstract *LSCs M = $(pre, full)$ and M' = $(pre', full')$, s.t. $full'$ has full as a prefix (with isomorphic binding), $supp(M') \leq supp(M)$.*

We explore the search space by first considering frequent single-event abstract charts. We then grow each of these by adding events one by one in a depth first fashion following a lexicographic ordering. The frequent charts mined are then combined into pre-chart main-chart pairs. Given a pair of charts pre and $full$, the LSC M = $L(pre, full)$ is generated if the confidence of M is above the minimum confidence threshold.

For example, we first try $\langle A \rangle$. If $\langle A \rangle$ is frequent, that is, its support in the traces is greater or equal than *min_sup*, we grow $\langle A \rangle$ to $\langle A, A \rangle$. If $\langle A, A \rangle$ is infrequent, from Property 1, all charts having $\langle A, A \rangle$ as prefix will not be frequent as well. We then try $\langle A, B \rangle$, and the process repeats. Note that without taking advantage of Property 1, this approach would not have been scalable. Relying on Property 1, we are able to prune the search space containing infrequent charts. Knowing 'apriori' that $\langle A, A \rangle$ is infrequent we can prune all charts $\langle A, A, A \rangle$, $\langle A, A, B \rangle$, $\langle A, A, A, A \rangle$, ... Thus, a large part of the search space is pruned. These systematic traversal of search space and smart applications of pruning properties enable our approach to work well.

Since we are interested in mining abstract LSCs and the trace consists of concrete events, this complicates the process of growing the charts by adding events one-by-one. First, an abstract chart may correspond to a number of different concrete charts. During the mining process, we need to be able to map an abstract chart to its concrete charts. Instances of different concrete charts need to be grown separately to ensure satisfaction of LSC semantics which involves reasoning on identities of objects involved in a chart. However, computation of support value must be considered in the abstract level by aggregating the support of each of the individual concrete charts. Second, there are a number of possible resultant bindings when growing a chart with an abstract event. These possible bindings need to be identified and corresponding concrete charts need to be mapped to the appropriate abstract level chart following the isomorphic binding preserving abstraction. To simplify the outline of the pseudo-code of our algorithm we define the following operation.

DEFINITION 3.1 **(Possible Bindings)**. *Given an abstract chart AC and an abstract event AE, concatenating AE to AC results in a set of (non-isomorphic) possible abstract charts. Each of the charts in the set corresponds to a different valid pair of abstract lifelines assigned as the bindings of AE. We denote the set of possible resultant charts from AC and AE by $PBDG$(AC,AE).*

To illustrate the set $PBDG(\cdot,\cdot)$, consider the left-most chart Abs shown in Figure 3. Performing the operation $PBDG(Abs, (B, A, m2()))$ will return four possible charts as shown by the subsequent charts $AbsExt$-1,2,3 and 4 in Figure 3.

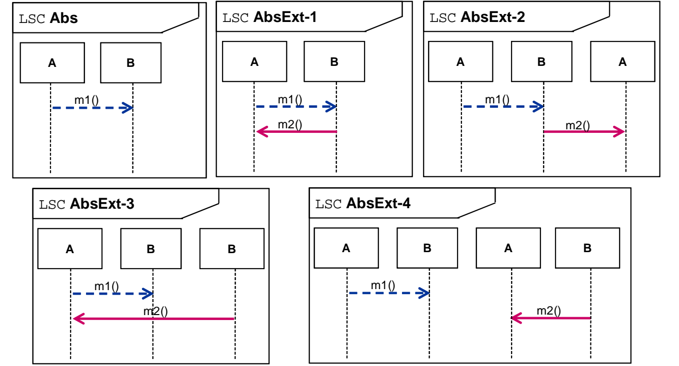

Figure 3: PBDG Examples

Our algorithm pseudo-code is shown in the algorithm boxes – Figure 4. The algorithm explores the search space of all possible abstract charts. At every point of the search, a particular abstract chart is considered. This abstract chart is initially a chart of single event. At each step, an abstract event is added to the abstract chart under consideration. The two procedures are described below.

MineSymbolicLSCs Procedure. The procedure accepts as parameters the input trace and the minimum support and confidence thresholds. It will output all statistically significant LSCs.

At line 1, the procedure computes the set of all single-event abstract chart (*i.e.*, AEV) whose number of instances is above the minimum support threshold. From Property 1, if a single abstract event aev is not frequent, so will all other charts containing aev. Hence, we only need to consider growing charts by events in AEV. The instances of a single abstract event aev correspond to all occurrences of concrete events whose abstract event is aev.

At line 2, we initialize the list *PrefixList*. *PrefixList* contains the prefixes of the current chart under consideration during search space traversal. These prefixes are used to compose LSCs corresponding to the current chart under consideration and its prefixes (see Procedure $MineRecurse$). At line 4, we grow each single-event chart recursively by calling procedure $MineRecurse$. When the procedure returns, the *PrefixList* data structure is updated ac-

cordingly at line 5.

MineRecurse Procedure. The procedure accepts as parameters the input trace, the minimum support and confidence thresholds, the set of frequent single-event charts, the current chart under consideration $CurC$, and the current list of prefixes $\mathit{PrefixList}$.

At lines 6 & 7, it outputs significant LSCs corresponding to the current chart and its prefixes. At line 8, we update the $\mathit{PrefixList}$ data structure by adding the current chart $CurC$.

At lines 9-10, the procedure tries to extend the current abstract chart with frequent single-event charts in AEV (note the $PDBG$ operator defined in Definition 3.1).

From Property 1, if the support of $CurC$ is below the minimum support threshold there is no need to extend $CurC$ anymore. The check of this property is performed at line 12. So, the procedure only makes a recursive call to itself at line 13 if, $CurC$'s positive witness is larger than the minimum support threshold. Finally, at line 14, we set the $\mathit{PrefixList}$ data structure accordingly.

Procedure MineSymbolicLSC
Inputs:
  TR : Input Trace
  min_sup, min_conf: Min. sup. and conf. Thresholds
Output:
  A set of statistically significant LSCs
Method:
1: Let AEV = Frequent single-event charts satisfying min_sup
2: Let $\mathit{PrefixList}$ = {}
3: For every f_aev in AEV
4:   Call MineRecurse (TR,min_sup,min_conf,AEV,f_aev, $\mathit{PrefixList}$)
5:   Remove last element of $\mathit{PrefixList}$

Procedure MineRecurse
Inputs:
  TR : Input trace
  min_sup, min_conf: Min. sup. and conf. thresholds
  AEV: Frequent single events
  $CurC$: Current chart considered
  $\mathit{PrefixList}$: Prefixes of $CurC$
Method:
6: For every pre in *PrefixList*
7:   Create LSC L(pre,$CurC$)
8:   If conf(L) $\geq min_conf$
9:     **Output** L
10: Append $\mathit{PrefixList}$ with $CurC$
11: For every f_aev in AEV
12:   Let $nxtCSet$ = $PBDG$(C,f_aev)
13:   For every $nxtC$ in $nxtCSet$
14:     If (|Instances of $nxtC| \geq min_sup$)
15:       Call MineRecurse(TR,min_sup,min_conf,AEV, $nxtC$,$\mathit{PrefixList}$)
16: Remove last element of $\mathit{PrefixList}$

Figure 4: Symbolic LSC Mining Algorithm

Finally, we present the following theorem describing the guarantee given by our mining algorithm. The proof sketch is available in [24].

THEOREM 1. *The algorithm shown in Figure 4 computes a statistically sound and complete set of statistically significant abstract Live Sequence Charts from input traces.*

4. MINING EXAMPLE

We demonstrate the mining algorithm using (a short snippet from) an example trace, taken from the Space Invaders game application (trace generation is done using AspectJ, see [25]).

```
1  (GameCtrl&0,Monster&0,void Monster.fire())
...
2  (Monster&0,Laser&0,void Laser.fire())
3  (GameCtrl&1,Monster&1,void Monster.fire())
4  (Monster&1,Laser&1,void Laser.fire())
5  (Laser&1,GameCtrl&1,void GameCtrl.add(Laser))
6  (Laser&0,GameCtrl&0,void GameCtrl.add(Laser))
...
7  (GameCtrl&0,Laser&0,void Laser.remove())
8  (GameCtrl&1,Laser&1,void Laser.remove())
9  (Bullet&0,Monster&2,void Monster.hit(Bullet))
10 (Monster&2,GameCtrl&0,void GameCtrl.hit(Bullet))
11 (GameCtrl&0,Monster&2,void Monster.remove())
...
12 (GameCtrl&0,Bullet&0,void Bullet.remove())
13 (Bullet&1,Monster&3,void Monster.hit(Bullet))
14 (Monster&3,GameCtrl&0,void GameCtrl.hit(Bullet))
15 (GameCtrl&0,Monster&3,void Monster.remove())
16 (GameCtrl&0,Bullet&1,void Bullet.remove())
17 (Bullet&1,Monster&3,void Monster.hit(Bullet))
18 (Bullet&2,Monster&4,void Monster.hit(Bullet))
...
```

Figure 5: A Snippet from an Example Trace

Each line corresponds to a concrete event, presented as a tuple, showing the caller, callee, and method signature. The integer after the "&" denotes the identifier of the object (originally this is the object's hash key but we reduce it here to small integers to save space and improve readability). The "..." shown in the sample trace above correspond to occurrences of unrelated events.

Using the example, we illustrate how the abstract class-level LSC of the `firing` scenario in Fig. 1 (left) is mined. Note that concrete instances of this abstract LSC, each of which involves different concrete objects, may be interleaved with one another.

Consider running the mining algorithm with $min_sup = 2$. The algorithm first mines frequent single-event abstract charts. One of them corresponds to the pre-chart of the LSC shown in Fig. 1 (left). It then grows the single event chart to form longer charts in a depth-first order. Eventually the chart corresponding to the full chart of the LSC is considered. This chart together with its corresponding pre-chart are then used to construct the LSC shown in Fig. 1 (left).

Let us show in more detail how the `firing` LSC shown in Fig. 1 (left) is formed. For a succinct representation, consider the following mapping of abstract events:

ID	Abstract Event Details
AE_1	(GameCtrl, Monster, void Monster.fire())
AE_2	(Monster, Laser, void Laser.fire())
AE_3	(Laser, GameCtrl, void GameCtrl.add(Laser))
AE_4	(GameCtrl, Laser, void Laser.remove())

The single-event chart $\langle AE_1\rangle$ is first formed. The algorithm eventually grows $\langle AE_1\rangle$ to $\langle AE_1, AE_2\rangle$. To grow $\langle AE_1\rangle$, it considers all concrete instances of AE_1. This corresponds to the first and third events shown in the snippet. It then tries to see whether there are concrete events in the set $\{e|proj(e) = AE_2\}$ with which the concrete instances of AE_1 can be extended. We note that it is possible. There are actually three of them, corresponding to the substrings denoted as (1,2),(1,4) and (3,4). Among the three, only (1,2) and (3,4) are correct instances of the abstract chart $\langle AE_1, AE_2\rangle$ (following the "monitoring" semantics of symbolic instances in LSC – c.f. [28]). Since the number of instances of $\langle AE_1, AE_2\rangle$ is greater than the minimum support set, we continue to grow it.

At the next traversal of the search space in lexicographical depth-first order, we consider $\langle AE_1, AE_2, AE_1\rangle$; however, the number

of instances of this chart is less than the minimum support. Hence, following Prop. 1, we prune the search space that extends it. Eventually, we will consider the chart $\langle AE_1, AE_2, AE_3, AE_4\rangle$. This chart together with chart $\langle AE_1\rangle$ which is a prefix found earlier, are composed to form the class level LSC $L(\langle AE_1\rangle,\langle AE_1, AE_2, AE_3, AE_4\rangle)$. This is the LSC shown in Fig. 1 (left).

5. DISCUSSION

The complexity of the algorithm is linear in the number of the significant abstract LSCs mined. It can be easily seen that this is the case from the fact that every time the procedure $MineRecurse$ is called, a set of LSCs is outputted. Still, depending on the length of the trace and the thresholds set, the number of significant LSCs may be very large. In [23] we proposed several techniques to reduce the number of the mined LSCs by incorporating additional user input. With this strategy, additional speed-up can be achieved. A more detailed complexity analysis is given in the technical report [24].

One limitation of our approach, common to dynamic analysis methods in general, is that the quality of its results depends on how the traces used are representative of the system under investigation. One possible way to address this may be to investigate an integration with static analysis methods. A related weakness in our approach is its reliance on user-defined thresholds. A possible way to dealing with this is to construct specification mining as a user-guided interactive iterative process, where the user may begin with relatively high thresholds and refine them later as needed. This seems a pragmatic solution.

6. RELATED WORK

In our own previous work on mining LSCs (a short paper [25]), we refer briefly to class level LSCs. There, the support and confidence of class level LSCs are aggregated from those of *significant* concrete LSCs. Thus, the work assumes that an abstract LSC is significant only if at least one of its corresponding concrete LSC is significant. If this does not hold, the algorithm might miss mining some abstract LSCs or introduce spurious abstract LSCs that are insignificant (thus, the algorithm there is neither sound nor complete for class level LSCs). In another work [23], we focused on an extension of this algorithm to incorporate more user-guidance into the mining process using the concepts of triggers and effects.

Rountev and Connell [31] considered the problem of object naming analysis for reverse engineered sequence diagrams. This problem has some similarities with our problem of binding preserving abstraction. However,[31] uses static analysis of program code, and is aimed at finding reverse engineered, concrete, non-interleaving, complete sequence diagrams. Our work uses dynamic analysis to reveal statistically significant recurrent, interleaved, universal scenario-based specs, and our 'naming' problem arises in the attempt to generalize from objects' concrete names to their classes.

Ernst et al. [8] describe the Daikon tool, which discovers likely program invariants from execution traces. However, Daikon mines value based invariants, not temporal constraints. In this sense our work is complementary to Daikon.

Most specification miners extracting temporal constraints from traces produce an automaton (e.g., [3, 5, 20]). Unlike these miners, we mine a set of LSCs from traces of program executions. We believe sequence diagrams in general and LSCs in particular, are suitable for the specification of inter-object behavior, as they make the different role of each participating object and the communications between the different objects explicit. Thus, our work is not aimed at discovering the complete behavior or APIs of certain components, but, rather, to capture the way components cooperate to implement certain system features. Indeed, inter-object scenarios are popular means to specify system requirements (see, e.g., [12, 15]). The addition of symbolic scenarios is important, as it results in more expressive and succinct specification models, which are critical in the context of real world object oriented systems.

Yang et al. [37] present mining two-event temporal rules that are statistically significant with regard to a user-defined 'satisfaction rate'. Lo et al.[22] generalize this work to multi-event rules. In this work, we mine multi-event LSCs rather than temporal rules. Different from temporal rules, LSCs capture caller and callee relationship.

It is interesting that in the work of Yang *et al.* [37], two different mining scenarios are considered namely context-sensitive and context-insensitive. The earlier refer to the case where information relating to the context of a function calls including the object identities is recorded in the execution traces. Yang *et al.* also introduce the concept of context slicing where a trace is sliced based on object identities. Mining can the be performed on the set of sliced traces. This can be viewed as a degree of generalization from concrete to abstract. However, the slicing proposed in [37] only slice single object instance at a time. Hence, their generalization is less powerful than the one proposed in this paper where multiple objects from the same or multiple classes can exist in the specification and can be differentiated.

Many work suggest and implement different variants of dynamic analysis based reverse engineering of objects' interactions from program traces and their visualization using sequence diagrams (see, e.g., [1, 4, 30]), which may seem similar to our current 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 scenarios; 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 not only as concrete but also as 'existential' (see the distinction between existential and universal LSC in [6, 13]). In contrast, we 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 specifications, at the object-level as well as the more expressive and succinct class-level, ultimately suggesting scenario-based system requirements.

There are also other work mining specifications from program code, e.g., [35, 2, 7]. Different from their work we employ dynamic rather than static analysis in inferring specifications. Dynamic and static analysis complement each other and each has its own pros and cons which have been documented in the literature.

7. CONCLUSION AND FUTURE WORK

We presented our approach to scenario-based specification mining, which is a novel example of a dynamic analysis specification mining work that considers object identities, and generalizes from concrete objects to their classes while maintaining soundness and completeness. We use data mining methods to extract symbolic class level scenarios from concrete execution traces with guaranteed soundness and completeness modulo user defined support and confidence thresholds. The key to the approach is the mechanism for binding preserving abstraction. The mined symbolic scenarios are succinct and expressive behavioral specifications. They may be presented visually as UML sequence diagrams (with the *modal* profile [13]) within standard modeling tools, and further used for program comprehension and runtime monitors generation (see, e.g., [18, 27]). As an immediate future work, we are in the process of implementing a tool and conducting case studies based

on the proposed algorithm.

Acknowledgements We thank Yishai Feldman, David Harel, and Itai Segall, for their valuable comments and advice on our work on mining Live Sequence Charts.

8. REFERENCES

[1] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/.

[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 *SIGSOFT FSE*, 2007.

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

[4] 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.

[5] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *TOSEM*, 7(3):215–249, July 1998.

[6] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of Symp. on Operating Systems Principles*, 2001.

[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 27(2):99–123, 2001.

[9] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE*, 2008.

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

[11] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[12] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.

[13] D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.

[14] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[15] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards Formal Design with Sequence Diagrams. *Software and Systems Modeling (SoSyM)*, 4(4):355–367, 2005.

[16] J. Klose, T. Toben, B. Westphal, and H. Wittke. Check it out: On the efficient formal verification of Live Sequence Charts. In *CAV*, 2006.

[17] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *TACAS*, 2005.

[18] M. Lettrari and J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In *UML*, 2001.

[19] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.

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

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

[22] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of recurrent rules from a sequence database. In *Proc. of Int. Conf. on Database Systems for Advanced Applications*, 2008.

[23] D. Lo and S. Maoz. Mining Scenario-Based Triggers and Effects. In *ASE*, 2008.

[24] D. Lo and S. Maoz. Specification mining of symbolic scenario-based models (tech. report. version). *Technical Report at www.comp.nus.edu.sg/∼dlo/symbolicmining.pdf*, 2008.

[25] D. Lo, S. Maoz, and S.-C. Khoo. Mining Modal Scenario-Based Specifications from Execution Traces of Reactive Systems. In *ASE*, 2007.

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

[27] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.

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

[29] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis for component integration. In *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[30] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (VET): an interactive plugin-based visualisation tool. In *Proc. 7th Australasian User Interface Conf. (AUIC'06)*, pages 153–160. Australian Computer Society, Inc., 2006.

[31] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *ICSE*, pages 254–263, 2005.

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

[33] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *WCRE*, 2007.

[34] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

[35] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.

[36] X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. of SIAM Int. Conf. on Data Mining*, 2003.

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