9-2010

Scenario-based and value-based specification mining: better together

David LO
Singapore Management University, davidlo@smu.edu.sg

Shahar MAOZ
Aachen University

Citation

LO, David and MAOZ, Shahar. Scenario-based and value-based specification mining: better together. (2010). *ASE '10: Proccedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, 20-24 September, Antwerp, Belgium*. 387-396.
Available at: https://ink.library.smu.edu.sg/sis_research/1348

Scenario-Based and Value-Based Specification Mining: Better Together

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

Shahar Maoz
Dept. of Computer Science 3
RWTH Aachen University, Germany
maoz@se-rwth.de

ABSTRACT

Specification mining takes execution traces as input and extracts likely program invariants, which can be used for comprehension, verification, and evolution related tasks. In this work we integrate scenario-based specification mining, which uses data-mining algorithms to suggest ordering constraints in the form of live sequence charts, an inter-object, visual, modal, scenario-based specification language, with mining of value-based invariants, which detects likely invariants holding at specific program points. The key to the integration is a technique we call *scenario-based slicing*, running on top of the mining algorithms to distinguish the scenario-specific invariants from the general ones. The resulting suggested specifications are rich, consisting of modal scenarios annotated with scenario-specific value-based invariants, referring to event parameters and participating object properties.

An evaluation of our work over a number of case studies shows promising results in extracting expressive specifications from real programs, which could not be extracted previously. The more expressive the mined specifications, the higher their potential to support program comprehension and testing.

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

1. INTRODUCTION

A specification typically imposes constraints both on sequencing of method calls or statement executions (ordering constraints), and on values that method parameters or some variables at a program point could have (value constraints). One takes a separate viewpoint from the other, and each independently, although interesting, is unable to present the full picture on the specification that a system should follow.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

Motivated by the lack of documented specifications, recently a number of studies have investigated mining of suggested specifications from program executions, e.g., [13, 26, 31]. The mined specifications, whether value-based invariants, automata/finite state machines, or scenario-based behavioral models, may be used for tasks related to program comprehension, verification, and evolution.

One pioneering work, Daikon, mines for value-based invariants that hold at user-specified program points [13]. Values of method parameters, object properties etc. are collected at selected program points during execution, and are then generalized in order to suggest invariants that hold at these points. Also, recently, we have investigated mining an expressive visual sequence-diagram-like scenario-based specification in the form of live sequence charts (LSC) [9,16] using a data-mining approach [26,28,29]. However, these have only considered ordering constraints among method calls.

In this paper, we merge the two specification mining approaches – the value-based approach of Daikon [13] and our scenario-based approach – resulting in one that mines a combination of ordering and value-based invariants. The key to the merging is a multi-step mining process and a novel dynamic slicing technique we call *scenario-based slicing*, where the mined scenarios are used as a slicing criteria over the input traces. Following the initial scenario-based mining, value-based invariants found over the sliced traces are compared against value-based invariants found over the original traces, so as to distinguish the ones unique to the scenarios context. Finally, the invariants found are attached to the mined scenarios. Thus, the resulting approach strengthens the expressive power of the mined scenarios by enriching them with *scenario-specific value-based invariants*.

To illustrate the advantages and challenges of mining scenarios with value-based invariants consider the following example, taken from one of our case study applications, CrossFTP [1], a commercial open-source FTP server. Using the scenario-based specification mining technique presented in previous work [29], we were able to mine the scenario shown in Fig. 1. Roughly, this scenario specifies that *"whenever a* `PASV` *command object calls the method* `setPasvCommand(...)` *of the* `FtpDataConnector` *(DC), and the* `DC` *calls the* `getSSL(...)` *method of a* `FtpDataConnectionConfig` *(DCC), it must eventually call the* `createServerSocket` *of an* `Ssl` *object (SSL)".* However, the mined scenario does not provide information on the values of parameters used and participating object properties *whenever this scenario indeed happened.* Are there any value-based invariants related and unique to this scenario?

Note that discovering general value-based invariants related to the methods that appear in this scenario or to its participating objects may not be good enough. The same method may be called with different parameters in different contexts and thus the invariant we may extract from its calls would be too general - in essence, too weak - not contributing to the understanding of the scenario at hand. Similarly, participating object properties may hold different values in different contexts.

Scenario-based slicing is used to address this problem. Following the process of scenario-based specification mining, we construct a sliced trace by selecting from the original traces used for mining a concatenation of only the sub-traces representing instances – positive witnesses – of the mined scenario at hand. We then look for value-based invariants twice – on the original trace and on the sliced trace – and compare the results in order to identify the scenario-specific invariants, those value-based invariants that are unique to the witnesses of the scenario.

Indeed, to continue the example just presented, we were able to find that whenever this scenario happens, the property `secure` of the `FtpDataConnection` (DC) is `true`. This invariant does *not* hold in general in our traces and hence is not suggested by Daikon when running on the original traces. However, it does hold whenever the scenario we examine happens!

Thus, the combination of value-based specification mining and scenario-based specification mining, through the use of scenario-based slicing, is able to produce expressive candidate specifications that each of the mining approaches alone is unable to produce. As shown in previous work [6,12,33,37, 41], the mined specifications may be used for tasks related to program comprehension, testing, and verification. Naturally, the more expressive the specification mined, the better it may support these tasks. Specifically, program comprehension is enhanced with stronger candidate invariants, combining execution order and values. Tests that are induced by these invariants are more accurate and hence more valuable.

We have implemented our ideas and evaluated them using a number of case study applications; see Sec. 5. The examples throughout the paper are taken from two of these, CrossFTP (mentioned above), and Jeti [3], a feature-rich instant messaging application.

Specification mining in general, and combining mining of value-based invariants with mining of ordering constraints in particular, has been recently considered and implemented (see, e.g., [31]). We discuss related work in Sec. 7.

Paper organization: Sec. 2 covers background material on LSC, scenario-based specification mining, and value-based specification mining. The syntax and semantics of scenarios with value-based invariants, our target specification formalism, are presented in Sec. 3. Sec. 4 describes the mining framework and algorithms. The results of case studies are given in Sec. 5. Sec.6 discusses some advanced issues of our work, its advantages and its limitations, Sec. 7 discusses related work, and Sec. 8 concludes.

2. BACKGROUND

We provide background material on LSC, scenario-based specification mining, and value-based invariants mining.

2.1 Live Sequence Charts

Live sequence charts (LSC) [9,16] extend classical sequence diagrams with a universal interpretation and must/may modalities. They thus allow the specification of scenario-based temporal invariants describing interactions between system objects. The language has been used in the context of execution, verification, and synthesis (see, e.g., [20, 23, 32]). A translation of LSC into temporal logics appears in [22]. A trace-based semantics for a UML2-compliant variant of LSC appears in [16]. We use here a subset of the language, with total-order events.

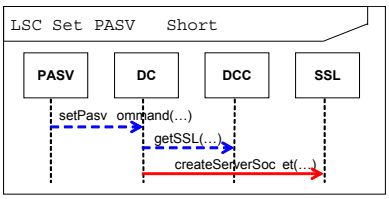


Figure 1: Example LSC: SECURE PASV

An LSC is composed of two basic charts: a *pre-chart* and a *main-chart*. A basic chart is a tuple $C = (C_L, C_E, C_<)$ where C_L is a set of lifelines representing system objects, C_E is a set of inter-object events involving the objects represented by the lifelines in C_L, and $C_<$ is a total order on C_E. Thus, a chart can also be represented as a chain of events $\langle e_1, \ldots, e_n \rangle$. We denote an LSC by $L(pre, full)$, where pre is the pre-chart and $full$ is the concatenation of the pre-chart and main-chart.

Syntactically, lifelines are drawn using vertical lines. Inter-object events are drawn using horizontal arrows from caller to callee; pre-chart events use dashed blue lines, main-chart events use solid red lines.

Semantically, an LSC specifies a temporal invariant: whenever the events in the pre-chart occur in the specified order, eventually the events in the main-chart must occur in the specified order. An LSC does not restrict events not appearing in it to occur or not to occur during a run.

Fig. 1 shows an example LSC. Roughly, this LSC means that *"whenever a `PASV` command object calls the method `setPasvCommand(...)` of the `FtpDataConnection` (DC), and the DC calls the `getSSL(...)` method of a `FtpDataConnectionConfig` (DCC), it must eventually call the `createServerSocket` of an `Ssl` object (SSL)".*

2.2 Scenario-Based Specification Mining

Scenario-based specification mining [26–29] is concerned with extracting statistically significant LSCs from inter-object traces of a system under investigation.

Inter-object trace, event. A *concrete inter-object trace* is a sequence of inter-object events. A *concrete inter-object event* ev is a tuple $\langle el_1, el_2, m \rangle$ representing an object el_1 (the caller) calling method m of object el_2 (the callee).

We define the *significance* of an LSC based on its occurrences in the traces and measure it using *support* and *confidence*, commonly used metrics in data mining [14]. Below we recall the concepts of scenario instance, positive and negative witnesses, support, and confidence, defined in [29].

Chart instance. Satisfaction of a 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 as an instance of C. A segment of a trace is said to be an instance of a chart C if it obeys the ordering specified by C. Each event in the chart must map to a corresponding event in the segment appearing in the specified order. Other events not

specified by the chart can occur in any order, unrestrictedly.

To describe an LSC chart instance, we use Quantified Regular Expressions (QRE) [35]. In our context, quantified regular expression is similar to standard regular expression with ';' as concatenation operator, '[-]' as exclusion operator (*i.e.* [-P,S] means any event except P and S[1]), and * as the standard kleene-star. The formal definition of an instance of a chart is given in Defn. 2.1 (see [29]):

DEFINITION 2.1 (**Instance of a Concrete Chart**). *Given a concrete chart $C = (C_L, C_E, C_<)$, a trace segment $SB = \langle sb_i, sb_{i+1}, \ldots, sb_{i+m-1} \rangle$ is an instance of C if SB follows the QRE expression*

$$e_1; [-G]*; e_2; \ldots; [-G]*; e_n \qquad \textit{where},$$

$C_E = \{e_1, e_2, \ldots, e_n\}$, $\forall_{0<i<n}.e_i <_C e_{i+1}$, *and* $G = C_E$.

Fig. 2 shows a short sample from a trace. The trace includes 2 instances of the LSC shown in Fig. 1: $I_1 = \langle 1, 2, 3 \rangle$, $I_2 = \langle 8, 10, 11 \rangle$.

```
1  PASV DC    setPasvCommand()
2  DC   DCC   getSSL()
3  DC   SSL   createServerSocket()
4  FRI  DC    getDataSocket()
5  PASV DC    setPasvCommand()
6  FW   FRI   getUserArgument()
7  DC   DCC   getSSL()
8  PASV DC    setPasvCommand()
9  FW   FRI   getUserArgument()
10 DC   DCC   getSSL()
11 DC   SSL   createServerSocket()
12 FRI  DC    getDataSocket()
13 PASV DC    setPasvCommand()
```

Figure 2: Part of a sample trace (`PASV` stands for the `PASV` class, `DC` for `FtpDataController`, `DCC` for `FtpDataConnectionConfig`; the actual trace includes the full qualified signatures of the classes and methods involved)

Witnesses. Based on the above definition of a chart instance, we define the notion of positive and negative witnesses of an LSC. Recall that an LSC is composed of a pre-chart and a main-chart. A *positive witness* of an LSC $L = L(pre, full)$, is a trace segment satisfying (i.e., is an instance of) the $full$ chart – by extension the *pre* chart as well, since *pre* is a prefix of $full$. A *negative witness* of L is a positive witness of *pre* that can not be extended to a positive witness of L (or $full$). We say that a negative witness is a *weak negative witness* if the positive witness of *pre* cannot be extended due to end-of-trace being reached (see discussion in [29]). We denote the set of all positive witnesses of an LSC L in a trace T by $pos(L,T)$. Similarly, we denote the set of negative witnesses as $neg(L,T)$.

Support & confidence. We use the above notions of witnesses to define the statistical *support* and *confidence* metrics for LSC. Support and confidence are commonly used statistics in data mining [14]. Given a trace T, the *support* of an LSC $L = L(pre, full)$, denoted by $sup(L)$, is simply defined as the number of positive witnesses of $full$ found in T. The *confidence* of an LSC L, denoted by $conf(L)$, measures the likelihood of a sub-trace in T satisfying L's pre-chart, to be extended such that L'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:

$$conf(L,T) \equiv_{def} \frac{|pos(full,T)| + |w\_neg(full,T)|}{|pos(pre,T)|}$$

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

The support metric is used to limit the extraction to frequently observed interactions. The confidence metric restricts mining to such pre-charts that are followed by particular main-charts with high likelihood. In scenario-based specification mining we are interested in mining statistically significant LSCs: those which occur frequently in the trace (have high support) and in which the pre- is followed by the main- chart with high likelihood (have high confidence). A chart is said to be *significant* if it obeys minimum thresholds of support and confidence, denoted by *min_sup* and *min_conf* respectively.

For the LSC shown in Fig. 1 and the trace shown in Fig. 2, $sup(L) = 2$, $conf(L) = 2/3$.

Data mining algorithms to compute a statistically sound and complete set of LSCs, given a trace (or a set of traces) and thresholds for minimal support and confidence, were presented in [29]. These were extended in [27], to handle symbolic scenario-based specifications (at the class level rather than the object level), in [26], to handle the special case of trigger and effect mining, and in [28], to take advantage of architectural hierarchies.

2.3 Value-Based Invariants Mining

Value-based dynamic detection of likely invariants is concerned with reporting likely program invariants, which hold at a certain point or points in a program's execution. Basically, dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions.

A primary example of a dynamic invariants detector is Daikon [13]. Other examples are described in, e.g., [7, 36]. As opposed to scenario-based specification mining, which, like, e.g., [6, 25, 41], is concerned with detecting temporal invariants in the form of ordering constraints over program events, these tools aim at detecting value-based invariants, e.g., in the form of `arg1 == false` or `return != null` for a certain method, `this.field has only one value` for a certain object, etc.

In our present work we integrate temporal invariants with value-based invariants. For value-based invariants detection we use Daikon. For details on Daikon internals see [13].

3. SCENARIOS WITH VALUE-BASED INVARIANTS

We now describe our target formalism, namely scenario-based specifications with value-based invariants.

We consider three types of value-based invariants inside LSCs: inv_{pre}, inv_{post}, and inv_{global}; inv_{pre} and inv_{post} are attached to LSC events, and may refer to event parameters or properties of the objects (caller and callee) involved in the event; inv_{global} invariants are attached to the LSC as a whole, and involve properties of objects participating in the LSC.

More formally, a chart with value-based invariants is a tuple $CA = (C_L, C_E, C_<, A)$ where the events in C_E are

[1]The original notation is slightly modified for brevity.

tuples $\langle el_1, el_2, m, inv_{pre}, inv_{post} \rangle$ representing an object el_1 (the caller) calling method m of object el_2 (the callee) with inv_{pre} holding immediately before the call, inv_{post} holding right after the call, and A is a set of global invariants, holding throughout the chart instance occurrence.

Semantically, an LSC $L(pre, full)$ made of basic charts annotated with value-based expressions specifies a temporal invariant: whenever the events in the pre-chart occur in the specified order, their corresponding inv_{pre} and inv_{post} expressions hold, and the pre-chart's global invariants hold throughout its occurrence, eventually the events in the main-chart must occur in the specified order, their corresponding inv_{pre} and inv_{post} expressions must hold, and the main-chart's global invariants must hold throughout the occurrence of the main-chart. Naturally, an LSC does not restrict events not appearing in it to occur or not to occur during a run, and does not restrict the properties appearing in its value-based invariants to take any value outside the LSC context.

In the visual syntax of the LSC, inv_{pre} and inv_{post} expressions may be drawn adjacent to the arrow representing their corresponding event, or in a table below the chart, together with the inv_{global} expressions.

Fig. 9 shows an example LSC annotated with a value-based invariant. The invariant found, `this.secure==true`, is a global one, related to a property of `DC`, one of the objects participating in the scenario. Additional examples are shown in Sec. 5.

Note that the LSC language as described in [9, 16] includes *conditions* (also called *state-invariants*), which specify hot/cold conditions that must/may hold during the occurrence of a scenario. Also, the variant of LSC defined in [17] includes *forbidden conditions*, which may be used as invariants over the scope of the entire scenario. Our target formalism is similar, with cold pre-chart conditions and hot main-chart conditions. However, it is also somewhat different, tying conditions directly to events pre- and post-occurrence, and specifying not what should never happen but what should always happen throughout the occurrence of a chart.

4. MINING FRAMEWORK

Our mining solution integrates Daikon [13], a value-based specification miner, with our previous solution for mining scenario-based specifications in the form of LSC [26, 28, 29]. As is shown in Fig. 3, the framework involves a number of steps: trace generation and conversion, scenario-based specification mining, scenario-based slicing, value-based invariant generation via Daikon, and selection and integration of scenario-specific invariants.

First, the input application is instrumented using the Daikon front end. Running the instrumented program produces a trace file (Daikon Trace File (DT)), which is converted to the format accepted by the scenario-based specification miner (LSC Miner Trace File (LT)). Running the scenario miner produces a set of scenarios, all of which may be further enriched with value-based invariants. For each of the scenarios, we take DT and transform it to a scenario-based sliced trace (SDT). Daikon is then invoked on the sliced and original traces, i.e., DT and SDT. A comparison of the invariants found on the sliced trace and the original trace allows us to identify scenario-specific invariants, used to enrich and strengthen the suggested scenario-based specifications. The steps are described in further detail below.

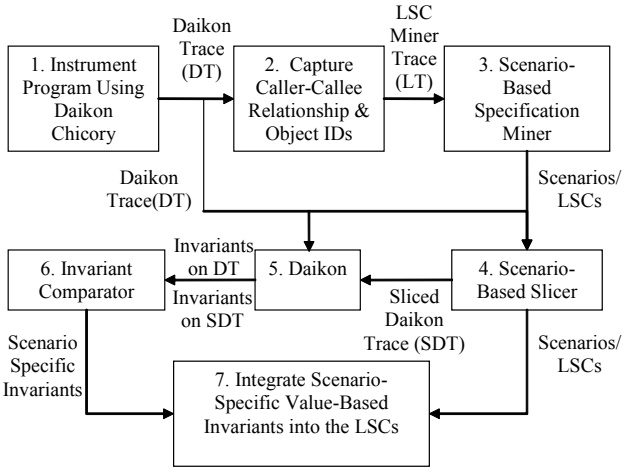


Figure 3: Mining Framework

4.1 Trace Generation and Conversion

We use the Daikon tool's front end to generate traces. Daikon provides a number of front ends for Java, C, etc., all of which produce a common trace format for Daikon's input. The trace files of Daikon contain the list of records corresponding method entries and exits during the run. Each record contains information on method signature along with the values associated with different parameter values and global variables when each of the methods was entered or exited. These traces are very rich as compared to the typical traces collected by most specification mining tools that mine for temporal ordering constraints/invariants. This is particularly needed by Daikon, so as to be able to infer value-based invariants.

On the one hand, the scenario-based specification miner looks only for temporal relationships and does not need to know about parameters and global variables. On the other hand, the scenario-based specification miner needs more information pertaining to the caller and callee of method calls. Thus, we employ a converter to extract caller-callee information based on the method entry and exit entries in Daikon trace. The converter also remove unneeded information for the scenario-based mining process including values of global variables, parameters, etc..

4.2 Scenario-Based Specification Mining

Given the converted traces, we run a scenario-based specification mining algorithm. We are interested in finding scenarios that appear more times than a specified user-defined *min_sup* threshold. Each extracted scenario must also have its main-chart appearing after each pre-chart with likelihood higher than a *min_conf* threshold.

The scenario-based specification mining algorithm works in three steps: mining frequent charts, chart composition to LSC, and chart redundancy elimination and post processing.

Frequent chart mining. The frequent chart mining algorithm is a variant of a pattern mining algorithm that models mining as a search space exploration. Different from a standard pattern mining algorithm that is agnostic to semantics of program specifications, our specification mining algorithm follows the semantics of LSC when identifying and

counting the chart/pattern occurrences in the traces. Also, since we consider scenario-based specifications in the form of sequence diagrams, the input events are not atomic symbols but rather triplets of caller, callee, and method call signature. A simplified pseudo-code is shown in Fig. 4.

Procedure MineFrequentCharts
Inputs:
T : Input trace
min_sup : Minimum support threshold
Output: All frequent charts
Method:
1: Let FEv = All frequent events in trace T
2: For each ev in FEv
3: GrowChart($ev, T, FEv, min\_sup, min\_conf$)

Procedure GrowChart
Inputs:
P : Chart so far
T : Input trace
FEv : Frequent event
min_sup : Minimum support threshold
Output: All frequent charts in the form of P++ evs
Method:
4: Let $NxtPat$ = {P++ e|e$\in$ FEv}
5: For each nxt in $NxtPat$
6: If sup(nxt) $\geq$ min_sup
7: Output nxt
8: GrowChart ($nxt, T, FEv, min\_sup, min\_conf$)

Figure 4: Frequent Chart Mining Algorithm

The frequent chart mining process starts with chart of size 1 and then tries to form longer patterns. Based on an anti-monotonicity property (see [29]), the support or number of occurrences of a pattern P should be larger or equal to the support of pattern P++ evs, where evs is one or more events. In this case, we only need to consider patterns of length one with support larger than the min_sup threshold (i.e., frequent ones) (Line 1).

Each of the frequent events is then grown to form longer frequent patterns (Lines 2-3). The search space of all patterns is traversed in a depth first fashion by appending one event at a time (Line 4). At each step one would compare the number of occurrences of a pattern to the min_sup threshold (Lines 5-6). If the minimum support threshold is not met, then based on the anti-monotonicity property, there is no need to grow the pattern further, as longer patterns would not be frequent. If the threshold is met, the algorithm will output the pattern (Line 7) and continue to try to grow the pattern further (Line 8). The algorithm will eventually terminate with the set of all frequent charts.

Chart composition to LSC. An LSC consists of pre- and main- charts and has the semantics that dictates that the pre-chart must be followed by the main-chart. Given the set of frequent charts mined, one could form LSCs by composing these charts. In particular one could pair two charts, one being a prefix of the other. Consider two charts pre and pre++$post$, one could then form the LSC having pre as the pre-chart and $post$ as the main-chart. The process is illustrated in Fig. 5.

Note that due to the anti-monotonicity property, the support of the pre-chart is larger or equal to the support of the

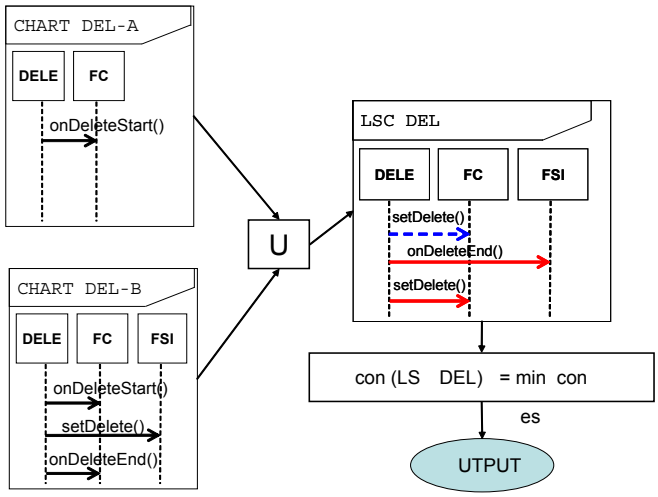


Figure 5: Chart Composition to LSC

pre-chart concatenated with the main-chart. Following the semantics of LSC, we are only interested in retrieving LSCs where the pre- is followed by the main-chart. Since the trace could be incomplete, there could be bugs in the system, and we are analyzing long running reactive systems, we provide users with the ability to extract near perfect scenarios where the pre-chart is only followed by the main-chart with likelihood less than 100%. We refer to this notion of likelihood based on the observed traces as the confidence of the LSC. Hence, given a composition of two frequent charts resulting in an LSC with confidence greater than a minimum confidence threshold min_conf, we would output the LSC. We refer to the LSCs obeying the min_sup and min_conf thresholds as the set of *significant* LSCs.

Redundancy elimination and post-processing. Often, there are too many significant LSCs. One potential root cause is that all sub-LSCs of a large and significant LSC are potentially significant too. Thus there could be a combinatorial number of mined LSCs. Thus, it would be better to mine a representative set of LSCs. To do this we only extract maximal LSCs without any larger LSCs having the same significance values of support and confidence. To do this efficiently, we first bucketize the LSCs into support and confidence value buckets. A one-to-all comparison to look for non-redundant LSCs should only then be performed among LSCs in each bucket rather than over all frequent and confident LSCs. The number of buckets depends on the number of unique combinations of support of confidence values of the LSCs. The more spread-out the distribution of support and confidence value pairs the more effective the proposed process would be. The process is illustrated in Fig. 6.

4.3 Scenario-Based Slicing

After a set of scenarios is mined, each scenario (or selected ones) may be enriched with value-based invariants. To do this, the parts of the traces that correspond to a scenario under consideration are selected. We refer to this process as *scenario-based slicing*. Consider a scenario L and a trace T, the slice of the trace T with respect to L is the sequence of positive witnesses of L in the trace T. Let ++ and $\sqsubseteq$ represent the concatenation of two sequences of events and the sub-sequence relationship between two sequences of events.

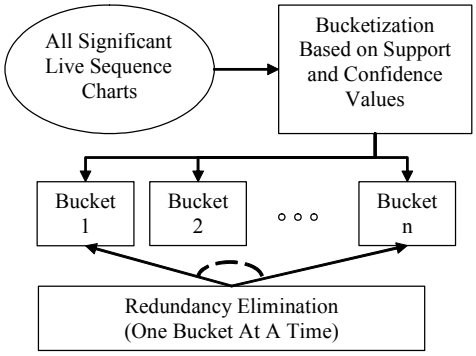


Figure 6: Redundancy Elimination

The scenario-based slicing operation is defined as follows.

DEFINITION 4.1 (**Scenario-Based Slicing**). *Consider a trace $T=\langle ev_1,\ldots,ev_n\rangle$ and an LSC L. Slicing T with respect to L produces a sub-trace ST, such that ST is the maximal sub-sequence of T composed of series of positive witnesses of L. Formally, $ST = ST_1 +\!\!+ \ldots +\!\!+ ST_n$, where $ST \sqsubseteq T$ and $\{ST_1,\ldots,ST_n\} = pos(L,T)$.*

As an example, consider the following trace:

```
1: USER | FTPWriter | send()
2: FTPWriter | FTPRequestImpl | getUserArgument()
3: USER | FTPRequestImpl | resetState()
4: PWD | FTPRequestImpl | getSystemFileView()
5: PWD | FtpWriter | send()
6: USER | FTPWriter | send()
7: FTPWriter | FTPRequestImpl | getUserArgument()
8: PWD | FtpWriter | send()
9: USER | FTPRequestImpl | resetState()
```

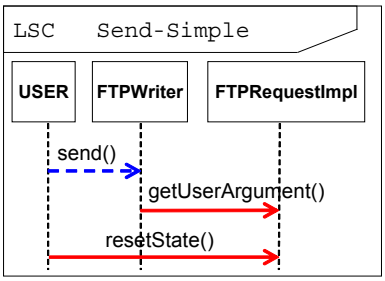


Figure 7: LSC - Send Simple

Slicing the above trace with the LSC shown in Fig. 7 results in the following sliced trace:

```
1: USER | FTPWriter | send()
2: FTPWriter | FTPRequestImpl | getUserArgument()
3: USER | FTPRequestImpl | resetState()
6: USER | FTPWriter | send()
7: FTPWriter | FTPRequestImpl | getUserArgument()
9: USER | FTPRequestImpl | resetState()
```

Note that the 4th, 5th, and 8th events in the original trace are removed. The above shows the simplified representation used for scenario-based mining. The system maintains a one-to-one correspondence between the events in the LSC mining trace and the events in the Daikon trace. Converting the events in the sliced traces back to Daikon events produces a sliced Daikon trace. With the original Daikon trace and the sliced Daikon trace, we are ready for the next step.

4.4 Scenario-Specific Invariants

Running Daikon on the original trace produces invariants for entry and exit points of each of the instrumented methods. These invariants hold for every invocation of the respective method in the traces. While these are useful, they are too general, or generic, and not scenario-specific. Thus, they are not good enough for our purpose.

On the other hand, running Daikon on the sliced trace produces invariants that hold at entry and exit points of each of the instrumented methods *only when these methods participate in the witnesses of the scenario at hand.* This is because method invocations that do not participate in the scenario are not included in the sliced trace and hence are not considered for the computation of the invariants. Based on the invariants mined on the original and sliced traces, we define the concept of *scenario-specific invariants* below.

DEFINITION 4.2 (**Scenario-Specific Invariants**). *Let inv_{orig} and inv_{sliced} be the set of invariants mined by Daikon on the original trace and sliced trace respectively. As the sliced trace is a sub-trace of the original trace, the invariants found on the former are equal to or stronger than the ones found on the latter. We distinguish the strictly stronger invariants using a comparison of the two sets. We call these invariants scenario-specific. This is the set $inv_{sliced} \setminus inv_{orig}$.*

For example, for the `PASV` scenario shown in Fig. 9, Daikon has not found an invariant related to the Boolean property `this.secure` for `DC` in the original trace (because in the original trace, this property was sometimes true and sometimes false). However, Daikon did find the invariant `this.secure==true` in the sliced trace, which included only the sub-traces representing witnesses of the `PASV` scenario.

Note that a syntactic comparison of the two sets is typically good enough for our purposes, based on the assumption that Daikon outputs semantically equivalent invariants using syntactically identical representations. We discuss this assumption and its consequences in Sec. 6.

At the end of the process, we have collected a (potentially empty) set of scenario-specific invariants including: inv_{pre}, inv_{post} for each method, and inv_{global} for the pre-chart and the main-chart. Finally, we output the mined scenarios annotated with these invariants.

5. EXPERIMENTS & EVALUATION

We have implemented our ideas and evaluated them on two case study applications: CrossFTP server [1] and Jeti instant messaging application [3]. CrossFTP is a commercial open-source FTP server built on top of Apache FTP server. It consists of 18841 LOC, 15 packages, 165 classes, and 1148 methods. Jeti [3] is a popular open-source instant messaging application. Its core contains 49K LOC, 62 packages, 511 classes, and 3400 methods.

We report here on the results of our experiments (running on an Intel Core Duo 2.4 GHz, 3.24 GB RAM Windows XP Tablet PC). The algorithms are implemented in C#.Net compiled using VS.Net 2005. We used Daikon Chicory to generate traces from CrossFTP, running it on usage scenarios involving start up, file transfers, administrator login and query, server maintenance, etc. Similarly, we used Daikon Chicory to generate traces from Jeti, running it to chat or communicate with a remote client. Various details of the

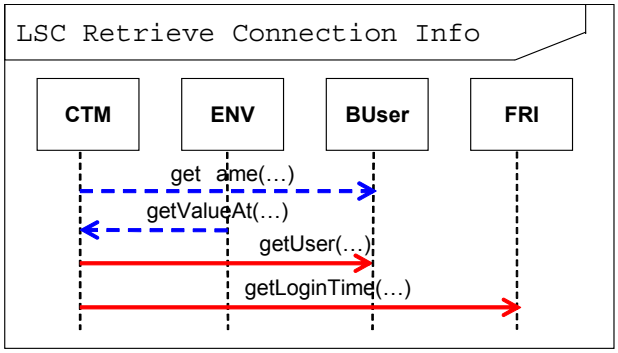
Program	CrossFTP	Jeti
Trace Size	12,217 evs	3,182 evs
Scenario Mining	53s	2s
Daikon (All)	132s	54s
Slicing Time	11s	3s
Daikon (Sliced)	31s	23s

Table 1: Experiment Details: Program, Trace Size, Scenario Mining Time, Daikon Time, Trace Slicing Time, and Daikon Mining on Sliced Traces Time

experiments including trace sizes and runtimes are shown in Table 1. We illustrate some miend scenarios in the following paragraphs.

Retrieving connection information. Fig. 8 shows a scenario from CrossFTP, specifying how the server retrieves information about the connection it is serving. In the figure, the `CTM` (`FTPConnectionTableManager`) retrieves the name and login time of the connecting user. The table at the bottom of the LSC shows the scenario-specific value based invariants found: The `col` argument of method `getValueAt()` is always set to 1 when the scenario occurs. This is *not* a general invariant for the method `CTM`, but rather a scenario-specific invariant related to the context of this scenario. Different values of `col` argument are found in the traces. Additional scenarios involving other values of `col` are given in the technical report [4].

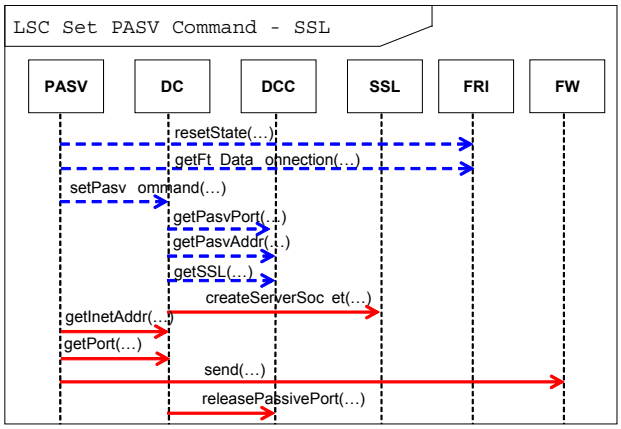
For the scenario shown in Fig. 8, a total of 37 scenario-specific invariants are reported. This is much less than the value-based invariants reported by Daikon on the original trace, consisting 5910 invariants. Indeed, out of the 11 cases described in this section and in the accompanying technical report, the number of invariants on the sliced traces is only 0.10% - 4.86% of that found in the original traces. Thus, this demonstrates an additional benefit of scenario-specific invariants: limiting the number of value-based invariants presented, focusing particularly on a scenario context under investigation.



Method	Value-Invariant Pre	Value-Invariant Post
getValueAt()	{col==1}	{}

Figure 8: Mined LSC: Retrieve Connection Info-1

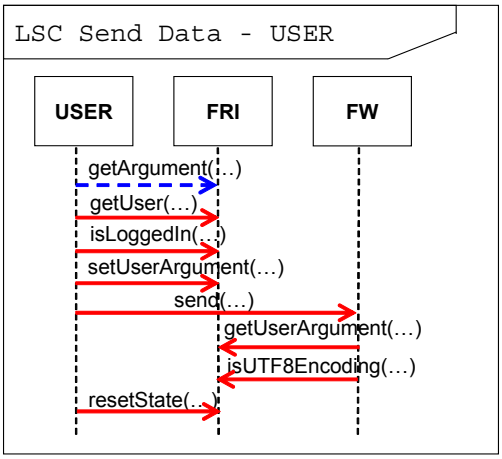
PASV FTP command. Fig. 9 shows another mined scenario, which holds when the PASV command is set. An FTP has two modes of operation, namely PORT and PASV, in addition to secure (using TLS or SSL) or regular. The scenario captures the case when PASV command is set together with SSL. We highlighted the most relevant scenario-specific invariant namely `isSecure==true`. `isSecure` is a property of the class `DC` (`FtpDataConnection`).



b ect	Value-Invariant
D	{this.secure == true,…}

Figure 9: Mined LSC: PASV Command - Secure

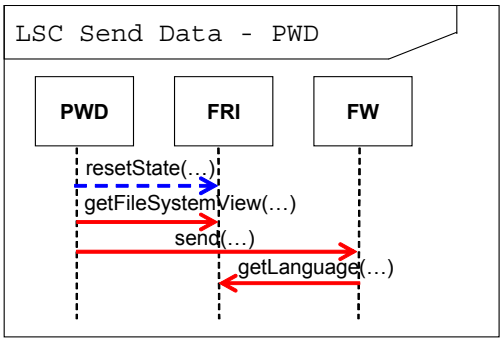
Sending data - several commands. Figs. 10 and 11 show two different scenarios for CrossFTP, where data packets corresponding to FTP commands USER and PWD are issued. Two particularly interesting scenario-specific value-based invariants related to the parameters `code` and `subId` of the method `send(...)` are found. Each of the two scenarios has unique scenario-specific value-based invariants for these two parameters. Similar examples are available in [4].



Method	Value-Invariant Pre	Value-Invariant Post
send()	{code==331, subId.toString == "USER",…}	{subId.toString == "USER",…}

Figure 10: Mined LSC: Send Data - USER, Port

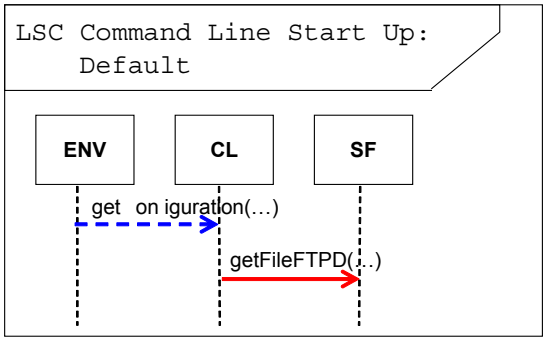
Command-line startup. Fig. 12 shows one of the scenarios when the CrossFTP server starts using the command-line option. It shows the default case when the server is started up with no parameters. It is also possible to start the server by passing an XML file. Note that for this scenario, we capture the scenario-specific invariants about the method `getConfiguration` including: the size of the input `args[]` array must be empty, and the type of the returned object must be equal to `PropertiesConfiguration`. Another vari-



Method	Value-Invariant Pre	Value-Invariant Post
send(...)	{code==257, subId.toString == "PWD", ...}	{subId.toString == "PWD", ...}

Figure 11: Mined LSC: Send Data - PWD, Port

ant involving start-up using XML file was also mined and is given in [4].

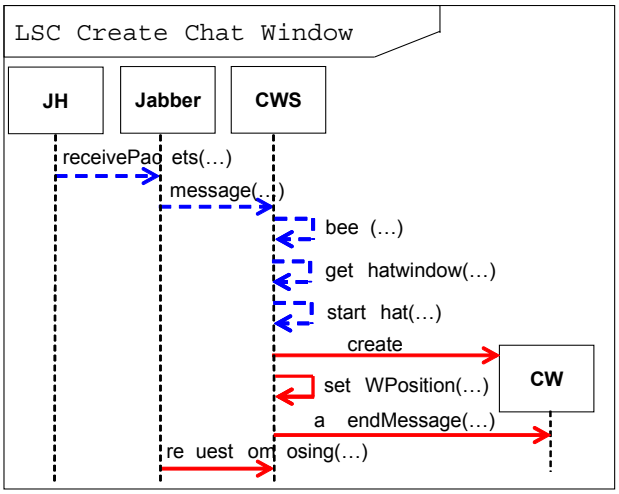


Method	Value-Invariant Pre	Value-Invariant Post
get on iguration(...)	{args .toString == , ...}	{ args .toString == , return.get lass() == org.a ache. t server.con ig. Pro erties on iguration.cla ss, ...}

Figure 12: Mined LSC: Command Line Startup Normal

Start chat window. Fig. 13 shows one of the scenarios when Jeti is used to communicate with a remote client. It specifies the scenario where a message comes, the system beeps, and a window is popped up by the Jeti client. Eventually the chat window is set up and the system is ready to accept reply messages from the user. This scenario involves JH (`JabberHandler`), Jabber, CWS (`ChatWindows`) and CW (`ChatWindow`). Note that for this scenario we capture scenario-specific invariants about the method `receivePackets`. The method accepts many different types of packets involving presence updates (e.g., Busy, Away, Extended Away, etc), error messages, etc. However, in the context of this scenario, there should be only one type of packet being received by method `receivePackets`, namely `Message`. Also, we capture the scenario-specific invariants involving the return type of method `getChatwindow`, which, in this scenario, is always null: a chat window creation occurs (the <<`create`>> event), which only happens if two parties have not communicated before, causing the call to method `getChatwindow` to return a null value.

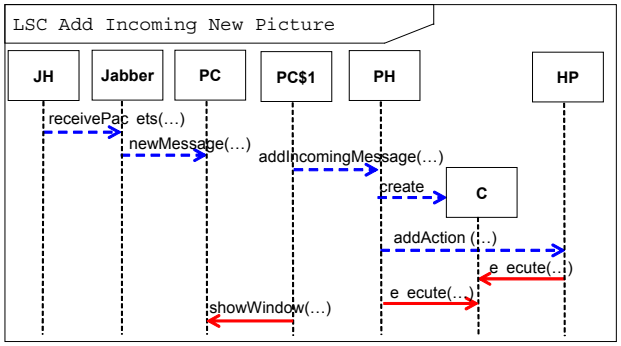
Add incoming new picture. Fig. 14 shows a scenario where Jeti received a new picture creation message from



Method	Value-Invariant Pre	Value-Invariant Post
receivePac ets(...)	{arg .get lass() == nu. w. eti. abber.elements. Message.class}	{}
get hatwindow(...)	{}	{return == null}

Figure 13: Mined LSC: Create Chat Window

another client: the packet is received, `newMessage` arrival message is passed, the history is updated, a `Creation` object is created, appropriate new picture creation methods are executed, and finally the updated window is shown. The scenario involves JH (`JabberHandler`), Jabber, PC (`PictureChat`), PC$1 (a nameless internal class of `PictureChat`), PH (`PictureHistory`), C (`Creation`) and HP (`HistoryPanel`). Note the scenario-specific invariant found for method `addIncomingMessage`. The method accepts many different types of messages involving picture updates (e.g., creation, display, deletion, change in background setting, etc.) as the first argument (i.e., arg0). However, for this scenario, there is only one type of message being received by the method, namely `CreationMessage`.



Method	Value-Invariant Pre	Value-Invariant Post
addIncomingMessage(...)	{arg .get lass() == nu. w. eti. lugins.drawing.elements. reati onMessage.class, ...}	{...}

Figure 14: Mined LSC: Add Incoming Picture

6. DISCUSSION

Choice of the target formalism. The popularity and intuitive visual nature of sequence diagrams as a specification language in general, together with the additional unique features of LSC – in particular, the universal interpretation

and its expressive power, motivate our choice for the target formalism of our mining work. Moreover, the choice is supported by previous work on LSC (see, e.g., [20, 24, 32]), which can potentially be used to visualize, analyze, manipulate, test, and verify the specifications we mine (see [29]).

Still, one may consider other scenario-based formalisms with different semantics as targets for mining (e.g., [38]), or, more generally, the mining of other useful behavioral patterns [11]. These alternatives require further investigation.

Soundness and completeness. Our previous work on scenario-based specification mining [29] was sound and complete; all mined scenarios met the minimum support and confidence thresholds (soundness), and all the scenarios meeting these thresholds were mined (completeness). Hence it is important to note that our present work on adding value-based invariants to the scenarios preserves soundness but gives up completeness. Since scenario-based mining is done first, independent of value-based mining, our current method might miss scenarios whose confidence statistics depends on their restriction with value-based invariants: when the pre-chart is restricted, the actual confidence may be higher than the one we computed without the restriction. Still, our present work is sound: all mined scenarios with their value-based invariants indeed meet the minimum support and confidence thresholds. Developing a sound *and complete* method to mining scenario-based specifications with value-based invariants is left for future work.

Identifying scenario-specific invariants. A rather simple syntactic comparison of the invariants found on the original trace and on the sliced trace suffices to identify the scenario-specific invariants we are looking for. Comparison correctness relies on the fixed and simple default syntax of Daikon's output textual representation of these invariants.

However, in some cases, a simple syntactic comparison may not be good enough because two equivalent invariants may be represented syntactically different. This indeed happened in one of our experiments, where Daikon reported `this.language == orig(this.language)` for the original trace, and `this.userArgument == this.language; this.userArgument == orig(this.language)` for the sliced trace (see Fig. 10, just after the return of `getArgument()`). To handle such cases in general, a constraint solver should be used to identify *semantic* differences, regardless of the syntactic representation. We leave this for future work.

Additional limitations. Two additional limitations of our present work should be mentioned. First, we handle only fully ordered scenarios and cannot handle partial orders. Second, in a multi-threaded environment one may be interested in mining thread-specific specifications; however, our present work is agnostic to threads. We leave these two as challenges for future work.

Integration with previous work. It is important to note that our method of adding value-based invariants to scenario-based specification mining is applicable to the various variants of the latter, that is, to the mining of scenario-based triggers and effects presented in [26] and to the mining of hierarchical scenario-based specifications presented in [28]. Also, in the present work, object IDs are abstracted away from the input traces. As discussed in previous work [27], this cannot be done in the general case; thus, we implicitly assume no overlapping LSCs. Relaxing this restriction requires further work, see [27]. We leave the implementation of these integrations and their evaluation for future work.

7. RELATED WORK

Reverse engineering of sequence diagrams. Many works present various variants of reverse engineering of objects' interactions from program traces and their visualization using sequence diagrams (see, e.g., [2, 18]). These 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'). In contrast, we look for universal (modal) sequence diagrams, which aim to abstract away from the concrete trace and reveal significant recurring potentially universal scenario-based specification, ultimately suggesting scenario-based system requirements.

Automata-based specification mining. Most specification miners produce an automaton (e.g., [5, 6, 8, 31]), and have been used for various purposes from program comprehension to verification. 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., [15, 19, 21, 38, 39]). The addition of value-based invariants strengthens the expressive power of the mined scenarios.

Mining of automata with value-based invariants. Most studies on specification mining extracting automata to date do not capture value-based invariants. Some recent works do. We briefly discuss these below.

Mariani and Pezze [34] work reports both value-based invariants and automata to help component integration. The value-based invariants are mined using Daikon, while the automata are mined using an automata learner. In contrast, we do not mine general invariants; rather, we merge value-based invariants and sequencing constraints to form scenario-specific specifications. Also, while an automaton describes the entire behavior of a system, a scenario describes only a certain aspect of it. Different from the model mined in [34], our mined scenarios also capture caller and callee relationships and present them in the intuitive visual syntax of sequence diagrams.

Lorenzoli et al. [31] integrate Daikon invariants with an automaton using a four steps approach: (1) merging all traces with the same sequence of methods invoked with different values of the parameter, (2) inferring Daikon invariants from each of the merged traces, (3) creating an initial automaton, and (4) merging locally equivalent states (based on the next k-steps) to obtain the final automaton. Our work is substantially different. First, we mine scenarios in the form of LSC, which express universal properties in the form of "Whenever the pre-chart occurs, the main chart must eventually occur". The automaton mined in [31] expresses a global/existential property, modeling all executions that are allowed in the traces. Due to this difference, the mining algorithms are very different (see [40]). Second, [31] may

'mix' between different behaviors, as it merges similar methods into one. The different context of each invocation is lost in the merging. In contrast, our use of scenario-based slicing and differencing ensures the mining of scenario-specific invariants, where the context information is preserved and highlighted.

Mining of temporal rules with equality constraints. Lo et al. [30] mine length-2 *quantified* temporal rules in the form "For all x, whenever method `A` is called with the nth parameter equals to x, method `C` would eventually be called with the mth parameter equals to x". [30] permits equality constraints. In our present work, we mine for scenarios in the form of LSCs, not limited to length two. LSCs capture caller and callee relationships not considered in [30]. While the approach in [30] is shown to work only on equality constraints, we support a wider subset of Daikon invariants. We introduce the concept of scenario-specific invariants and realize it by scenario-based mining, slicing, and differencing. However, [30] captures quantified variables, not supported in our present work. Adding quantification to our approach is left for future research.

8. CONCLUSION & FUTURE WORK

We presented scenario-based mining with value-based invariants, as an expressive extension of scenario-based specification mining in general. The key to the extension is a new technique we call scenario-based slicing, to distinguish scenario-specific invariants from general ones. The resulting suggested specifications are rich, consisting of modal scenarios annotated with scenario-specific value-based invariants, referring to event parameters and participating object properties.

An evaluation over a number of case studies shows promising results in extracting expressive specifications from real programs, which could not be extracted previously. The more expressive the mined specifications, the higher their potential to support program comprehension, testing, and verification tasks. The work is part of the larger framework of specification mining, integrating behavioral models mining with value-based invariants mining to improve the state-of-the-art support for property discovering tasks.

Future work directions relate to the challenges discussed in Sec. 6. These include, among others, the development of a complete (rather than only sound) solution, handling of partial orders, and integration with previous work through LM, the LSC mining tool [10].

Acknowledgements. This work is partially supported by Office of Research, Singapore Management University (Grant number: 09-C220-LEE-001). The second listed author acknowledges support from a postdoctoral Minerva Fellowship, German Federal Ministry for Education and Research.

9. REFERENCES

[1] CrossFTP Server. sourceforge.net/projects/crossftpserver/.
[2] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/.
[3] Jeti. Version 0.7.6 (Oct. 2006). http://jeti.sourceforge.net/.
[4] http://www.mysmu.edu/faculty/davidlo/inv/invariants.html.
[5] 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.
[6] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *POPL*, 2002.
[7] M. Boshernitsan, R.-K. Doong, and A. Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *ISSTA*, 2006.
[8] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, 2006.
[9] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
[10] T. Doan, D. Lo, S. Maoz, and S.-C. Khoo. LM: A tool for scenario-based specification mining. In *ICSE*, 2010.
[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.
[12] M. El-Ramly, E. Stroulia, and P. Sorenson. Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces. In *KDD*, 2002.
[13] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 27(2):99–123, 2001.
[14] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.
[15] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.
[16] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.
[17] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
[18] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE*, 1997.
[19] F. Klein and H. Giese. Joint structural and temporal property specification using timed story scenario diagrams. In *FASE*, 2007.
[20] J. Klose, T. Toben, B. Westphal, and H. Wittke. Check it out: On the efficient formal verification of Live Sequence Charts. In *CAV*, 2006.
[21] I. Krüger. Capturing overlapping, triggered, and preemptive collaborations using MSCs. In *FASE*, 2003.
[22] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In *TACAS*, 2005.
[23] H. Kugler and I. Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *TACAS*, 2009.
[24] M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time UML models. In *UML*, 2001.
[25] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.
[26] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *ASE*, 2008.
[27] D. Lo and S. Maoz. Mining symbolic scenario-based specifications. In *PASTE*, 2008.
[28] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *ASE*, 2009.
[29] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *ASE*, 2007.
[30] D. Lo, G. Ramalingam, V. Ranganath, and K. Vaswani. Mining Quantified Temporal Rules: Formalisms, Algorithms, and Evaluation. In *WCRE*, 2009.
[31] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, 2008.
[32] S. Maoz and D. Harel. From multi-modal scenarios to code: Compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.
[33] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *ICSE*, 2007.
[34] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis for component integration. In *Proc. IEEE Int. Conf. on Eng. of Complex Computer Systems*, 2005.
[35] K. Olender and L. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE TSE*, 16:268–280, 1990.
[36] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *CoRR*, cs.SE/0310040, 2003.
[37] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *ICPC*, 2006.
[38] G. Sibay, S. Uchitel, and V. A. Braberman. Existential live sequence charts revisited. In *ICSE*, 2008.
[39] J. Sun and J. S. Dong. Synthesis of distributed processes from scenario-based specifications. In *FM*, 2005.
[40] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data mining for software engineering. *IEEE Computer*, 42(8):35–42, 2009.
[41] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.