Revolution: Automatic Evolution of Mined Specifications

Leonardo Mariani
Università di Milano Bicocca
Milano, Italy
mariani@disco.unimib.it

Alessandro Marchetto, Cu D. Nguyen, Paolo Tonella
Fondazione Bruno Kessler
Trento, Italy
{marchetto,cunduy,tonella}@fbk.eu

Arthur Baars
Universidad Politécnica de Valencia
Valencia, Spain
abaars@pros.upv.es

***Abstract*—Specifications mined from execution traces are largely used to support testing and analysis of software applications with little runtime variability. However, when models are mined from applications that evolve at runtime, the resulting models become quickly obsolete, and thus of little support for any testing and analysis activity.**

To cope with such systems, mined specifications must be consistently updated every time the software changes. In principle, models can be periodically mined from scratch, but in many cases this solution is too expensive or even impossible.

In this paper we describe *Revolution,* an approach for the automatic evolution of specifications mined by applying state abstraction techniques. Revolution produces models that are continuously updated and thus remain aligned with the actual implementation. Empirical results show that Revolution can suitably address run-time evolving applications.

***Keywords*-Specification mining; Model-based analysis; Finite State Machine.**

I. Introduction

Modern software applications are increasingly developed with a high degree of configurability and adaptability. For instance, more and more applications have the capability of monitoring the execution environment, taking decision at runtime, and dynamically installing and removing components. Since these applications handle unforeseen situations and make decisions at runtime, modeling such applications once for all at design time is extremely hard and could result in inadequate models (e.g., models not aligned with the implementation). Specification models should be conceived as runtime artifacts that evolve consistently with the applications, that is, when the applications and the environment change, runtime models should be promptly modified, to suitably reflect the new behavior of the modeled entities [3].

An alternative to specifying software applications at design-time is mining the specification models by observing the run-time behavior of the applications. Specification mining can naturally take into account the variability and dynamicity intrinsic of runtime evolving systems by producing new models on demand when the applications change.

In this work, we specifically concentrate on state-based mining of finite state models, that is mining finite state models from traces that include information about both the state of the application under consideration and the events that cause the transitions among the states. These models can be recovered by tools such as ADABU [4], Crawljax [11] and ReAjax [10], and have been already exploited to support a number of testing and analysis techniques [5], [10], [11].

State of the art approaches can produce a model from a set of traces, but cannot update the model when additional traces are provided. In particular, they include no mechanism to identify the outdated behaviors, and models can only grow in size and complexity. In other terms, state of the art techniques neglect the alignment between the mined models and the implementation, when the latter evolves at runtime. The only option for taking into account outdated behaviors is periodically mining a model from scratch using the most recent traces. This strategy is straightforward, yet expensive: when the system being modeled is highly dynamic, models are continuously rebuilt, wasting an enormous amount of computational resources.

In this paper we present *Revolution,* the first algorithm for continuously updating and keeping the implementation aligned with the corresponding finite state model, mined through state-based specification mining. Revolution adds expiration times to the elements of the mined specification model, and exploits such expiration time to identify the new and the outdated behaviors.

The key contributions of this paper are: (1) the definition of Revolution, an algorithm for continuous model update in state-based mining; (2) the presentation of multiple strategies to handle the expiration time; (3) the extension of ADABU and ReAjax with the automatic model update capability described in Revolution; (4) the theoretical and empirical investigation of the effectiveness of Revolution, for different configurations and strategies. The empirical results provide indications about how setting up the parameters that influence the behaviors of Revolution, and show that automatic model evolution is viable and efficient.

The paper is organized as follows: Section II provides background definitions. Sections III and IV present Revolution and analyze its computational complexity, respectively. Section V describes the experiments. Section VI discusses the empirical results. Section VII compares our work with related work. Section VIII provides final remarks.

II. State-Based Specification Mining

In this section we provide some definitions used throughout the paper; a brief description of the classic algorithm for mining specification models by state abstraction; and an example of a concrete abstraction function.

A *model* M is a tuple (N, E, T, I, F), where N is a set of abstract states (nodes), E is a set of events, T is a set of abstract transitions (transitions, for simplicity), $I \subseteq N$ is a set of abstract initial states, and $F \subseteq N$ is a set of abstract final states. A *transition* is a triple $t = (n_1, n_2, e)$, where n_1 and n_2 are abstract states (*source* and *target*, respectively), while e is an event.

An *execution trace* (trace, for simplicity) $tr = \langle ct_1, \ldots, ct_M \rangle$ is a sequence of concrete transitions, where a *concrete transition* ct is a triple (m_1, m_2, e), with m_1 and m_2 concrete states and e an event. A concrete state m is a set of variable-value associations that is logged at run time for the application being modeled. Values are concrete values traced during execution. An abstract state n is a set of variable-abstract value associations, where abstract values belong to abstract domains which define an equivalence partitioning of the concrete domains. The relationship between concrete and abstract domains is defined by an abstraction function abs, such that for every concrete state m there exists a unique abstract state n, with $n = abs(m)$. By extension, application of abs to a concrete transition gives the associated abstract transition in the model: given $ct = (m_1, m_2, e), t = abs(ct) = (abs(m_1), abs(m_2), e)$.

The input to the mining algorithm is a set of traces, each consisting of a sequence of concrete transitions. The first and last concrete transition in each trace defines the initial and final states of the model. Processing a concrete transition in a trace involves the abstraction of the transition into an abstract transition t and its addition to the model, when missing.

Different techniques for state-based mining define different abstraction functions. For instance, the ADABU tool implements an abstraction function that targets Java classes. ADABU works by first observing the execution traces of a given Java object and then mining a state machine that describes the behavior of such an object. The mined state machine mainly captures the effects of method calls on the state of the observed object. For example, let us consider an object of type *Vector*. A call to the method *add()* on an empty object has the effect of adding one object to the vector, that is, after the execution of the method *add()* the empty vector will be a non-empty vector. The resulting finite state machine will have a transition, labeled with "add()", from the empty to the non-empty state of the vector object [4].

Differently from ADABU, which mines specification models for Java classes, ReAjax implements an abstraction function that targets Ajax web applications. ReAjax works by first observing the execution traces of an Ajax application and then mining a state machine that describes the behavior of such an application. In ReAjax [10], the abstraction function is applied to the concrete state of an Ajax application, i.e., the Document Object Model (DOM) of the web page dynamically generated by the application.

Traces collected from Ajax applications contain information about the DOM states and the callbacks causing state transitions. The state of an application is inferred by analyzing the state of its DOM elements (e.g., tables, text fields and areas). DOM states are abstracted from the concrete states by means of a state abstraction function [10]. Figure 1 shows the model built by ReAjax using a set of traces obtained from the executions of a simple Ajax-based shopping cart application.

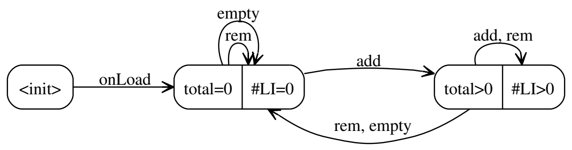

Figure 1. A mined model of a simple shopping cart application.

III. Automatic Evolution of Mined Specifications

Revolution incrementally modifies a mined specification according to an input sequence of traces. The purpose of the algorithm is to keep the behavior specified in the model consistent with the behavior of the system that produces the traces. To this end, our algorithm updates a model by adding and removing states and transitions to/from it. It is easy to recognize when a model must be extended. When the algorithm analyzes a trace that includes behaviors that are not represented in the current model, the model must be extended with new transitions and states, to incorporate these new behaviors. In absence of additional information about the system under analysis, it is more difficult to recognize when some behaviors should be removed from a model. Revolution addresses this issue according to the following heuristic: when some behaviors represented in the model are not part of traces for a "long period" of time, the algorithm removes from the model the states and transitions that correspond to these behaviors, inferring that they are outdated behaviors. The time after which a transition can be removed (the *expiration time*) is a critical parameter of our algorithm that requires a calibration procedure. In Revolution the calibration can be a preliminary activity conducted to setup the algorithm, or an online activity that takes place while the algorithm is running.

A. Revolution

Algorithm 1 shows the pseudocode for automatic model evolution. The algorithm requires two inputs: a trace tr and a model M. The algorithm returns a model M' obtained

Algorithm 1 Revolution: Automatic model update

```
Input $tr = \langle ct_1, \ldots, ct_k \rangle$: newly collected trace, possibly triggering a model update
Input $M$: model to be updated
Output $M'$: updated model
1: $M'$ := $M$

   //check and add the initial state
2: ...

   //check and add the final state
3: ...

   //check and add remaining states and edges
4: for each $i$:= 1 to $k$ do
5:     setGlobalTime(getGlobalTime() + 1)
6:     $t$ := $abs(ct_i)$
7:     $n$ := $target(t)$
8:     if $n \notin nodes(M')$ then
9:         $M'$ := $addNode(n, M')$
10:    end if
11:    $e$ := $edge(t)$
12:    if $e \notin edges(M')$ then
13:        $M'$ := $addEdge(e, M')$
14:    end if
15:    $scheduleDeletion(t)$
       //remove outdates edges
16:    for each $t$ with $deletionTime[t] \leq getGlobalTime()$ do
17:        $e$ := $edge(t)$
18:        $M'$ := $removeEdge(e, M')$
19:        $deletionTime[t]$ := $\perp$
20:    end for
21: end for
    //removes unreachable elements
22: $M'$ := $fixModel(M')$
23: return $M'$
```

Procedure 2 scheduleDeletion : Automatic Model Update

```
Configuration $x$: expiration time (in number of events)
Input $t$ : transition
1: $deletionTime[t]$ := $getGlobalTime() + x$
```

from M by adding the new behaviors in tr and removing the outdated behaviors.

The algorithm uses a notion of discrete time. The unit of measure for time is the number of events traced so far. We are aware that this measure of time is not perfect (e.g., the distance between events belonging to different user sessions is affected by the somewhat arbitrary length of the sessions considered). However, alternative measures of time, such as the number of traces collected so far or the clock time, suffer problems that are even more severe than the number of events. In fact, separation of events by number of traces is highly dependent on the duration of user sessions. Very short sessions, which generate lots of short traces, would make events old even when they are separated by just a few intermediate events. Clock time is even worse, since it depends on external factors, such as minute, hour, or time of the day (morning, afternoon, night) that have no relationship with the usage of the system (e.g., peak time will have many more accesses than idle time, but clock time proceeds uniformly). We believe that the number of events is a reasonable and universally applicable measure, preferable to its alternatives.

Revolution assigns every transition in the model with its own expiration time. When the global time is greater than or equal to the expiration time of a transition, the transition is assumed to be outdated and is removed. The expiration time of a transition is defined when the transition is created, and it is refreshed every time a trace covers the same transition. The technique to tune the expiration time of a transition is discussed in Section III-B.

When a new trace tr is processed by the automatic model update algorithm, the initial and final states are extracted from the trace and are marked as initial and final nodes in the model (line 2 and 3 of Algorithm 1). In case they are not present in the model, they are added.

Then, the core loop of the algorithm is entered (lines 4-21). In each iteration, the loop adds the states and transitions necessary to accept an event in the trace and removes the outdated edges. When a new event is processed, the first operation in the loop increments the global time (line 5). The concrete transition ct_i, which represents the concrete event that is processed, is abstracted into the abstract transition t. The edge and the target states of t are added to the model, if not already there. Finally, the expiration time associated with transition t is updated using the procedure $scheduleDeletion$. Note that the expiration time of a transition is updated both for transitions already in the model and for newly added ones (line 15). The main loop finishes by removing every obsolete transition (inner loop at line 16).

The last step of the main algorithm ensures that only nodes reachable from the initial nodes are kept in the model (line 22). Since this operation might be expensive to execute, it can be run just before the model is used, instead of running it after processing every new trace.

The implementation of $scheduleDeletion$ is shown in Procedure 2. The procedure is configured by the expiration time x. The implementation of the procedure is straightforward since it attaches to t an expiration time defined as x time ticks in the future (line 1).

B. Calibration procedure

The behavior of Revolution is influenced by the expiration time assigned to transitions. If the value of the expiration time is too low, many transitions could be deleted too early, resulting in the generation of an incomplete specification model. If the value of the expiration time is too high, many transitions could be deleted too late, resulting in the generation of an unsound model.

In this section we describe three calibration strategies that can be used to estimate the expiration time of the transitions:

calibration of the global expiration time, calibration of the per-transition expiration time, and online calibration. The purpose of the calibration algorithms is to determine a value for the expiration time parameter that ensures: (1) no false positive within the same version (i.e., the mined model remains stable if the software does not change); (2) minimum delay in recognition of obsolete transitions (i.e., outdated transitions are removed soon). We designed two calibration procedures that can be applied to traces collected from a same program version, when no discontinuity should be detected, before the system is operating in the field, and one calibration procedure designed to run in the field.

Algorithm 3 Calibration

```
Input $T$: collection of traces, used for calibration
Output $x$: expiration time, in number of events
Param $\alpha$: safety margin ($> 1$) on length of calibration (e.g., 100)
1: $x$ := 0
2: Estimate the transition probabilities $p_{ij}$ for each transition $t = \langle n_i, n_j, e\rangle = abs(ct), ct \in T$
3: Estimate the steady state probabilities $p_i$ for each state $n_i$ appearing (as source or target) in any transition $t = abs(ct), ct \in T$
4: $tm := \alpha \max_{i,j} 1/(p_i p_{ij})$    // calibration time: $\alpha$ multiplied by the expected time between two occurrences of the same transition
5: if $\sum_{t \in T} len(t) < tm$ then
6:     return error("Insufficient number of transitions in input traces")
7: end if
8: $M := \emptyset$
9: for each $tr \in T$ do
10:    $M := autModelUpdate'(tr, M)$  // no edge removal using procedure 4
11: end for
12: return $x$
```

Procedure 4 scheduleDeletion : Calibration

```
Global $\overline{x}$: computed expiration time
Input $t$ : transition
1: $timestamp[t] := getGlobalTime()$
2: $\overline{x} := max(\overline{x}, getGlobalTime() - min(timestamp[.]) + 1)$
```

Global Expiration Time: Algorithm 3 shows the pseudocode for the calibration procedure. We make the assumption that a training set of traces T collected from the same program version is available. For the training set it is required that every transition is present at least α times (we call this parameter the *safety margin* of the calibration procedure), otherwise the training set is considered too small to be used for calibration. Algorithm 3 estimates the transition probabilities p_{ij} and the steady state probabilities p_i, and determines the expected mean time between two occurrences of the same transition. This estimation is used to compute the minimum length of the traces in T (line 4). The size of the available traces is checked against this threshold value (line 5), and if the check is not passed the calibration procedure is aborted, and a better training set must be obtained.

Then, for each trace the automatic model update algorithm `autModelUpdate'` is applied. It differs from the basic automatic model update algorithm `autModelUpdate`, whose pseudocode is shown in Algorithm 1, in that it redefines the procedure *scheduleDeletion*. The redefined *scheduleDeletion* is shown in Procedure 4. The redefined procedure does not mark any transition for deletion, thereby ensuring that no edge removal is performed by the edge removal loop at line 16 of Algorithm 1. Instead it puts a timestamp on the current transition t (line 1) and it determines the expiration time $\overline{x}$ (line 2) as the maximum difference between the current time and the oldest time stamp. This ensures that no edge removal would occur when this trace is processed by the original algorithm `autModelUpdate`. The resulting expiration time $\overline{x}$ is thus the minimum value such that no false positive (no edge removal within the same version of the application) occurs if assigned in the original algorithm as the value for x.

Per-Transition Expiration Time: In order to further optimize the time of edge removal from the model, it is possible to define the expiration time of each model transition, instead of a single, global expiration time for all transitions (as done in Algorithm 3). The change to be made to Procedure 4 to obtain such effect is limited to line 2, which should be replaced by the following computation:

$$\overline{x}[t] := max(\overline{x}[t], getGlobalTime() - timestamp[t] + 1)$$

The expiration time, which is now computed separately for each abstract transition, is defined as the maximum time between two consecutive occurrences of the same transition. In this way, transitions that occur frequently will have a lower expiration time, hence allowing for a faster adaptation to changes (i.e., faster removal of obsolete edges). The global expected effect of the per-transition expiration time is a lower delay between the change of the application and the incremental update of the associated model.

The expiration time of each transition is initialized to $\perp$ ($\forall t\ \overline{x}[t] := \perp$, initially). If at the end of the calibration there exists an abstract transition t with expiration time equal to $\perp$, this must be interpreted as an infinite expiration time, since in the traces used for calibration there is no pair of consecutive occurrences of transition t.

A variant of the per-transition expiration time includes the computation of support $sup[t]$, a measure of the degree of confidence in the estimated expiration time for a given transition t. The support is initially zero for all transitions and it is incremented whenever transition t is observed in Algorithm 3. At the end of the calibration, every expiration time $\overline{x}[t]$ having a support less than S (a configurable parameter that represents the minimum confidence required for estimating the expiration time) is replaced with infinite.

Procedure 5 scheduleDeletion : Online calibration

```
Configuration $B$: buffer size (by default, unbounded)
Configuration $S$: minimum support (by default, 2)
Input $t$ : transition
1: $buf[t] := addCircularBuffer_B(buf[t], getGlobalTime())$
2: $sup[t] := sup[t] + 1$
3: if $sup[t] > S$ then
4:     $x := maxDelta(buf[t]) + 1$
5:     $deletionTime[t] := getGlobalTime() + x$
6: end if
```

Online Calibration: It is also possible to combine the calibration algorithm with the model mining algorithm, in order to avoid the need for a separate calibration phase. This solution has the benefit of adaptively reflecting changes both on the model level and on the expiration time. Calibration is computed for a transition t only if its support $sup[t]$ is above the threshold S when the most recently traced events are considered. To this aim, a buffer is maintained for each transition as the algorithm executes, containing only a fixed number of occurrences (actually, the latest event time stamps), up to the buffer size B. In this way, only recent behaviors of the application contribute to the determination of each per-transition expiration time. When changes occur, they are reflected in the statistics of the most recently observed events. Of course, before any calibration can be done (i.e., when $sup[t]$ is below threshold), we will have $x[t] := \perp$, which means infinite expiration time. The algorithm for incremental model update with online calibration differs from Algorithm 1 only in the definition of $scheduleDeletion$, which is replaced by Procedure 5. Procedure 5 has two configuration parameters: the buffer size B and the minimum support threshold S, and schedules a transition t for deletion only when its associated support is higher than S. The expiration time x for the transition t is computed by taking the maximum distances between the observation times of the last B occurrences of t.

IV. Computational complexity analysis

In this section, we compare the computational complexity of the automatic model update algorithm with global expiration time (Algorithm 1) with the complexity of periodic model mining from scratch. By adopting proper data structures, in both algorithms the following operations have unitary cost: addition/removal of a node, addition/removal of an edge, and check for the existence of a node or an edge in the model. Operation *fixModel* is computationally expensive, but we can make the assumption that it is executed only after a possibly long sequence of model update activities, just before the model is used, or can even be skipped when the presence of unreachable nodes in the model is not an issue. Hence, it will not be included in the analysis of the computational complexity.

Let us consider the behavior of the *automatic model update algorithm* (Algorithm 1) when the concrete transition ct_i is processed (lines 4-21). Let us indicate by e the event in this transition. The algorithm checks the presence of the target node and of the event in the model (lines 8, 12), which corresponds to a cost of 2. The edge is added if event e was not added during the last x time slots (where x is the expiration time). If we indicate as $P(e,t)$ the probability that event e appears in a trace of length t, event e determines the addition of an edge e with probability $1 - P(e,x)$. For the probability of adding also the target node, because it is not present in the model, we can use $1 - P(e,x)$ as an upper bound (in some cases the target node will be there, while the edge will be missing, resulting in a lower probability).

Removal of an edge e occurs whenever the timestamp for e expires at the current time slot. This means that edge e was added $x+1$ time slots before and was never added again in the following x time slots. The probability for this is also $(1-P(e,x))$ and the associated cost is unitary[1]. Overall, the expected cost for processing all concrete transitions in a trace is thus: $\sum_e 2 + 2(1 - P(e,x)) + (1 - P(e,x))$. Assuming uniform event probabilities, with $p = P(e,1) = 1/|E|$, and indicating by L the total number of events in a trace (trace length), the equation above simplifies to: $L(2 + 3(1-p)^x)$.

The computational cost for *model construction from scratch* performed periodically, after every T events, is the cost of performing x checks for node and edge additions, plus the related additions when the event is not in the previous trace portion (formally the probability of addition is $1 - P(e_i, i-1)$ where e_i indicates the event at position i of the trace). The total cost for model construction from scratch is thus: $\sum_{i=1}^{x} 2 + 2(1 - P(e_i, i-1))$.

Under the assumption of uniform event probabilities, with $p = P(e,1) = 1/|E|$, we get: $\sum_{i=1}^{x} 2 + 2(1-p)^{(i-1)} = 2x + 2\frac{1-(1-p)^x}{p}$.

Since model construction from scratch is repeated every T time slots, given a trace of length $L \geq T$, the cost computed above must be multiplied by the number of times the model is reconstructed, i.e., L/T, becoming: $\frac{L}{T}(2x + 2\frac{1-(1-p)^x}{p})$.

If we compare automatic model update and model mining from scratch, executed periodically after every T events, we get a different situation depending on the relation among the various parameters involved. For instance, when the trace contains only one event that is always already in the model ($p = 1$), the cost of model update will be just the cost of the checks: $2L$, while model mining from scratch has a cost equal to $L/T(2x + 2)$. As soon as the ratio L/T increases, the advantages of automatic model update become higher and higher. On the other hand, when the trace contains events that are always new ($p = 0$), the cost of model update must account for node/edge addition and edge removal at each time slot, for a total cost of $5L$, while model

[1]In case of online calibration this cost is $\log x$, since an ordered queue of transitions scheduled for removal must be maintained. However, the qualitative observations in this section are not affected by this change.

mining from scratch will have cost $L/T(4x) = 4xL/T$. Under the (reasonable) assumption that $x < T$ (i.e., the model is reconstructed only when x or more events have been observed), for $p = 0$ (or, more generally, low values of p), periodic model construction from scratch is more convenient than incremental model update. However, even in this situation the advantages of model construction from scratch diminish as L/T grows.

Since the comparative benefits of incremental model update depend on the actual parameters $L, T, x, P(e, x), p$, increasing with the frequency of reconstruction from scratch, L/T, and with the probability p, it is not possible to claim that the proposed algorithm is always beneficial and an empirical comparison between automatic model update and periodic model construction from scratch is needed to see if the conditions under which incremental model update is convenient hold in practice (see RQ4 of Section V).

V. Empirical Evaluation

The aim of the empirical evaluation is to assess three major variants of Revolution and compare Revolution with periodic mining from scratch. The three variants differ on the handling of the expiration time: (1) global expiration time (*GX*); (2) per-transition expiration time (*TX*); (3) online calibration of the expiration time (*online TX*).

In the following we describe the four research questions that we investigated in this paper. Since the traces obtained from the subject systems vary from one execution to another, we repeated every experimental activity required by each research question 20 times, collecting each time a new set of traces, to evaluate the average behavior of the algorithms.

RQ1 (Global vs. per-transition x): *What is the delay removal reduction and the false positive rate increase associated with per-transition expiration times, compared to the use of a global expiration time?*

The usage of per-transition expiration time compared to global expiration time reduces the delay of edge removal for frequently occurring transitions, while nothing changes for the least frequent transitions. On one hand this is beneficial for the mining, because the expiration times are precisely tailored to the characteristics of each event. On the other hand, reducing the expiration time of frequent transitions might expose them to incorrect removals (i.e., false positives). Research question RQ1 deals with the tradeoff between faster edge removal and higher false positive rate possibly associated with TX as compared to GX.

To address this research question we use a within-version collection of traces Tr split into two disjoint subsets Tr_1 and Tr_2. Traces in Tr_1 are used to compute per-transition and global expiration times using TX and GX respectively. Traces in Tr_2 are used to measure the false positive rates of TX and GX. A false positive occurs every time an edge is removed, although no behavior has changed in the application since the same version of the application is used.

The distribution of the expiration times associated with transitions quantifies the delay removal reduction achieved by using TX instead of GX. In fact the maximum value in the distribution is exactly the expiration time computed by GX. The ratio between the various per-transition expiration times and the maximum expiration time gives the reduction in the removal delay achieved by TX with respect to GX.

To answer RQ1 we collect the following metrics:

- *Expiration Times* **X**: including the global (X_g) and per transition ($X[t]$) expiration times; we collect values after Tr_1 has been fully processed.
- *Delay Reduction* **DR**: $X[t]/X_g$, i.e., the ratio between the median value of the per-transition expiration time (computed among the expiration times associated to every transition) and the global expiration time.
- *False Positives* **FP**: number of deletions occurring while processing the set of traces Tr_2, obtained from the same version of the application.

RQ2 (Convergence to initial values): *How fast do the estimated per-transition expiration times converge to the initial, finite values, for different choices of the minimum support S, and what are the associated false positive rates?*

Online calibration (online TX) is an interesting option, since it does not require any preliminary analysis. Initially every expiration time is infinite, hence no edge removal occurs. The key point is thus to understand how fast the online calibration of per-transition expiration times converges to finite values. This depends on the choice of the minimum support level S. A low value of S ensures fast convergence, but it exposes the model to incorrect removals (false positives), due to underestimated expiration times. On the other hands, higher S reduces the false positive rate, but also increases the time to convergence. Research question RQ2 deals with this tradeoff.

To address this research question we use a within-version collection of traces Tr. Automatic model update with online calibration is executed initially with $S = 2$ and then with greater values of S. We plot the number of transitions having infinite expiration time to see how fast the algorithm converges to finite expiration times. The false positive rate shows the associated cost, in terms of number of incorrect edge removals, occurring when a given support level is used. In this experiment, every edge removal is a false positive because traces are collected from the same version of the application, as for RQ1.

To answer RQ2 we collect the following metrics:

- *False Positives* **FP**: number of deletions occurring when trace Tr is processed.
- *Infinite Expiration* **IX**: number of transitions having infinite expiration time, while trace Tr is processed.

We measure IX over time to see how it evolves depending on the choice of S. We also associate each choice of S with the corresponding false positive rate FP.

RQ3 (Adaptation to changes): *How fast does the per-transition expiration time adapt to a change in the frequency of an event, for different values of the buffer size B, and what are the associated false positive rates?*

Revolution with online calibration based on a buffer of fixed size B adapts the mined model to changes in the implementation and, more generally, to changes in the relative frequency distribution for the observed events. In particular, the mined model takes into account new ways of using the application that may be observed in the field. Thus, if the application being traced and modeled is changed, or the way it is used changes, the time stamps associated with past events stored at the end of the buffer will be flushed out, to leave room to the new time stamps, which better represent the new statistics of the observed events. The buffer size must be always greater than or equal to the support ($B \geq S$). A small buffer size ($B = S$) results in fast adaptation to changes, but it may also lead to an underestimation of the expiration time, because the older time stamps, associated with a higher expiration time, are thrown out of the buffer too quickly. This may result in some false positives. On the other hand, a larger buffer ensures that the expiration time will decrease more slowly, with less false positives, but also with a slower reaction to changes. This tradeoff is investigated in this research question.

To address this research question we conduct an across-version experiment, requiring traces from an initial and from a changed version of the application. The initial traces Tr are used for initial model mining and per-transition expiration time estimation. The new traces Tr' are obtained from the same application that generated Tr, but the probability of one transition t is randomly multiplied or divided by a factor between 50, 100 or 1000, which means the associated probability $P[t]$ changes into $P'[t]$, which can be either much greater or much smaller than $P[t]$. For the transitions having a changed probability between Tr and Tr', we measure the false positive rate.

To answer R3 we collect the following metrics:

- *Buffer Size* **B**: size of the time stamp buffer used for expiration time estimation.
- *False Positives* **FP**: number of deletions of the transition t with changed execution probability $P'[t]$, occurring when trace Tr' is processed.

The buffer size B gives an approximate measurement of the adaptation rate, with a smaller B being associated with a faster adaptation rate. The number of false positives associated with the transition having a changed probability provides the cost associated with each buffer size.

RQ4 (Efficiency of automatic model update): *What is the benefit of automatic model update, compared to periodic model mining from scratch?*

The automatic model update is expected to be more efficient (in terms of computational cost) than periodic model mining from scratch only in case the frequency of model re-computation from scratch is high enough. We empirically compare the cost of automatic model update with global expiration time and periodic model mining from scratch with the aim of identifying the frequency of model re-construction that makes automatic update convenient.

Table I
SUBJECTS OF THE STUDY

Name	Type	Application	$\lvert Traces \rvert$	$\lvert Events \rvert$
CharsetDecoder	class	JavaSE	275	16750
CharsetEncoder	class	JavaSE	247	135181
WeakHashMap	class	JavaSE	158	42803
CodeWriter	class	ASM	141	6681
Uncond.FlowInfo	class	Eclipse	5352	70769
BinaryTypeBinding	class	Eclipse	305	266425
MethodScope	class	Eclipse	263	44732
View	class	Eclipse	158	1282
Expression	class	HSQLDB	1436	25972
jdbcResultSet	class	HSQLDB	61	616
TableFilter	class	Hsqldb	56	4706
TUDU	app	TUDU	629	5195

We measure the cost of model update (with add, check and remove node/edge considered as unitary costs) when a trace of length L is processed, and we compare it with the cost of model mining from scratch, performed periodically (after every T events) by processing the last x events (where $x \leq T \leq L$). We study such cost when the frequency of model mining from scratch $F = L/T$ is varied from infrequent to frequent model reconstruction.

We use a within-version trace of length L and we measure the cost of model mining from scratch at increasing values of F, every time re-constructing the model using x events.

To answer RQ4 we collect the following metrics:

- *Number of Operations* **NOP**: number of add, remove and check operations executed by the automatic model update or by model mining from scratch when processing Tr.

Since model mining from scratch is run F times on different segments of the trace Tr, NOP is computed as the sum of the values of NOP obtained for the individual segments, so as to get the cumulative cost for processing the entire trace Tr with periodic model mining from scratch.

A. Subjects

To answer research questions RQ1-4, we evaluated Revolution on 12 subjects: 11 Java classes taken from four Java applications (the Java Platform Standard Edition[2]; the Java bytecode manipulation and analysis framework ASM[3]; the Eclipse IDE[4]; and the HSQLDB[5] relational database engine) and 1 Ajax application (TUDU[6]). We use the 11 Java classes

[2]http://www.java.com
[3]http://asm.ow2.org
[4]http://www.eclipse.org
[5]http://hsqldb.org
[6]http://tudu.ess.ch

Table II
MEDIAN OF THE ACHIEVED FP AND X

App./Class	FP		X		
	X_g	$T[x]$	X_g	$T[x]$	**DR**
CharsetDecoder	0	4	3485	1716	0.49
CharsetEncoder	18	18.5	37976	25312.5	0.66
WeakHashMap	3	5	15107	934	0.06
CodeWriter	0	39.5	2652.5	591	0.22
Uncond.FlowInfo	1	66.5	30484.5	5449	0.17
BinaryTypeBinding	1.5	148	105947	32403	0.3
MethodScope	0	43	17197.5	3797	0.22
View	0	2.5	537.5	361	0.67
Expression	0	223.5	12153	2619	0.21
jdbcResultSet	1.5	8.5	219	81	0.36
TableFilter	0.5	10.5	2079.5	560	0.26
TUDU	0.5	65	2466	473	0.19

to obtain extensive empirical evidence that can answer RQ1-4 both quantitatively and qualitatively. We use the Ajax application as an in-depth case study, to confirm and further investigate the findings obtained for the Java applications on the domain of web applications.

Table I summarizes some data about the subjects and the traces we used in our study. We run the subjects with the aim of covering their functional requirements; in other terms, we traced one execution for each relevant functionality provided by the subjects. It should be noticed that for the Java subjects the monitored events are method calls, while for the Ajax subject events are calls to the GUI event handlers.

To run the experiment, we have integrated the proposed algorithm, Revolution, into two specification mining tools: ADABU (for Java applications) and ReAjax (for Ajax applications). The two modified tools can either mine a model from scratch or by incremental model update. In these two tools, we have implemented all the variants described in previous sections, including global and per-transition expiration time.

VI. RESULTS

A. RQ1: Global vs. per-transition expiration time

Table II shows the median values of false positives (FP) and delay of removal (X) for both global (X_g) and per-transition ($T[x]$) expiration time. As expected, there is a tradeoff between the number of false positives that are generated and the ability of promptly removing outdated behaviors. Looking at the results for the Java classes, we can notice that model mining based on the global-expiration time frequently produced no false positives, and at most 3 false positives (excluding the outlier *CharsetEncoder*); whereas the per-transition expiration time never produced less than 4 false positives, and in 8 out of 11 cases it produced more than 30 false positives. On the other hand, the strategy based on per-transition expiration time reacts much faster to changes than the strategy based on a global expiration time. In particular, the per-transition expiration time required less than half of the number of events required by global

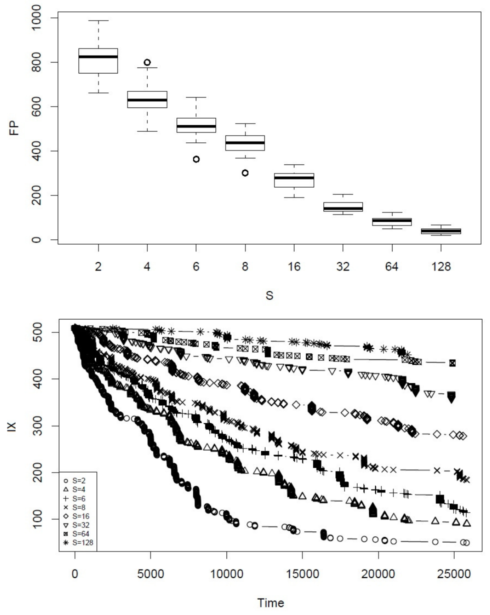

Figure 2. org.hsqldb.Expression: FP w.r.t. S (top) and IX w.r.t. Time (bottom)

expiration time ($DR < 0.5$) to detect outdated behaviors in 9 out of 11 cases.

The results with the TUDU application are consistent with these findings: the use of the global-expiration time produced 0.5 false positives on average, while the use of per-transition expiration time produced 65 false positives. On the other hand, per-transition expiration time reduces the delay of reaction by 81% compared to global expiration time.

While false positives introduce instabilities, since behaviors are incorrectly removed and are later reintroduced when they are observed again, we think that for modern, highly dynamic applications, such instabilities are acceptable (being around 30 FP for thousands of events processed), given the much shorter reaction time of automatic update (around five times faster) when per-transition expiration time is used.

B. RQ2: Convergence to initial values

Figure 2 (top) shows how false positives vary with respect to the change of S, while Figure 2 (bottom) shows how the number of transitions having infinite expiration time decreases with time for different values of S. The data plotted in the figure have been collected for the Java class org.hsqldb.Expression of HSQLDB. We do not report the plots for all 12 case studies, but similar trends have been observed also for the other 11 subjects. The results show that a low value of S ensures fast convergence but it exposes the model to many false positives, while a high value of S reduces the false positives but increases the time to convergence. We see that a value of $S = 8$ or $S = 16$ is a good trade-off, balancing false positives and speed of convergence.

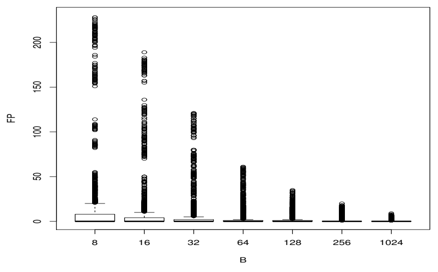

Figure 3. Java classes: FP w.r.t. B

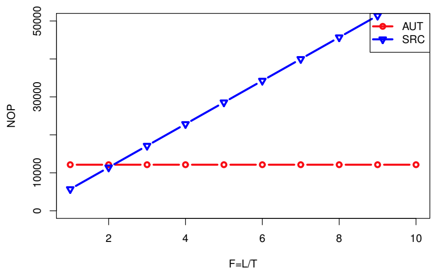

Figure 4. TUDU: NOP for AUT and SCR

C. RQ3: Adaptation to changes

Figure 3 shows the boxplots corresponding to the average number of false positives obtained on the considered Java applications when changing the probability of one event, for different values of B. Such probability change mimics a dynamic change of behavior of the application being modeled. This boxplot confirms the initial hypothesis that having a small B (e.g., $B < 64$) could lead to a high false positive rate (on average more than 15 in our experiments) while a higher value of B (e.g., $B \geq 64$) leads to a number of false positive near to 0, but also to a slower reaction to changes in probabilities. A good trade off could be around 16-32, where the false positive rate is still quite low and at the same time obsolete behaviors are completely flushed out of the buffer after just 16-32 events. A similar boxplot has been obtained for the TUDU application.

D. RQ4: Efficiency of automatic model update

Figure 4 plots the NOP (number of operations) performed by model mining from scratch (SCR) and automatic model update (AUT), for different values of the frequency of model reconstruction from scratch, $F = L/T$, for the TUDU application. Similar plots have been obtained for the other subjects of the study. We can observe that in Figure 4, SCR starts being less convenient than AUT when $F = L/T \geq 3$, that is when SCR is applied more than two times on the whole history of monitored events. We can notice that the cost of SCR increases linearly with F since in our case study we assume SCR processes exactly x events every time the model is rebuilt (i.e., $F = L/T$ times).

We computed the cut-off frequency of model mining ($F* = L/T*$) for which SCR becomes more expensive than AUT in all the application subjects. The result is that SCR is computationally more expensive than AUT for a frequency of model reconstructions $\geq$3, on all considered subjects except two, for which the cut-off frequency $F*$ is respectively 2 and 5. As a consequence, we can observe that if the model is expected to be reconstructed more than twice (four times, in the worst case) during the entire operation of the application, Revolution should be preferred to periodic model mining from scratch.

E. Discussion

The experiments produced key insights about the use of Revolution for automatically keeping an application aligned with its specification model, obtained through state-based mining algorithms. The application of our algorithm to twelve different subject systems has increased our confidence over the feasibility and viability of the approach.

The comparison between periodically rebuilding the model from scratch and automatic model update provides empirical evidence of the trade-off between the two solutions: periodically rebuilding the model from scratch is more convenient only if done once or twice (four times, in the worst case) during the entire observation period. Hence Revolution is preferable especially when the system being modeled has a (relatively) high dynamic behavior, so that its model needs to be updated with a rate that is definitely higher than twice over its entire history.

The experiments also provided indications about how to set up Revolution to satisfy the requirements imposed by the specific context of use of the algorithm. The empirical results show that per-transition expiration time should be preferred with highly dynamic systems, for which a relatively low number of false positives can be tolerated in exchange for a much faster reaction to changes. When per-transition expiration time is used, a value of the support between 8 and 16 and a buffer size between 16 and 32 appear to be a good compromise between speed of convergence, prompt reaction to changes and number of false positives.

F. Threats to validity

Some threats can potentially affect the achieved results.

Conclusion validity threats concern the relationship between the treatment and the outcome. In the study, we did not perform statistical tests to check our hypotheses about the investigated research questions. However, we observed strong trends in the data we collected that let us clearly answer the research questions.

Construct validity threats concern the relationship between theory and observation. In the study, to answer RQ3 we did not consider actual changes of the application but we approximated these changes by artificially decreasing/increasing the probability of an event. In this way, we have a better control for studying the relation between probabilities and effectiveness of our algorithm.

Internal validity threats concern external factors that may affect our dependent variables. In the case studies, we deal with the randomness of the algorithms by repeating the non-deterministic steps of the study 20 times, and reasoning on the average behavior.

External validity threats concern the generalization of our findings. A threat that can limit the generalization of the results concerns number and size of subjects used in the experiments. We have applied Revolution to extend both ReAjax and ADABU, which are state-of-the-art approaches in state-based model mining. Further investigation will be devoted to repeat the study with additional subjects.

VII. Related works

Mining of FSMs has been successfully employed to test and analyze software systems with little runtime variability [1], [4], [10]. However, the increasing runtime dynamicity of applications [7] demands for techniques that can update models according to the runtime changes affecting the underlying software. To this end, it is necessary to design incremental algorithms that can both update the models when new traces are collected, and remove the outdated behaviors that are inconsistent with the actual behavior of the software.

FSM-based models can be mined by means of two main approaches: event sequence or state abstraction inference. Event sequence abstraction takes advantage of regular language mining algorithms, such as k-tail [2], or its variants [8], [9], to produce a model that generalizes a set of event sequences. State based abstraction maps the concrete (and traced) states into abstract states, and events into transitions between abstract states [4]. Even if there are algorithms for incremental mining based on event sequence abstraction [12], and state based abstraction can be used to incrementally augment a model, none of these algorithms take into account the identification and removal of outdated behaviors. Some event-sequence abstraction algorithms can remove behaviors from a model when a negative trace is provided (i.e., a trace that represents a behavior that should not be accepted by the model) [6], but negative traces are seldom available in practice. Rather, algorithms should incorporate mechanisms to automatically eliminate outdated behaviors, based on the observation of positive traces only.

VIII. Conclusions and future work

Mined specification models can support testing and analysis activities [1], [4], [10]. However, existing solutions do not take into account the high dynamicity of modern software systems, that evolve continuously. Hence, mined models are outdated quickly and misaligned with the systems they represent. Models can be periodically re-built from scratch to incorporate the changes that happen in the software. However, when changes are frequent, periodic model re-mining can have prohibitive costs.

In this paper we presented Revolution, a technique that can incrementally and automatically update a model obtained by state abstraction at runtime. Revolution continuously incorporates in the model not only the new behaviors represented in the traces, but also automatically identifies and removes the outdated behaviors. The experimentation we conducted shows the effectiveness of Revolution and provides guidelines for configuring the algorithm.

Acknowledgment. This work has been funded by the European Union FP7 projects FITTEST (grant agreement n. 257574) and PINCETTE (grant agreement n. 257647).

References

[1] A. Babenko, L. Mariani, and F. Pastore. AVA: automated interpretation of dynamically detected anomalies. In *proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[2] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6), 1972.

[3] G. Blair, N. Bencomo, and R. France, editors. *Models@run.time*, volume 42 of *IEEE Computer*, October 2009.

[4] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *proceedings of the International Workshop on Dynamic Systems Analysis*, 2006.

[5] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *proceedings of the International Conference on Automated Software Engineering*, 2009.

[6] P. Dupont. Incremental regular inference. In *Proceedings of the International Colloquium on Grammatical Inference: Learning Syntax from Sentences*. Springer, 1996.

[7] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[8] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *proceedings of the International Conference on Software Engineering - NIER Track*, 2010.

[9] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *proceedings of the International Conference on Software Engineering*, 2008.

[10] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *proceedings of the International Conference on Software Testing, Verification and Validation*, 2008.

[11] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *proceedings of the International Conference on Software Engineering*, 2009.

[12] R. Parekh, C. Nichitiu, and V. Honavar. A polynomial time incremental algorithm for learning DFA. In V. Honavar and G. Slutzki, editors, *Grammatical Inference*, volume 1433 of *Lecture Notes in Computer Science*. Springer, 1998.