Fast-Forward Runtime Monitoring — An Industrial Case Study

Christian Colombo[1] and Gordon J. Pace[1]

Department of Computer Science, University of Malta
{christian.colombo | gordon.pace}@um.edu.mt

Abstract. Amongst the challenges of statefully monitoring large-scale industrial systems is the ability to efficiently advance the monitors to the current state of the system. This problem presents itself under various guises such as when a monitoring system is being deployed for the first time, when monitors are changed and redeployed, and when asynchronous monitors fall too much behind the system.
We propose fast-forward monitoring — a means of reaching the monitoring state at a particular point in time in an efficient manner, without actually traversing all the transitions leading to that state, and which we applied to a financial transaction system with millions of transactions already affected. In this paper we discuss our experience and present a generic theory of monitor fast-forwarding instantiating it for efficient monitor deployment in real-life systems.

1 Introduction

Our experience with the financial transactions industry has shown us that runtime verification is generally perceived as an intrusive addition which modifies the system code even if not its functionality — slowing down the system at best and introducing extra bugs at worst. The reason is mainly that, with possibly few exceptions, the existing industrial systems have not been designed with runtime verification in mind. To address the industry's concern, we have advocated asynchronous runtime verification which is completely non-intrusive given that all system events were already being logged in a database. Our architecture, embodied in the tool asyncLarva[1] and depicted in Fig. 1, consists of a system recording events (represented by a circle as in a tape recorder) in a database and subsequently, the monitor plays back (represented by a triangle) the events to check correctness. Note that, as in standard asynchronous monitoring, the system head can move forward without waiting for the monitor head to keep up. Any detected problems are communicated to an administrator since by the time the problem is detected it would usually be too late to take corrective action.

Asynchronous runtime monitoring has been successfully applied to an industrial case study [1] and proved effective in discovering issues which would have

[1] `http://www.cs.um.edu.mt/svrg/Tools/asynclarva`

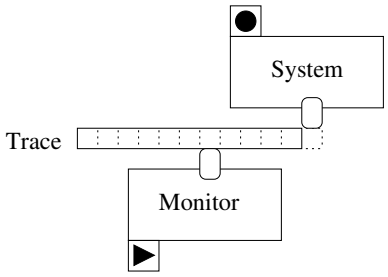

Fig. 1. Asynchronous monitoring

otherwise gone unnoticed. While this approach gives administrators a monitoring tool, it does not provide any possibility of triggering corrective action automatically to mitigate discovered problems.

As a means of enabling asynchronous monitors to take corrective action, we introduced a compensation mechanism [2] — although violations may be discovered late, the system is stopped and its state may be reversed to the point of violation. This could be relatively easily done in a financial transaction context where compensations are inherently part of the system. Also, since frequently financial systems have short bursts of peak load but a lower load on average, the monitor does not fall behind the system indefinitely. Using a box to represent stopping the system and a backward triangle to represent compensations, Fig. 2 depicts the modified architecture.

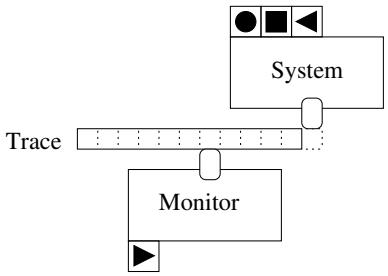

Fig. 2. Asynchronous monitoring with the possibility of compensation

As more confidence was gained in the runtime monitoring system, the next phase was to introduce the possibility of synchrony in which some monitors can run synchronously while the rest of the less crucial monitors are asynchronous. This setup [3] was particularly useful for monitoring untrusted system users who should not be allowed to proceed unless each step is approved by their monitor, *i.e.,* synchronous monitoring. In this way trusted users can still be monitored asynchronously without experiencing any service deterioration due to

monitoring. To implement synchronous and asynchronous monitoring, the system should now be capable of pausing and unpausing to wait for the monitor verdict — shown in Fig. 3 as the rightmost button with a diagonal signifying the two actions. Furthermore, it was noted that sometimes fine-grained compensations have to be discarded and replaced by coarser-grained ones. For example, if a fraud has been detected when it is too late to reverse the related purchases, it might be enough to block the offending user's account. This is shown in the diagram as a double backward triangle.

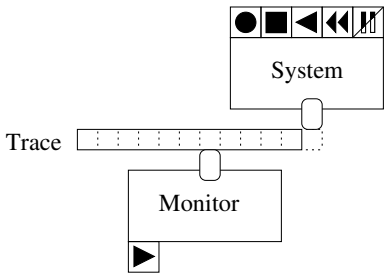

Fig. 3. Synchronous and asynchronous monitoring with the possibility of coarse-grained compensations

In the same way as coarse-grained compensations may be useful because of out-dated compensations, there are cases when 'coarse-grained' monitoring is required. For example, given a system which has already been running for a number of years, this approach can be particularly useful to bring the monitor up to scratch, taking into account the past transactions of the users. If a violation occurred over a year ago, we might not be interested in discovering it, but we still require the monitoring state to be as though the monitors have been running along with the system throughout the years it has been in operation. Another application for coarse-grained monitoring is to restart monitoring from a particular point in time if for some reason monitoring data is lost, or the monitor has been significantly modified. Coarse-grained monitoring can be thought of as a way of fast-forwarding monitoring through a trace by allowing it to skip parts of it. The full architecture is depicted in Fig. 4 showing fast-forward monitoring as two forward triangles at the monitor side.

While the full architecture without fast-forwarding has been presented in [3], in this paper we focus solely on fast-forward monitoring which proved to be necessary to monitor an industrial case study where properties are changed and augmented regularly. This dynamicity requires the monitors to fast-forward through years of data so that the property monitors can update their state with that of the system as fast as possible. To this extent, we propose a methodology whereby apart from the normal (slow) monitor processing the system trace of events, a (fast) abstract monitor is available which can process an abstraction of the system trace. By having translation functions which enable going back and

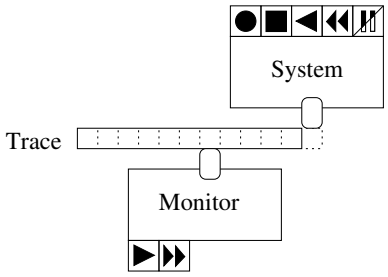

Fig. 4. The proposed architecture with coarse-grained monitoring

forth between the slow and the fast version, monitoring can be fast-forwarded to ignore irrelevant intermediate monitoring states. To the best of our knowledge, monitor fast-forwarding has not been treated in the literature and thus our contributions are as follows:

- We formalise the notion of monitor fast-forwarding in general and explain what it means for fast-forwarding to be correct. (Section 3.1)
- Next, we show how the theory is applicable to monitoring with a particular monitoring tool, LARVA. (Section 3.2)
- Through an industrial case study we show the usefulness of our approach, particularly in the context of an industrial case study (Section 4).

2 Background

To instantiate fast-forward monitoring, we have used the monitoring tool, LARVA [4], which consists of dynamic automata with timers and events (DATEs). These automata trigger on particular system events after checking that a corresponding condition holds. Conditions can be specified both on the system state and on the monitor state, the latter possibly including stopwatches and Java objects. Following a transition trigger, an action can be executed to modify stopwatches, the system or monitor state, or synchronise with other automata. Furthermore, DATEs enable monitoring on a *foreach* object basis such that a monitor is replicated for each object of a particular type. Thus, a LARVA monitor is in fact a vector of automata accompanied by a function which may dynamically launch new automata and add them to the vector.

Example 1. As an example of a LARVA monitor, below is a property which manages inactive users of a financial system. To ensure inactive accounts are safe, users who are inactive for more than six month are suspended, *i.e.,* put in *dormancy mode*, and an administration fee is charged. If a user asks for his account to be reactivated, then the request is granted but the account is switched to dormant once more if the user still remains inactive for another three months. The corresponding LARVA property checks a number of sub-properties:

1. The account is switched to dormant after six/three months of inactivity (otherwise *expiredDorm* bad state is reached).
2. The account is not put to dormant before six/three months of inactivity (otherwise *unexpectedDorm* bad state is reached).
3. The applicable fee is paid correctly (otherwise *failedPay* bad state is reached).
4. No transaction is carried out if the account is dormant (otherwise *unexpectedTx* bad state is reached).

This property is monitored for each system user and thus a replica of each monitor is instantiated for each user. Excerpts of the LARVA script which specifies the dormancy property is given in Fig. 5 (top) while the depiction of the DATE automaton is given in Fig. 5 (bottom). Note that each transition is an $event \backslash condition \backslash action$ triple and octagons represent bad states. For brevity we use *dorm* for *dormancy*, *Tx* for *transaction*, and *T* for *Timer*.

System events in the context of DATEs can be system methods calls, timer events (*e.g.,* a monitor stopwatch triggers an event after 30 minutes since it was reset), synchronisation events, or a disjunction of events. Since basic events contribute to disjunctions, then at any time instant several events may fire simultaneously.

Definition 1. *A system trace $s \in \mathcal{S}$ is a sequence of time instants such that each instant, s_i, is composed of a set of events, $E \in 2^{event}$ and a timestamp $t \in \mathbb{R}_0^+$: $s_i = (E_i, t_i)$. The alphabet of possible instants will be written as Σ: $\Sigma \stackrel{df}{=} 2^{event} \times \mathbb{R}_0^+$.*

A DATE automaton has a set of states Q connected with transitions triggering on system or timer events and guarded by conditions on the system and monitor symbolic state (ranging over Θ) and stopwatches (ranging over $\mathcal{CT}$). A monitor consists of a set of initial automata and directives to instantiate new automata dynamically upon receiving certain events. Full details of the the formal semantics of DATEs can be found in [4]. For the needs of this paper, it suffices to identify the configuration of a vector of DATE automata and explain how they form a run depending on the system events observed.

Definition 2. *A configuration $c \in \mathcal{C}$ of a vector of DATE automata, $\overline{M} \in \overline{\mathcal{M}}$, consists of the current system and monitor state, $\theta \in \Theta$, the current state of the stopwatches, $ct \in \mathcal{CT}$, a vector of locations representing the location each automaton is in, $\overline{q} \in \overline{Q}$, and the vector of automata itself: $c = (\theta, ct, \overline{q}, \overline{M})$.*

We can now outline the semantics of a vector of DATEs.

Definition 3. *The semantics of a vector of DATEs, $\overline{M} \in \overline{\mathcal{M}}$, can be given by transforming $\overline{M}$ into a labelled transition system over the configurations $\langle \mathcal{C}, c_0, \rightarrow_c \rangle$ — with states $\mathcal{C}$ (the configurations of $\overline{M}$), initial state $c_0 \in \mathcal{C}$, and transition function labelled by events and timestamps, $\rightarrow_c \in (\mathcal{C} \times \Sigma) \rightarrow \mathcal{C}$. We write $c \xrightarrow{a}_c c'$ to refer to particular $(c, a, c') \in \rightarrow_c$ and $c \stackrel{w}{\Rightarrow}_c c'$ (with $w \in \Sigma^*$) for the transitive closure of $\rightarrow_c$.*

```
Foreach (String user) {
   Variables {
      Clock T6, T3; }
   Events {
      expired6 = {T6 @ 183 days}
      anyTx = {event(currency, amount, type) where type = "generic"}
      ... }
   Property dormancy {
      States {
         Bad { expiredDorm, unexpectedDorm, failedPay, unexpectedTx }
         Normal { dorm, thawed, feepaid }
         Starting { nonDorm } }
      Transitions {
         nondorm -> nondorm [anyTx\\resetT6]
         nondorm -> dorm [dorm\T6>180 days]
         ... } } }
```

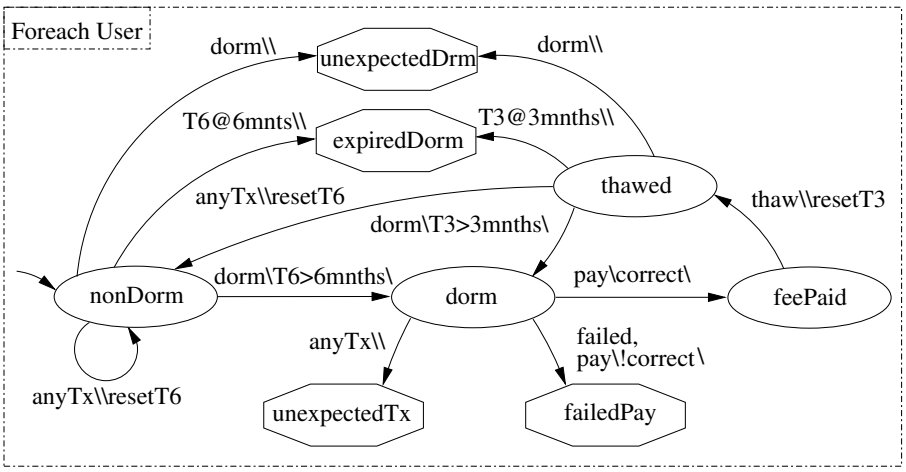

Fig. 5. The dormancy property expressed as a LARVA script (top) and as a DATE (bottom)

The set of bad configurations, $\mathcal{C}_B \subseteq \mathcal{C}$, correspond to the configurations arising from the states which are tagged as undesirable in the original DATEs. We will assume that the transition system guarantees that recovery from a bad state is not possible — if $c \in \mathcal{C}_B$ and $c \xrightarrow{a}_c c'$, then $c' \in \mathcal{C}_B$.

The set of bad traces starting from a configuration c, written $\mathcal{B}(c)$, are the strings leading to a bad configuration: $\mathcal{B}(c) = \{w \mid \exists c' \in \mathcal{C}_B \cdot c \overset{w}{\Rightarrow} c'\}$. A configuration c_2 is said to be as strict or stricter than c_1 (written $c_1 \sqsubseteq_c c_2$) if c_2 rejects all traces rejected by c_1 (and possibly more): $\mathcal{B}(c_1) \subseteq \mathcal{B}(c_2)$. We say that they are equivalent if they reject the same traces: $c_1 =_c c_2 \overset{df}{=} c_1 \sqsubseteq_c c_2 \wedge c_2 \sqsubseteq_c c_1$.

3 Monitor Fast-Forwarding

Whenever new or modified stateful properties are to be monitored on a system, monitoring has to start processing the system trace from its beginning (which might be years worth of logs) since ignoring any part of the trace might yield wrong monitoring results — the monitor state would not have correctly evolved over the whole trace. Similar problems occur when monitor state is lost due to a system crash or when asynchronous monitors fall too much behind the system. Fast-forward monitoring provides a means of going through the trace (or an abstraction of it) to infer the state (or a somewhat similar state) the monitor would have reached if the trace was monitored normally. By ignoring the intermediate monitoring states, fast-forward monitoring promises to be significantly faster than normal monitoring.

In this section we define a generic theory of monitor fast-forwarding and then we instantiate the approach on the monitoring tool Larva and explain how we support users in specifying an application of monitor fast-forwarding.

3.1 A Theory of Monitor Fast-Forwarding

The idea behind monitor fast-forwarding is to allow the monitor to skip parts of the trace but still reach the same configuration that would have been reached had the monitor progressed normally. At its simplest level, this would just involve a function to abstract traces into shorter ones. However, in practice, this is usually not sufficient, and one would require abstracting also the transition system induced by the DATE against which the trace is verified. Given a monitoring system, its fast-forward version is another monitoring system, with three additional components: (*i*) a mapping from the original monitoring system to the fast one; (*ii*) a trace abstraction which transforms an actual trace into a shorter or a more efficiently processable one; and (*iii*) a mapping from the fast monitoring system to the original one.

Definition 4. *Given a monitoring transition system $M = \langle \mathcal{C}, c_0, \rightarrow_c \rangle$ with bad states $\mathcal{C}_B$, the transition system $\langle \mathcal{A}, a_0, \rightarrow_A \rangle$ with total domain translation functions $\blacktriangleright\!\blacktriangleright \in \mathcal{C} \rightarrow \mathcal{A}$, $\blacktriangleleft\!\blacktriangleleft \in \mathcal{A} \rightarrow \mathcal{C}$, and a trace abstraction function $\alpha \in \Sigma^* \rightarrow \Sigma^*$, is said to be a fast-forward version of M.*

Typically, for actual fast-forwarding, the length of an abstracted trace is shorter than the original trace:[2] $length(\alpha(w)) \leq length(w)$. The trace abstraction function is assumed to map the empty string to itself: $\alpha(\varepsilon) = \varepsilon$ (which follows directly if the abstraction always shortens traces).

Definition 5. *A transition system A is an* exact *fast-forward version of a monitoring transition system M, with functions $\blacktriangleright\!\blacktriangleright$, $\blacktriangleleft\!\blacktriangleleft$ and α if whenever $c \overset{w}{\Rightarrow}_c c'$:*

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies c' =_c \blacktriangleleft\!\blacktriangleleft(a')$$

It is said to be an over-approximated *fast-forward version if whenever $c \overset{w}{\Rightarrow}_c c'$:*

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies c' \sqsubseteq_c \blacktriangleleft\!\blacktriangleleft(a')$$

It is an under-approximation *if $c \overset{w}{\Rightarrow}_c c'$ implies:*

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies \blacktriangleleft\!\blacktriangleleft(a') \sqsubseteq_c c'$$

We will write $c \overset{w}{\Rightarrow}_{\blacktriangleright\!\blacktriangleright} c'$ if there exist a and a' such that (i) $a = \blacktriangleright\!\blacktriangleright(c)$; (ii) $c' = \blacktriangleleft\!\blacktriangleleft(a')$; and (iii) $a \overset{\alpha(w)}{\Longrightarrow}_A a'$.

Proposition 1. *Abstracting to an exact fast-forwarded system and transforming back does not change the observable behaviour of the resulting monitor: $\blacktriangleleft\!\blacktriangleleft \circ \blacktriangleright\!\blacktriangleright \subseteq =_c$.*

Proof. The proof follows directly from the definition of exact fast-forwarded systems and the fact that $\alpha(\varepsilon) = \varepsilon$.

Example 2. Referring back to the dormancy example, we illustrate how the above theory can be instantiated for initialising dormancy monitors for the existing users of a system. For the sake of this example we will focus on deciding whether a user is in the *nonDorm* state, or the *dorm* state and set the clocks accordingly. To facilitate this decision, we need two pieces of information: (i) a list of system users, and for each user (ii) the timestamp of the latest transaction or the latest switch to dormancy, whichever is most recent.

Thus, assuming a monitor $\langle \mathcal{C}, c_0, \rightarrow_c \rangle$ and a system trace s, the corresponding translation function $\blacktriangleright\!\blacktriangleright$ transforms c_0 into a configuration of type $\mathcal{C} \times \Psi$ (using information from the system state) where Ψ includes a list of users currently active in the system. The reverse translation function $\blacktriangleleft\!\blacktriangleleft$ simply drops Ψ from the resulting configuration. The trace abstraction function α drops all events except for the most recent activity of each user (with respect to the point of initialisation). As for the fast-forward transition system, for each user there are conceptually two options: either his or her last activity was a normal transaction, in which case the user is in the *nonDorm* state and the stopwatch should be set to trigger six months from the latest transaction, or the last activity turned the user into dormant and hence the user is in the *dorm* state.

Given a monitoring transition system M and an exact fast-forwarded version M_A, then monitoring a trace partially in M and partially in the fast-forwarded

[2] We do not enforce this in the definition to allow for abstractions to an extended alphabet which although lengthens the trace, would still be processed faster.

version M_A is equivalent to monitoring it completely with the original monitor M.

Theorem 1. *Given a monitoring transition system $M = \langle \mathcal{C}, c_0, \rightarrow_c \rangle$ with bad states $\mathcal{C}_B$, and an exact fast-forward transition system $M_A = \langle \mathcal{A}, a_0, \rightarrow_A \rangle$ with functions $\blacktriangleright\!\blacktriangleright$, $\blacktriangleleft\!\blacktriangleleft$ and α, given $w = w_1 w_2 \dots w_n$, then $w \in \mathcal{B}(c_0)$ if and only if there exists $c_n \in \mathcal{C}_B$ and states $c_i \in \mathcal{C}$ such that:*

$$c_0 \xrightarrow{w_1}_C c_1 \xrightarrow{w_2}_{\blacktriangleright\!\blacktriangleright} c_2 \xrightarrow{w_3}_C c_3 \xrightarrow{w_4}_{\blacktriangleright\!\blacktriangleright} \dots \xrightarrow{w_{n-1}}_{\blacktriangleright\!\blacktriangleright} c_{n-1} \xrightarrow{w_n}_C c_n$$

If the fast-forward system is an over-approximation, then the above is a forward implication. Similarly, for an under-approximation, only the backward implication is guaranteed to hold.

This result follows by induction on the number of parts string w is split into. Interestingly, while it is generally undesirable to use over- or under-approximations, in certain scenarios one might be ready to compromise having false negatives in the case of an over-approximation and false positives in the case of an under-approximation. In the following list we suggest a number of applications of fast-forwarding monitoring, highlighting where it is preferable to use exact or approximate fast-forward monitoring:

Fast monitor bootstrapping: Whenever monitors have to be instantiated on a system with a long recorded history, running the standard monitor on the long traces may take prohibitively long. An alternative is to process the traces using an exact fast-forwarded version of the monitor. Approximate fast-forwarding can also be useful if, either we are assured that there are no errors to be caught on the stored history (in which case we can use an under-approximation) or if we prefer to process the history quickly ensuring that any errors are caught (in which case, an over-approximation would be applicable).

Burst monitoring: In systems where resources are committed only at particular points in time, it can be beneficial to accumulate and process the system trace only at these moments in time. For instance, in a transaction processing system where all database modifications are committed at the end of a transaction one may, for example, collect the full trace of a transaction and process it using a fast-forwarded monitor. If an exact fast-forward may still be too expensive to check and performance is an issue, one may choose to apply over-approximations for transactions by blacklisted users and under-approximations for whitelisted ones.

Synchronous/asynchronous monitoring: In the case of asynchronous monitoring, fast-forward monitoring can be used whenever the monitor is lagging too much behind the system. Moreover, in monitoring systems such as [3] where asynchronous monitoring can be synchronised at runtime, fast-forward monitoring can be used for a quick synchronisation.

In the rest of the paper, we will focus on the first of the above applications of fast-forwarded monitoring, showing the applicability of the approach and its gains.

3.2 Instantiating Fast-Forwarding to Larva

In Larva we use monitor fast-forwarding to start monitors from a particular point in a system's history, *i.e.,* fast monitor bootstrapping. This is crucial for industrial systems so that when properties are modified, the monitor comes up to scratch with the system as soon as possible — monitoring years' worth of data would waste monitoring time which could be used to start monitoring more recent (and thus more relevant) data.

Instantiating the theory of fast-forwarding to monitor bootstrapping in Larva would include deciding the two translation functions $\blacktriangleright\blacktriangleright$ and $\blacktriangleleft\blacktriangleleft$, the trace abstraction function α, and the fast-forward transition relation $\rightarrow_A$. The following list expands each of these aspects, generalising the approach taken in Example 2:

- In the case of Larva it is assumed that the translation function $\blacktriangleright\blacktriangleright$ obtains the list of objects which should be monitored during fast-forwarding while the reverse translation function $\blacktriangleleft\blacktriangleleft$ drops the additional information.
- The trace which originally includes all the system events, is collapsed to the trace elements which are required for deciding the state of each monitor.
- The fast-forward monitor which handles fast bootstrapping, first obtains $\overline{M}$ by instantiating a monitor for each entity indicated by $\blacktriangleright\blacktriangleright$ and subsequently allows each entity to perform one step updating its monitor state based on the abstracted trace. Such a configuration step should include three aspects corresponding to the configuration components $\overline{q}$, ct, and θ respectively:
  1. The state of the respective monitor (an element of $\overline{q}$) is updated (*e.g.,* monitors of users whose account has been put into dormant state in the past should be in the state *dorm*).
  2. The clocks of each monitor are set (*e.g.,* when the last financial transaction took place so that it can be ensured that inactive users are actually put into the dormant state).
  3. The values of the variables of each monitor are set (*e.g.,* counting the number of transactions which the user has carried out so that monitors can check that the allowed quota has not been exceeded).

To facilitate the specification of the abstracted monitor for Larva users, we have introduced some minor additions to Larva scripts as explained in the following.

3.3 Adapting Larva Scripts

To enable users to easily program fast monitor bootstrapping, we have augmented the Larva script structure. Recall that Larva provides the *foreach* construct which enables monitors to be replicated for distinct objects of a particular type. Furthermore, *foreach* components can be nested (*e.g.,* for monitoring each credit card of each user) and the outermost *foreach* components are enclosed in a *global* component which can be used to monitor properties which are not replicated, *i.e.,* not related to particular objects. To initialise a *global* component

what is required is to give a value to variables, clocks and update the state of any global property automata. For this reason we have added the *initializeIf* component which triggers on a particular condition (indicating that the monitor is in fast-forward mode) where the user can specify a Java method which returns a hashmap with variable/clock/automata names as keys and the corresponding intended values as the hashmap values. Note that no setting needs to be done if the variables/clocks/states have not progressed from their default initialisation. In Fig. 6 we show how using two SQL queries we deduce when the last successful transaction occurred for a particular user and whether the user has been recently (since the last successful transaction) put into dormant state. Using this information we set the corresponding clocks to trigger when the user should be put to dormant in the future. Moreover, if the user is currently dormant, then the corresponding *dormancy* automaton (shown in Fig. 5) is to be in state *dorm*. Note that we assume that the current system state does not contain errors and consequently our fast-forwarding is an under-approximation.

Yet it is not enough to be able to initialise a monitor for one particular user — the LARVA script should also allow the script writer to specify a means of deducing the number of users in the system for whom a monitor should be replicated and initialised. Note that this is useless for a *global* component for which no replication takes place anyway. For this reason, each *foreach* may contain an *initially* component (apart from an *initializeIf* component) which can specify a method returning an array with all the objects for which a monitor should be created[3]. In our example the *initially* method returns an array of user ids.

The approach described in this section has been successfully applied to the live data of an industrial case study with promising results as elaborated in the following section.

4 Case Study

We have applied our architecture on Entropay, an online prepaid payment service offered by Ixaris Systems Ltd[4]. Entropay users deposit funds through funding instruments (such as their own personal credit card or through a bank transfer mechanism) and spend such funds through spending instruments (such as a virtual VISA card or a Plastic Mastercard). The service is used worldwide and thousands of transactions are processed on a daily basis.

In our case study 15 properties written as DATEs (each including several sub-properties as explained in the *dormancy* example) have been monitored on Entropay which can be loosely classified under:

Life cycle properties checking that operations occur at the right stage of a user's life cycle (*e.g.*, a user cannot carry out financial transactions if his account is dormant).

[3] LARVA supports *foreach* components for tuples of objects. Thus, the *initially* method actually returns an array of arrays where each array supplies an element of the tuple.

[4] `http://www.ixaris.com`

```
Foreach (String user) {
  Initializeif (init) {
    static HashMap<String, Object> initializeifUser(String user) {
      HashMap<String, Object> list = new HashMap<String, Object>();

    //obtain last successful user transaction
      rs = st.executeQuery(
           SELECT timestamp FROM transaction_table
           WHERE id=@user AND timestamp < @initializationTime
           ORDER BY timestamp DESC);
      latestTrans = rs.getLong("timestamp");

    //check if user is currently dormant
      rs = st.executeQuery(
           SELECT timestamp FROM log_table WHERE id=@user
           AND event="USER_DORMANT" AND timestamp < @initializationTime
           ORDER BY timestamp DESC);
      latestDorm = rs.getLong("timestamp");

      if (latestDorm > latestTrans) {
        //i.e. user is currently dormant
        //therefore put automaton into "dorm" state
      } else {
        //i.e. user is not dormant
        //set clock to expire 6 months after last transaction occurred
      }
...
     return list; } }

//code given in the previous example starts here
  Variables {...} Events {...} Property ...
//code given in the previous example ends here

  Initially {
    static ArrayList initiallyUsers() {
      ...
      rs = s.executeQuery(SELECT id FROM users_table;);
      while (rs.next())
        list.add(rs.getString("id"));
      return list;
  } } }
```

Fig. 6. The dormancy example augmented with fast bootstrapping code

Real-time properties checking that actions which should be system-triggered are carried out on time (*e.g.,* the system should automatically put to dormant accounts which have been inactive for more than six months).

Rights properties checking that the user has the appropriate rights before a transaction is permitted (*e.g.,* for a user to log into the system he or she must have the *login* right).

Limits properties checking that the frequency and value of certain transactions fall within the stipulated limits (*e.g.,* no more than 100 purchases are allowed each month).

The case study was successfully executed on a sanitized[5] database of 300,000 users with around a million virtual cards and a number of issues have been detected through the monitoring system[6].

Since Entropay had been up and running for more than a year at the time of applying monitoring and it was envisaged that monitors would have to be modified or added regularly, fast monitor bootstrapping was crucial in making monitoring feasible. More details are given in the following subsection.

4.1 Results

The monitor was deployed on data representing activities starting from 23 December 2008. Data before this date was considered to be too old and would waste monitoring time which could more beneficially be used to monitor more recent data pertaining to users which are more probably still active at the time of monitoring. To quickly bootstrap the monitors up to 23 December 2008, we used the fast-forward technique on 58 weeks of data starting from 8 November 2007.

Using a Dual Core AMD Opteron Processor at 1.81GHz running Windows XP x64 with 2Gb RAM, the monitors successfully fast-forwarded through 58 weeks in 35 hours. Subsequently, the monitors were run on the available data (at the time when this case study was carried out) dating till 8 September 2009 (including 37 weeks of data) consisting of millions of transactions. This process took 552 hours (approximately 23 days) equating to less than 15 hours of processing per one week's data. Proportionately, monitoring the 58 weeks of data would have roughly taken the monitor 36 days to come at par with the live system as opposed to the day and a half with fast-forwarded initialisation. This time saving is crucial when one would need to receive immediate feedback upon deploying new monitors, particularly if remedy actions can be taken based on monitoring results.

We have not tried out this case study using other runtime verification tools. Thus, although past experiments [4] have shown our tool's performance to be comparable to that of other tools, it is difficult to discuss the performance in itself. However, these experiments do clearly highlight the effectiveness of fast-forward monitoring in significantly reducing the monitoring time.

[5] User information was obfuscated for the purpose of this study.

[6] The issues have been extensively reported in [2, 3].

4.2 Discussion

The downside of the current instantiation of monitor fast-forwarding is that the user has to program the fast-forwarding abstractions manually. From our experience, coming up with fast-forwarding monitors is more challenging than devising normal monitors. The reason is that normal monitoring is usually more similar to the typical specifications accompanying industrial systems, while the logic needed for fast-forward monitoring can only be obtained by having an intimate knowledge of the system at hand. For example, referring back to the dormancy example, the industrial specifications are written in the following imperative style: (*i*) *The cycle starts when a registered user is inactive for six months, at that point the user account must be put to dormant.* (*ii*) *Whilst dormant, the user may not perform any transactions but may ask to be reactivated.* (*iii*) *If the user has been reactivated by does not carry out a financial transaction for another three months, the user account is deactivated again.* This logic can almost be directly translated into normal monitors with states and transitions. On the other hand, programming under-approximating (*i.e.,* assuming the system worked correctly before monitoring started) fast-forward monitoring would requires declarative knowledge such as: (*i*) *If the user has performed financial transactions since the last time he or she has been dormant, then the user must be active.* (*ii*) *On the other hand, if no transactions have been carried out since, then the user is still dormant.* (*iii*) *Yet another possibility is that if the user has carried out (only) non-financial transactions, then the user has been thawed but not fully activated yet.* Although statements such as the latter can usually be inferred from the former, they are not typically written in technical specifications and engineers are more accustomed to the imperative style of specifications.

One use of more generic fast-forwarding in our case study would be to enable the monitor to keep up with the system in case asynchronous monitoring is consistently slower than the system. However, as yet, we have never encountered monitors which are not able to keep up with the system. In our case study results it is noteworthy that once monitors come at par with the system, it is not a problem for the monitors to keep up; with slightly more than two hours of processing for a day's events. Still, if one had to adapt the above given code for fast-forwarding asynchronous monitoring, this can be done by simply adding a condition in the SQL statements to ignore entries before a particular date (the date where the slow asynchronous monitors have reached).

5 Conclusion

To the best of our knowledge the idea of fast-forward monitoring is novel. A notion which relates to fast-forward monitoring is counterexample shrinking [6, 7] from the area of testing. For example QuickCheck [5], a model-based testing tool for Erlang, attempts to find a shorter trace when a bug is found. This simplifies the developers' task of debugging since it is easier to understand what happened in a simpler trace. Note that counterexample shrinking is a special case of our fast-forwarding theory: (*i*) the translation functions are the identity

functions, (*ii*) the trace abstraction function returns a shorter or simpler trace, and (*iii*) the same identical oracle that is used during testing is used during shrinking. Exact fast-forwarding ensures that the same bug that was exhibited during the original trace is also exhibited when monitoring the simpler one.

In a monitoring environment where the monitoring impact on system performance should be strictly minimal, fast-forward monitoring can be useful to enable the monitor to keep up with the system so that the monitoring effort is spent on monitoring the most relevant events. This problem is not only encountered the first time monitoring is deployed on the live system but also whenever a new version of monitoring code is used or a new property is added. In all such cases, the monitor has to update its state so that it comes in line with the system state. The theory of fast-forwarding has been instantiated for the monitoring tool LARVA by enhancing its script with two new components which enable users to specify fast monitor bootstrapping. We have shown the usefulness of this approach for an industrial case study where monitors have to process millions of records before starting to process relevant events. In the future this technique can be incorporated with heuristics (in a similar fashion to [3]) so that the monitor can be fast-forwarded at the start of a critical section where timely monitoring is crucial. This approach might be more practical than the approach proposed in [3] of simply pausing the system to wait for the monitor to keep up.

References

1. Colombo, C., Pace, G.J., Abela, P.: Offline runtime verification with real-time properties: A case study. Tech. rep., Department of Computer Science, University of Malta (2009), internal report 01-WICT-2009
2. Colombo, C., Pace, G.J., Abela, P.: Compensation-aware runtime monitoring. In: Runtime Verification - First International Conference, (RV). Lecture Notes in Computer Science, vol. 6418, pp. 214–228. Springer (2010)
3. Colombo, C., Pace, G.J., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods in System Design 40, 1–26 (2012)
4. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Formal Methods for Industrial Critical Systems (FMICS). Lecture Notes in Computer Science, vol. 5596, pp. 135–149. Springer (2008)
5. Hughes, J.: Quickcheck testing for fun and profit. In: Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 4354, pp. 1–32. Springer (2007)
6. Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient unit test case minimization. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 417–420. ACM (2007)
7. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. 28(2), 183–200 (2002)