Autom Softw Eng
DOI 10.1007/s10515-012-0103-x

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

David Lo · Shahar Maoz

Received: 22 August 2011 / Accepted: 19 February 2012

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 a data-mining algorithm 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.

We have implemented the mining algorithm and the visual presentation of the mined scenarios within a standard development environment. 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.

Keywords Specification mining · Dynamic analysis · Live sequence charts · Value-based invariants

D. Lo (✉)
School of Information Systems, Singapore Management University, Singapore, Singapore
e-mail: davidlo@smu.edu.sg

S. Maoz
Dept. of Computer Science 3, Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: maoz@se-rwth.de

Published online: 06 March 2012

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.

Motivated by the lack of documented specifications in real-world applications, recently a number of studies have investigated mining of suggested specifications from program executions, e.g., Ernst et al. (2001); Lo and Maoz (2008a); Lorenzoli et al. (2008), and from source code, e.g., Li and Zhou (2005); Wasylkowski and Zeller (2011). The mined specifications, whether value-based invariants, automata/finite state machines, temporal rules, 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 (Ernst et al. 2001). 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) (Damm and Harel 2001; Harel and Maoz 2008), using a data-mining approach (Lo and Maoz 2008a, 2009; Lo et al. 2007). 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 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 server, a commercial open-source FTP server. Using the scenario-based specification mining technique presented in previous work (Lo et al. 2007), we were able to mine the scenario shown in Fig. 1, presented as an LSC (we show here a shortened version of this LSC, the complete mined LSC is shown in Fig. 11). 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 an* `FtpDataConnectionConfig` *object (*`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 happens*. Are there any value-based invariants related and unique to this scenario?

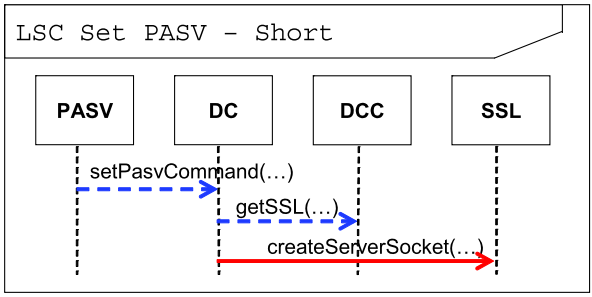


Fig. 1 Example LSC: SECURE PASV (shortened version, the complete mined LSC is shown in Fig. 11). Roughly, the LSC specifies that "*whenever a* `PASV` *command object calls the* `setPasvCommand` *method of the* `FtpDataConnector` *(DC), and the DC calls the* `getSSL` *method of an* `FtpDataConnectionConfig` *(DCC), it must eventually call the* `createServerSocket` *method of an* `SSL` *object (*`SSL`*)*"

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 a `FtpDataConnection` object (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 (Ammons et al. 2002; El-Ramly et al. 2002; Mariani et al. 2007; Safyallah and Sartipi 2006; Yang et al. 2006), 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 four case study applications. The examples throughout the paper are taken from these case study applications: CrossFTP (mentioned above), Jeti, a feature-rich instant messaging application,

Columba, an email client, and Thingamablog, a blogging utility. The implementation consists of two main parts: first, tracing and mining, and second, visualization.

Our choice of LSC as the target formalism is motivated by the popularity of the sequence diagrams notation, the additional features of LSC which support may/must modalities, and the availability of many tools that could further process the specifications that we mine. We discuss these and other advanced issues of our work, its advantages and limitations, in Sect. 7.

It is important to note that scenario-based specification mining is not aimed at finding a complete specification of the system under investigation. The scenario-based approach to modeling, in general, is not aimed at providing complete systems specifications. Rather, the strength of the scenario-based approach to modeling is that it allows the specifier to break up the specification into 'pieces of behavior', or 'scenarios', each of which cuts across multiple objects (see, e.g., Harel 2001; Uchitel et al. 2001).

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., Lorenzoli et al. 2008). These studies however, focus on mining an automaton enriched with value-based invariants. In our present work, we extract scenarios, which express temporal invariants, enriched with value-based invariants. We discuss these studies and other related work in Sect. 8.

Our earlier paper with the same title (Lo and Maoz 2010) has motivated the use of a combination of scenario-based and value-based specification mining, introduced our solution, and evaluated it on two case studies. This paper extends our previous work by (1) including additional background material that makes the paper more self-contained, (2) describing our solution in more detail, (3) presenting an end-to-end implementation, from tracing to mining to visualizing the mined LSCs using a new UML profile, together with their related value-based invariants, (4) reporting on experiments conducted on additional case studies, and (5) providing a deeper discussion of limitations and of comparison to related work.

The remainder of the paper is organized as follows. Section 2 covers important 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 Sect. 3. Section 4 describes the mining framework and algorithms. The visual presentation of the mining results using a UML profile is discussed in Sect. 5. The results of the four case studies are given in Sect. 6. Section 7 discusses some advanced issues of our work, its advantages and its limitations, Sect. 8 discusses related work, and Sect. 9 concludes.

2 Background

In this section we provide background material on LSC, on scenario-based specification mining, and on value-based invariants mining.

2.1 Live sequence charts

Live sequence charts (LSC) (Damm and Harel 2001; Harel and Maoz 2008) extend classical sequence diagrams mainly by adding a universal interpretation and

must/may modalities. They thus allow the specification of scenario-based temporal invariants describing interactions between system objects. The LSC language has been used in the context of execution, verification, testing, trace visualization, and synthesis (see, e.g., Klose et al. 2006; Kugler and Segall 2009; Maoz and Harel 2006, 2011; Maoz et al. 2009, 2011). A translation of LSC into various temporal logics appears in Kugler et al. (2005). A trace-based semantics for a UML2-compliant variant of LSC appears in Harel and Maoz (2008). We use here a subset of the language, with total-ordered events.

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. We use $+\!+$ and $\sqsubseteq$ to represent the concatenation of two sequences of events and the sub-sequence relationship between two sequences of events respectively.

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 and 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 (even in between events that do appear in the LSC).

Figure 1 shows an example LSC. Note how the difference between pre-chart and main-chart methods is reflected in the semantics of the LSC.

2.2 Scenario-based specification mining

Scenario-based specification mining (Lo and Maoz 2008a, 2008b, 2009; Lo et al. 2007) 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 (Han and Kamber 2006). Below we recall the concepts of scenario instance, positive and negative witnesses, support, and confidence, first defined in Lo et al. (2007).

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.

```
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()
```

Fig. 2 Part of a sample trace (`PASV` stands for the `PASV` class, `DC` for `FtpDataController`, `DCC` for `FtpDataConnection-Config`). Space separates caller, callee, and (shortened) method signature. The actual trace includes the full qualified names of the classes and methods involved

To describe an LSC chart instance, we use quantified regular expressions (QRE) (Olender and Osterweil 1990). In our context, a quantified regular expression is similar to the 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 Definition 1:

Definition 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

$$e_1; [-G]*; e_2; \ldots; [-G]*; e_n$$

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

Figure 2 shows a short sample from a trace. The trace includes two instances of the LSC shown in Fig. 1: $I_1 = \langle 1, 2, 3 \rangle$, $I_2 = \langle 8, 9, 10, 11 \rangle$.

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 Lo et al. 2007). 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 by $neg(L, T)$.

Support and 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 (Han and Kamber 2006). We use them here to identify significant LSCs, modulo an input trace T. 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

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

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 is used to restrict mining of 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). An LSC 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, $sup(L) = 2$, and $conf(L) = 2/3$.

A data mining algorithm 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 Lo et al. (2007). This was extended in Lo and Maoz (2008b), to handle symbolic scenario-based specifications (at the class level rather than the object level); in Lo and Maoz (2008a), to handle the special case of trigger and effect mining; and in Lo and Maoz (2009), to take advantage of architectural hierarchies.

2.3 Value-based invariants mining

Value-based dynamic detection of 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 (Ernst et al. 2001). Other examples are described in, e.g., Boshernitsan et al. (2006); Pytlik et al. (2003). As opposed to scenario-based specification mining, which, like, e.g., Ammons et al. (2002); Yang et al. (2006); Lo and Khoo (2006), is concerned with detecting temporal properties 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 (Ernst et al. 2001). Daikon can mine various kinds of value-based invariants that relate program elements of interest (e.g., global variables, parameter values, return value, fields, etc.) to constant values, other program elements, or a particular property. These relationships include equality (with regard to a constant, a null value, another program element, etc.), inequality,

containment, sortedness, etc. The invariants are reported as pre- and post-conditions of various procedures and also as object (class) invariants.

Daikon provides a number of front ends that allow one to instrument programs written in various programming languages. One front end that we are especially interested in is Chicory. Chicory instruments Java programs and produces a trace file that can later be fed to the invariant inference engine of Daikon. The invariant inference engine produces candidate invariants based on a set of templates. Candidate invariants are generated by a generate-and-check strategy. The checking process investigates whether a particular invariant of interest is observed in the input execution traces acceptably (based on a certain threshold). The invariants are then further filtered to remove redundant ones. Abstract types could also be used to remove invariants that are semantically meaningless (Guo et al. 2006). Daikon also supports a wide number of output formats to allow its output to be easily consumed by other downstream program analysis solutions. In the experiments, we use the default setting of Daikon.

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 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.

Figure 11 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 Sect. 6.

Note that the LSC language as described in Damm and Harel (2001); Harel and Maoz (2008) 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 Harel and Marelly (2003) 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

4.1 Framework overview

Our mining solution integrates Daikon (Ernst et al. 2001), a value-based specification miner, with our previous solution for mining scenario-based specifications in the form of LSC (Lo et al. 2007; Lo and Maoz 2008a, 2009). 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.

We begin by instrumenting the program at hand 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.

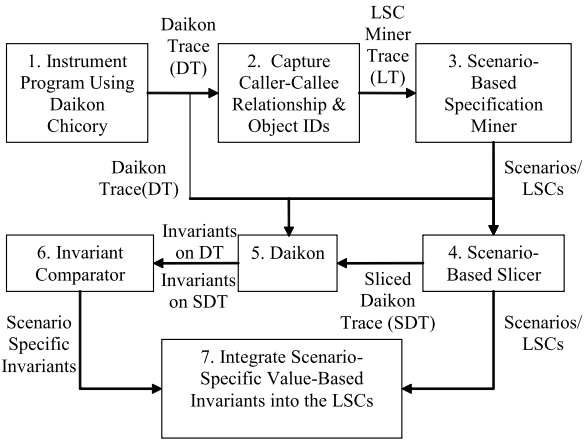


Fig. 3 An overview of our mining framework (see Sect. 4.1)

4.2 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 to 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. For an application that uses much multi-threading, we use thread identifiers to infer the caller and callee relations among method entries and exits. We modify Daikon's Java frontend (i.e., Chicory) to also output thread identifiers. The converter also removes unneeded information for the scenario-based mining process including values of global variables, parameters, etc. In sum, the converter is used to extract caller-callee relationships and object information and abstract away value-based information from Daikon traces.

4.3 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. These steps are shown in Fig. 4.

Frequent chart mining. The frequent chart mining algorithm is a variant of pattern mining algorithms that model mining as a search space exploration. The latter include algorithms mining frequent subsets (Agrawal and Srikant 1994), frequent sub-

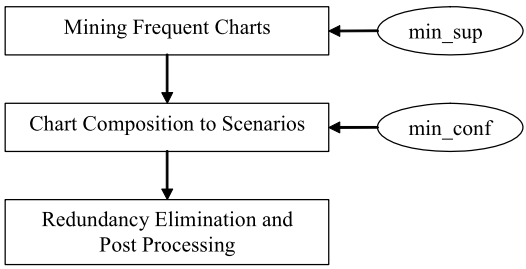


Fig. 4 The three steps of the scenario-based specification mining process: Mining Frequent Charts, Chart Composition to Scenarios, and Redundancy Elimination and Post Processing (see Sect. 4.3)

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*)

Procedure GrowChart
Inputs:
P : Chart so far
T : Input trace
FEv : Frequent events
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*)

Fig. 5 Frequent chart mining algorithm

sequences (Agrawal and Srikant 1995), etc.[2] 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. 5.

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 Lo et al. 2007), the support or number of occurrences of a pattern P should be greater than 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 greater 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

[2]Please refer to the detailed description in Sect. 8.6.

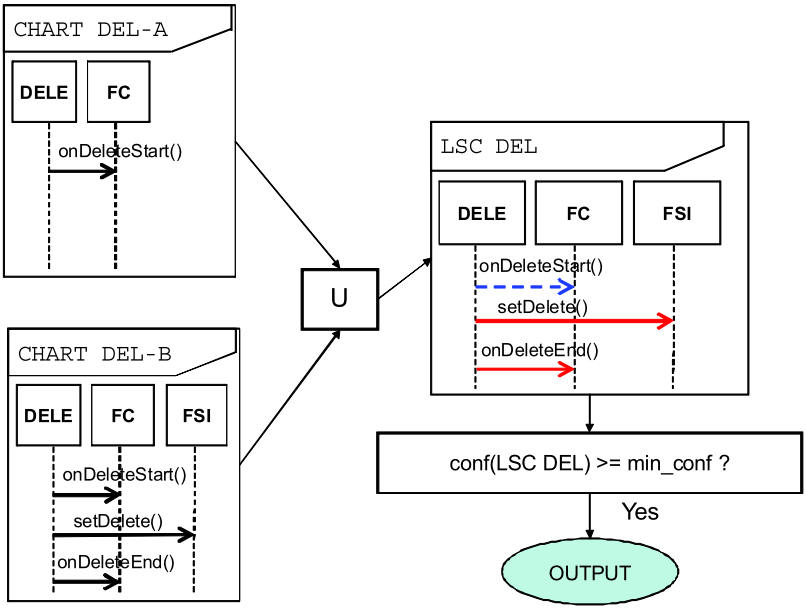


Fig. 6 Composing two charts, DEL-A (*pre*) and DEL-B (*pre++post*), to an LSC DEL ($L(pre, pre{+}{+}post)$)

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. 6.

Note that due to the anti-monotonicity property, the support of the pre-chart is greater than or equal to the support of the 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. Thus, the output of this step is the set of significant LSCs, i.e., the ones obeying the *min_sup* and *min_conf* thresholds.

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

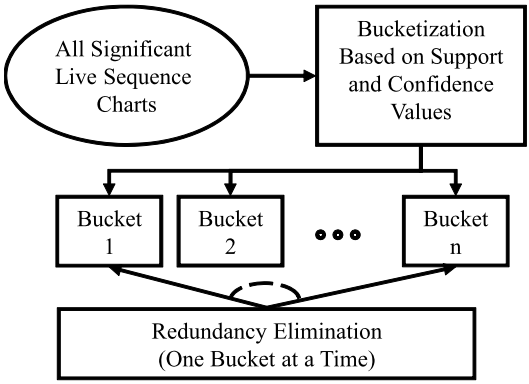


Fig. 7 Redundancy elimination. Significant LSCs are put into buckets based on support and confidence values. Each bucket is processed independently. Only maximal LSCs in each bucket would be kept

LSC are 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 is, the more effective the proposed process would be. The process is illustrated in Fig. 7.

4.4 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. The scenario-based slicing operation is defined as follows.

Definition 2 (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, $ST = ST_1 +\!\!+ \cdots +\!\!+ ST_m$, where $ST \sqsubseteq T$ and $\{ST_1, \ldots, ST_m\} = pos(L, T)$.

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()
```

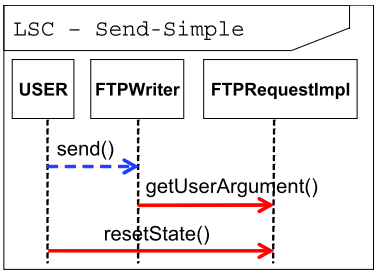


Fig. 8 LSC: Send Simple. Roughly, the LSC specifies that "*whenever a* `USER` *object calls the* `send` *method of an* `FTPWriter` *object, eventually the* `FTPWriter` *object must call the* `getUserArgument` *method of an* `FTPRequestImpl` *object, and the* `USER` *object must call the* `resetState` *method of the* `FTPRequestImpl` *object*"

```
8: PWD FTPWriter send()
9: USER FTPRequestImpl resetState()
```

Slicing the above trace with the LSC shown in Fig. 8 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.

4.5 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* as follows.

Definition 3 (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. 11, 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 Sect. 7.

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 Presenting the mined specifications

The end result of any specification mining approach consists of candidate mined specifications. For the usefulness of the approach, it is best if the presentation of these specifications to the engineer is done in a way that is intuitive, clear, and accessible on the one hand, yet formal and amenable to automated manipulation on the other hand. In this section we describe our solution for the presentation of the results of our mining work.

The output of our mining work consists of a set of scenarios with value-based invariants. For their formal presentation we use the UML2-compliant variant of LSC defined by Harel and Maoz (2008), extended with a new UML profile. A UML profile is the standard way of extending the abstract and concrete syntax of UML diagrams. We now describe the new profile we have defined.

At the abstract syntax level, our profile consists of two parts. First, it includes the temperature property defined by Harel and Maoz (2008): each message has a `Temperature`, which can be either `cold` or `hot`, corresponding to pre-chart and main-chart messages respectively. Second, it extends this basic LSC profile with five additional (optional) properties: `Support` and `Confidence` properties for each LSC, an `Invariant` property for each lifeline, and a `Pre Condition` property and a `Post Condition` property for each message.

At the concrete (visual) syntax level, the new profile's properties are manifested as follows. The temperature of every message affects its color and style: hot messages use a solid red line while cold messages use a dashed blue line. The support and confidence values of a mined LSC appear in parenthesis at the top left corner of the chart, near its name. A lifeline invariant, if any, is marked using a diamond, attached to the lifeline's head. Pre and post conditions, if any, are marked using similar diamonds, located at the beginning and end of the relevant message arrow.

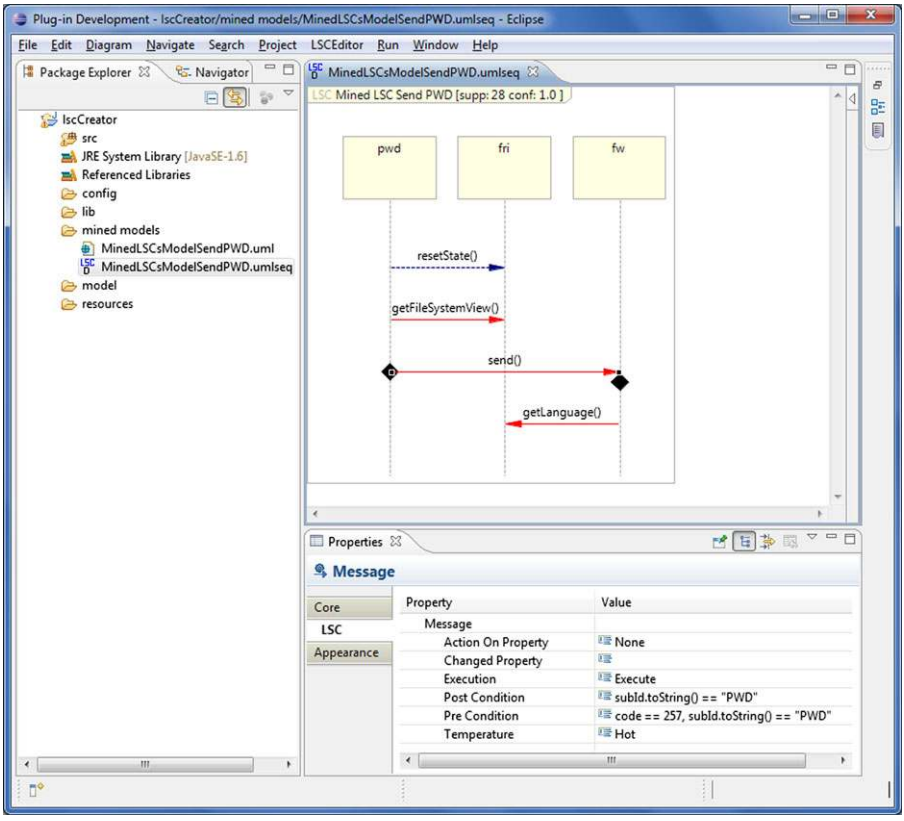


Fig. 9 Screen capture from PlayGo, showing a mined LSC. Note the support and confidence values for this mined LSC, shown at the top part of the chart, and the two *black diamonds* at the beginning and end of the method `send(...)`, marking the pre and post conditions for this method; the details of the conditions (the expressions) are shown in the message's LSC properties, displayed in the table at the lower pane of the screen. Also note the `Temperature` property of this message: the temperature is `Hot` because it is part of this LSC's main-chart

Finally, we have implemented the presentation of our results, using the new profile, in Eclipse, on top of components from PlayGo (Harel et al. 2010). The components we used provide a mechanism for extending UML2-compliant LSCs with profiles, an API to programmatically 'draw' LSCs, and means to edit and manipulate these LSCs. Figure 9 shows a screen capture from our tool, presenting one of the mined LSCs together with some of its profile properties. Note the support and confidence values for this mined LSC, shown at the top part of the chart, and the two black diamonds at the beginning and end of the method `send(...)`, marking the pre and post conditions for this method; the details of the conditions (the expressions) are shown in the message's LSC properties, displayed in the table at the lower pane of the screen. Also note the `Temperature` property of this message: the temperature is `Hot` because it is part of this LSC's main-chart.

The use of the components from PlayGo enables us to present the mining results visually and formally, within a standard setting (PlayGo is a UML2-compliant

Eclipse-based tool). It allows the engineer to browse the mined LSCs, to edit them, to print them etc. Finally, in the future it may provide support for automatic manipulation of the mined scenarios, e.g., for test-code generation, as PlayGo back-end includes the S2A compiler described in Maoz and Harel (2006); Harel et al. (2007); Maoz et al. (2011).

6 Experiments and evaluation

We have implemented our ideas and evaluated them on four case study applications: CrossFTP server, Jeti instant messaging application, Columba email client, and Thingamablog blogging utility.[3] CrossFTP is a commercial open-source FTP server built on top of Apache FTP server. It consists of about 19 K LOC, 15 packages, 165 classes, and 1148 methods. Jeti is a popular open-source instant messaging application. Its core contains about 49 K LOC, 62 packages, 511 classes, and 3400 methods. Columba is an open-source email client. Its core contains 1432 LOC, 27 packages, 93 classes, and 564 methods. Thingamablog is an open-source desktop blogging platform. Its core contains about 28 K LOC, 12 packages, 352 classes, and 2096 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. Also, we used Daikon Chicory to generate traces from Columba, running it to store an appointment, manage a contact, and prepare and send an email. Finally, we used Daikon Chicory to generate traces from Thingamablog, running it to manage (e.g., create, edit, delete, search) blogs and feeds. Various details of the experiments including trace lengths, running times etc., are shown in Table 1.

We collected traces with a total length of 12,187 events from CrossFTP, 3,182 events from Jeti, 13,015 events from Columba, and 19,511 events from Thingamablog. We ran the mining algorithms implementation with minimum support and confidence set at 25 and 100% respectively. For the first step of mining for scenarios, mining completed within 53 sec for CrossFTP, 2 sec for Jeti, 2 sec for Columba, and 6 sec for Thingamablog. For every selected scenario, aside from Daikon operations, of the 22 cases reported in this paper and accompanying website, slicing typically only took a few seconds with a few cases going closer to one minute (average was 11 sec, 3 sec, 1 sec, and 20 sec for CrossFTP, Jeti, Columba, and Thingamablog respectively). Running Daikon on the original traces took about 132 sec, 54 sec, 11 sec, and 1,038 sec for CrossFTP, Jeti, Columba, and Thingamablog respectively. Daikon took a long time to process the traces of Thingamablog as the Daikon traces contain much information—the time for it to load the traces alone is about 14 minutes.

[3]CrossFTP server is available from http://sourceforge.net/projects/crossftpserver/, Jeti is available from http://jeti.sourceforge.net/, Columba is available from http://sourceforge.net/projects/columba/, and Thingamablog is available from http://www.thingamablog.com/.

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

Program	CrossFTP	Jeti	Columba	Thingamablog
Trace Length (events)	12,217	3,182	13,015	19,511
Scenario Mining (sec)	53	2	2	6
Daikon (All) (sec)	132	54	11	1,038
Avg. Slicing Time (sec)	11	3	1	20
Avg. Daikon (Sliced) (sec)	31	23	5	138

Table 2 Experiment details: statistics of mined scenarios

Program	CrossFTP	Jeti	Columba	Thingamablog
Number of Mined LSCs	272	61	25	45
Min Length	2	2	2	2
Max Length	14	9	10	11
Average Length	6.5	4.9	4.6	5.7
Min Pre-Chart Length	1	1	1	1
Max Pre-Chart Length	13	8	8	8
Avg Pre-Chart Length	3.83	2.7	2.24	2.8
Min Main-Chart Length	1	1	1	1
Max Main-Chart Length	10	6	7	8
Avg Main-Chart Length	2.7	2.18	2.36	2.9
Min Lifelines	2	2	3	2
Max Lifelines	7	7	6	7
Average Lifelines	4.0	3.8	4.1	4

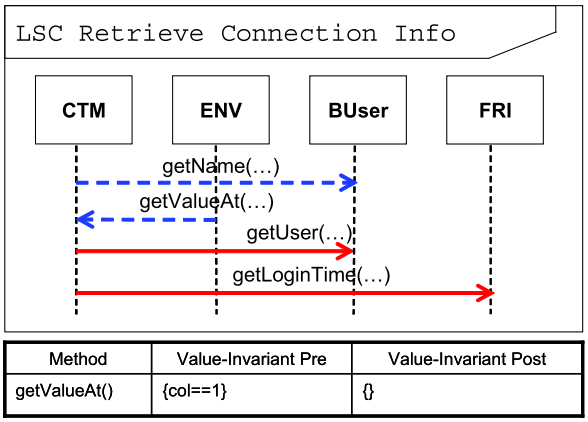
Running Daikon on the sliced traces took between 2 to 107 sec, with an average of 31 sec, 23 sec, 5 sec, and 138 sec for CrossFTP, Jeti, Columba, and Thingamablog respectively. Some statistics of the mined LSCs are provided in Table 2.

6.1 Example mined LSCs

We now describe some of the mined scenarios from the four case study applications.

6.1.1 CrossFTP

Retrieving connection information. Figure 10 shows a scenario from CrossFTP, specifying how the server retrieves information about the connection it is serving; specifically, whenever the first two methods occur, eventually the last two methods occur. 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 this method of `CTM`, but rather a scenario-specific invariant related to



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

Fig. 10 Mined LSC: Retrieve Connection Info. The LSC shows a scenario from CrossFTP, specifying how the server retrieves information about the connection it is serving; specifically, whenever the first two methods occur, eventually the last two methods occur. See Sect. 6.1.1

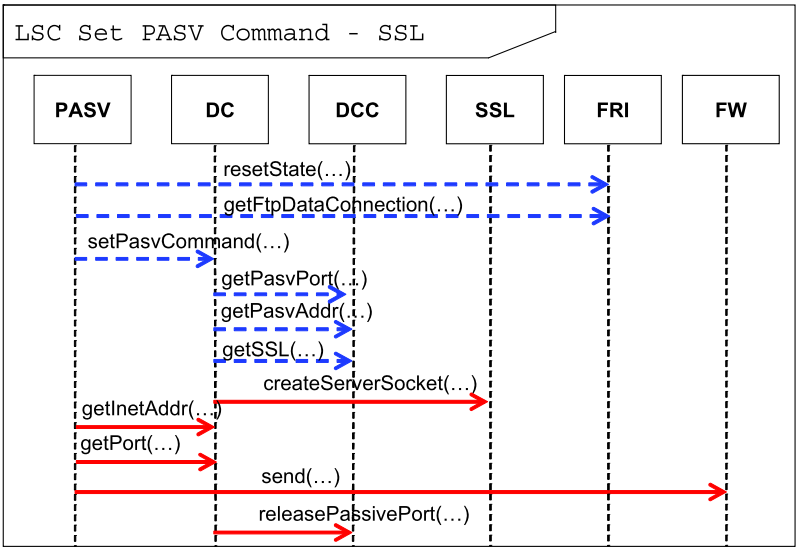
the context of this scenario. Different values of `col` argument are found in the traces. Additional scenarios involving other values of `col` are available in Accompanying Website & Technical Report (2011).

We note that there are other invariants reported aside from the *col* argument. They are there because Daikon reports a different invariant format representing the same invariant in the original and sliced trace (see our discussion of semantically equivalent invariants in Sect. 7).

PASV FTP command. Figure 11 shows another mined scenario from CrossFTP, 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`).

Sending data—several commands. Figures 12 and 13 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. Additional examples are available in Accompanying Website & Technical Report (2011).

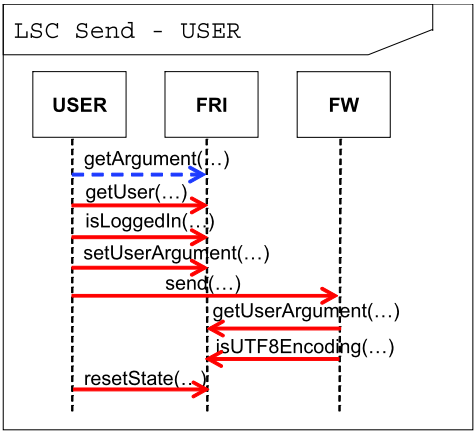
Command-line startup. Figure 14 shows one of the scenarios when the CrossFTP server starts using the command-line option. It shows the default case when the server is started up with no parameters. It is also possible to start the server by passing an XML file. Note that for this scenario, we capture the scenario-specific invariants about the method `getConfiguration(...)`: the size of the input `args[]` array must be empty, and the type of the returned object must be equal to `PropertiesConfiguration`. Another variant involving start-up using XML file was also mined (see Accompanying Website & Technical Report 2011).



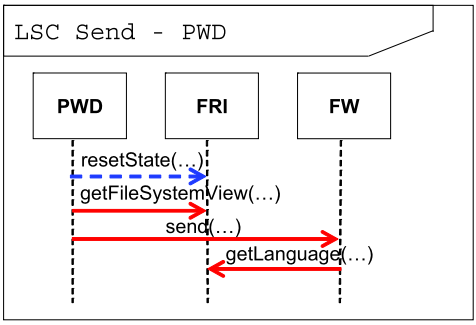
Object	Global Value-Invariant
DC	{this.secure == true,…}

Fig. 11 Mined LSC: PASV Command—Secure. The LSC shows a scenario from CrossFTP, specifying how the PASV command is set; specifically, whenever the first six methods occur, eventually the last five methods occur. See Sect. 6.1.1

Fig. 12 Mined LSC: Send Data—USER, Port. The LSC shows a scenario from CrossFTP, specifying the sending of data related to the USER command; specifically, whenever the first method occur, eventually the last seven methods occur. See Sect. 6.1.1

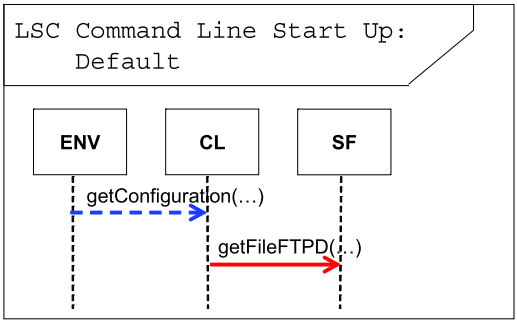


Method	Value-Invariant Pre	Value-Invariant Post
send()	{code==331, subId.toString == “USER”,…}	{subId.toString == “USER”,…}



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

Fig. 13 Mined LSC: Send Data—PWD, Port. The LSC shows a scenario from CrossFTP, specifying the sending of data related to the PWD command; specifically, whenever the first method occur, eventually the last three methods occur. See Sect. 6.1.1

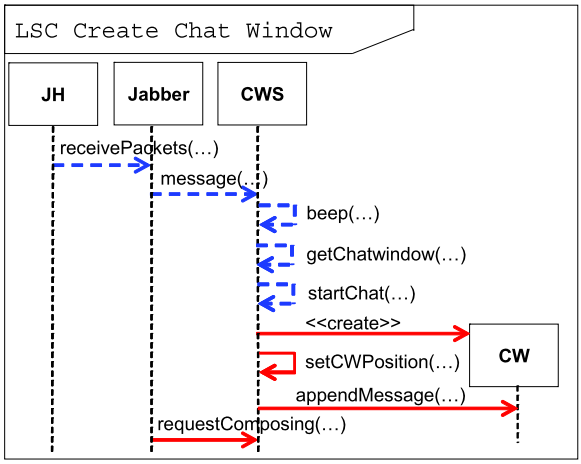


Method	Value-Invariant Pre	Value-Invariant Post
getConfiguration(…)	{args[].toString == [], …}	{ args[].toString == [], return.getClass() == org.apache.ftpserver.config.PropertiesConfiguration.class, …}

Fig. 14 Mined LSC: Command Line Startup Normal. The LSC shows a scenario from CrossFTP, specifying the startup of the server from a command-line; specifically, whenever the first method occur, eventually the last method occur. See Sect. 6.1.1

6.1.2 Jeti

Start chat window. Figure 15 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 al-



Method	Value-Invariant Pre	Value-Invariant Post
receivePackets(…)	{arg0.getClass() == nu.fw.jeti.jabber.elements.Message.class}	{}
getChatwindow(…)	{}	{return == null}

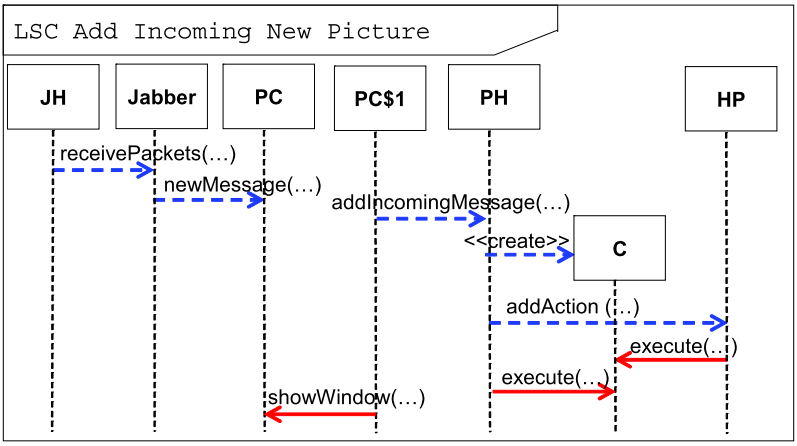
Fig. 15 Mined LSC: Create Chat Window. The LSC shows a scenario from Jeti, specifying how a new chat window is created; specifically, whenever the first five methods occur, eventually the last four methods occur. See Sect. 6.1.2

ways 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. Figure 16 shows a scenario where Jeti received a new picture creation message from 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 (`Picture-Chat`), 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`.

6.1.3 Columba

Editing a contact. Figure 17 shows a scenario from Columba, when a user edits a contact: information on appropriate folder/group containing the contact is obtained, a `ContactEditorDialog` object captures the edit made and the user's confirmation if he/she would like to proceed with the edit, the appropriate folder is modified, and finally the `AddressbookFrameController` is updated. The scenario involves EA (`EditPropertiesAction`), TC (`TreeController`), LF (`LocalFolder`), CD (`ContactEditorDialog`), and AC (`Addressbook-FrameController`). Note the scenario-specific invariant found for method

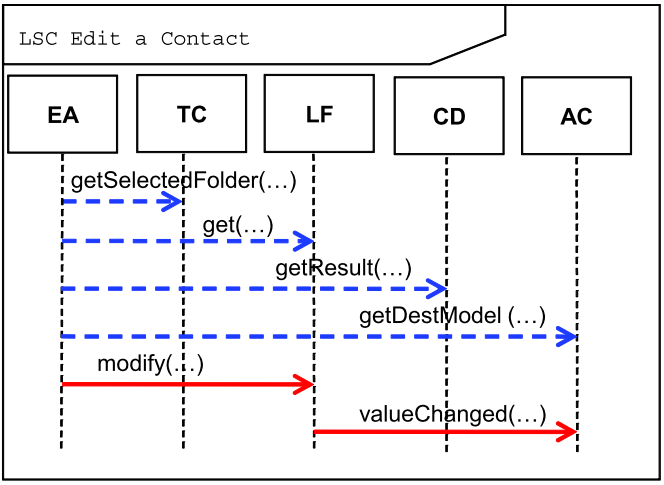


Method	Value-Invariant Pre	Value-Invariant Post
addIncomingMessage(…)	{arg0.getClass() == nu.fw.jeti.plugins.drawing.elements.CreationMessage.class, …}	{…}

Fig. 16 Mined LSC: Add Incoming Picture. The LSC shows a scenario from Jeti, specifying how an incoming picture is handled; specifically, whenever the first five methods occur, eventually the last three methods occur. See Sect. 6.1.2

`getResult(...)` and object of class `ContactEditorDialog`; both specify that the instance variable `result` must be true. This happens when a user confirms a contact's edit. Note also the scenario-specific invariant found for method `getSelectedFolder(...)`. The method returns an object of type `AbstractFolder`, which is an abstract class. The scenario-specific value invariant highlights that the object must be of type `AddressbookFolder`, which is a sub-class of `AbstractFolder`.

Adding an appointment. Figure 18 shows a scenario for Columba when a user adds an appointment: an `EditEventDialog` stores the content of the new appointment and captures the user's confirmation to store the new appointment, information on calendar is obtained, and finally `addOp(...)` message of `CommandProcessor` is invoked to instruct the `CommandProcessor` to add the appointment. The scenario involves NA (`NewAppointmentAction`), ED (`EditEventDialog`), CSF (`CalendarStoreFactory`), and CP (`CommandProcessor`). Note the scenario-specific invariant found for method `success` and object of class `EditEventDialog`; both specify that the instance variable `success` must be true. This happens when a user confirms the creation of a new appointment. Note the scenario-specific invariant found for method `addOp(...)`. The method can take a parameter of type `Command`, however, for this scenario the parameter need to be of type `AddEventCommand` which is a sub-class of `Command`. Note also the object invariant for `CommandProcessor`; the invariant specifies that Singleton design pattern is likely to be used here (which is indeed the case).



Object	Global Value-Invariant
ContactEditorDialog	{this.result == true,…}

Method	Value-Invariant Pre	Value-Invariant Post
getResult(…)	{}	{return == true}
getSelectedFolder…)	{}	{return.getClass() == org.columba.addressbook.folder.AddressbookFolder.class}

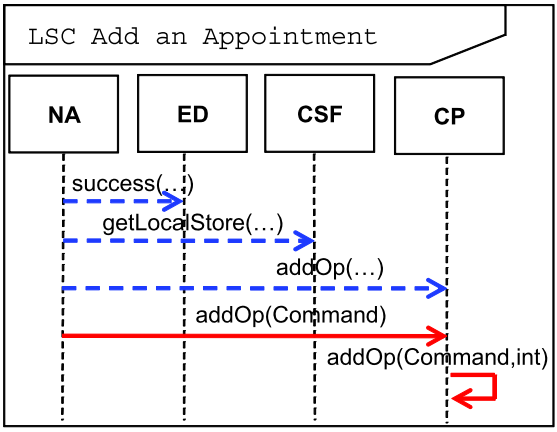
Fig. 17 Mined LSC: Edit a Contact. The LSC shows a scenario from Columba, specifying how a contact is edited; specifically, whenever the first four methods occur, eventually the last two methods occur. See Sect. 6.1.3

6.1.4 Thingamablog

Adding and updating a blog entry. Figures 19 and 20 show two related scenarios on adding and updating a blog entry in Thingamablog. The scenarios are almost the same—they mainly specify the series of events corresponding to extraction of information about the blog, followed by the actual add or update commands, and finally ending with user interface updates. The differences between the two LSCs are the methods `addEntry(...)` and `updateEntry(...)`. The scenario involves PL (`PostListener`), EE (`EntryEditor`), WL (`Weblog`), DB (`HSQLDatabaseBackend`), WTM (`WeblogTableModel`), and TF (`ThingamablogFrame`). Note the scenario-specific invariant found for method `getMode(...)` and object of class `EntryEditor`: each specifies the appropriate `mode` when a blog entry is added (i.e., `NEW_ENTRY_MODE`) and when a blog entry is updated (i.e., `UPDATE_ENTRY_MODE`).

6.2 Number of scenario-specific invariants

For the scenario shown in Fig. 10, a total of 37 scenario-specific invariants are reported. This is much less than the set of value-based invariants reported by Daikon



Object	Global Value-Invariant
EditEventDialog	{this.success == true}
CommandProcessor	this.timeStamp == org.columba.core.command.CommandProcessor.instance.timeStamp

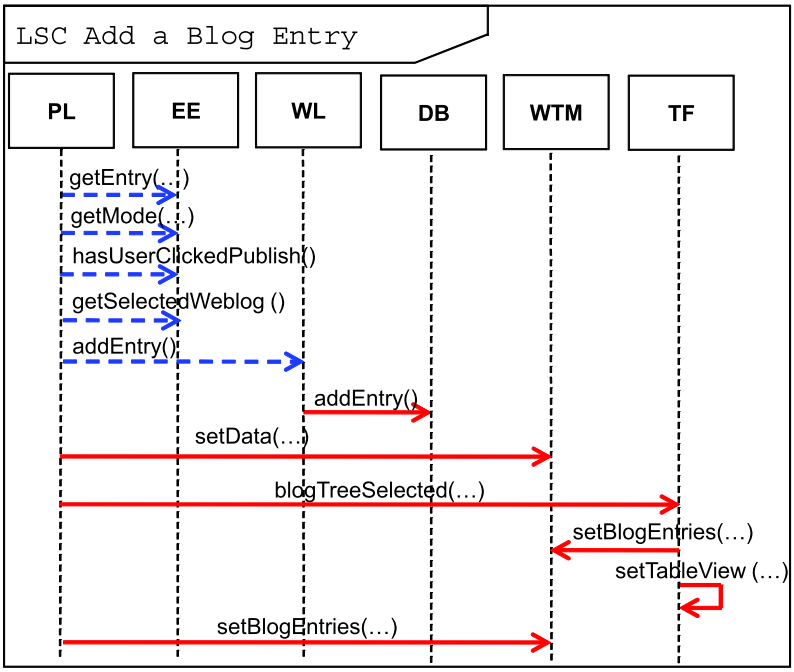
Method	Value-Invariant Pre	Value-Invariant Post
success(…)	{}	{return == true}
addOp(Command)	{arg0.getClass() == org.columba.calendar.command.AddEventCommand.class}	{}

Fig. 18 Mined LSC: Adding an Appointment. The LSC shows a scenario from Columba, specifying how an appointment is added; specifically, whenever the first three methods occur, eventually the last two methods occur. See Sect. 6.1.3

on the original trace, which consists of 5910 invariants. For the scenario shown in Fig. 15, a total of 2 scenario-specific invariants are reported. This again is much less than the set of value-based invariants reported on the original trace, which consists of 387 invariants. Indeed, out of the 22 cases described in this section and in the accompanying website, 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.

7 Discussion

We now discuss some important design choices, limitations, and future work directions to consider in specification mining in general and in the combination of scenario-based specification mining and value-based invariants in particular.



Object	Global Value-Invariant
EntryEditor	{net.sf.thingamablog.gui.editor.EntryEditor.NEW_ENTRY_MODE == this.mode}

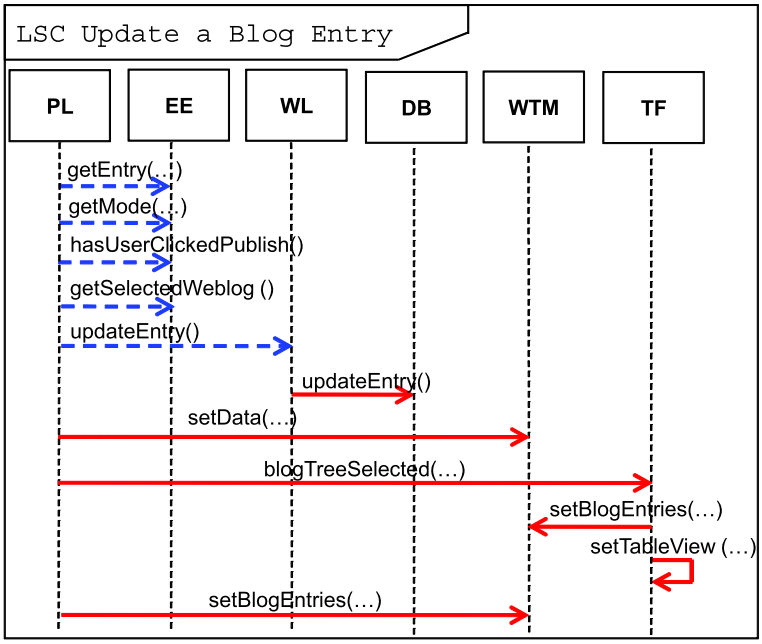
Method	Value-Invariant Pre	Value-Invariant Post
getMode(…)	{}	{net.sf.thingamablog.gui.editor.EntryEditor.NEW_ENTRY_MODE == return, …}

Fig. 19 Mined LSC: Add a Blog Entry. The LSC shows a scenario from Thingamablog, specifying how a blog entry is added; specifically, whenever the first five methods occur, eventually the last six methods occur. See Sect. 6.1.4

7.1 Choice of the target formalism

The popularity and intuitive visual nature of sequence diagrams as a specification language in general, their support in the UML standard, 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., Klose et al. 2006; Lettrari and Klose 2001; Maoz and Harel 2006; Maoz et al. 2011), which can potentially be used to visualize, analyze, manipulate, test, and verify the specifications we mine (see Lo et al. 2007).

Still, one may consider the use of scenario-based formalisms with different semantics as targets for mining (e.g., the branching time semantics of the existential scenarios described by Sibay et al. 2008), or, more generally, the mining of other,



Object	Global Value-Invariant
EntryEditor	{net.sf.thingamablog.gui.editor.EntryEditor.UPDATE_ENTRY_MODE == this.mode}

Method	Value-Invariant Pre	Value-Invariant Post
getMode(…)	{}	{net.sf.thingamablog.gui.editor.EntryEditor.UPDATE_ENTRY_MODE == return, …}

Fig. 20 Mined LSC: Update a Blog Entry. The LSC shows a scenario from Thingamablog, specifying how a blog entry is updated; specifically, whenever the first five methods occur, eventually the last six methods occur. See Sect. 6.1.4

common useful behavioral patterns (e.g., as described by Dwyer et al. 1999). These alternatives require further investigation.

7.2 Soundness and completeness

In frequent pattern mining algorithms (e.g., Li et al. 2006), soundness (or correctness) is defined as the ability to mine for *only* significant patterns or rules. Completeness is defined as the ability to mine *all* significant patterns or rules. We follow these definitions. Similar to other dynamic analysis techniques, we can only be as sound and as complete as the input traces are. There are other studies that investigate ways to generate test cases of good coverage, e.g., by Xu et al. (2010) and by Santelices et al. (2008), which is beyond the scope of our present work.

Our previous work on scenario-based specification mining (Lo et al. 2007) was (statistically) sound and complete; all mined scenarios met the minimum support and confidence thresholds (soundness), and all the scenarios meeting these thresh-

Table 3 An example trace to illustrate the incompleteness of our current approach

Caller's Class	Callee's Class	Method	Parameter value
X	A	a(int mode)	mode = 1
X	B	b(int mode)	mode = 1
X	A	a(int mode)	mode = 1
X	B	b(int mode)	mode = 1
X	A	a(int mode)	mode = 2
X	C	c(int mode)	mode = 2
X	A	a(int mode)	mode = 2

olds 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. We explain why and give an example below.

Since scenario-based mining is done first, independently of value-based mining, our current method might miss scenarios whose confidence statistics depends on their restriction with value-based invariants: when a scenario is restricted, its actual confidence may be higher than the one we computed without the restriction. Thus, our current method is incomplete. Still, our current method is sound: all mined scenarios with their value-based invariants indeed meet the minimum support and confidence thresholds.

The following example illustrates the incompleteness of our current approach. Consider the trace shown in Table 3, the thresholds *min_sup* and *min_conf* set at 2 and 100% respectively, and the LSC `L1` specifying that whenever `X` calls `A.a(1)`, eventually `X` must call `B.b(1)`. Unfortunately, although `L1` meets the minimum support and confidence thresholds, it will not be included in our results. This is the case because without the value-based invariant (i.e., by omitting the condition *mode* = 1), the confidence of `L1` is less than 100%: it is not true that whenever `X` calls `A.a(...)`, eventually `X` must call `B.b(...)`—when *mode* = 2, there is a case where `X` call to `A.a(...)` is not followed by a call from `X` to `B.b(...)`.

Developing a sound *and complete* method to mining scenario-based specifications with value-based invariants is left for future work.

7.3 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 semantically 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. 12, 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 of the invariants. We leave this for future work.

7.4 Additional limitations

A number of additional limitations of our work deserve further discussion.

First, in some cases, the number of mined scenarios and value-based invariants could be too high to be effectively presented to the engineer. To partly address this issue, in Lo and Maoz (2011) we have proposed a number of strategies to summarize mined scenario-based specifications (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 in Lo and Maoz (2011) are indeed necessary). Moreover, Daikon allows one to "turn-off" some of the invariant templates and thus reduce the number of reported invariants (Ernst et al. 2007). Employing these summarization techniques to the integrated solution of scenarios with value-based invariants is left for future work.

Second, in some cases there may be no scenario-specific invariants found; this happens in applications where most methods are only used in a single context. Thus, it is not always the case that scenario-based mining and value-based mining are 'better together'. It is recommended that the engineer would also examine the value-based invariants found by Daikon independently of the scenarios.

Third, just like all other dynamic analysis approaches, the quality of our analysis results depends on the quality of the input traces. We believe this limitation could be addressed by employing the many studies that investigate ways to augment test suites to generate a reasonable code coverage (Xu et al. 2010; Santelices et al. 2008).

Fourth, our current method only handles fully ordered scenarios and cannot handle partial orders. Extending our approach to handle partial orders is left for future work.

Fifth, similar to other frequent pattern mining algorithm, (e.g., Agrawal and Srikant 1994, 1995; Wang and Han 2004), it is often not easy to decide the exact thresholds of minimum support and confidence for a pattern (in our case LSC) to be considered significant. In practice, one could first try a relatively high minimum support threshold and gradually decrease this threshold until a reasonable number of LSCs are mined. If a user would like to increase or reduce the number of LSCs mined, he could simply increase or reduce the minimum support and confidence thresholds respectively.

Finally, in a multi-threaded environment one may be interested in mining thread-specific specifications; however, our present work is agnostic to threads. We leave this too as a challenge for future work.

7.5 Integration with other studies

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 Lo and Maoz (2008a) and to the mining of hierarchical scenario-based specifications presented in Lo and Maoz (2009). Also, in the present work, object IDs are abstracted away from the input traces. As discussed in our previous work (Lo and Maoz 2008b), this cannot be done in the general case; thus, we implicitly assume no overlapping LSCs. Relaxing this restriction requires further work, see Lo and Maoz (2008b). We leave the implementation of these integrations and their evaluation for future work.

There have been a number of studies that could verify either statically or during runtime the satisfaction of a particular LSC (Klose et al. 2006; Maoz et al. 2011). By integrating the approaches together, potentially we could flag bugs or even security violations (see, e.g., the work by Milea et al. 2011). We leave the integration of our work with these other studies for future work.

8 Related work

We now discuss related studies in the areas of reverse engineering sequence diagrams, automata-based specification mining, mining of automata with value-based invariants, mining of temporal rules with equality constraints, static specification mining, and frequent pattern mining.

8.1 Reverse engineering of sequence diagrams

Many studies present various variants of reverse engineering of objects' interactions from program traces and their visualization using sequence diagrams (see, e.g., EclipseTPTP 2011 (Eclipse TPTP) and the work of Jerding et al. 1997). 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. These 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 specifications, ultimately suggesting scenario-based system requirements.

8.2 Automata-based specification mining

Most specification miners, dynamic and static (see below), produce an automaton (e.g., Ammons et al. 2002; Acharya et al. 2007; Dallmeier et al. 2006; Lorenzoli et al. 2008), 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., Harel 2001; Klein and Giese 2007; Krüger 2003; Sibay et al. 2008; Sun and Dong 2005). The addition of value-based invariants strengthens the expressive power of the mined scenarios.

8.3 Mining of automata with value-based invariants

Most existing studies on specification mining that extract automata do not capture value-based invariants. Some recent works do. We briefly discuss these below.

Mariani and Pezzè (2005) report 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 by Mariani and Pezzè (2005), our mined scenarios also capture caller and callee relationships and present them in the intuitive visual syntax of sequence diagrams.

Lorenzoli et al. (2008) 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 and we believe the two studies enrich each other. First, the automaton mined by Lorenzoli et al. (2008) combines all scenarios together into one representation, which could result in a complicated structure. In our work, we report each significant scenario one-by-one. Second, 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 Lorenzoli et al. (2008) 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 Xie et al. 2009). Third, the work of Lorenzoli et al. (2008) 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.

8.4 Mining of temporal rules with equality constraints

Lo et al. (2009) 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". Lo et al. (2009) 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 by Lo et al. (2009). While the approach in Lo et al. (2009) 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, Lo et al. (2009) capture quantified variables, not supported in our present work. Adding quantification to our approach is left for future research.

8.5 Static specification mining

The above mentioned related studies all mine program specifications in various formats from execution traces. There are also a number of studies that mine from source code (or byte code), i.e., by performing static analysis. We highlight some of them below.

Acharya et al. use a model checker to generate static traces that are later used to infer an automaton given a particular trigger automaton (Acharya et al. 2006). In Acharya et al. (2007), the authors also use a model checker to generate static traces and infer an automaton by means of a partial order mining algorithm. Quante and Koschke (2007) develop a static analysis approach that extracts multiple object-process graphs that are a projection of a control-flow graph on a single object. These object-process graphs are then converted to a finite-state machine by some transformation operations.

Li and Zhou (2005) propose PR-Miner, which mines for rules from program code by utilizing association rule mining algorithm. The association rule mining algorithm is agnostic to the order in which various program elements appear. Weimer and Necula (2005) extract temporal rules by statically analyzing program code and contrasting two sets of statically generated traces (error vs. normal traces). Goues and Weimer (2009) propose an approach to eliminate false positives in the mined specifications. Gabel and Su (2008) propose a symbolic algorithm based upon Binary Decision Diagram (BDD) to mine specifications. Shoham et al. (2007) propose a two-phase inter-procedural static analysis approach, which performs abstract trace collection and summarization, to mine specifications. Ramanathan et al. (2007a) perform an inter-procedural path-sensitive static analysis to infer function precedence protocols, i.e., relationships among function calls. They later extend their approach to infer data-flow and control-flow pre-conditions that must be satisfied before a function is entered, by performing an inter-procedural path-sensitive static analysis (Ramanathan et al. 2007b).

Wasylkowski and Zeller (2009, 2011) mine temporal rules as Computational Tree Logic (CTL) properties by leveraging a model checking algorithm and using concept analysis. Livshits et al. (2009) infer information flow specifications by classifying nodes in an interprocedural explicit information flow graph into sources, sinks, and sanitizers.

8.6 Frequent pattern mining

In the data mining community, frequent pattern mining is a research topic that has caught much interest (Agrawal and Srikant 1994, 1995; Wang et al. 2003; Han and Pei 2000; Wang and Han 2004). We highlight some well-known studies.

Association rule mining proposed by Agrawal and Srikant (1994) is the pioneer in this research topic. Given a multi-set of transactions, where each transaction is

simply a set of items, association rule mining finds rules that state: "If a transaction contains a set of items X, then it will also contain the set of items Y". Association rule mining first finds sets of items that appear in many transactions (aka. frequent itemsets); these itemsets are then composed to form association rules. The concepts of support and confidence are used to remove insignificant rules based on user defined thresholds. There have been various studies that improve the efficiency of the first algorithm proposed by Agrawal and Srikant (1994), e.g., by Wang et al. (2003) and by Han and Pei (2000).

Sequential pattern mining, also proposed by Agrawal and Srikant, extends association rule mining to mine for frequent patterns in sequences of transactions (Agrawal and Srikant 1995). Given a multi-set of sequences of transactions (aka. sequence database), sequential pattern mining finds patterns that are sub-sequences of many sequences in the sequence database. The concept of support is used to remove infrequent patterns based on a user defined threshold. There have been various studies that improve the efficiency of the original sequential pattern mining algorithm, e.g., by Wang and Han (2004) and by Lo et al. (2008).

Different from association rule mining and sequential pattern mining, in our present work we mine a different form of patterns/rules; we recover modal sequence diagrams decorated with value-based invariants. To mine these modal sequence diagrams, we analyze sequences of events, where each event is a triple containing caller, callee, and method signature information. Furthermore, our notion of chart instance, support, and confidence is based on the formal semantics of LSCs (Damm and Harel 2001; Harel and Maoz 2008). Association rule mining and sequential pattern mining make use of subset and sub-sequence relations to count the support and confidence of the patterns and rules. In our work, we use the semantics of LSCs to decide trace segments that are instances of a given LSC.

9 Conclusion and future work

We have presented scenario-based mining with value-based invariants, as an expressive extension of scenario-based specification mining. 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. The candidate specifications are presented to the engineer in a visual, accessible, standard way, using a UML profile.

An evaluation over four 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 include addressing the challenges and opportunities discussed in Sect. 7. These include, among others, the development of a complete (rather

than only sound) solution, handling of partial orders, and integration with previous work, e.g., enhancing triggers and effects mining (Lo and Maoz 2008a) with value-based invariants. It would also be interesting to perform a user study to further evaluate the utility of mined specifications for program understanding.

Acknowledgements We would like to thank Smadar Szekely for her advice and assistance in using components from PlayGo for the profile definition and the presentation of the mined scenarios.

References

Accompanying Website & Technical Report: LSC mining with value-based invariants. Supplementary material (2011). http://www.mysmu.edu/faculty/davidlo/inv/invariants.html

Acharya, M., Xie, T., Xu, J.: Mining interface specifications for generating checkable robustness properties. In: Proc. of International Symposium on Software Reliability Engineering (ISSRE), pp. 311–320 (2006)

Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proc. of Joint Symposium on the Foundations of Software Engineering and European Software Engineering Conference (ESEC/SIGSOFT FSE), pp. 25–34 (2007)

Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. of International Conference on Very Large Data Bases (VLDB), pp. 487–499 (1994)

Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. of International Conference on Data Engineering (ICDE), pp. 3–14 (1995)

Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proc. of Symposium on Principles of Programming Languages (POPL), pp. 4–16 (2002)

Boshernitsan, M., Doong, R.K., Savoia, A.: From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In: Proc. of International Symposium on Software Testing and Analysis (ISSTA), pp. 169–180 (2006)

Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: Proc. of International Workshop on Dynamic Analysis (WODA), pp. 17–24 (2006)

Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Form. Methods Syst. Des. **19**(1), 45–80 (2001)

Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. of International Conference on Software Engineering (ICSE), pp. 411–420 (1999)

EclipseTPTP: Eclipse test and performance tools platform (2011). http://www.eclipse.org/tptp/

El-Ramly, M., Stroulia, E., Sorenson, P.G.: From run-time behavior to usage scenarios: an interaction-pattern mining approach. In: Proc. of International Conference on Knowledge Discovery and Data Mining (KDD), pp. 315–324 (2002)

Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. **27**(2), 99–123 (2001)

Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)

Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: Proc. of International Conference on Software Engineering (ICSE), pp. 51–60 (2008)

Goues, C.L., Weimer, W.: Specification mining with few false positives. In: Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 292–306 (2009)

Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D.: Dynamic inference of abstract types. In: Proc. of International Symposium on Software Testing and Analysis (ISSTA), pp. 255–265 (2006)

Han, J., Kamber, M.: Data Mining Concepts and Techniques. Morgan Kaufmann, San Mateo (2006)

Han, J., Pei, J.: Mining frequent patterns by pattern-growth: methodology and implications. ACM SIGKDD Explor. **2**(2), 14–20 (2000)

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

Harel, D., Maoz, S.: Assert and negate revisited: modal semantics for UML sequence diagrams. Softw. Syst. Model. **7**(2), 237–252 (2008)

Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Berlin (2003)

Harel, D., Kleinbort, A., Maoz, S.: S2A: A compiler for multi-modal UML sequence diagrams. In: Proc. of International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 121–124 (2007)

Harel, D., Maoz, S., Szekely, S., Barkan, D.: PlayGo: towards a comprehensive tool for scenario based programming. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 359–360 (2010)

Jerding, D.F., Stasko, J.T., Ball, T.: Visualizing interactions in program executions. In: Proc. of International Conference on Software Engineering (ICSE), pp. 360–370 (1997)

Klein, F., Giese, H.: Joint structural and temporal property specification using timed story scenario diagrams. In: Proc. of International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 185–199 (2007)

Klose, J., Toben, T., Westphal, B., Wittke, H.: Check it out: on the efficient formal verification of live sequence charts. In: Proc. of International Conference on Computer Aided Verification (CAV), pp. 219–233 (2006)

Krüger, I.: Capturing overlapping, triggered, and preemptive collaborations using MSCs. In: Proc. of International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 387–402 (2003)

Kugler, H., Segall, I.: Compositional synthesis of reactive systems from live sequence chart specifications. In: Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 77–91 (2009)

Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 445–460 (2005)

Lettrari, M., Klose, J.: Scenario-based monitoring and testing of real-time UML models. In: Proc. of International Conference on the Unified Modeling Language (UML), pp. 317–328 (2001)

Li, J., Li, H., Wong, L., Pei, J., Dong, G.: Minimum description length principle: generators are preferable to closed patterns. In: Proc. of Conference on Artificial Intelligence (AAAI) (2006)

Li, Z., Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. of Joint Symposium on the Foundations of Software Engineering and European Software Engineering Conference (ESEC/SIGSOFT FSE), pp. 306–315 (2005)

Livshits, V.B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: Proc. of Conference on Programming Language Design and Implementation (PLDI), pp. 75–86 (2009)

Lo, D., Khoo, S.C.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: Proc. of Symposium on the Foundations of Software Engineering (SIGSOFT FSE), pp. 265–275 (2006)

Lo, D., Maoz, S.: Mining scenario-based triggers and effects. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 109–118 (2008a)

Lo, D., Maoz, S.: Specification mining of symbolic scenario-based models. In: Proc. of Workshop on Program Analysis For Software Tools and Engineering (PASTE), pp. 29–35 (2008b)

Lo, D., Maoz, S.: Mining hierarchical scenario-based specifications. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 359–370 (2009)

Lo, D., Maoz, S.: Scenario-based and value-based specification mining: better together. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 387–396 (2010)

Lo, D., Maoz, S.: Towards succinctness in mining scenario-based specifications. In: Proc. of International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 231–240 (2011)

Lo, D., Maoz, S., Khoo, S.C.: Mining modal scenario-based specifications from execution traces of reactive systems. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 465–468 (2007)

Lo, D., Khoo, S.C., Li, J.: Mining and ranking generators of sequential patterns. In: Proc. of SIAM International Conference on Data Mining (SDM), pp. 553–564 (2008)

Lo, D., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Mining quantified temporal rules: formalism, algorithms, and evaluation. In: Proc. of Working Conference on Reverse Engineering (WCRE), pp. 62–71 (2009)

Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. of International Conference on Software Engineering (ICSE), pp. 501–510 (2008)

Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: Proc. of Symposium on the Foundations of Software Engineering (SIGSOFT FSE), pp. 219–230 (2006)

Maoz, S., Harel, D.: On tracing reactive systems. Softw. Syst. Model. **10**(4), 447–468 (2011)
Maoz, S., Metsä, J., Katara, M.: Model-based testing using LSCs and S2A. In: Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 301–306 (2009)
Maoz, S., Harel, D., Kleinbort, A.: A compiler for multimodal scenarios: transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. **20**(4), 18 (2011)
Mariani, L., Pezzè, M.: Behavior capture and test: automated analysis of component integration. In: Proc. of International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 292–301 (2005)
Mariani, L., Papagiannakis, S., Pezzè, M.: Compatibility and regression testing of COTS-component-based software. In: Proc. of International Conference on Software Engineering (ICSE), pp. 85–95 (2007)
Milea, N., Khoo, S.C., Lo, D., Pop, C.: Nort: Runtime anomaly-based monitoring of malicious behavior for windows. In: Proc. of International Conference on Runtime Verification (RV) (2011)
Olender, K., Osterweil, L.: Cecil: a sequencing constraint language for automatic static analysis generation. IEEE Trans. Softw. Eng. **16**, 268–280 (1990)
Pytlik, B., Renieris, M., Krishnamurthi, S., Reiss, S.P.: Automated fault localization using potential invariants. CoRR cs.SE/0310040 (2003)
Quante, J., Koschke, R.: Dynamic protocol recovery. In: Proc. of Working Conference on Reverse Engineering (WCRE), pp. 219–228 (2007)
Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proc. of International Conference on Software Engineering (ICSE), pp. 240–250 (2007a)
Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: Proc. of Conference on Programming Language Design and Implementation (PLDI), pp. 123–134 (2007b)
Safyallah, H., Sartipi, K.: Dynamic analysis of software systems using execution pattern mining. In: Proc. of International Conference on Program Comprehension (ICPC), pp. 84–88 (2006)
Santelices, RA, Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J.: Test-suite augmentation for evolving software. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 218–227 (2008)
Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: Proc. of International Symposium on Software Testing and Analysis (ISSTA), pp. 174–184 (2007)
Sibay, G., Uchitel, S., Braberman, VA: Existential live sequence charts revisited. In: Proc. of International Conference on Software Engineering (ICSE), pp. 41–50 (2008)
Sun, J., Dong, J.S.: Synthesis of distributed processes from scenario-based specifications. In: Proc. of International Symposium on Formal Methods (FM), pp. 415–431 (2005)
Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. In: Proc. of Symposium on the Foundations of Software Engineering (SIGSOFT FSE), pp. 74–82 (2001)
Wang, J., Han, J.: BIDE: efficient mining of frequent closed sequences. In: Proc. of International Conference on Data Engineering (ICDE), pp. 79–90 (2004)
Wang, J., Han, J., Pei, J.: Closet+: searching for the best strategies for mining frequent closed itemsets. In: Proc. of International Conference on Knowledge Discovery and Data Mining (KDD), pp. 236–245 (2003)
Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. In: Proc. of International Conference on Automated Software Engineering (ASE), pp. 295–306 (2009)
Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. **18**(3–4), 263–292 (2011)
Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 461–476 (2005)
Xie, T., Thummalapenta, S., Lo, D., Liu, C.: Data mining for software engineering. Computer **42**(8), 35–42 (2009)
Xu, Z., Cohen, M.B., Rothermel, G.: Factors affecting the use of genetic algorithms in test suite augmentation. In: Proc. of Annual Genetic and Evolutionary Computation Conference (GECCO), pp. 1365–1372 (2010)
Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proc. of International Conference on Software Engineering (ICSE), pp. 282–291 (2006)