Mining Branching-Time Scenarios

Dirk Fahland[1], David Lo[2], and Shahar Maoz[3]
[1]Eindhoven University of Technology, The Netherlands
[2]Singapore Management University, Singapore
[3]Tel Aviv University, Israel
d.fahland@tue.nl, davidlo@smu.edu.sg, maoz@cs.tau.ac.il

***Abstract*—Specification mining extracts candidate specification from existing systems, to be used for downstream tasks such as testing and verification. Specifically, we are interested in the extraction of behavior models from execution traces.**

In this paper we introduce mining of branching-time scenarios in the form of existential, conditional Live Sequence Charts, using a statistical data-mining algorithm. We show the power of branching scenarios to reveal alternative scenario-based behaviors, which could not be mined by previous approaches.

The work contrasts and complements previous works on mining linear-time scenarios. An implementation and evaluation over execution trace sets recorded from several real-world applications shows the unique contribution of mining branching-time scenarios to the state-of-the-art in specification mining.

I. Introduction

Specification mining methods, which extract candidate specifications from source code or execution traces, have attracted much research efforts in recent years. The mined specifications serve as input for downstream tasks such as testing, formal verification, and program comprehension. Specifically, we are interested in mining scenario-based specifications, where the mined specifications consist of a set of typically short sequences of events, intuitively depicted using variants of classic sequence diagrams.

In recent work [23], [25], [28], two of the co-authors of the present paper have presented mining of scenario-based specifications in the form of universal Live Sequence Charts (LSC) [5], [18]. Universal LSCs specify rules of the form "whenever a sequence of events happens, eventually another sequence of events will happen". That is, they specify a universal linear temporal logic property.

However, a specification of the behavior of a system may often be characterized not only by linear invariants but also by possible choices. In the context of scenario-based specifications, these are best represented using existential conditional LSCs, as defined in [37]. Such a branching LSC specifies a rule of the form "when a sequence of events happens, another sequence of events is a possible continuation". That is, they specify a conditional branching temporal logic property.

In this work we present mining branching time scenario-based specifications in the form of the branching, existential conditional LSCs described in [37]. The input for our technique are a set of execution traces (a *log*) that have been recorded from a running application, and two parameters: a *support* threshold sets how often a particular LSC has to occur in the log (i.e., when it is "relevant" enough), and a *confidence* threshold sets the fraction of times the property specified in the discovered LSC has to hold (allowing the LSC to be violated on some occurrences). The output consists of a set of existential conditional LSCs that is correct and complete with regard to the support and confidence: the given log satisfies each returned existential conditional LSC, and any other existential conditional LSC occurs less often than the given support threshold, or the log violates the LSC more often than allowed by the confidence threshold.

To better understand the difference between mining linear LSCs and mining branching LSCs, and thus motivate our work, consider the following two example LSCs, `Delete` and `Download`, shown in Fig. 1 and Fig. 2 respectively. Both LSCs were mined in our experiments from an execution trace set of *crossFTP*, an open-source FTP server (see Sect. V for details). As these are branching LSCs, each specifies a possible continuation to the onConnect call, a continuation which was observed with high support and confidence in the trace set, executing the `Delete` or the `Download` FTP commands (we mined several additional LSCs with the same pre-chart onConnect, all together covering several additional commands of the FTP protocol). However, when mining for linear LSCs (as in [28]), from the same trace set, none of these LSCs is found, because in the trace set, onLogin is not *always* followed by the setDelete command, by the setDownload command, or by any other single FTP command. Instead, only linear invariants are mined with high support and confidence, such as the linear LSC shown in Fig. 3, which specifies that "whenever onConnect is called in the trace, eventually there will be calls to onLogin, setUser, and setLogout" (in this order). This example shows the difference between (mined) linear and branching scenarios in terms of their expressiveness and their ability to explain and reveal the behaviors embedded in an execution trace set.

We have implemented our ideas and evaluated them on execution trace sets recorded from two applications. All trace sets and mining results are available at [14].

We summarize the contribution of our work as follows:

- Definition and algorithm for mining branching-time scenarios;
- Evaluation over a number of trace sets, including a discussion and comparison of branching vs. linear scenarios in the context of specification mining;
- Implementation for mining of linear and of branching scenarios available for download with all trace sets data,

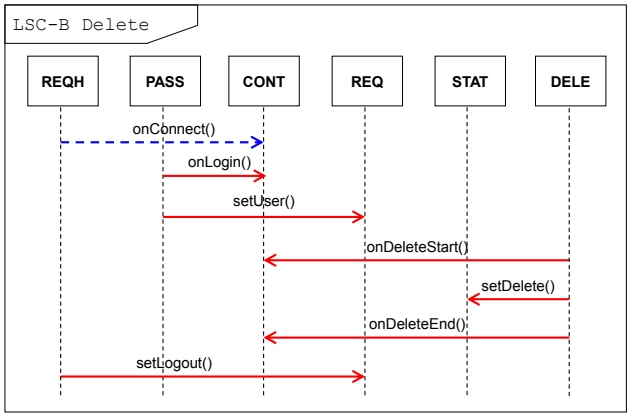

Fig. 1. The branching LSC `Delete`, specifying one possible continuation after login, dealing with the FTP delete command.

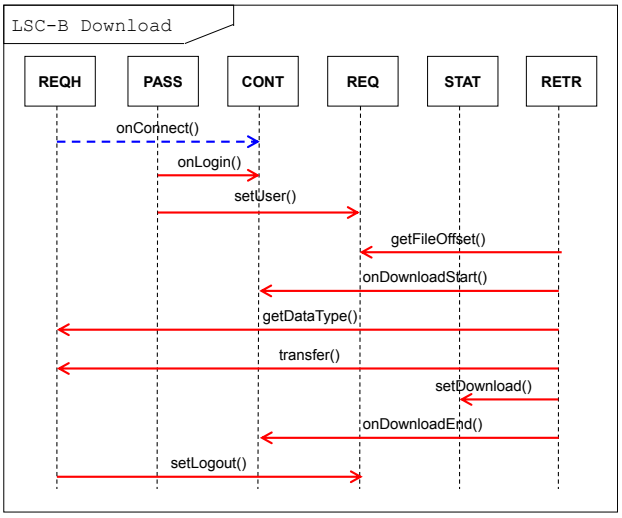

Fig. 2. The branching LSC `Download`, specifying one possible continuation after login, dealing with the FTP download command.

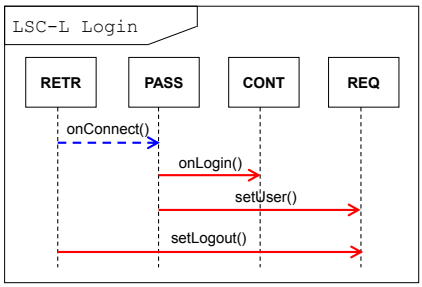

Fig. 3. The linear LSC `Login`, specifying that "whenever onConnect is called, eventually there will be calls to onLogin, setUser, setLogin, and setLogout" (in this order).

to allow readers to reproduce our experiments.

The formal methods community has had a long debate about the relative merits of linear vs. branching-time logic as they are used for system specifications; which one is easier for engineers to formulate and understand, which one may have better performing synthesis and verification algorithms, which one allows compositional reasoning etc. (see, e.g., [8], [39]). One may view our present work as another contribution to this long debate, specifically in the context of specification mining.

The paper is organized as follows. Sect. II provides background and definitions. The mining algorithm is presented in Sect. III. An extension to leverage additional user inputs to improve the mining process is presented in Sect. IV. Sect. V presents an evaluation, Sect. VI discusses related works, and Sect. VII concludes.

II. Background and Definitions

We recall the semantics of branching LSCs and their formal relation to execution trees. We further define a number of measures required in the context of scenario mining.

A. Branching LSCs and traces

1) Branching LSCs: We use the branching LSCs defined in [37], with totally ordered events. An LSC consists of lifelines representing participating objects and of events between them. Events are partitioned into pre-chart and main-chart events, with the following conditional branching semantics: "whenever a sequence of events in the pre-chart happens, the sequence of events in the main-chart must be a possible continuation". Syntactically, pre-chart events use blue, dashed lines, and main-chart events use red, solid lines. Figures 1-3 show examples.

Fig. 4 shows an example of an *execution tree*, which illustrates the semantics of branching LSCs. The tree represents three different merged execution traces all starting with onConnect and ending with setLogout. The tree *satisfies* the branching LSC `Delete` (Fig. 1): whenever its pre-chart onConnect occurs, *there exists* a continuation with its main-chart (including onDeleteStart etc). Like a linear LSC, a branching LSC abstracts from events not mentioned in its pre- or main-chart as illustrated by the occurrence of `Delete` in the right branch of Fig. 4. For the same reason, the branching LSC `Download` (Fig. 2) is satisfied. There are even *two different* traces that continue with the main-chart of the branching LSC `Delete`, as highlighted in Fig. 4. In contrast, the tree *violates* branching LSC `Rename2` (Fig. 5). There is no trace that continues *every occurrence* of onDeleteEnd with event onRenameStart etc. (the right branch does continue with `Rename2` whereas the left branch does not).

2) Branching LSCs vs. Linear LSCs: Intuitively, to be counted for branching, an LSC needs to occur on one branch, and to be counted for linear it has to occur on *all* branches. The tree of Fig. 4 illustrates this difference. If we would interpret `Download` of Fig. 2 as a linear LSC, then the tree of Fig. 4 would *violate* this LSC, because one trace continues without completing the occurrence of the main-chart of `Delete`. More precisely, the occurrence of setLogout at the right-most branch of the tree will violate the main-chart of LSC `Download` which requires (in a linear interpretation) getFileOffset to occur after setUser.

Thus, branching LSCs allow one to describe *alternative* continuations of a pre-chart. In contrast, linear LSCs, such as `Login` of Fig. 3, specify *invariants* of each trace. The tree of Fig. 3 satisfies `Login`, which specifies that each call to onConnect leads to a subsequent setLogout (along the sequence of events specified in the chart).

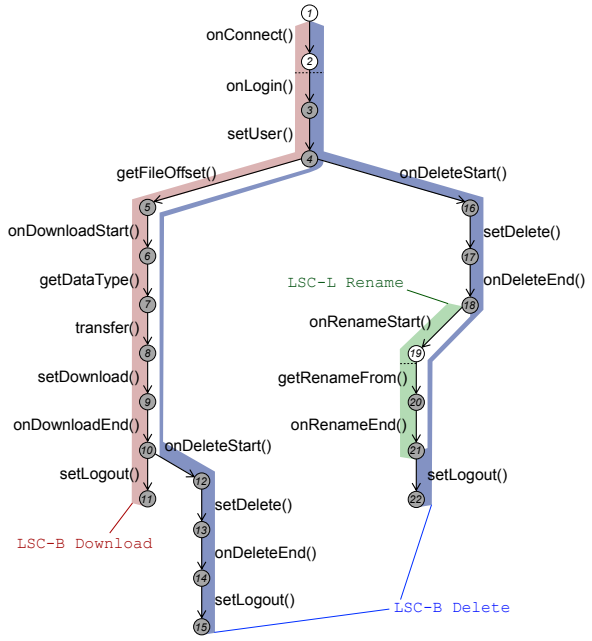

Fig. 4. Execution tree of a trace set from crossFTP (represents 3 merged traces), highlighting occurrences of the LSCs of Figures 1-3.

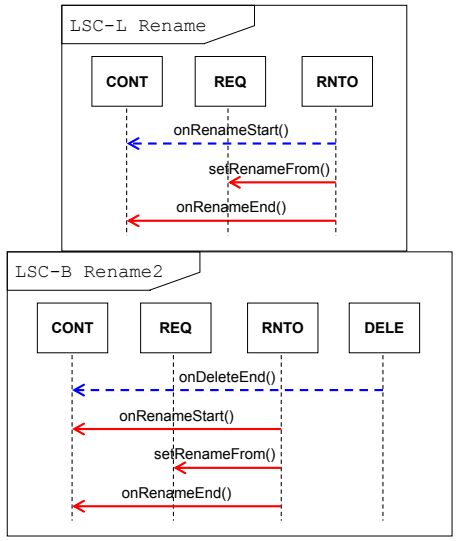

Fig. 5. Linear LSC `Rename` and Branching LSC `Rename2` specifying renaming in different contexts.

3) Basic notions: LSCs, traces, and positive witnesses: More formally, an LSC event is defined by a triple (s, r, m) where s denotes the sender object (method caller), r the receiver object (callee), and m the method that s invoked to communicate with r. If all events of an LSC are totally ordered, an LSC $L = (pre, main)$ can be formalized as two sequences of events describing the pre-chart pre and the main-chart $main$. For example, LSC `Login` reads as $(\langle(\mathsf{RETR}, \mathsf{PASS}, \mathsf{onConnect})\rangle, \langle(\mathsf{PASS}, \mathsf{CONT}, \mathsf{onLogin}) \ldots (\mathsf{RETR}, \mathsf{REQ}, \mathsf{setLogout})\rangle)$.

In the following we abstract from the inner structure of events and treat each triple $a = \langle s, r, m\rangle$ as a single letter a. In this abstract setting, we consider traces over a finite alphabet $\Sigma = \{a, b, c, \ldots\}$ of events. The symbol $+\!\!+$ denotes the concatenation of two finite words. Then, an LSC $L = (pre, main)$ induces a word $w = pre +\!\!+ main$. Let Σ_L denote the alphabet of "visible" events appearing in w.

Semantics of linear LSCs intuitively require that in every execution, an occurrence of the pre-chart pre is followed by an occurrence of the main-chart $main$. We formalize these occurrences as positive witnesses as follows. A (possibly empty) word p is *prefix* of u, written $p \sqsubseteq u$ iff there is a (possibly empty) word u' s.t. $u = p +\!\!+ u'$. A word s is a *subword* of a word u if there are (possibly empty) words u' and u'' s.t. $u = u' +\!\!+ s +\!\!+ u''$. For a word s and a set $\Sigma' \subseteq \Sigma$ of "visible" events, the projection of s onto Σ' is written $s|_{\Sigma'}$. A *positive witness* of a word w wrt. the visible events Σ' in a word u is a *length-minimal* subword s of u such that $w = s|_{\Sigma'}$.

4) Execution trees and branching-time semantics: The branching time semantics of LSCs [37] requires that each positive witness of pre is followed by a positive witness of $main$ *in some run*, which is naturally expressed on the semantic domain of execution trees. An *execution tree* (over alphabet Σ) is a rooted, labeled $\mathcal{T} = (V, E, v_0, \ell)$ with nodes V, edges E, root v_0, and a labeling ℓ of nodes and edges such that

1) $\ell(v_0) = \varepsilon$ (the empty word), and
2) for each edge $(v, v') \in E$ holds: $\ell(v, v') = a \in \Sigma$ such that $\ell(v) +\!\!+ a = \ell(v')$.

Each node node $v \in V$ describes a run $\ell(v)$ from the root v_0 to v. A node $v \in V$ is a *leaf* of $\mathcal{T}$ if for no node $v' \in V$, $(v, v') \in E$. A node v' *is reachable from* a node v, written $v \leq v'$ iff $(v, v') \in E^*$ (the reflexive-transitive closure of E). $\mathcal{T}$ is *deterministic* iff for any two edges $(v, v'), (v, v'') \in E$ with $\ell(v, v') = \ell(v, v'')$ holds $v' = v''$. Fig. 4 shows an example of a deterministic execution tree with three leaf nodes $11, 15, 22$.

The semantics of branching LSCs as defined in [37] reads on execution trees as follows. Let $L = (pre, main)$ be an LSC. A word u *ends with* the pre-chart pre iff u has a suffix s, $u = pre +\!\!+ s$, such that s is a positive witness of pre wrt. Σ_L in u. Let $\mathcal{T} = (V, E, v_0, \ell)$ be an execution tree. A node v is a *positive witness* of pre iff $\ell(v)$ ends with pre. $\mathcal{T}$ *satisfies* L iff for each positive witness v of pre exists a node $v' \in V$, $v \leq v'$, such that $\ell(v')$ is a positive witness of $pre +\!\!+ main$ wrt. Σ_L; we call v' a *positive witness* of L in $\mathcal{T}$. Thus, note that the semantics of branching LSCs (and of linear LSCs) do not constrain events not appearing in the LSC to appear or not to appear in the tree (and trace set), including in between events that do appear in the LSC.

For example, node 2 of Fig. 4 is a positive witness of the pre-charts of the LSCs of Figures 1-3. Node 11 is a positive witness of `Download` (Fig. 2) and `Login` (Fig. 3), 15 and 22 are positive witness of `Delete` (Fig. 1) and `Login`. Node 21 is a positive witness of `Rename2` (Fig. 5).

In the following we consider *weighted* execution trees $\mathcal{T} = (V, E, v_0, \ell, \kappa)$ where the *weight function* κ assigns each node $v \in V$ a positive integer $\kappa(v)$.

B. Relating execution trees and traces

A weighted deterministic tree represents a (multi-)set of traces: if two traces share the same prefix w, then this prefix is represented in the tree only once by a node $v, \ell(v) = w$. We use the weight of v to represent how many times $\ell(v)$ occurs as a prefix of a trace in this set. The weights will be relevant to discover how many times a particular scenario occurs in a set of traces.

A *trace* T is a finite word over Σ; a *log* $\mathcal{L}$ is a finite *multi-set* of traces, the natural number $\mathcal{L}(T)$ describes how often T occurs in L. Each log $\mathcal{L}$ has a canonical representation as a tree: the unique deterministic tree $\mathcal{T}$ where each leaf represents exactly one distinct trace. The number of occurrences of traces in $\mathcal{L}$ is captured by node weights of $\mathcal{T}$.

The formal definition requires an auxiliary notion: for any word p, let $cont(p, \mathcal{L}) = \{T \in \mathcal{T} \mid p \sqsubseteq T\}$ denote the set of traces of $\mathcal{L}$ that continue a prefix p. Now a weighted tree $\mathcal{T} = (V, E, v_0, \ell, \kappa)$ *represents* the log $\mathcal{L}$ iff (1) for each $T \in \mathcal{L}$, exists node $v \in V$ with $\ell(v) = T$, (2) for each leaf v of V, $\ell(v) \in \mathcal{L}$, (3) $\ell(v_1) = \ell(v_2)$ implies $v_1 = v_2$, for all nodes $v_1, v_2 \in V$, and (4) for each $v \in V$, holds $\kappa(v) = \sum_{T \in cont(\ell(v), \mathcal{L})} \mathcal{L}(T)$. Condition (3) implies that $\mathcal{T}$ is deterministic. Condition (4) states that the weight of each node is the sum of all occurrences of all traces that "run through it"; note that this implies $\kappa(v) \geq \kappa(v')$ for all $(v, v') \in E$. In the following, we assume that all traces in $\mathcal{L}$ start with the same event, that is, v_0 of $\mathcal{T}$ has exactly one outgoing edge.

For example, the tree of Fig. 4 represents three traces (induced by its leaf nodes). Nodes 1-4 have weight 3, nodes 5-10 have weight 2, and all other nodes have weight 1. The weight function allows us to see that the pre-chart onConnect of Fig. 1 occurs 3 times in the tree. We use the weight function to define measures on scenarios, as follows.

C. Measures for LSCs on execution trees

Our goal is to discover from a given log of execution traces a set of LSCs that describe the behavior in the log according to the branching-time semantics given in Sect. II-A. We introduce basic measures that express how good a set of (discovered) LSCs may describe the behavior recorded in a log.

1) Negative witnesses, support, and confidence: *Support* measures the number of times an LSC $L = (pre, main)$ occurs in the execution tree $\mathcal{T}$ that represents the log. *Confidence* measures the likelihood that a positive witness of pre is followed by a positive witness of $main$ in some branch of $\mathcal{T}$.

For this, we introduce *(weak and strong) negative witnesses*. A *negative* witness of L is a witness of pre that is not followed by a witness of $pre +\!\!+ main$. The witness is *weak negative* if some trace could be extended to a witness of $pre +\!\!+ main$ (by adding missing events), and *strong negative* otherwise (e.g., events occur out of order).

Formally, let $\mathcal{T} = (V, E, v_0, \ell)$ be an execution tree and $L = (pre, main)$ be an LSC.

1) A *weak negative witness* of L in $\mathcal{T}$ is a positive witness $v \in V$ of pre such that for no node $v' \in V$, $v \leq v'$ holds $\ell(v')$ is a positive witness of $pre +\!\!+ main$.
2) A *strong negative witness* of L in $\mathcal{T}$ is a positive witness $v \in V$ of pre such that for every leaf $v' \in V$ and every finite word w holds: $\ell(v') +\!\!+ w$ is *not* a positive witness of $pre +\!\!+ main$.

Support and confidence measures are based on the *weights* of all witnesses. Let $pos(L, \mathcal{T})$, $wneg(L, \mathcal{T})$, and $sneg(L, \mathcal{T})$ denote the positive, negative, and strong negative witnesses of L in $\mathcal{T}$, respectively; $pos(pre, \mathcal{T})$ denotes the positive witnesses of pre-chart pre. The *weight of a witness* v in $\mathcal{T}$ is $\kappa(v)$. For a set W of witnesses, let $\kappa(W) = \sum_{v \in W} \kappa(v)$ be the sum of all weights of all witnesses.

1) The support of L in $\mathcal{T}$ is $sup(L, \mathcal{T}) = \kappa(pos(L, \mathcal{T}))$,
2) the confidence of L in $\mathcal{T}$ is $conf(L, \mathcal{T}) = \frac{\kappa(pos(L,\mathcal{T})) + \kappa(wneg(L,\mathcal{T})) - \kappa(sneg(L,\mathcal{T}))}{\kappa(pos(pre,\mathcal{T}))}$.

To account for incomplete traces, confidence only penalizes strong negative witnesses of L, as weak negative witnesses of L could be extended to positive ones.

For example, in the tree of Fig. 4, LSC `Delete` (Fig. 1) has support 2 and confidence 1.0, LSC `Download` (Fig. 2) has support 1 and confidence 1.0, linear LSC `Login` (Fig. 3) has support 3 and confidence 1.0. In contrast, LSC `Rename2` (Fig. 5) has support 1 and confidence 0.5. Its pre-chart has a positive witness at nodes 14 and 18, but its main-chart has only a witness at node 21. Thus, 14 is a weak negative witness for `Rename2` as the trace could be extended (after node 15) to a positive witness of `Rename2`. If the edge from 14 to 15 was labeled with setRenameFrom, 14 would be a strong negative witness as event onRenameStart would be missing.

As in the case of linear LSCs, the support measure for the branching case is monotonic. For an execution tree $\mathcal{T}$, an LSC L, and a word w: $|pos(pre +\!\!+ main, \mathcal{T})| \geq |pos(pre +\!\!+ main +\!\!+ w, \mathcal{T})|$ [28]. This monotonicity extends to $\kappa(pos(pre +\!\!+ main, \mathcal{T})) \geq \kappa(pos(pre +\!\!+ main +\!\!+ w, \mathcal{T}))$ because $\kappa(v) \geq \kappa(v')$, for all edges (v, v') of $\mathcal{T}$. We take advantage of this monotonicity to prune search when support falls below a threshold.

Note that the weight function is needed for mining branching scenarios because the tree merges prefixes of traces into the same node. When mining linear scenarios, one can count the scenarios in the trace set and no weight function is needed.

2) Two notions of coverage: Finally, the branching semantics of LSCs gives rise to another measure. A *set* of LSCs is comprehensive if it can explain every branch in the execution tree. We formally capture this property in two *coverage* measures: the fraction of tree nodes covered by an LSC (pre- and main-chart), and the fraction of nodes only covered by main-chart events.

For example, the tree of Fig. 4 is completely covered by LSCs `Delete` (Fig. 1), `Upload` (Fig. 2), and `Rename` (Fig. 5 top). When considering *main-chart* coverage, node 19 *is not covered*, neither by `Delete` nor by `Rename`, as event onRenameStart is not a main-chart event of these LSCs. We can cover node 19 with LSC `Rename2` (Fig. 5 bottom). Main-chart coverage measures the fraction of events in the

tree that are *not explained as the consequence of some pre-chart*. For example, `Rename2` explains onRenameStart as the consequence of onDeleteEnd whereas `Rename` assumes onRenameStart to be given. More formally:

1) A node $v \neq v_0$ of an execution tree $\mathcal{T}$ is *covered* by an LSC $L = (pre, main)$, written $v \; cov \; L$ iff there exist nodes $v_1, v_2 \in V$, $v_1 \leq v \leq v_2$ such that v_2 is a positive witness of L in $\mathcal{T}$, v_1 is the corresponding positive witness of pre in $\mathcal{T}$, and the incoming edge $(v', v) \in E$ is labeled with an event $\ell(v', v) \in \Sigma_L$ of L; v is *main-chart covered* by L, $v \; cov^m \; L$, iff $(v', v) \in E$ is labeled with a *main chart* event of L.
2) For a set $\mathcal{S}$ of LSCs, we write $v \; cov \; \mathcal{S}$ ($v \; cov^m \; \mathcal{S}$) iff $v \; cov \; L$ ($v \; cov^m \; L$) for some $L \in \mathcal{S}$.
3) The *coverage* and *main-chart coverage* of $\mathcal{T}$ by $\mathcal{S}$ are defined as $cov(\mathcal{S}, \mathcal{T}) = \frac{|\{v \in V | v \; cov \; \mathcal{S}|\}}{|V \backslash \{v_0\}|}$ and $cov^m(\mathcal{S}, \mathcal{T}) = \frac{|\{v \in V | v \; cov^m \; \mathcal{S}|\}}{|V'|}$, respectively, where $V' = V \backslash (\{v_0\} \cup \{v \mid (v_0, v) \in E\}|)$.

We use these notions of coverage as a yardstick for the quality of the mined branching LSCs[1].

III. Mining Algorithm

We now present the main contribution of this paper: a mining algorithm for branching LSCs. Sect. III-A introduces measures to characterize the output of a mining algorithm and discusses variants of mining. Sect. III-B explains the basic algorithmic idea and Sect. III-C presents the actual algorithm. In Sect. III-D we discuss correctness and complexity.

A. Characterizing mining results

A mining algorithm $\mathcal{M}$ extracts from a given log $\mathcal{L}$ a finite set $\mathcal{S}$ of LSCs. One can impose properties on $\mathcal{S}$ to characterize desirable results of a mining algorithm. For the branching semantics, three properties are of interest: correctness, completeness, and coverage. We state these properties in terms of the execution tree $\mathcal{T}$ that represents $\mathcal{L}$. Intuitively, a set $\mathcal{S}$ of LSCs is correct wrt. $\mathcal{T}$ if each LSC exceeds a given confidence threshold c (has no or only a specified ratio of negative witnesses in $\mathcal{T}$) and its support exceeds a given threshold s (occurs often enough). $\mathcal{S}$ is complete wrt. $\mathcal{T}$ if for any other LSC L that is satisfied by $\mathcal{T}$ (confidence exceeds threshold c) the support of L is below threshold s. Finally, a set of LSCs covers $\mathcal{T}$ if a given percentage of nodes of the execution tree is explained by at least one LSC. Formally:

1) A set $\mathcal{S}$ of LSCs is *correct* with respect to $\mathcal{T}$ and thresholds s and c iff each $L \in \mathcal{S}$ has support $sup(L, \mathcal{T}) \geq s$ and confidence $conf(L, \mathcal{T}) \geq c$ in the execution tree $\mathcal{T}$ that represents $\mathcal{L}$.
2) A set $\mathcal{S}$ of LSCs is *complete* with respect to $\mathcal{T}$ and thresholds s and c iff for each LSC L holds: $sup(L, \mathcal{T}) < s$ or $conf(L, \mathcal{T}) < c$ or $L \in \mathcal{S}$.
3) A set $\mathcal{S}$ of LSCs *covers* $\mathcal{T}$ with respect to threshold k iff $cov(\mathcal{S}, \mathcal{T}) \geq k$.

The mining algorithm's objective is to return a set of LSCs that is correct and complete wrt. the chosen thresholds.

B. Algorithm idea and monotonicity

The mining algorithm proposed in this paper builds on the mining algorithm for universal LSCs that was proposed first in [28]. The algorithm has two major steps. In the first step, the set of *candidate words* that exceed the chosen support threshold s are discovered. In the second step, each candidate word w is split into all possible pre- and main-charts $pre +\!\!+ main = w$; each splitting gives rise to a unique candidate scenario $L = (pre, main)$ for which the confidence is checked, that is, whether each occurrence of pre is followed by an occurrence of $main$ (up to the chosen confidence threshold). All LSCs that reach the chosen confidence threshold are returned, the others are discarded.

The key to efficiently find a *complete* set of LSCs above a certain threshold is a *monotonicity* property on the number of positive witnesses of words. For two finite non-empty words w, u, the word $w +\!\!+ u$ will occur at most as often in a tree $\mathcal{T}$ as the word w, because not each occurrence of w is followed by an occurrence of u. Thus, the discovery starts with single events that are extended to words by exhausting all possible event combinations. Once an explored word w occurs less often than the support threshold s, exploration stops for w, as no extension $w +\!\!+ u$ will occur more often than s times. Hence, every LSC $L = (pre, main)$ that can be built from a discovered word $w = pre +\!\!+ main$ will satisfy $sup(L, \mathcal{T}) \geq s$, and any other LSC has a support $< s$ in $\mathcal{T}$.

C. Mining algorithm

In the following we consider only LSCs $L = (pre, main)$ where in $pre +\!\!+ main$ each event occurs at most once.

The algorithm's main procedure **MineLSC** is shown in Fig. 6. It contains the two main steps described above (discover candidate words, and discover LSCs from candidate words). Its input consist of the tree and of thresholds for support and confidence. Its output is a correct and complete set of *statistically significant* LSCs, i.e., exactly all branching LSCs that meet the thresholds.

Discovering candidate words begins with discovering the *frequent* events that each occur in $\mathcal{T}$ more often than the support threshold s (an event occurring less than s times cannot be part of an LSC that exceeds s), see procedure **MineSupportedWords**. Then procedure **MineRecurse** recursively builds candidate words from frequent events as follows. Extend a word w with each frequent event e to a word $w +\!\!+ e$. If $w +\!\!+ e$ occurs more often than threshold s, then $w +\!\!+ e$ is a candidate word, and then continue recursively with $w +\!\!+ e$. Eventually $w +\!\!+ e$ falls below the threshold (or w represents a complete trace *ending* at a leaf of $\mathcal{T}$), and the recursion terminates.

The main difference between our branching LSC mining algorithm and the original linear LSC mining algorithm of [28]

[1]Note that our definition of coverage relates a set of (mined) branching LSCs wrt. a set of traces, as it is represented in the execution tree. We do not consider here how well does the set of traces cover the behaviors of the application from which they were extracted.

Procedure MineLSC
Inputs: $\mathcal{T}$: Input Tree; min_sup: Min. Sup. Thresh.; min_conf: Min. Conf. Thresh.
Output: A set of statistically significant LSCs

Let $WSet$ = MineSupportedWords($\mathcal{T}, min\_sup$)
Let $LSCResult = \{\}$
For every word $w \in WSet$
  For every prefix pre of w
    Let $main$ s.t. $pre +\!\!+ main = w$
    Let $NewLSC$ = Create new LSC (pre, $main$)
    If ($conf(NewLSC, \mathcal{T}) \geq min\_conf$)
      Add $NewLSC$ to $LSCResult$
Return $LSCResult$

Procedure MineSupportedWords
Inputs: $\mathcal{T}$: Input Tree; min_sup: Min. Sup. Thresh.
Output: A set of supported words

Let EV = Single-events occurring $\geq min\_sup$ in $\mathcal{T}$
  Let $WSet = \{\}$
  For every $ev \in EV$
    Call MineRecurse($\mathcal{T}, min\_sup, EV, ev, WSet$)
Return $WSet$

Procedure MineRecurse
Inputs: $\mathcal{T}$: Input Tree; min_sup: Min. Sup. Thresh.; EV : Frequent Events; $curW$: Current word considered; $WSet$: Current set of supported words
Output: Updated set of supported words ($WSet$)

Add $curW$ to $WSet$
For every $ev \in EV$ with ev not in $curW$
  Let $nxtW = curW +\!\!+ ev$
  If ($|pos(nxtW, \mathcal{T})| \geq min\_sup$)
    Call MineRecurse($\mathcal{T}, min\_sup, EV, nxtW, WSet$)

Fig. 6. Mining algorithm

lies in the computation of positive and negative witnesses of an LSC $L = (pre, main)$, which has to be performed on the canonical tree $\mathcal{T}$ to determine L's support and confidence. Function $conf(L, \mathcal{T})$ called in **MineLSC** is straightforward. Find all nodes $v_1, v_2, \ldots$ of $\mathcal{T}$ where $\ell(v_i), i > 0$ ends with pre. Then for each v_i, perform a depth-first search on $\mathcal{T}$ starting at v_i to find a positive witness of $main$. Collect all positive and (weak) negative witnesses to compute $conf(NewLSC, \mathcal{T})$.

D. Correctness and complexity

The mining algorithm of Fig. 6 is correct: the set $LSCResult$ returned by **MineLSC** is correct and complete w.r.t. the given tree $\mathcal{T}$ and given support and confidence thresholds min_sup and min_conf. The proof holds by the same arguments as for the linear case [28] as follows.

First, **MineSupportedWords** returns in $WSet$ each word w with support $\geq min\_sup$. This proposition holds as **MineSupportedWords** recurses over all variations (permutations of subsets) of events that occur at least min_sup times (weighted occurrences). Recursion stops only for every word w with support $< min\_sup$, and by the monotonicity of support no word $w +\!\!+ u$ has support $\geq min\_sup$.

Second, all LSCs that can be built from $WSet$ (as assumed, all LSCs where each event occurs at most once), will be considered in **MineLSC**. Thus, each LSC with support $\geq min\_sup$ is considered.

Third, $LSCResult$ contains only those LSCs with confidence $\geq min\_conf$. Thus, $LSCResult$ is the set of all LSCs having support $\geq min\_sup$ and confidence $\geq min\_conf$.

Regarding complexity, let n be the number of single events occurring at least min_sup times in $\mathcal{T}$ (weighted occurrences). Then up to $\sum_{k=1}^{n} \frac{n!}{(n-k)!}$ supported words can be found. Each supported word w of length $k > 1$ gives rise to $k-1$ LSCs $L(pre, main)$ with $pre +\!\!+ main = w$. Thus, in worst case $\sum_{k=2}^{n} k \cdot \frac{n!}{(n-k)!}$ candidate LSCs have to be checked for confidence. Checking confidence of one candidate LSC requires basically one depth-first search on $\mathcal{T}$. Thus, if $\mathcal{T}$ has m nodes, the check succeeds in $\mathcal{O}(2 \cdot m)$ steps. Altogether, this gives a worst case complexity of $\mathcal{O}(\sum_{k=1}^{n} \frac{n!}{(n-k)!} + \sum_{k=2}^{n} k \cdot \frac{n!}{(n-k)!} \cdot m) = \mathcal{O}(\sum_{k=2}^{n} k \cdot \frac{n!}{(n-k)!} \cdot m)$. This is a very coarse approximation. By far not each possible variation of events is a supported word and recursion stops as soon as the support drops. Also, heuristics allow to prune the search space (number of recursions) significantly. The complexity of the branching case deviates from the linear case in the final step when checking confidence of LSCs. The linear case has to consider the entire log (number of cases times average length of all cases), which is usually larger than the tree.

IV. Mining Triggers and Effects

In addition to finding a *complete* set of LSCs, discovery can also be driven by the questions "Which triggers are needed to observe a given behavior $main$?" and "What behaviors are triggered by a given pre-chart pre?". These two questions lead to *trigger* and *effect* mining, respectively [23]. Here, the goal is to find for a given pre-chart pre (main-chart $main$) the correct and complete subset $\mathcal{S}$ of LSCs that have this pre-chart (main-chart).

Mining scenarios given triggers and effects can be beneficial for two reasons. First, the number of mined scenarios is typically small and users are presented only with scenarios of interest. Second, the efficiency of the mining process could be improved significantly. This section shows how to extend the basic mining algorithm of Fig. 6 to solve this problem variant.

As an example of trigger and effect mining, users can provide the pre-chart (or main-chart) of the scenario shown in Fig. 1 as a trigger (or an effect). The mining algorithm would then only return the set of branching scenarios with pre-charts matching the given trigger (or main-chart matching the given effect). This would include the scenario shown in Fig. 1 and possibly many others, but exclude other scenarios that are not of interest, e.g., the one shown in Fig. 5 (bottom).

We first describe a straightforward solution and give a more efficient one afterwards. To mine from a given tree $\mathcal{T}$ scenarios that are triggered by a given pre-chart pre (or have the given effect $main$) with support s and confidence c, we can compute $\mathcal{S} = \mathbf{MineLSC}(\mathcal{T}, s, c)$ and then filter $\mathcal{S}$ by the given pre-chart or main-chart:
$trigger(\mathcal{S}, pre) = \{(pre, main') \mid (pre, main') \in \mathcal{S}\}$ and

$effect(\mathcal{S}, main) = \{(pre', main) \mid (pre', main) \in \mathcal{S}\}$. The result is correct and complete in the following sense.

1) $\mathcal{S}$ is *correct* with respect to tree $\mathcal{T}$, support s, confidence c and pre-chart pre (main-chart $main$) iff $\mathcal{S}$ correct with respect to $\mathcal{T}$, s, and c (as before) and additionally $(pre', main') \in \mathcal{S}$ implies $pre' = pre$ ($main' = main$).
2) $\mathcal{S}$ is *complete* wrt. $\mathcal{T}$, s, c and pre ($main$) iff for each LSC $L = (pre', main') \notin \mathcal{S}$, one of the following holds: $sup(L, \mathcal{T}) \leq s$ or $conf(L, \mathcal{T}) \leq c$ or $pre \neq pre'$ ($main \neq main'$).

Note that in trigger/effect mining, a given pre-chart pre defines its own support value $sup(pre, \mathcal{T})$ as upper bound for the support threshold s. If $s > sup(pre, \mathcal{T})$ then the resulting set $\mathcal{S} = \emptyset$. A similar observation is also valid when mining triggers for a main-chart $main$.

To improve efficiency, we adapt procedure **MineSupportedWords** of Fig. 6 for trigger and effect mining. For a given pre-chart pre we call **MineRecurse** with pre as first supported word (if pre is supported in $\mathcal{T}$). For a given main-chart $main$ call **MineRecurse** with $main$ as first supported word (if $main$ is supported in $\mathcal{T}$) and compute the next word as $nextW = ev +\!\!+ curW$ in the recursion step (to build up the pre-chart for $main$). In Sect. V, we show that this improves the efficiency of the mining process significantly.

V. Implementation and Evaluation

We report on experimental evaluation of mining branching scenarios, in particular compared to linear-time scenarios.

A. Experiment design

The main aim of our experiments was threefold: to show the feasibility of our algorithm to discover branching-time scenarios on actual data sets, to gain insights on how branching-time scenarios relate to linear ones *on actual data*, and to evaluate the benefit of trigger and effect mining for branching scenarios. We thus formulated four research questions.

- RQ I: In what ways is mining of branching scenarios different than mining of linear ones?
- RQ II: Which properties of traces impact mining branching or linear scenarios? Is it possible to estimate useful support/confidence thresholds from the traces prior to mining?
- RQ III: Is there an advantage to mine both linear and branching scenarios from the same set of traces?
- RQ IV: Could we improve the efficiency of the mining process by allowing users to specify triggers (or effects) of interest? What insights can be gained with this technique?

B. Experiment setup

1) Implementation: We implemented the mining algorithm of Sect. III (as well as the linear mining algorithm of [23]) in the Java-based command-line tool *Sam* that is available at https://github.com/scenario-based-tools/sam/wiki. *Sam* takes as input a log file in *XES* format [38] and support and confidence thresholds and returns the discovered scenarios as modal UML sequence diagrams [18]. The results are shown on an HTML page, scenarios are grouped by equivalence classes sharing the same pre-charts. *Sam* also visualizes occurrences of the discovered scenarios on the execution tree of the given log similar to Fig. 4.

TABLE I
Basic data on the logs.

log	traces	$\lvert\Sigma\rvert$	nodes	max. degree	width	depth
CrossFTP	54	50	921	7	51	64
Columba	104	79	2028	6	70	210

The implementation improves over the algorithm of Sect. III in two ways. (1) We use a heuristics to prune the search for supported words as follows. Instead of extending a found supported word w with all supported events, we prefer extending w with the successor events of all occurrences of w that have been found in the tree. When w cannot be extended further, we return to naive recursive search and also check each subword of w (by removing single events) that allows for further extension not possible for w. We found this heuristics to greatly reduce the search space and hence the runtime needed to identify candidate words. (2) We filter from the resulting set of LSCs all LSCs that are *subsumed* by some other LSC. Intuitively, LSC L subsumes LSC K if K holds whenever L holds. Formally, LSC $L = (pre_L, main_L)$ subsumes LSC $K = (pre_K, main_K)$ iff either pre_L is a subword of pre_K and $main_K$ is a subword of $main_L$ (L requires less in the pre-chart and provides a larger main-chart), or $pre_K +\!\!+ main_K$ is a subword of $main_L$ and pre_L is not a subword of $main_K$ (K's pre-chart and main-chart occur while L's main-chart occurs, and not earlier). In both cases, every positive witness of pre_K succeeds a positive witness of pre_L (hence L occurs on some branch), and every positive witness of L succeeds a positive witness of K (hence an occurrence of L implies an occurrence of K). As a consequence, all smaller LSCs contained in some larger LSC are omitted from the results.

2) Data: We validated our approach on logs obtained by tracing method calls of running applications, where each method call was logged as an event (calling class, called class, called method). For the experiment, we used logs obtained from CrossFTP server [4] and from Columba mail client [3]. Tracing was repeated several times from a well-defined start state (for CrossFTP: no open connections, for Columba: showing the main screen of the application), which yielded several traces per log. The logs are available at [14]; Tab. I shows basic data.

3) Experiment execution: For each log, we mined branching LSCs and linear LSCs for several support and confidence thresholds on a standard Laptop at 2.4GHz and 4GB RAM. The resulting LSCs were grouped by equivalence classes of identical pre-charts and classified as *strictly branching* (if an LSC was found by the branching miner of this paper and not by the linear miner), *strictly linear* (vice versa), and *both* otherwise. Additionally, we measured (main-chart) coverage of the tree for all kinds of scenarios.

C. Results and analysis

As expected, we did not find any strictly linear scenario: each scenario found by the linear miner was also found by the branching miner. All results for both logs are available [14]; Tab. II gives a summary.

1) Research question I: As each linear-time scenario is also a branching-time scenario (each LSC satisfied according to linear-time semantics of LSC is also satisfied according to branching-time semantics), we focused our evaluation on *strictly* branching-time scenarios (ones that are not mined as linear-time scenarios). In all the execution trace sets we analyzed, we were able to mine linear as well as branching-time scenarios. For all logs and parameters the branching miner returned at least the scenarios that also the linear miner returned, plus additional, strictly branching-time scenarios.

We observed in all logs that strictly branching-time scenarios have a lower support than linear scenarios; for high enough support thresholds, only linear LSCs are found (see '# LSCs' in Tab. II). For given support and confidence thresholds, the maximal and average number of events and number of participating objects in the mined branching LSCs was higher than those in the mined linear LSCs; however pre-charts of linear LSCs tend to be longer (see length pre-/main-chart in Tab. II). Moreover, we often found linear LSCs to abstract from many behaviors between two events whereas branching LSCs tended to contain more contiguous sequences of events without abstraction. These observation make sense, as the branching ones are in a sense 'weaker' than the linear ones: for a behavior to be counted for branching, it is enough if it appears on one branch; to be counted for linear, it has to occur on all branches. As a concrete example, recall the two branching LSCs, `Delete` (Fig. 1) and `Download` (Fig. 2), and the linear LSC `Login` (Fig. 3).

Linear and branching scenarios also differ regarding the coverage measure. Branching-time scenarios yield at least the same coverage as linear-time scenarios and the lower the support value, the higher the coverage. We observed consistently higher main-chart coverage for branching-time scenarios (see 'coverage' in Tab. II). Depending on the log, linear scenario can cover the entire log with pre- and main-charts (e.g., CrossFTP for support 20). However, linear scenarios are unable to give complete main-chart coverage with confidence 1.0 if the tree has width ≥ 2. As a consequence, reasons for occurrences of a particular branch cannot be discovered with linear scenarios. For branching scenarios, main-chart coverage with confidence 1.0 is possible but not guaranteed, as discussed in Sect. V-C2.

We found the branching scenarios most informative when a number of mined branching LSCs share an identical pre-chart and thus the overall interpretation is that of several alternative continuations. For example, for the traces of crossFTP and support ≤ 14, we rediscovered the six main commands the server can handle: upload, download, delete, and rename a file, create, delete, and rename a directory. For instance, LSCs `Delete` and `Download`, shown in Fig. 1 and Fig. 2 have been mined. In contrast, the linear miner was only able to discover the invariants of each individual FTP command and returned only the LSCs corresponding to `Rename` of Fig. 5, which provide no context of their occurrences relative to each other. The main commands of Columba were harder to rediscover as we discuss next.

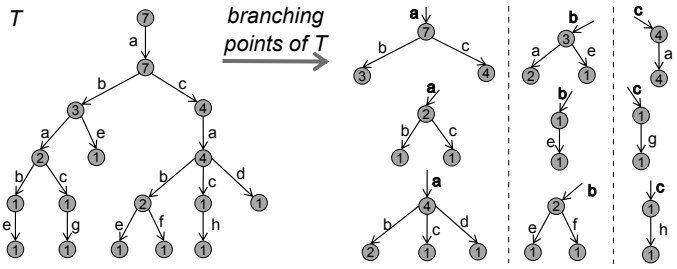

Fig. 7. Branching diversity in execution trees (node weights inscribed).

2) Research question II: We observed that for given support and confidence thresholds, some execution trace sets revealed relatively more branching LSCs than others. While simple tree properties such as maximum out degree or width did not explain the relative share of branching LSCs compared to linear LSCs, we found the "consistency of branching" in the tree to have a significant effect on branching scenarios. Tree T of Fig. 7(left) shows an example: a is always followed by b and c and only occasionally by d (as alternatives), whereas c never has the same successor. Thus, only b and c follow a with confidence 1.0. However by lowering confidence thresholds, also d becomes a possible continuation of a.

This variation with respect to branching can be quantified in a measure on the weighted tree $\mathcal{T} = (V, E, \ell, v_0, \kappa)$. For any two events a, b, let $\kappa(a) = \sum_{v \in V(a)} \kappa(v)$ where $V(a) = \{v \mid (v', v) \in E, \ell(v', v) = a\}$ (number of occurrences of a) and $\kappa^1(a, b) = \sum_{(v,v') \in E(a,b)} \kappa(v)$ where $E(a, b) = \{(v, v') \in E \mid v \in V(a), v' \in V(b)\}$ (number of a's having b as successor) and $\kappa^2(a, b) = \sum_{(v,v') \in E(a,b)} \kappa(v')$ (number of b's having a as predecessor). A branching LSC that contains a and b has *at most confidence* $\kappa^1(a, b)/\kappa(a)$ and *at least support* $\kappa^2(a, b)$. By the branching points of T shown in Fig. 7(right), $\kappa^2(a, b) = \kappa^2(a, c) = 13/13 = 1.0$, $\kappa^2(a, d) = 4/13 \approx .31$, $\kappa^2(b, e) = 6/6 = 1.0$, $\kappa^2(b, a) = 3/6 = .5$, $\kappa^2(b, f) = 2/6 \approx .33$, and $\kappa^2(c, h) = 1/6 \approx 0.17$.

Let $conf(a) = (\kappa^1(a, b_1)/\kappa(a), \ldots, \kappa^1(a, b_n)/\kappa(a))$ be the vector of confidence estimates for direct successors $b_1, \ldots, b_n$ of a in descending order. Also, let $supp(a) = (\kappa^2(a, b_1), \ldots, \kappa^2(a, b_n))$ be the corresponding support vector. In Fig. 7, $conf(a) = (1, 1, .31)$, $supp(a) = (6, 6, 1)$ (for b, c, d), $conf(b) = (1, .5, .33)$, $supp(b) = (3, 2, 1)$ (for e, a, f), $conf(c) = (.67, .17, .17)$, $supp(c) = (4, 1, 1)$ (for a, g, h).

This profile lifts to entire $\mathcal{T}$ by the vectors $conf(\mathcal{T})$ and $supp(\mathcal{T})$ with $conf(\mathcal{T})_i = \max_{a \in \Sigma} conf(a)_i$ and $supp(\mathcal{T})_i = supp(a)_i$ if $conf(\mathcal{T})_i = conf(a)_i$, for $1 \leq i \leq$ max outdegree of $\mathcal{T}$. In Fig. 7, $conf(\mathcal{T}) = (1, 1, .33)$, $supp(\mathcal{T}) = (6, 6, 1)$ telling that we find two alternative scenarios with confidence 1 and support 6 (b and c after a) and three alternatives with confidence .33 and support 1 (e,a,f after b).

TABLE II
RESULTS OF THE MINING ALGORITHMS FOR BRANCHING AND LINEAR MINER.

log	supp.	conf.	# LSCs		length pre-chart		length main-chart		coverage		runtime
			all ($\times 10^3$)	non-subsumed	avg.	max.	avg.	max.	all	main-chart	
CrossFTP	20	1.0	0/26	0/7	-/ 1	-/1	-/6.6	-/23	-/1.0	-/.9	2.5s/3.5s
	14	1.0	103/125	12/9	1/2.3	1/7	13.3/5.4	21/23	1.0/1.0	.95/.9	26s/31s
	10	1.0	6675/4055	18/9	1/2.3	1/7	18.2/5.4	33/23	1.0/1.0	.95/.9	685s/1008s
Columba	40	1.0	0/.7	0/10	-/1	-/1	-/2	-/5	-/.49	-/.34	<2s / <2s
	20	1.0	5/157	1/57	1/1.3	1/2	9/4.1	9/14	.93/.90	.71/.70	154s / 159s
	10	1.0	1136/1601	53/205	1.9/1.8	3/3	9.3/5.7	20/18	.95/.91	.75/.72	2055s / 2191s
	10	.5	1097/3742	44/163	1/1.4	1/3	6.3/5.7	26/20	.97/.93	.84/.78	2125s / 2256s

all values: strictly branching LSCs / linear LSCs

TABLE III
BRANCHING PROFILE OF THE LOGS.

	k=2	3	4	5	6	7	8	9
CrossFTP								
supp	9	9	9	9	9	1		
conf	1.0	1.0	1.0	1.0	1.0	0.45		
Columba								
supp	33	19	14	15	11	2	2	1
conf	1.0	1.0	.8	.52	.5	.25	.15	.12

Table III shows the vectors for the two logs. We can clearly see distinct characteristics of the two trees regarding consistency of branching.

Intuitively, setting confidence and support to $conf(\mathcal{T})_k$ and $supp(\mathcal{T})_k$ allows to find at least k different branching scenarios that contain the same event (as for these thresholds, there is some event having k different successors). We could confirm on our data that $conf(\mathcal{T})$ and $supp(\mathcal{T})$ roughly correlates with the number of branching scenarios having the same pre-chart. For CrossFTP, we found for support 10 and confidence 1.0, two equivalence classes each having 6 branching scenarios with the same pre-chart. We did not find any branching scenario for support values > 12. For Columba, we found 1 branching scenario for support 20 and confidence 1.0. Support 10 and confidence 1.0 yielded 2 classes of 2 branching LSCs and 1 class of 3 branching LSCs with the same pre-chart each. When lowering confidence to .5 (see Tab. II), we discovered among others 1 class of 9 branching LSCs with the same pre-chart which also revealed the main uses case of Columba: read message, send message, view account, view sub-folder, and variations of these; see [14] for details.

As $conf(\mathcal{T})$ and $supp(\mathcal{T})$ can be computed by a depth-first search on $\mathcal{T}$, this profile allows to quickly check whether the tree branches consistently enough to allow mining relevant branching scenarios. If branching is too diverse, mining linear-time LSCs (with high confidence) is preferable over mining branching-time LSCs with low confidence. The branching diversity of a tree is not necessarily inherent to the original application; branching diversity may also be high if some branches were "just not recorded" in the log. This is comparable to a log having only few traces. Then discovering linear-time LSCs with high support is also impossible.

3) Research question III: Our experiments show that mining both branching and linear scenarios from the same trace set is indeed valuable, as the two sets of mined scenarios reveal different, somewhat complementary, information about the behaviors of the application at hand.

TABLE IV
MINING ALL LSCS VS. TRIGGER AND EFFECT MINING ON CROSSFTP AT SUPPORT 10, CONFIDENCE 1.0, RESULTS FOR STRICTLY BRANCHING/LINEAR SCENARIOS.

Setting	# LSCs all		# LSCs red.	max. length	Runtime
All	6.6/4.1	($\times 10^6$)	34	18/9	685s/1008s
Trigger 1	101/3	($\times 10^3$)	26	12/1	18s/18s
Trigger 2	0/17		9	0/1	<1s/<1s
Trigger 3	53/0		15	1/0	<1s/<1s
Effect 1	0/1		4	0/1	<1s/<1s
Effect 2	2290/20		4	9/6	27s/27s
Effect 3	16/17		6	4/1	1s/1s

Trigger/Effect 1 = ⟨onConnect⟩, Trigger/Effect 2 = ⟨setLogOut⟩,
Trigger/Effect 3 = ⟨onRenameStart, setRenameFrom, onRenameEnd⟩

This is particularly evident when considering sets of branching and linear scenarios that share the same pre-chart. The branching scenarios usually show alternative continuations of the same behavior, such as the 6 commands of crossFTP. Often, these branching scenarios are accompanied by one or more linear scenarios describing invariants that hold in all alternative scenarios. There may be several such invariants. The two branching LSCs `Delete` and `Upload` (Fig. 1 and 2) and the linear LSC `Login` (Fig. 3) illustrate this.

Usually, the linear scenarios capture the behaviors that are identical and common to all alternative scenarios. We found this combination helpful to quickly understand where alternatives do differ. In addition to invariants across all alternatives, we often could find for each strictly branching-time scenario a shorter linear-time scenario that describes the "essence" of the branch; for instance `Rename` (Fig. 5). By combining both, the linear-time scenarios give insight into "framework" behaviors and "core" behaviors of an application (`Login` (Fig. 3) and `Rename` (Fig. 5) are respective examples), while the branching-time scenarios "glue" the linear-time ones together.

4) Research question IV: Finally, we investigated the proposed extension of mining scenarios with given a trigger or an effect of interest in the context of branching scenarios. First, we checked whether efficiency was improved by reduced runtime and smaller, and more focused sets of scenarios.

For this, we chose 3 triggers and effects for the CrossFTP log shown in Tab. IV (based on the LSCs in Figs. 1-5) and employed trigger and effect mining at support 10 and confidence 1.0. Table IV compares the number of found scenarios and runtimes to the case of discovering all LSCs of given support and confidence.

The number of mined LSCs reduced significantly from

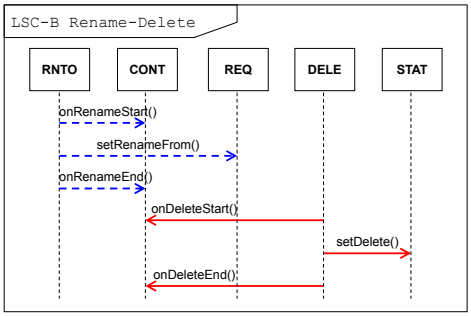

Fig. 8. Branching LSC `Rename-Delete` specifying subsequent occurrences of rename and delete operations.

10.7×10^6 scenarios (strictly branching and linear) to about 1-104 $\times 10^3$ scenarios in total. This comes with a significant reduction of runtime down to a few seconds. After removing subsumed scenarios, result sets are of similar size, but scenarios are shorter (see "max. length" in Tab. IV) which allows to analyze behavior of interest in a more focused way.

Because of significant improvements in running time, trigger/effect mining allows to drill down in the given log and investigate less frequent behaviors. Based on the branching profile of CrossFTP (Tab. III), we ran a trigger/effect analysis for support 1 and confidence .45 for Trigger/Effect 3 (Tab. III). This allowed us to find branching scenarios that were not found before at support 10. In particular, we found `Rename2` of Fig. 5 and `Rename-Delete` of Fig. 8. Together, these two strictly branching scenarios describe that the FTP operation rename can follow the FTP operation delete, and delete can follow rename (at confidence .45). In other words, we see that one FTP command can follow another FTP command without logging out, which was not observable at support 10.

VI. Discussion and Related Work

Linear vs. branching-time scenarios. The formal methods community has had a long debate about the relative merits of linear vs. branching-time logic as they are used for system specifications; which one is easier for engineers to formulate and understand, which one may have better performing synthesis and verification algorithms, which one allows compositional reasoning etc. (see, e.g., [8], [39]). Linear and branching-time logics correspond to two distinct views of time [33], and the logics LTL and CTL are expressively incomparable [8]. Thus, we consider also the choice between a universal, linear semantics for LSC, as presented in [5], [18], and a branching semantics for LSC, as presented in [37], to be open, and depend on the specific way one wishes to use the scenarios. One may view our present paper as another contribution to this long debate: specifically in the context of specification mining it makes sense to consider both, linear-time and branching-time behavior, *together*.

Two of the three authors of the present paper have introduced mining of (linear) scenario-based specifications in [28]. Several variants and extensions of this initial work have been presented [23]–[27]. The present work is different from all of the above as it mines for branching scenarios. Some of the variants and extensions applied to mining linear scenarios may be applied to branching-time scenarios as well. Other notions of branching (existential conditional) scenarios exist [11]–[13], though these cannot be discovered by our approach as they are based on partial-order semantics.

A possible use of (mined) linear and branching scenarios is formal verification. Recently, Ben-David et al. [2] have presented a translation of branching scenarios into CSSL, a new logic based on an extension of LTL with limited branching expressiveness, with an efficient model-checking procedure, specifically suitable for branching scenarios. Our mined branching scenarios may be used for model-checking using the logic and procedure presented in [2].

Other approaches to specification mining. Specification mining has attracted much research efforts in recent years; many approaches and techniques have been suggested. The different approaches vary and can be roughly categorized along two dimensions.

Some works use static specification mining and learn candidate specifications from source code (see, e.g., [1], [9], [17], [21], [32], [36], [41]), while others, like our present work, use dynamic specification mining, i.e., look for candidate specifications in execution traces (see, e.g., [6], [7], [10], [16], [19], [20], [29]–[31], [34], [35], [40], [42]). An advantage of the former is that it does not depend on the quality or coverage of a specific set of executions. On the other hand, like other static analysis methods, it may only provide an over approximation of the actual behaviors the program may exhibit at runtime. As future work, we consider the use of source code for mining branching-time scenarios on the one hand, as well as advancing the results of our present dynamic approach by using techniques that can improve the coverage of the set of executions we use as input.

Some works look for value-based invariants that hold at specific points in the program (see, e.g., [10]), while others, like our present work, look for temporal, behavioral rules, regarding the order of program actions over time (see, e.g., [15], [22], [31], [42]). As shown in [26], these two approaches may be combined. As future work, we consider such a combination in the context of branching-time scenarios.

VII. Conclusion

We introduced mining of branching-time scenarios in the form of existential, conditional live sequence charts, using a statistical data-mining algorithm. We showed the power of branching scenarios to reveal alternative behaviors, which could not be mined by previous approaches. Our work contrasts and complements previous works on mining linear-time scenarios. An implementation and evaluation over execution trace sets recorded from several real-world applications shows the unique contribution of mining branching-time scenarios to the state-of-the-art in specification mining. We consider several directions for future research. First, the integration of our present work with value-based invariants, following the ideas presented in [26]. Second, a parametric extension of our present work, following the ideas presented in [20], [24].

REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC/SIGSOFT FSE*, pages 25–34, 2007.

[2] S. Ben-David, M. Chechik, A. Gurfinkel, and S. Uchitel. CSSL: a logic for specifying conditional scenarios. In *SIGSOFT FSE*, pages 37–47, 2011.

[3] Columba, Java Email Client. http://sourceforge.net/projects/columba.

[4] CrossFTP Server. sourceforge.net/projects/crossftpserver/.

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

[6] F. C. de Sousa, N. C. Mendonça, S. Uchitel, and J. Kramer. Detecting implied scenarios from execution traces. In *WCRE*, pages 50–59, 2007.

[7] M. El-Ramly, E. Stroulia, and P. G. Sorenson. From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *KDD*, pages 315–324. ACM, 2002.

[8] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[9] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

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

[11] D. Fahland. Oclets - scenario-based modeling with Petri nets. In *PetriNets 2009*, volume 5606 of *LNCS*, pages 223–242. Springer, 2009.

[12] D. Fahland. *From Scenarios To Components*. PhD thesis, Humboldt-Universität zu Berlin, 2010.

[13] D. Fahland and A. Kantor. Synthesizing Decentralized Components from a Variant of Live Sequence Charts. In *Modelsward 2013 - 1st International Conference on Model-driven Engineering and Software Development - Proceedings*, pages 25–38. INSTICC, 2013. ISBN: 9789898565426.

[14] D. Fahland, D. Lo, and S. Maoz. Mining Branching LSCs: CrossFTP, Columba traces and results. 3TU.DataCentrum, 2013. doi:10.4121/uuid:aa7db920-aae6-4750-8975-cb739262f432.

[15] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, pages 51–60, 2008.

[16] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ICSE*, pages 15–24, 2010.

[17] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6, 000 projects: lightweight cross-project anomaly detection. In *ISSTA*, pages 119–130, 2010.

[18] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.

[19] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Mining message sequence graphs. In *ICSE*, pages 91–100, 2011.

[20] C. Lee, F. Chen, and G. Rosu. Mining parametric specifications. In *ICSE*, pages 591–600, 2011.

[21] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/SIGSOFT FSE*, pages 306–315, 2005.

[22] D. Lo and S.-C. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, pages 265–275, 2006.

[23] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *ASE*, pages 109–118, 2008.

[24] D. Lo and S. Maoz. Specification mining of symbolic scenario-based models. In S. Krishnamurthi and M. Young, editors, *PASTE*, pages 29–35. ACM, 2008.

[25] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *ASE*, pages 359–370, 2009.

[26] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *ASE*, pages 387–396, 2010.

[27] D. Lo and S. Maoz. Towards succinctness in mining scenario-based specifications. In *ICECCS*, pages 231–240, 2011.

[28] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 465–468. ACM, 2007.

[29] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 501–510. ACM, 2008.

[30] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *ICSE*, pages 85–95. IEEE Computer Society, 2007.

[31] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS*, pages 292–301, 2005.

[32] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/SIGSOFT FSE*, pages 383–392, 2009.

[33] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *ICALP*, volume 194 of *LNCS*, pages 15–32. Springer, 1985.

[34] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382. IEEE Computer Society, 2009.

[35] J. Quante and R. Koschke. Dynamic protocol recovery. In *WCRE*, pages 219–228, 2007.

[36] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, pages 174–184, 2007.

[37] G. Sibay, S. Uchitel, and V. A. Braberman. Existential live sequence charts revisited. In *ICSE*, pages 41–50, 2008.

[38] W. M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[39] M. Y. Vardi. Branching vs. linear time: Final showdown. In *TACAS*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.

[40] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *ASE*, 2008.

[41] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Autom. Softw. Eng.*, 18(3-4):263–292, 2011.

[42] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006.