Succinctness in Scenario-Based Specification Mining

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

Shahar Maoz
Dept. of Computer Science 3 (Software Engineering)
RWTH-Aachen University, Aachen, Germany
maoz@se.rwth-aachen.de

ABSTRACT

Specification mining methods are used to extract candidate specifications from system execution traces. A major challenge for specification mining is succinctness. That is, in addition to the soundness, completeness, and scalable performance of the specification mining method, one is interested in producing a succinct result, which conveys a lot of information about the system under investigation but uses a short, machine and human-readable representation.

In this paper we address the succinctness challenge in the context of scenario-based specification mining, whose target formalism is live sequence charts (LSC), an expressive extension of classical sequence diagrams. We do this by adapting three classical notions: a definition of an equivalence relation over LSCs, a definition of a redundancy and inclusion relation based on isomorphic embeddings among LSCs, and a delta-discriminative measure based on an information gain metric on a sorted set of LSCs. These are used on top of the commonly used statistical metrics of support and confidence.

A number of case studies show the utility of our approach towards succinct mined specifications.

1. INTRODUCTION

Specification mining, the dynamic analysis process of extracting models from software execution traces, in order to aid in program comprehension, testing, and formal verification tasks in the absence of complete and up-to-date documented specifications, has attracted many and diverse research efforts in recent years (see, e.g., [2, 3, 5, 10, 29]). The underlying common goal of these work is to extract a set of invariants and present them to the user. In the literature, extracted invariants vary from boolean expressions capturing a relationship between two variables in a particular program point, to frequent patterns of usage behavior, to various automata and temporal rules. The extracted candidate invariants are used to support comprehension, testing, and formal verification tasks (see, e.g., [3, 8, 33, 35, 39]).

In addition to the soundness, completeness, and scalable performance of the specification mining method, one is interested in producing a succinct result, which conveys a lot of information about the system under investigation but uses a short, machine and human-readable representation. Thus, an important challenge of specification mining in general is the definition of what constitutes a likely invariant and, in particular, the selection and presentation of the extracted invariants. Obviously, a specification mining framework that returns 'too many' likely invariants is not useful. The need arises to produce succinct mined specifications.

In this work we address the succinctness challenge by adapting three classical notions: equivalence relation, statistical redundancy, and delta-discriminative information gain. We do this in the context of *scenario-based specification mining* [25, 26, 27, 28].

In scenario-based specification mining, data mining methods are employed to extract statistically significant inter-object scenario-based specifications in the form of Damm and Harel's *live sequence charts* (LSC) [6], a visual formalism that extends classical sequence diagrams with modalities. An LSC is composed of a pre-chart and a main-chart both consisting an ordered sequence of events. An LSC specifies a temporal invariant: whenever the pre-chart events happen in the specified order, eventually the main-chart events must happen in the specified order. LSC has a formal semantics [6], a UML2-compliant variant [14], and a translation into various temporal logics [18].

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 target formalism. The choice is supported by previous work on LSC (e.g., [17, 19, 31]), which can be used to visualize, analyze, manipulate, test, and verify the specifications we mine.

The statistical significance of an LSC in a set of traces is measured using *support* and *confidence* – two metrics commonly used in data mining [12]. The main inputs are a set of traces and thresholds for minimum support and confidence: the first tells the number of times an LSC need to be observed in the traces, the second dictates the likelihood that the pre-chart of the LSC is indeed followed by the main-chart. Only LSCs obeying the minimum support and confidence are deemed significant (see [28]).

Most importantly, to achieve succinctness we introduce the following three main features:

1. **A single representative of each equivalence class**. We define an equivalence relation between scenarios based on the notion of symbolic lifelines. The symbolic LSC corresponding to all concrete LSCs in an

equivalence class is considered a representative of this class. Statistical significance is computed at the level of the symbolic representative LSC.

2. **Statistical redundancy via isomorphic embeddings**. We define an inclusion relation between LSCs based on a variant of subgraph isomorphism. When two LSCs have the same statistical significance but one is embedded in the other, the smaller one is considered redundant and is removed from the result.
3. **Delta-discriminative summarization via information gain**. We rank the mined scenarios and then filter out the ones that are not at least delta-discriminative relative to higher ranked scenarios.

The formal definitions and algorithms we use for the above features are given in Sec. 3 and 4. Overall, the result is a succinct set of statistically significant LSCs.

It is important to note that setting higher support and confidence thresholds may have the effect of reducing the number of significant LSCs mined. However, this would have removed some informative LSCs while keeping others whose contribution to the complete specification mined is minor. Thus, summarization methods such as the ones we introduce and evaluate are indeed necessary.

The examples throughout the paper are taken from CrossFTPServer [4], a commercial open source FTP server built on top of Apache FTP server, and from Jeti [1], a full featured open-source instance messaging application. The examples show the utility of our approach in mining summarized non-redundant symbolic scenarios from a set of execution traces. Experimental results appear in Sec. 5.

Finally, the challenge of succinctness in specification mining is not limited to the scenario-based approach. Some work present mining of rules and patterns (e.g., [8, 35, 39, 34]) where the number of rules and patterns could be very large and make it hard for users to investigate the results. In Daikon [9], the number of reported invariants could be too large, and in mining finite-state machines (e.g., [3, 21, 33, 29]), at times the size of the automaton could be too large to be useful. We further discuss these related works and their succinctness challenges in Sec. 6.

Paper organization: Sec. 2 provides background on LSC and on statistical significance in scenario-based specification mining. Sec. 3 formally defines the three main succinctness features used in our work. The mining framework and algorithms appear in Sec. 4. Experimental results appear in Sec. 5. Related work is discussed in Sec. 6 and Sec. 7 concludes with a discussion and future work directions.

2. BACKGROUND

In this section we give background on LSC, describe basic notations, outline basic definitions of significance of LSC in a set of traces, and state some important properties.

2.1 Live Sequence Charts

The language of live sequence charts (LSC) [6, 14] is a formal expressive variant of classical sequence diagrams. In this study we use a restricted subset of LSC. A chart 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 chart are not restricted in any way to appear or not to appear during the run (including between the events that are mentioned in the chart). For a thorough description of the language and its semantics see [6, 15]. A UML2-compliant variant of LSC using the *modal* profile is defined in [14]. A translation of LSC into temporal logics appears in [18].

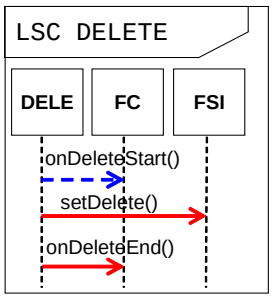

Figure 1: [CrossFTP] Mined LSC: Delete

Syntactically, instance lifelines are drawn as vertical lines, pre-chart (main-chart) events are colored in blue (red) and drawn using horizontal dashed (solid) lines.

An important feature of LSC is its semantics of *symbolic instances* [32]. Rather than referring to concrete objects, lifelines may be labeled with a class name and marked symbolic, formally representing any object of this class. This allows a designer to take advantage of object-orientation and create more expressive and succinct specifications.

Fig. 1 shows an example LSC. Roughly, the semantics of this LSC is: *"Whenever an object of type DELE calls the method onDeleteStart() of an object of the type FtpletContainer, eventually the DELE object must call the method setDelete() of an object of type FtpStatisticsImpl and call the method onDeleteEnd() of the FtpletContainer"*. Note that this is a temporal invariant. Also note the symbolic interpretation of the lifelines.

We denote an LSC by $L(pre, full)$, where pre is the pre-chart and $full$ is the complete chart containing pre followed by the main chart. For concrete LSC, the pre and $full$ correspond to concrete charts, and for symbolic LSC, the pre and $full$ correspond to symbolic charts.

2.2 Basic Notations

Let C be the set of concrete events in the traces. A concrete event is a triplet (*caller*,*callee*,*signature*) corresponding to caller unique object identifier (obtained in Java using identityHashCode()), callee object identifier, and the signature of the method being called, respectively. A trace is a series of concrete events. The input under consideration is a multi-set of traces.

We consider the input multi-set of traces as a trace database TDB. The ith member of the trace database is denoted TDB[i]. A contiguous sub-trace within a trace t is denoted $t[i..j]$, where i and j refer to the starting and ending positions of the sub-trace in the trace t respectively.

A concrete event could be abstracted to form a symbolic event. A symbolic event is a triplet (*caller*,*callee*,*signature*) corresponding to caller class, callee class, and method signature. While a symbolic event may have one or more corresponding concrete events, a concrete event maps to a single

symbolic event. A simple map from a concrete event to its symbolic event is defined as a projection: given a concrete event e, $proj(e)$ returns the symbolic event of e, where the caller and callee objects identifiers are replaced by the names of their classes.

2.3 Statistical Significance

To mine *all significant* LSCs, we first describe a simpler problem of computing the statistical values of a given chart. The definitions provided in this sub-section apply to both concrete and symbolic LSCs. Given a concrete or a symbolic LSC M and a trace T, we are interested in finding statistics denoting the significance of M in T. To do so, we introduce the concepts of positive and negative witness. We further categorize a negative witness as either strong- or weak-negative witness.

A positive witness of a concrete or symbolic LSC $M = L(pre,full)$, is a trace segment satisfying the $full$ chart – by extension the pre chart as well, since pre is a prefix of $full$. A negative witness of M is a positive witness of pre which cannot be extended to a positive witness of M (or $full$). Satisfaction of a chart follows the semantics of LSC described in [6, 32]. A trace segment satisfying a chart would have all events prescribed in the chart appearing in the correct order, a correct number of times as defined by the chart (recall that the chart does not restrict events not appearing in it to appear or not to appear in the trace, including in between the events that appear in the chart). Also, the trace need to have correct bindings of respective objects appearing in the trace to corresponding lifelines in the chart. A more operational semantics for instances of concrete and symbolic charts is provided in [26].

The semantics of LSC (like most formal specification languages used for reactive systems, e.g., LTL) 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 adapt the semantics of LSC, and in particular, 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.

Hence, a negative witness of M corresponding to a positive witness pos of pre may appear in two cases:

1. End of trace being reached when one tries to extend pos to a positive witness of M.
2. The positive instance pos of pre can not be extended to a positive witness of M due to violation of instance order constraint.

We refer to the two cases as strong- and weak- negative witness of M respectively. We denote positive, negative, strong-negative and weak-negative witnesses of an LSC M by $pos(M)$, $neg(M)$, $strong\_neg(M)$ and $weak\_neg(M)$ respectively.

The above notions of witnesses are used to define the *support* and *confidence* metrics for an LSC, two statistical metrics commonly used in data mining. Given a trace T, the *support* of an LSC $M= L(pre, full)$, denoted by $sup(M)$, is simply defined as the number of positive witnesses of M found in T. The *confidence* of an LSC M, denoted by $conf(M)$, measures the likelihood of a sub-trace in T satisfying M's pre-chart to be extended such that M's main-chart is satisfied or the end of the trace is reached. Hence, confidence is expressed as the ratio between the number of positive witnesses and weak-negative witnesses of the LSC and the number of positive witnesses of the LSC's pre-chart. Formally:

$$\begin{array}{|l|}\hline sup(L,T) \equiv_{def} |pos(L,T)| \\ conf(L,T) \equiv_{def} \dfrac{|pos(L,T)|+|weak\_neg(L,T)|}{|pos(pre,T)|} \\ \hline\end{array}$$

Notation-wise, when T is understood from the context, it can be omitted.

Thus, 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 to be followed by a main-chart instance in the trace. 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.[1] 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 that satisfies both thresholds is referred to as being *significant*.

2.4 Monotonicity, Soundness & Completeness

The following monotonicity property for symbolic LSCs is used in our mining algorithm [26].

PROPERTY 1 (**Monotonicity**). *Given a* symbolic *LSC M = (pre, full) and M' = (pre′, full′). If full is a prefix (or suffix) of full′, every positive witness of M' is a positive witness of M. Hence, $sup(M') \leq sup(M)$.*

The monotonicity property can be effectively used to prune the search space. For example, after ascertaining that the support of an symbolic chart A is less than the minimum support threshold, one can prune the search space of all charts having A as a prefix.

Following data mining algorithms in general, and frequent pattern mining algorithms in particular (see, [12, 20]), our goal is to achieve statistical soundness and completeness described in Definition 2.1. Our algorithms are sound and complete modulo the given traces and user-defined thresholds (our notion of soundness and completeness is thus independent of the quality of the traces used in terms of coverage etc., that is, in contrast to, e.g., [16, 40]).

DEFINITION 2.1 (**Stat. Sound. & Completeness**). *A mining algorithm is* statistically sound *with respect to an input dataset and given user thresholds iff every mined result obeys the thresholds based on the dataset. A mining algorithm is* statistically complete *with respect to an input dataset iff every potential result that obeys the thresholds based on the dataset are output.*

[1]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 [39]), since, in general, these may reveal errors in the program or in the trace generation process (see, e.g., [3, 21, 39]).

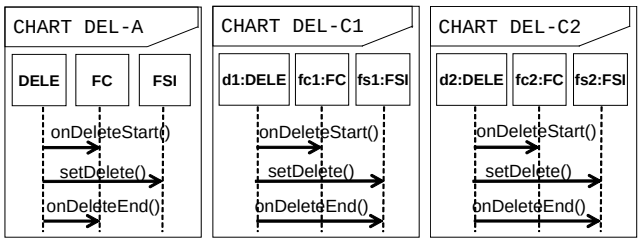

Figure 2: Charts: Symbolic and Concrete

3. TOWARDS SUCCINCTNESS

To mine for succinct specifications, we introduce three concepts: equivalence class of LSCs and symbolic LSCs, isomorphic embeddings of statistically non-redundant LSCs, and summary via extraction of delta- discriminative LSCs.

3.1 Symbolic LSCs as Equivalence Classes

The first succinctness feature builds on the mapping from concrete to symbolic LSCs. A symbolic LSC corresponds to a set of concrete LSCs having the same structure, lifelines and methods among the lifelines. The only differences among members of the equivalence class are the actual objects that are bound to the lifelines.

An illustration of concrete and symbolic full-charts is shown in Fig. 2. The one on the left is the symbolic LSC and the others are some examples of corresponding concrete LSCs. This is the full-chart of the LSC shown in Fig. 1. We formalize concrete and symbolic charts and the mapping between them via the definition of *binding preserving abstraction* (BPA).

A chart (concrete or symbolic) is composed of a list of (concrete or symbolic) events $\langle e_1, e_2, \ldots, e_n\rangle$, a set of (concrete or symbolic) 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 identified by its position in the chart 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 3.1 (**Concrete (Symbolic) Chart**). *A concrete (symbolic) chart C is a triplet $\langle E, L, bdg\rangle$ where E is a list of concrete (symbolic) events, L is a set of concrete/symbolic lifelines, and bdg is a binding map $bdg : I \rightarrow L \times L$, where $I = \{i | 1 \leq i \leq |E|\}$ (i.e., the set of indices of events in E). Notation-wise, we refer to the pair of lifelines corresponding to the i^{th} element of E as $bdg[i]$.*

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 symbolic events. Rather, given a list of symbolic events, one may possibly define more than one (non-isomorphic) sets of symbolic lifelines with corresponding bindings. This is so because a class corresponding to a symbolic caller or callee does not uniquely identify a symbolic lifeline in the set, as it may include a number of symbolic lifelines corresponding to the same class.

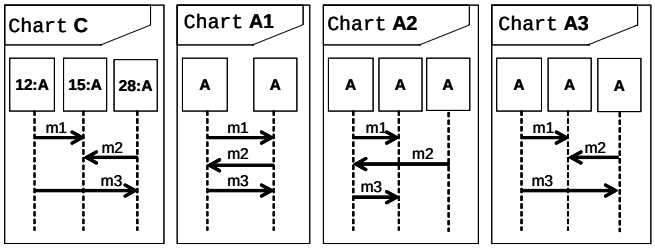

Figure 3: A Concrete Chart C and its Binding Preserving Abstraction (BPA) A3. Other charts (i.e., A1 and A2) are not BPA of C.

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. 3, $bdg((12 : A, 28 : A, m1())) = \langle 1, 2\rangle$.

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

DEFINITION 3.2 (**Binding Preserving Abstraction**). *Consider a concrete chart $CC = \langle\langle e_1, \ldots, e_n\rangle, \{l_1, \ldots, l_k\}, bdg\rangle$ and a symbolic 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 one-to-one mapping abs from the lifelines of CC to the lifelines of AC s.t. $\forall_{1\leq i,j\leq k}$ and $\forall_{1\leq v\leq k}$, $bdg(v) = \langle l_i, l_j\rangle$ iff $proj(e_v) = E_v$ and $BDG(v) = \langle abs(l_i), abs(l_j)\rangle$.*

We denote the binding preserving abstraction of CC by $abs(CC)$, representing the symbolic chart that is 'isomorphic' to CC.

To illustrate the concept of binding preserving abstraction consider Fig. 3. 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 symbolic charts. Two of the symbolic 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 a symbolic LSC L. When this is the case, more than one object instance of the class C participates in L.

We use the binding preserving abstraction *abs* to define an equivalence relation over concrete LSCs:

DEFINITION 3.3 (**Equivalence Relation**). *Consider two concrete LSCs $l_1 = L(pre_1, full_1)$ and $l_2 = L(pre_2, full_2)$. $l_1 \equiv_{abs} l_2$ iff $abs(pre_1) = abs(pre_2)$, and $abs(full_1) = abs(full_2)$.*

It is easy to see that $\equiv_{abs}$ is indeed reflexive, symmetric, and transitive, and thus partitions any set of concrete LSCs into equivalence classes. We use the symbolic LSC corresponding to all concrete LSCs in a class as a representative of the class. To extract a succinct set of LSCs from an input trace database, we look for representative symbolic LSCs.

3.2 Isomorphic Embeddings & Redundancy

Following the concept of BPA, two charts are said to be isomorphic if the series of events are the same, and there is

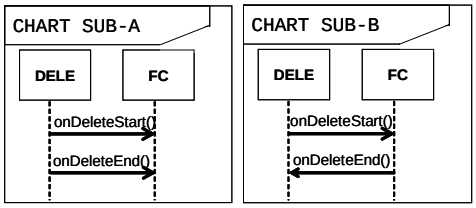

Figure 4: Iso. & Non-Iso. Charts

a one-to-one mapping between their lifelines that preserves the bindings. The formal definition is in Defn. 3.4. It could easily be seen a concrete chart having a chart C as a BPA will also have any isomorphic chart of C as a BPA.

DEFINITION 3.4 (**Isomorphic Charts**). *Two charts f1 = $\langle E1, L1, bdg1\rangle$ and f2 = $\langle E2, L2, bdg2\rangle$ are isomorphic if $\exists$ a one-to-one mapping map from L1 to L2 such that $\forall i, bdg1[i] = (k_x, k_y)$ iff $E1[i] = E2[i] \wedge bdg2[i] = (map(k_x), map(k_y))$.*

To illustrate isomorphic embeddings, consider the example chart in Fig. 4(left). This chart is an isomorphic embedding (i.e., it could be embedded isomorphically) of the chart shown in Fig. 2(left). On the other hand, the chart in Fig. 4(right) has no isomorphic embedding in the chart shown in Fig. 2(left). We formally define isomorphic embeddings below.

DEFINITION 3.5 (**Isomorphic Embeddings**). *Consider charts $C_A = \langle\langle a_1, \ldots, a_n\rangle, \{l_{a_1}, \ldots, l_{a_l}\}, bdg_a\rangle$ and $C_B = \langle\langle b_1, \ldots, b_m\rangle, \{l_{b_1}, \ldots, l_{b_k}\}, bdg_b\rangle$. C_A has an isomorphic embeddings within C_B iff there exists a mapping iso from the lifelines of C_A to the lifelines of C_B s.t. $\forall_{1\leq v\leq n}$, $bdg_a(a_v) = \langle l_i, l_j\rangle$ iff $\exists_{1\leq w\leq m}$. $a_v = b_w \wedge bdg_b(b_w) = \langle iso(l_i), iso(l_j)\rangle$. We refer to C_B as a super-chart of C_A, denoted $C_B \sqsupseteq C_A$.*

We use the above definition of isomorphic embeddings to define statistically redundant charts:

DEFINITION 3.6 (**Statistically Redundant**). *Consider two LSCs $l_1 = L(pre_1, full_1)$ and $l_2 = L(pre_2, full_2)$. The first LSC l_1 is rendered redundant by the second LSC l_2 iff sup(l_1)=sup(l_2), conf(l_1)=conf(l_2) and $full_2 \sqsupseteq full_1$.*

In our mining, we would like to eliminate statistically redundant charts. A chart is considered redundant if a super-chart of it with the same statistics appears in the mined set. Note that a significant LSC is likely to have many significant sub-LSCs having the same support and confidence. By eliminating statistically redundant charts, many uninteresting charts are removed.[2]

3.3 Delta-Discriminative Summarization

The third summarization feature concerns the information gained by adding each LSC to the mined set, with regard to other LSCs that were already selected.

We start by sorting the mined LSCs in an ascending order based on number of lifelines, followed by total length (i.e., number of events in the LSC), followed by the inverse length of the pre-chart, followed by their confidence and

[2]Note though, that statistical redundancy does not imply logical redundancy. In LSC semantics, a sub-chart and a super-chart are incomparable; one does not logically imply the other.

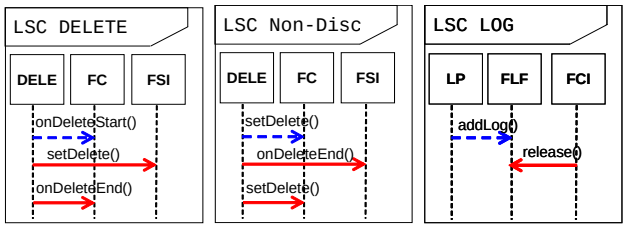

Figure 5: Non-Disc. & Disc. Charts

their support. The sorted list of LSCs is then summarized by removing those LSCs which do not contribute enough additional information – in terms of new event signatures – relative to the ones ranked above them. As a result, similar LSCs with minor variations are removed from the final set of mined LSCs which is presented to the user.

DEFINITION 3.7 (**InfoGain(l_1,l_2)**). *Consider two LSCs $l_1 = L(pre_1, full_1)$ and $l_2 = L(pre_2, full_2)$. Let $MSet_1$ and $MSet_2$ be the set of signatures contained in $full_1$ and $full_2$ respectively. The information gain of l_1 with respect to l_2 is defined as $|MSet_1 \setminus MSet_2|/|MSet_1|$.*

We use the above to define delta-discriminative LSC.

DEFINITION 3.8 (**Delta-Discriminative**). *Consider an ordered set of LSCs $ORD = \langle l_1, \ldots, l_n\rangle$. Given a minimum delta-discrimination threshold δ, an LSC $l_j \in ORD$ is delta discriminative iff $\nexists i < j$. InfoGain(l_i, l_j) $< \delta$. An ordered set of LSCs is delta-discriminative if all its LSCs are delta-discriminative.*

Thus, to produce a succinct set of LSCs which covers many different method signatures, only a set of delta-discriminative LSCs would be mined.

As an example, consider the LSCs shown in Fig. 5 ordered from left to right. With delta-discriminative threshold set at 10%, the second LSC would not be reported while the third LSC would. For every selected representative LSC, other similar LSCs with minor variations would be removed.

3.4 Additional Considerations

In addition to the above three features, we provide filters that could optionally be applied to restrict the format of mined LSCs to ones commonly used in the literature. We find that often methods are not repeated in an LSC, hence we introduce a filter to remove LSCs with repeated method. An example of such LSC is shown in Fig. 5(left) where method setDelete() is repeated twice. In addition, we remove LSCs which are not monotonically connected. An LSC L is monotonically connected if for every prefix of L, each lifeline in the prefix is connected, either directly or indirectly, to another lifeline. An example of non-monotonically connected chart is shown in Fig. 6. Note that if there is a third method from the 2nd to 3rd lifeline, this chart would be connected but not monotonically connected. We denote that an LSC L is monotonically connected by $MCONN(L)$.

4. MINING FRAMEWORK & ALGORITHM

This section describes our technique to mine a succinct set of LSCs from a set of traces, starting with the framework overview and continuing with the algorithm in detail.

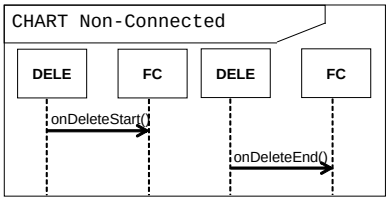

Figure 6: Non-Monotonically Connected Chart

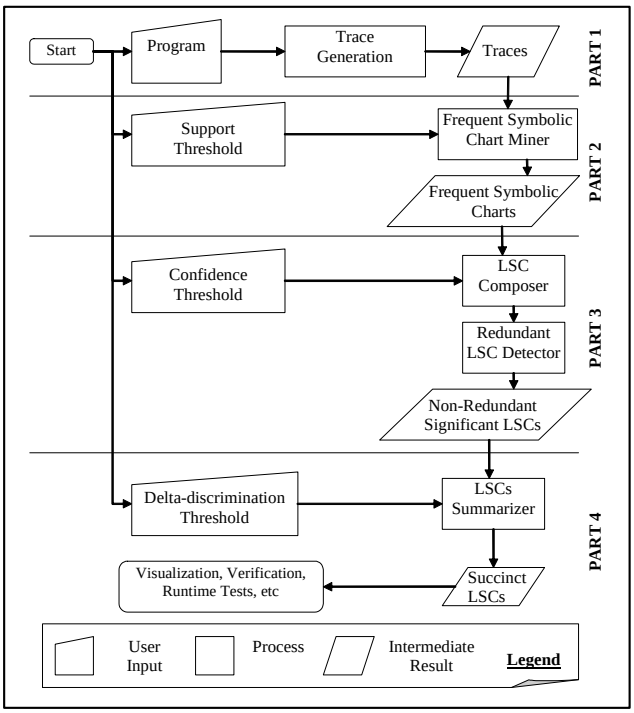

Figure 7: Mining Framework

4.1 Mining Framework

The mining framework starts with a program and ends with a succinct set of mined LSCs. The program under investigation is instrumented with 'print' statements at the entries of selected method calls; different instrumentation techniques could be used ranging from binary instrumentation, to byte code injection to aspect oriented programming. When executed, the instrumented program produces a set of traces. If a set of test cases is present, these test cases could be run to produce a set of traces. Otherwise, typical user interaction with the user interface component of the system could be performed and a corresponding set of traces could be collected.

The set of traces is fed to the miner (Part 2,3 and 4). The miner finds every LSC $L(pre, full)$ that satisfies the following criteria: (1) The full chart $full$ is observed at least min_sup number of times in the set of traces; (2) when the pre-chart pre occurs, the main-chart would eventually occur with at least min_conf likelihood in the input traces; (3) the set is non redundant; and (4) all LSCs are delta-discriminative with regard to higher ranked mined LSCs.

The mined LSCs could be visualized to help program understanding, fed to runtime monitoring tools [28, 31], used for software verification, etc.

4.2 Mining Algorithm

pos	caller	callee	signature
		Trace 1	
1	d1:DELE	fc1:FC	onDeleteStart(..)
2	d1:DELE	fs1:FSI	setDelete(..)
3	m1:MKD	fc1:FC	onMkdirStart(..)
4	m1:MKD	fs1:FSI	setMkdir(..)
5	d1:DELE	fc1:FC	onDeleteEnd(..)
6	m1:MKD	fc1:FC	onMkdirEnd(..)
		Trace 2	
1	d2:DELE	fc2:FC	onDeleteStart(..)
2	d2:DELE	fs2:FSI	setDelete(..)
3	m2:MKD	fc2:FC	onMkdirStart(..)
4	d2:DELE	fc2:FC	onDeleteEnd(..)
5	m2:MKD	fs2:FSI	setMkdir(..)
6	m2:MKD	fc2:FC	onMkdirEnd(..)

Table 1: Sample Trace Database - TDB

This section starts by describing the basic operation used during the mining process. It then follows with the algorithm to mine frequent symbolic charts. These charts are later composed to form significant LSCs. Redundant LSCs are removed on-the-fly during the composition process. As a final process, the mined significant non-redundant charts are further ranked and summarized.

Basic Operations. Before we describe the algorithm in detail we need to describe the projected-database operation to incrementally compute the instances of charts considered during the mining process.

DEFINITION 4.1 (**Projected Database**). *Given a trace database (i.e. a multi-set of traces) TDB and a chart C, the projection of TDB on C is the set of locations within the set of traces where instances of chart C ended. Formally:*

$TDB_C = \{(id, e) | t = TDB[id] \wedge (\exists (s<e).\ t[s..e]$ *is an instance of* $C)\}$

The first element of the pairs corresponds to the trace id, the second element corresponds to the ending index, in the corresponding trace, which points to the end of an instance of chart C. Thus, TDB_C contains locations within a trace database that serve as contexts where instances of C could be extended to form instances of its super-chart. The size of TDB_C is equal to the number of positive witnesses of C in TDB (when TDB_C is computed, one could automatically infer the end-of-trace instances of a chart C as well; these would correspond to locations in the projected database of C's immediate prefix that could not be extended to form an instance of C).

The instances of a length-1 chart $\langle e_1 \rangle$ are simply the occurrences of event e_1 throughout the sequences in TDB. After the projected database is computed, the instances of a length-k $\langle e_1, \ldots, e_k \rangle$ chart can be found from instances of length-(k-1) $\langle e_1, \ldots, e_{k-1} \rangle$ chart.

The support of a symbolic chart is the sum of instances of different concrete charts isomorphically obeying the bindings of the symbolic chart. In the actual implementation, we grow each corresponding concrete chart separately, as each of them has different object bindings that need to be checked. The support values of the corresponding set of concrete charts are then collated to be the support of their representative symbolic chart.

As an example, consider the trace database consisting of two traces shown in Table 1.

Given the symbolic chart C_A shown in Fig. 2(left) and the above trace database TDB, the projected database of C_A

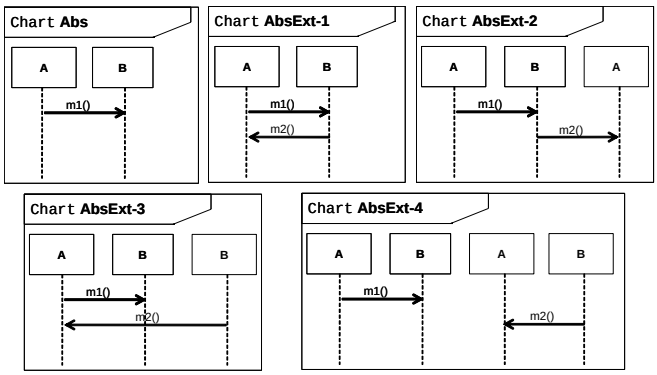

Figure 8: PBDG Examples

in TDB, denoted TDB_{C_A} is the set {(1,4),(2,3)}. The first chart instance refers to the trace segment located in the first trace, at position 0 to 4. The second chart instance refers to the trace segment located in the second trace, at position 0 to 3. To obtain the projected database of the symbolic chart, the algorithm computes the projected database for each of the corresponding concrete charts (shown in Fig. 2 (center,right)) and collate the results together.

Another basic operation computes possible bindings:

DEFINITION 4.2 (**Possible Bindings**). *Given a symbolic chart AC and a symbolic event AE, concatenating AE to AC results in a set of (non-isomorphic) possible symbolic charts. Each of the charts in the set corresponds to a different valid pair of symbolic 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* in Fig. 8. Computing $PBDG(Abs, (B, A, m2()))$ returns four possible charts as shown by charts $AbsExt$-1,2,3 and 4 in Fig. 8.

Mining Frequent Symbolic Charts. Our mining algorithm works in a depth first manner. First, chart of length 1 is considered. Consecutively, we incrementally increase the length of the chart by appending events to the previously considered charts. At each step of the mining process, we use the projected database to find instances of a particular chart. Following Property 1, when a chart under consideration is infrequent, all its extensions (i.e., all charts having this chart as a prefix) would be infrequent too. Hence, we safely stop growing the chart further when we find that a chart is infrequent. With this we prune a large search space containing infrequent charts. Furthermore, all additional filters, e.g., monotonic connectedness, etc., are applied at this phase when LSCs are composed from frequent charts. The algorithm to mine all symbolic charts is shown in Fig. 9.

It could be noted that the every time the recursive procedure MineWthPrefix is called, the parameter $CurC$ is a *frequent* symbolic chart. Also, every invocation of MineWthPrefix would be made with different $CurC$. Hence, he complexity of the procedure MineFreqCharts is then linear to the number of *frequent* symbolic charts. Representing the number of frequent symbolic chart as fsc, the complexity of the process of finding frequent symbolic charts is O(fsc) – it is linear to the size of the output charts.

Extraction of Non-Redundant Significant LSCs. Given a set of symbolic charts, significant LSCs are formed

Procedure MineFreqCharts
Inputs:
TDB : Input Trace Database
min_sup: Min. Sup. Thresh.
Output:
A set of frequent charts
Method:
1: Let AEV = Single-event charts satisfying min_sup
2: For each f_aev in AEV
3: Call MineWthPrefix(TDB_{f_aev},min_sup,AEV,f_aev)

Procedure MineWthPrefix
Inputs:
PDB : Projected database TDB_{CurC}
min_sup: Min. Sup. Thresh.
AEV: Frequent single events
$CurC$: Current chart considered
Method:
4: Output $CurC$
5: For each f_aev in AEV
6: Let $nxCSet = PBDG(C,f\_aev)$
7: Compute next projected databases from PDB
8: For each nxC in $nxCSet$
9: If ($|TDB_{nxC}| \geq min\_sup \wedge MCONN(nxC)$)
10: Call MineWthPrefix(TDB_{nxC},min_sup,AEV,nxC)

Figure 9: Mine Frequent Charts

by composing symbolic charts. Two charts pre and $full$, where pre is an isomorphic prefix of full, could be composed to form LSC $L(pre, full)$ with support $sup(full)$ and confidence $sup(full)/sup(pre)$. We are interested only in LSCs whose confidence is above the minimum confidence threshold. During composition we detect redundant LSCs on the fly. As defined in Defn. 3.6, an LSC L is redundant if there exists another LSC L' where there is an isomorphic embedding of L in L' and both LSCs have the same statistics (i.e., equal support and confidence).

A straightforward approach to extract non-redundant LSCs would be to compare each LSC with all other LSCs. However, this would be expensive – O(n^2). Since n could be large, we would like to have a more efficient approach. First, since we would like to check only the LSCs having the same support and confidence, we hash the LSCs based on their support and confidence values. Only LSCs in the same bucket would need to be compared with one another.

To further speed up the process, we check for redundant LSCs on-the-fly when new LSCs are formed and added to the bucket. A temporary data structure is used to hold candidate non-redundant LSCs. Whenever an LSC in the bucket is considered, this LSC is checked only against all candidate LSCs found so far. If there exists a candidate LSC that is a super-chart of the new LSC, this LSC would not be added as a candidate. Otherwise, it is added. During the process, candidate LSCs that are found to be embedded in newly formed LSCs are removed from the list of candidates. At the end, the candidate LSCs in each bucket would be the set of non-redundant LSCs with that particular statistics (i.e., support and confidence values).

Let us analyze the complexity of the procedure ComposeNRLSCs. The procedure combines the step of composing LSCs with detecting redundant LSCs to speed up the process. Let us split the two steps and analyze them separately.

Let fsc be the number of frequent symbolic charts. It could be noted that for a chart of size l, ignoring LSCs with empty pre- and main-charts, at most $l-1$ LSCs could be

Procedure ComposeNRLSCs
Inputs:
CHT : All frequent charts
min_conf: Min. Conf. Thresh.
Output:
A set of non-redundant significant LSCs
Method:
1: Initialize $BKTS$ as a hash table
2: For each $full$ in CHT
3: For each isomorphic prefix pre of $full$
4: Create LSC $L(pre,full)$
5: If $(conf(L) \geq min\_conf)$
6: Put L to $BKTS$ based on its support and conf.
7: Let $CandNR$ = bucket's candidate non-red. LSCs
8: Update $CandNR$ accordingly – see text
9: For each bucket in BKTS
10: Output all non-redundant LSCs in the bucket

Figure 10: Compose Non-Redundant Significant LSCs

formed. Hence, the complexity of forming LSCs satisfying minimum support and confidence thresholds is O($m \times fsc$), where m is the length of the largest frequent chart.

Let sl be the number of symbolic LSCs that satisfy minimum support and confidence thresholds. Checking of redundancy involves hashing LSCs into buckets. In the worst case, all LSCs map to the same bucket. The complexity of the redundancy check would then be O(sl^2).

Checking Isomorphic Embeddings. The algorithm performing redundancy checks described in the preceding paragraphs builds upon checking of isomorphic embeddings as a building block. To describe how isomorphic embeddings is checked, we first describe our approach to check for isomorphic charts. We then expand the approach to check for isomorphic embeddings.

Let us define a new operation LC(C) that takes in a chart C and outputs a set of constraints imposes on the lifelines as defined in Definition 4.3. The operation LC would explicitly enumerate all the conditions such as the left/right lifeline involves in the ith event is the same as the left/right lifeline involves in the jth event.

DEFINITION 4.3 (**LC(C)**). *Consider a chart C= $\langle E, L, bdg\rangle$. We define the operation LC(C) to return a set of pairs of sets:*

$\{\langle\{i|i \rightarrow (l_j, l_k) \in bdg\}, \{i|i \rightarrow (l_k, l_j) \in bdg\}\rangle | l_j \in L\}$

Each element in an LC(C) corresponds to a lifeline in L. For an element of LC(C) corresponding to lifeline l_j, the left element of the pair would list out the positions of the events in E that has l_j as the left lifeline. The right element of the pair would be the positions of the events in E that has l_k as the right lifeline.

PROPERTY 2 (**Isomorphic Check**). *Two charts c1= $\langle E1, L1, bdg1\rangle$ and c2= $\langle E2, L2, bdg2\rangle$ are isomorphic if $E1 = E2$ and $LC(c1) = LC(c2)$.*

PROOF. *Without the loss of generality, consider an arbitrary lifeline l1 in c1 that maps to the a^{th} element of LC(c1). Since LC(c1) = LC(c2), there must exists a lifeline l2 in c2 that maps to the b^{th} element of LC(c2) such that LC(c1)[a] = LC(c2)[b]. From Defn 4.3, it must be the case then for all i:*

1. *(l1,xi) in bdg1[i] iff (l2,yi) in bdg2[i]*
2. *(xi,l1) in bdg1[i] iff (yi,l2) in bdg2[i]*

Procedure IsoEmbeddingExists
Inputs:
$C1$: Smaller chart
$C2$: Larger chart
Output:
True/False
Method:
1: Return IsoEmbeddingExistsAt($C1$,$C2$,1,1)

Procedure IsoEmbeddingExistsAt
Inputs:
$C1$: Smaller chart
$C2$: Larger chart
$p1$: Index pointing to an event in C1
$p2$: Index pointing to an event in C2
Output:
True/False
Method:
2: If (p1 > |C1| ∧ p2 > |C2|)
3: Return False
4: Let matches = {p2n | p2n > p2 ∧ p-embed(C1[1..p1],C2[1..p2n])}
5: For each m in matches
6: Let exist = IsoEmbeddingExistsAt (C1,C2,p1+1,m+1)
7: if (exist) Return True
8: Return False

Figure 11: Check for Isomorphic Embeddings

Let us create a mapping map that maps such an l1 to the l2. Since LC(c1) = LC(c2), from Defn 4.3, it must be the case that map(xi) = yi. From the above, it must be the case that:

$\forall i, (k_x, k_y) = bdg1[i] iff (map(k_x), map(k_y)) = bdg2[i]$

Since $\forall i. E1[i] = E2[i]$ too, based on Defn 3.4, c1 is an isomorph of c2. □

Note that the above check could be performed in a linear time relative to the size of the LSCs. It is different from checking graph isomorphism in general, which has no known polynomial algorithm.

Next we describe our algorithm to check for the existence of an isomorphic embeddings of a chart C1 inside a chart C2 where |L1| ≤ |L2|. Let C[1..p] corresponds to the sub-chart of C containing events 1 until p in C. Let p-embed(C1,C2) be a boolean predicate that checks if there exists a sub-chart of C2 which is isomorphic to C1, and the last event of C1 is mapped to the last event of C2. Based on the above definitions, our algorithm is shown in Fig. 11.

The procedure IsoEmbeddingExists simply calls the procedure IsoEmbeddingExistsAt. The procedure IsoEmbeddingExistsAt will recursively check for embeddings of prefixes of $C1$ (the smaller chart) inside $C2$ (the larger chart). With every recursive call, a longer prefix is considered. The index $p1$ is maintained to point to the position of an event within $C1$ corresponding to the prefix under consideration. The index $p2$ is maintained to point to the position of an event within $C2$ that maps to the last event of $C1$ and $C1[1 \ldots p1]$ is embedded inside $C2[1 \ldots p2]$. At every recursion, we compute the next positions within $C2$ that is isomorphic with the prefix of $C1$ using the p-embed operation. Implementation-wise, we perform memoization to store past embeddings of the prefixes of $C1$ and $C2$ to fasten the p-embed operation. The algorithm eventually returns True if an embedding of $C1$ exists in $C2$. It returns False otherwise.

Since, for every recursive call to IsoEmbeddingExists the parameter p2 differs, the operation p-embed is called at most

Procedure SummarizeLSCs
Inputs:
NRLSC : All Non-Redundant LSCs
delta: Delta Discriminative Thresh.
Output:
A set of succinct LSCs
Method:
1: Let *Ranked* = Rank *NRLSC* – see text
2: For each *lsc* in *Ranked*
3: Let min_gain = 0
4: For each *lsc'* of higher rank than *lsc*
5: Let gain = infogain(*lsc*,*lsc'*)
6: If (gain < min_gain) min_gain = gain
7: If (min_gain ≥ *delta*)
8: Output *lsc*

Figure 12: Summarize LSCs

|C2| times. Each time p-embed is called, at most |C2| chart isomorphism checks are performed. Each isomorphic check costs |C1| + |C2| < 2×|C2| = O(|C2|). Hence, the total cost of checking isomorphic embeddings of C1 inside C2 is O(|C2|^3). This is lower than the cost of checking for subgraph isomorphism of general graphs, which is known to be an NP-Complete problem.

Ranking and Summarization. To perform the summarization, we first sort the LSCs in an ascending order based on number of lifelines, total length (i.e., number of messages), the inverse length of the premise, their confidence and their support. The sorted list is then summarized by removing those LSCs which do not give enough extra information (in terms of added messages) relative to the ones ranked above them. The delta-discrimination threshold is defined by the user; only those LSCs that have at least δ percent difference from LSCs ranked above them would be left in the summary. The algorithm to perform the above is shown in Fig. 12.

Let *nr* be the number of non-redundant symbolic LSCs that satisfy the minimum support and confidence thresholds. The complexity of the SummarizeLSCs procedure is O(nr^2) as for every non-redundant LSCs we need to compare it to all other LSCs of higher rank.

Output Guarantee. The above algorithm guarantees that all specifications mined are succinct (i.e., sound), and all succinct specifications are mined (i.e., complete). The definition and discussion on statistical soundness and completeness is given in Section 2.4.

THEOREM 1. *Our algorithm produces a statistically sound and complete set of succinct LSCs (i.e., it is significant, symbolic, non-redundant, delta-discriminative and obeys the set of filters specified by users).*

PROOF. *Soundness is guaranteed as we check each LSC output to see if it is succinct.*

Completeness is guaranteed due to the following. Our framework systematically consider the search space of all possible LSCs in a depth first search fashion. This search space is of infinite size. To make mining feasible, a pruning strategy based on Property 1 is used to cut the search space of non-frequent charts. Additional search space might be pruned based on the filters described in Section 3.4. Since only search spaces consisting of non-succinct LSCs (i.e., those not satisfying minimum support or are deemed not interesting based on the filteres) are pruned, and we exhaustive traversed all search space which are not pruned, our algorithm produces a complete set of succinct LSCs. □

App.	msup	Time(s)	|Syb.|	|NR|	|Suc.|
Jeti	10	527	323,269	50	16
Jeti	15	232	127,903	24	9
Jeti	20	1	20	7	6
CrossFTP	20	1,640	118,012	12	11
CrossFTP	30	1,635	115,234	5	4
CrossFTP	40	1,636	115,234	5	4

Table 2: Experiment result by varying trace sets and minimum support thresholds with min_conf=100%, and min_disc=10%. The columns correspond to the application where the traces come from (App.), the minimum support threshold (msup), the number of symbolic LSCs (|Syb.|), the number of non-redundant LSCs (|NR|), and the number of succinct LSCs mined (|Suc.|), respectively.

Given a specification format, a set of traces and some criteria (i.e., support, confidence, delta discrimination, etc), we would mine all specifications from the traces obeying the format and criteria. These properties are commonly obeyed by data mining tools [12, 20] and invariant generation tools like Daikon [9]. Admittedly, just like other dynamic analysis tools, the quality of the mined specifications would vary depending on the traces. This is common to dynamic analysis tasks and to data mining tools: results can only be as good as the quality of the input data.

5. EXPERIMENTAL RESULTS

We have implemented the ideas presented in this paper and evaluated them on two case study applications. All experiments were performed on a Intel Core 2 Duo 2.40GHz Tablet PC with 3.24 GB of RAM running Windows XP Professional. Algorithms were written using Visual C#.Net running under .Net Framework 2.0 with generics compiled using Visual Studio.Net 2005.

We first analyzed crossFTP server [4], a commercial open source FTP server built on top of Apache FTP server. The server consists of 15 packages containing 1148 methods in 165 classes spanning 18841 LOC. We used AspectJ to generate traces suitable for our needs, running the server with usage scenarios involving file transfers, deletions, renames, closing/ opening connections, starting/ closing the server etc. We collected 54 traces with a total length of 1876 events.

We also analyzed Jeti [1], a popular full featured open source instant messaging application. Jeti has an open plug-in architecture. It supports many features including file transfer, group chat, buddy lists, presence indicators, etc. Its core contains 49K LOC consisting of about 3400 methods, 511 classes in 62 packages. We collected 5 traces with a total length of 1797 events.

Succinctness Experiments. We vary the support values for the two sets of traces and run the mining. Table 2 contains information on the time needed to mine the scenarios for the various support thresholds, and the number of symbolic LSCs, non-redundant symbolic LSCs, and succinct LSCs mined.

From Table 2, running on traces from Jeti at the minimum support, confidence and delta-discrimination threshold set at 10, 100% and 10% respectively, the algorithm completed within 10 minutes and a total of 16 succinct scenarios were

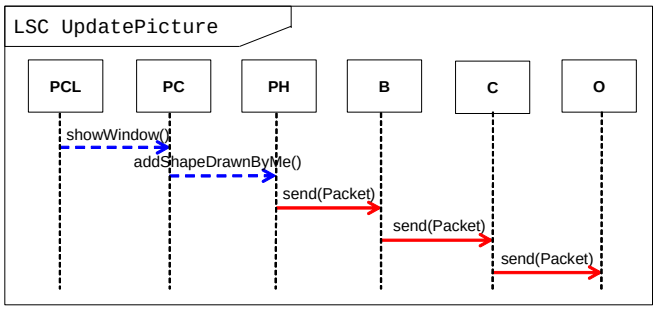

Figure 13: [Jeti] Mined LSC: Update Picture

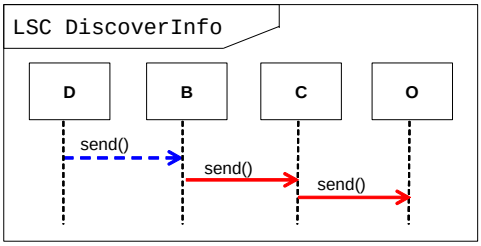

Figure 14: [Jeti] Mined LSC: Discover Information

mined. Note that mining symbolic LSCs without the redundancy filter and summarization produces 323,269 LSCs. That is, the algorithm reduced the number of LSCs mined by a factor greater than 10,000; many concrete LSCs could be merged into one symbolic LSC; many redundant and non-delta discriminative LSCs were removed.

In addition, for traces of CrossFTP at the minimum support, confidence and delta-discrimination threshold set at 20, 100% and 10% respectively, the algorithm completed within 35 minutes and a total of 11 succinct scenarios were mined. Note that mining symbolic LSCs without the redundancy filter and summarization produced 118,012 LSCs. That is, the algorithm reduced the number of LSCs mined by a factor greater than 10,000; again many concrete LSCs could be merged into one symbolic LSC; many redundant and non-delta discriminative LSCs were removed.

For Jeti traces, at higher support threshold (i.e., 20) the number of the mined LSCs is smaller. This affects also the number of redundancies found. Still the eventual number of succinct LSCs mined is smaller (6 instead of 9 or 16), showing that indeed some information was lost due to the use of the initial higher support threshold. For CrossFTP, even with higher support thresholds the number of redundant LSCs remains high as there are large mined LSCs appearing with a high support.

Some Mined Scenarios. We show some mined scenarios from Jeti and CrossFTP. Figs. 13 & 14 show two scenarios related to the drawing and internal information discovery functionalities of Jeti. Fig 13 shows that *whenever PictureChangeListener (PCL) (i.e., an object of type PCL) calls the showWindow() method of PictureChat (PC) and PC calls addShapeDrawnByMe() of PictureHistory (PH), it is the case that PH calls calls the send(Packet) method of Backend (B) which is then relayed by calling send(Packet) methods of Connect (C) and Output (C).* Fig 14 shows that *whenever Discovery (D) calls send(Packet) method of Backend (B), the packet is relayed via the send(Packet) methods of Connect (C) and Output (O).*

Figs. 15 shows an LSC mined from CrossFTP. It tells the 'story' of uploading a file to the server: *whenever a RequestHandler (RH) object calls the execute() method of a STOR command object, the latter eventually calls the getFileOffset() method of a FtpRequestImpl (FRI) object, the onUploadStart() method of a FtpletContainer (FC) object, the getDataInputStream() method of the FRI object, and the createOutputStream() of a NativeFileObject (NFO) object. Later the STOR command object calls the transfer() method of the RequestHandler (RH) object and the setUpload() method of a an FtpStatisticsImpl (FSI) object. Finally, the FSI object makes two calls to the notifyUpload() method of a StatisticsPanel (SP) object (each with a different signature, abstracted away in the diagram), and the scenario ends with the STOR command object calling the onUploadEnd() of the FC object.* Note how this mined LSC gives a comprehensive inter-object picture of the way upload requests are handled by the server. Also note the use of symbolic lifelines (notice the indefinite 'a' and definite 'the' in the description). Finally, recall that this scenario was extracted from a number of traces, where many events related to other features interleaved with the events in the scenario, including the simultaneous uploading of a number of files. Thus, abstraction from concrete to symbolic LSC had to be correctly employed.

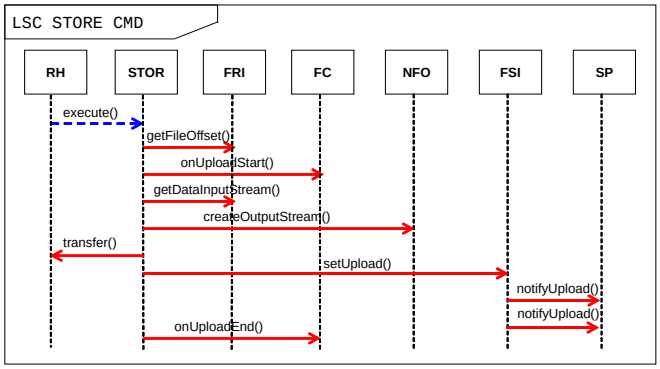

Figure 15: [CrossFTP] Mined LSC: Upload

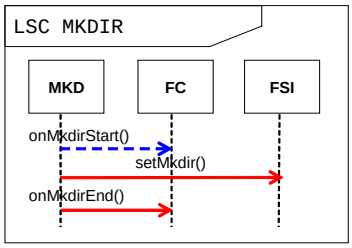

Figure 16: [CrossFTP] Mined LSC: Make Directory

Fig. 16 describes the scenario of creating a new directory on the server, involving an MKDIR command, an FC, and a FSI objects. It is interesting to note that two other mined scenarios, for moving and renaming files (not shown here), were also mined. The three scenarios share a very similar structure, differing only in the identity of the first lifeline and the names of the methods involved. As future work, it may be interesting to develop additional abstraction operators that will allow us to automatically recognize such similar structures and thus report them in a more succinct way.

More details of the case studies and additional examples of mined LSCs are available at [24].

6. RELATED WORK

Many work suggest and implement variants of dynamic

analysis based specification mining (e.g., [9] mine boolean expressions describing likely invariants on values of program variables, [3] mine temporal orderings of method calls in the form of a single finite state machines, [29] mine finite state machines with boolean expressions). Unlike these, 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 requirements (see, e.g., [13, 36]).

The challenge of succinctness in specification mining is not limited to the scenario-based approach. Some work present mining of rules and patterns (see, e.g., [8, 34, 35, 39]). At times, the number of rules and patterns could be very large. This makes it hard for users to investigate mined specifications as there are many reported rules and patterns which are very similar to one another. In Daikon [9], the number of reported invariants could also be too large. In order to address this issue, according to Daikon's documentation, Daikon may be configured to remove logically redundant invariants by using a theorem prover. In mining finite-state machines (e.g., [3, 21, 29]), at times the size of the automaton could be too large to be useful, with too many connections between nodes. This could happen especially if various un-related sub-specifications are interleaved together. To address this, Whaley et al. mine one automaton for each field in a program [37]. However, some interesting specifications capturing relations and interactions spanning across multiple fields/objects may be missed by this approach.

Some work mine frequent patterns of API usage [8, 22] and temporal invariants on API method calls [23, 39]. [22, 23] introduce non-redundancy in mining temporal patterns and rules. Instead, we consider not only method calls but also lifelines and isomorphic relationships among charts. We believe our combined succinctness strategies could be ported to these approaches and benefit them as well.

Several work propose summarization approaches. [38] proposes a summarization approach for itemset patterns, subsets that appear frequently in a set of transactions. Different from itemset patterns, which ignore the order among elements in a set, mined scenarios consider temporal relationship among events; moreover, they include object identities. Our work is thus very different than the one in [38]. [11] proposes to summarize traces by removing uninteresting events. Different from that work, we perform summarization on mined LSCs rather than events. The two approaches are orthogonal; we could use the approach in [11] to help in selecting the important events that we should trace.

7. CONCLUSION AND FUTURE WORK

In this paper we addressed the succinctness challenge of specification mining in the context of the scenario-based approach, mining a succinct set of significant LSCs from a set of program execution traces. We formulated and evaluated a definition of an equivalence relation over LSCs, a definition of a redundancy and inclusion relation based on isomorphic embeddings among LSCs, and a delta-discriminative measure based on an information gain metric on a sorted set of LSCs, all used on top of the statistical metrics of support and confidence. Our case studies showed the utility of our work in reducing the number of mined LSCs.

Work in progress includes the integration of the above methods into LM, the LSC mining tool [7]. Moreover, we are working on an integration with hierarchical scenario-based specification mining [27] and with the polymorphic extension of LSC [30], to further increase the expressive power of the mined scenarios while not compromising succinctness.

Future work includes potential generalization and application of the methods we presented in this paper to other specification mining approaches.

8. REFERENCES

[1] Jeti. Version 0.7.6 (Oct. 2006). http://jeti.sourceforge.net/.

[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 *FSE*, 2007.

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

[4] CrossFTPServer. sourceforge.net/projects/crossftpserver/.

[5] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining Object Behavior with ADABU. In *WODA*, 2006.

[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] T. Doan, D. Lo, S. Maoz, and S.-C. Khoo. LM: A Tool for Scenario-Based Specification Mining. In *ICSE*, 2010. To Appear.

[8] M. El-Ramly, E. Stroulia, and P. Sorenson. Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces. In *KDD*, 2002.

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

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

[11] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC*, 2006.

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

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

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

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

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *ICSE*, 1994.

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

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

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

[20] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI*, 2006.

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

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

[23] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[24] D. Lo and S. Maoz. Mining succinct LSCs, 2010. *www.mysmu.edu/faculty/davidlo/succinct/succinct.html*.

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

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

[27] D. Lo and S. Maoz. Mining Hierarchical Scenario-Based Specifications. In *ASE*, 2009.

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

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

[30] S. Maoz. Polymorphic Scenario-Based Specifications: Semantics and Applications. In *MoDELS*, 2009.

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

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

[33] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *ICSE*, 2007.

[34] M. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *PLDI*, 2007.

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

[36] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *FSE*, 2001.

[37] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object oriented component interfaces. In *ISSTA*, 2002.

[38] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: a profile-based approach. In *KDD*, 2005.

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

[40] Q. Yang, J. Li, and D. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 2007.