Formal Methods in Systems Design manuscript No.
(will be inserted by the editor)

Safer Asynchronous Runtime Monitoring Using Compensations

Christian Colombo · Gordon J. Pace · Patrick Abela

Received: date / Accepted: date

Abstract Asynchronous monitoring relieves the system from additional overheads induced through online runtime monitoring. The price paid with such monitoring approaches is that the system may proceed further despite having reached an anomalous state. Any actions performed by the system after the error occurring are undesirable, since for instance, an unchecked malicious user access may perform unauthorized actions. In this paper we investigate the use of compensations to enable the undoing of such undesired actions, thus enriching asynchronous monitoring with the ability to restore the system to the original state just after the anomaly had occurred. Furthermore, we show how adaptive synchronisation and desynchronisation of the monitor with the system can also be achieved and report on the use of the proposed approach on an industrial case study of a financial transaction handling system.

1 Introduction

Runtime verification techniques have addressed the increasing need for system correctness as a relatively lightweight approach for system verification, which scales up to large systems while still guaranteeing the detection of abnormal behaviour. Although monitoring of properties is usually computationally cheap when compared to the actual computation taking place in the system, the monitors induce an additional overhead which is not always desirable in real-time, reactive systems. In transaction processing systems, the additional overhead induced by each transaction can limit throughput and cripple user-experience at peak times of execution. This is particularly true in applications where load tends to converge at

The research work disclosed in this publication is partially funded by the Malta National Research and Innovation (R&I) Programme 2008 project number 052.

C. Colombo, G. J. Pace
Dept. of Computer Science,
University of Malta, Malta
E-mail: {christian.colombo, gordon.pace}@um.edu.mt

P. Abela
Ixaris Ltd, Malta
E-mail: patrick.abela@ixaris.com

particular times. For example, in an online betting setting, one would expect a pattern of usage which surges during particular intervals when response time and performance are at a premium, but with relatively low load for the rest of the time. One approach, sometimes adopted in such circumstances, is that of evaluating the monitors asynchronously with the system, possibly on a separate address space. The overhead is reduced to the cost of logging events of the system, which will be processed by the monitors. However, the downside of this approach is that by the time the monitor has identified a problem, the system may have proceeded further.

The problem is closely related to one found in long-lived transactions [16] — transactions which may last for too long a period to allow for locking of resources, but which could lead to an inconsistent internal state if the resources are released too early. To solve the problem, typically one defines *compensations*, to undo partially executed transactions if discovered to be infeasible halfway through. In the case of asynchronous monitoring, allowing the system to proceed before the monitor has completed its checks may lead to situations where the system should have been terminated earlier. As with long-lived transactions, we allow this run-ahead computation and adopt the use of compensations in our setting to enable the undoing of system behaviour when an asynchronous monitor discovers a problem late, thus enabling the system to rollback to a sane state. However, in many real-life cases, it is not realistic to assume that transactions can be undone at any time after their completion. Therefore, we enrich our compensation model with scopes marking boundaries beyond which compensation is no longer possible. Furthermore, in a setting such as transaction-processing systems, one can afford most of the time to run the monitors in synchrony with the system, falling back to asynchrony only when required due to high system load. In more stringent settings, monitors can be run asynchronously, only synchronising when there is a high risk of violation.

A compensation-aware monitoring architecture has been proposed [9] to enable loosely-coupled execution of monitors with the system, typically running synchronously, but allowing for de-synchronisation when required and re-synchronisation when desired. Although manual switching from synchronous monitoring to asynchronous monitoring is plausible, it is preferable to automate this process through the use of heuristics. Thus, we propose an extension to the existing architecture to support the incorporation and easy modification and extensibility of such heuristics. As an example, we choose a set of heuristics based on a real-life case study and implement them in terms of DATEs [10] (dynamic automata supporting event-based monitoring), facilitating the integration with the current DATE-based compensation-aware monitoring implementation, cLarva.

This paper is an extended and revised version of [9] with full proofs of the main results and the following new contributions: (i) the compensation model has been enriched to allow scoped compensations such that once the scope is closed, compensation of actions within that scope is no longer possible; (ii) we prove that the enriched compensation model is sound; (iii) we outline possible heuristics which can be used to automate the decision to switch between synchronous and asynchronous monitoring, a selection of which have been implemented in terms of other monitors; (iv) discuss how a heuristics component has been incorporated within the compensation-aware monitoring architecture; and (v) the case study is further elaborated upon.

The paper is organised as follows: in section 2 we present background necessary to reason about compensations, which we use to formally characterise compensation-aware monitoring in section 3. In section 4, we extend compensation-aware monitoring to include scopes and prove that the previous result still hold. An architecture implementing this mode of monitoring is presented in section 5, and we propose a number of heuristics and show

how these can be incorporated within the monitoring architecture in section 6. Next, we illustrate the use of our architecture on an industrial case study in section 7. Finally, we discuss related work in section 8 and conclude in section 9.

2 Compensations

Two major changes occurred which rendered traditional databases inadequate in certain circumstances [15,16]: (i) the advent of the Internet facilitated the participation of heterogeneous systems in a single transaction, and (ii) transactions became longer in terms of duration (frequently, the latter being a consequence of the former). These changes meant that it was possible for a travel agency to automatically book a flight and a hotel on behalf of a customer without any human intervention — a process which may take time (mainly due to communication with third parties and payment confirmation) and which may fail. Resource locking for the whole duration of the transaction became impractical since it may cause severe availability problems. This scenario motivated the need for a more flexible way of handling transactions amongst heterogeneous systems while at the same time ensuring correctness.

A possible solution is the use of compensations [15,16] which are able to deal with partially committed long-lived transactions with relative ease. Taking again the example of the flight and hotel booking, if the customer payment fails, the agency might need to reverse the bookings. This can be done by first cancelling the hotel reservation followed by the flight cancellation, giving the impression that the bookings never occurred. Although several notations supporting compensations have been proposed [3–5,17,23], little work [5,6] has been done to provide a mathematical basis for compensation correctness. For example, in the case of compensating CSP (cCSP) [5], to study the effect of the use of compensations, it is assumed that they are perfect cancellations of particular actions. This leads to the idea that executing an action followed by the execution of its compensation, is the same as if no action has been performed at all. In practice, it is rarely the case that two operations are perfect inverses of each other and that after their execution no trace is left. However, the notion of cancellation is useful as a check to the soundness of the formalism.

In this section we present the necessary background notions of cancellation compensations, based on [5].

2.1 Notation

To enable reasoning about system behaviour and compensations, we will be talking about finite strings of events. Given an alphabet Σ, we will write Σ^* to represent the set of all finite strings over Σ, with ε denoting the empty string. We will use variables a, b to range over Σ, and v, w to range over Σ^*. We will also assume action τ indicating internal system behaviour, which will be ignored when investigating the externally visible behaviour. We will write Σ_τ to refer to the alphabet consisting of $\Sigma \cup \{\tau\}$.

Definition 1 Given a string w over Σ_τ, its *external manifestation*, written $w^{-\tau}$, is the same string but dropping instances of τ.
Two strings v and w are said to be *externally (or observationally) equivalent*, written $v =_\tau w$, if their external manifestation is identical: $v^{-\tau} = w^{-\tau}$. We say that a set of strings W is contained in another set W' up to external manifestation, written $W \subseteq_\tau W'$, if for every string

in W, there is an externally equivalent string in W'. Set equality up to external manifestation, $W =_\tau W'$, is defined as containment in both directions.

External equivalence is an equivalence relation, and a congruence up to string concatenation.

2.2 Compensations

For every event that happens in the system, we will assume that we can automatically deduce a compensation which, in some sense, corresponds to the action to be taken to make up for the original event. Note that executing the two in sequence will not necessarily leave the state of the system unchanged — a typical example being that of a person withdrawing a sum of money from a bank ATM, with its compensation being that of returning the sum but less bank charges.

Definition 2 Corresponding to every event a in alphabet Σ, its compensation will be denoted by $\overline{a}$. We will write $\overline{\Sigma}$ to denote the set of all compensation actions. For simplicity of presentation, we will assume that the set of events and that of their compensations are disjoint[1]. Extending compensations to an alphabet enriched with the internal action τ, we assume that $\overline{\tau} = \tau$.

We also overload the compensation operator to strings over Σ_τ, in such a way that the individual events are individually compensated, but in reverse order: $\overline{\varepsilon} \stackrel{\text{def}}{=} \varepsilon$ and $\overline{aw} \stackrel{\text{def}}{=} \overline{w}\,\overline{a}$. For example, $\overline{abc} = \overline{c}\,\overline{b}\,\overline{a}$.

To check for consistency of use of compensations, the approach is typically to consider an ideal setting in which executing a, immediately followed by $\overline{a}$ will be just like doing nothing to the original state. Although not typically the case, this approach checks for sanity of the triggering of compensations.

Definition 3 The compensation cancellation of a string simplifies its operand by (i) dropping all internal actions τ; and (ii) removing actions followed immediately by their compensation. We define $cancel(w)$ to be the shortest string for which, after dropping all internal actions, there are no further reductions of the form $cancel(w_1 a \overline{a} w_2) = cancel(w_1 w_2)$.

Since the sets of normal and compensation events are disjoint, strings may change under cancellation only if they contain symbols from both Σ and $\overline{\Sigma}$. Cancellation reduction is confluent and terminates.

Definition 4 Two strings w and w' are said to be *cancellation-equivalent*, written $w =_c w'$, if they reduce via compensation cancellation to the same string: $cancel(w) = cancel(w')$. A set of strings W is said to be *included in set W' up-to-cancellation*, written $W \subseteq_c W'$, if for every string in W, there is a cancellation-equivalent string in W':

$$W \subseteq_c W' \stackrel{\text{def}}{=} \forall w \in W \cdot \exists w' \in W' \cdot w =_c w'$$

Two sets are said to be *equal up-to-cancellation*, written $W =_c W'$, if the inclusion relation holds in both directions.

[1] One may argue that the two could contain common elements — e.g. *deposit* can either be done during the normal forward execution of a system, or to compensate for a *withdraw* action. However, one usually would like to distinguish between actions taken during the normal forward behaviour and ones performed to compensate for errors, and we would thus much rather use *redeposit* as the name of the compensation of *withdraw*, even if it behaves just like *deposit*.

Cancellation equivalence is an equivalence relation, and is a congruence up to string (and language) concatenation. Furthermore, a string followed by its compensation cancels to the empty string:

Proposition 1 *The concatenation of a string with its compensation is cancellation equivalent to the empty string:* $\forall w \cdot w\overline{w} =_c \varepsilon$.

3 Compensations and Asynchronous Monitoring

In order to be able to reason about compensation-aware monitoring, and its correctness relative to regular monitoring strategies, we start by characterising synchronous and asynchronous monitoring. In the synchronous version, it is assumed that the system and monitor perform a handshake to synchronise upon each event. In contrast, in the asynchronous approach, the events the system produces are stored in a buffer, and consumed independently by the monitor, which may thus lag behind the system. We then define a compensation-aware monitoring strategy, which monitors asynchronously, but makes sure to undo any system behaviour which has taken place after the event which led to failure.

3.1 Synchronous and Asynchronous Monitoring

We will assume a labelled transition system semantics over alphabet Σ for both system and monitor. Given a class of system states S, we will assume the semantics $\longrightarrow_{sys} \subseteq S \times \Sigma \times S$, and similarly a relation $\longrightarrow_{mon}$ over the set of monitor states M. We also assume a distinct $\odot \in S$ identifying a stopped system, and $\otimes \in M$ denoting a monitor which has detected failure. Both $\odot$ and $\otimes$ are assumed to have no outgoing transitions.

Using standard notation, we will write $\sigma \xrightarrow{a}_{sys} \sigma'$ (resp. $m \xrightarrow{a}_{mon} m'$) as shorthand for $(\sigma, a, \sigma') \in \longrightarrow_{sys}$ (resp. $(m, a, m') \in \longrightarrow_{mon}$). We write $\overset{w}{\Longrightarrow}_{sys}$ and $\overset{w}{\Longrightarrow}_{mon}$ (where $w \in \Sigma^*$) to denote the reflexive transitive closure of $\longrightarrow_{sys}$ and $\longrightarrow_{mon}$ respectively.

Definition 5 The transition system semantics of the synchronous composition of a system and monitor is defined over $S \times M$ using the rules given in Fig. 1. The rule Sync defines how the system and monitor can take a step together, while SyncErr handles the case when the monitor discovers an anomaly. A state (σ, m) is said to be (i) *suspended* if $\sigma = \odot$; (ii) *faulty* if $m = \otimes$; and (iii) *sane* if it is not suspended unless faulty ($\sigma = \odot \implies m = \otimes$).

The set of traces generated through the synchronous composition of system σ and monitor m, written $\mathsf{traces}_{\|}(\sigma, m)$ is defined as follows:

$$\mathsf{traces}_{\|}(\sigma, m) = \{w \mid \exists (\sigma', m') \cdot (\sigma, m) \overset{w}{\Longrightarrow}_{\|} (\sigma', m')\}$$

Example 1 Consider a system P over alphabet $\{a, b\}$ and a monitor A which consumes an alternation of a and b events starting with a i.e. $abab\ldots$ but breaks upon receiving any other input. The synchronous composition of the system and monitor takes a step if and only if both the system and the monitor can take a step on the given input. Therefore, if the system performs event a: $(P, A) \xrightarrow{a}_{\|} (P', A')$. If system P performs a b instead, the system would break: $(P, A) \xrightarrow{b}_{\|} (\odot, \otimes)$.

Proposition 2 *A sequence of actions is accepted by the synchronous composition of a system and a monitor, if and only if it is accepted by both the monitor and the system acting independently. Provided that* $m' \neq \otimes$, $(\sigma, m) \overset{w}{\Longrightarrow}_{\|} (\sigma', m')$, *if and only if* $\sigma \overset{w}{\Longrightarrow}_{sys} \sigma'$ *and* $m \overset{w}{\Longrightarrow}_{mon} m'$.

In contrast to synchronous monitoring, asynchronous monitoring enables the system and the monitor to take steps independently of each other. The state of asynchronous monitoring also includes an intermediate buffer between the system and the monitor so as not to lose events emitted by the system which are not yet consumed by the monitor.

Definition 6 The asynchronous composition of a system and a monitor, is defined over $S \times \Sigma_\tau^* \times M$, in terms of the three rules given in Fig. 1. Rule $\textsc{Async}_S$ allows progress of the system adding the events to the intermediate buffer, while rule $\textsc{Async}_M$ allows the monitor to consume events from the buffer. Finally rule AsyncErr suspends the system once the monitor detects an anomaly. Suspended, faulty and sane states are defined as in the case of synchronous monitoring by ignoring the buffer.

The set of traces accepted by the asynchronous composition of system σ and monitor m, written $\mathsf{traces}_{|||}(\sigma, m)$ is defined as follows:

$$\mathsf{traces}_{|||}(\sigma, m) = \{w \mid \exists(\sigma', w', m') \cdot (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_{|||} (\sigma', w', m')\}$$

Example 2 Taking the same example as before, upon each step of the system an event is added to the buffer — if the system starts with an event b: $(P, \varepsilon, A) \xrightarrow{b}_{|||} (P', b, A)$. Subsequently, the system may either continue further, or the monitor can consume the event from the buffer and fail: $(P', b, A) \xrightarrow{\tau}_{|||} (P', \varepsilon, \otimes)$. At this stage the system can still progress further until it is stopped by the rule AsyncErr:

$$(P, \varepsilon, A) \xrightarrow{b}_{|||} (P', b, A) \xrightarrow{\tau}_{|||} (P', \varepsilon, \otimes) \xrightarrow{b}_{C} (P'', b, \otimes) \xrightarrow{\tau}_{|||} (\odot, b, \otimes)$$

Proposition 3 *The system can always proceed independently when asynchronously monitored, adding events to the buffer, while the monitor can also proceed independently, consuming events from the buffer: (i) if* $\sigma \overset{w}{\Longrightarrow}_{sys} \sigma'$, *then* $(\sigma, w', m) \overset{w}{\Longrightarrow}_{|||} (\sigma', w'w, m)$*; and (ii) if* $m \overset{w}{\Longrightarrow}_{mon} m'$, *then* $(\sigma, ww', m) \overset{\tau^*}{\Longrightarrow}_{|||} (\sigma, w', m')$.

3.2 Compensation-Aware Monitoring

The main problem with asynchronous monitoring is that the system can proceed beyond an anomaly before the monitor detects the problem and stops the system. We enrich asynchronous monitoring with compensation handling so as to 'undo' actions which the system has performed after an error is detected.

Definition 7 Compensation-aware monitoring semantics $\longrightarrow_C$ are identical to asynchronous monitoring rules, but include an additional rule, Comp, which performs a compensating action for each action still lying in the buffer once the monitor detects an anomaly. The rule is shown in Fig. 1.

The set of traces generated through the compensation-aware composition of system σ and monitor m, written $\mathsf{traces}_C(\sigma, m)$, is defined as follows:

$$\mathsf{traces}_C(\sigma, m) = \{w \mid \exists(\sigma', m') \cdot (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_{C} (\sigma', \varepsilon, m')\}$$

Sane, suspended and faulty states are defined as in asynchronous monitoring.

Synchronous Monitoring

$$\textsc{Sync}\ \frac{\sigma \xrightarrow{a}_{sys} \sigma',\ m \xrightarrow{a}_{mon} m'}{(\sigma,m) \xrightarrow{a}_{\|} (\sigma',m')}\ m' \neq \otimes \qquad \textsc{SyncErr}\ \frac{\sigma \xrightarrow{a}_{sys} \sigma',\ m \xrightarrow{a}_{mon} \otimes}{(\sigma,m) \xrightarrow{a}_{\|} (\odot,\otimes)}$$

Asynchronous Monitoring

$$\textsc{Async}_S\ \frac{\sigma \xrightarrow{a}_{sys} \sigma'}{(\sigma,w,m) \xrightarrow{a}_{|||} (\sigma',wa,m)} \qquad \textsc{Async}_M\ \frac{m \xrightarrow{a}_{mon} m'}{(\sigma,aw,m) \xrightarrow{\tau}_{|||} (\sigma,w,m')}$$

$$\textsc{AsyncErr}\ \frac{}{(\sigma,w,\otimes) \xrightarrow{\tau}_{|||} (\odot,w,\otimes)}\ \sigma \neq \odot$$

Compensation-Aware Monitoring

$$\textsc{Comp}\ \frac{}{(\odot,wa,\otimes) \xrightarrow{\bar{a}}_C (\odot,w,\otimes)}$$

Adaptive Monitoring

$$\textsc{ReSync}\ \frac{}{(\sigma,\varepsilon,m) \xrightarrow{\tau}_A (\sigma,m)} \qquad \textsc{DeSync}\ \frac{}{(\sigma,m) \xrightarrow{\tau}_A (\sigma,\varepsilon,m)}$$

Fig. 1 Semantics of different monitoring schemas

Example 3 Consider the previous example with:

$(P,\varepsilon,A) \xrightarrow{b}_C (P',b,A) \xrightarrow{b}_C (P'',bb,A) \xrightarrow{\tau}_C (P'',b,\otimes) \xrightarrow{a}_C (P''',ba,\otimes) \xrightarrow{\tau}_C (\odot,ba,\otimes)$

At this stage, compensation actions are executed for the actions remaining in the buffer in reverse order:

$$(\odot,ba,\otimes) \xrightarrow{\bar{a}}_C (\odot,b,\otimes) \xrightarrow{\bar{b}}_C (\odot,\varepsilon,\otimes)$$

Proposition 4 *States reachable (under synchronous, asynchronous and compensation-aware monitoring) from a sane state are themselves sane. Similarly, for suspended and faulty states.*

Strings accepted by compensation-aware monitoring can be shown to follow a regular pattern.

Lemma 1 *For an unsuspended state (σ,ε,m), if $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\odot,v,\otimes)$, then there exist some $w_1,w_2 \in \Sigma^*$ such that the following three properties hold: (i) $w =_\tau w_1 v w_2 \overline{w}_2$; (ii) $m \overset{w_1}{\Longrightarrow}_{mon} \otimes$; (iii) $\exists \sigma'' \cdot \sigma \overset{w_1 v w_2}{\Longrightarrow}_{sys} \sigma''$.*

Similarly, for an unsuspended state (σ,ε,m), if $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\sigma',v,m')$ (with $\sigma' \neq \odot$), then there exists $w_1 \in \Sigma^*$ such that the following three properties hold: (i) $w =_\tau w_1 v$; (ii) $m \overset{w_1}{\Longrightarrow}_{mon} m'$; (iii) $\sigma \overset{w_1 v}{\Longrightarrow}_{sys} \sigma'$.

Proof The proof of the lemma is by induction on string w.
For the base case, with $w = \varepsilon$, we consider the two possible cases separately:

- *Given that* $(\sigma,\varepsilon,m) \overset{\varepsilon}{\Longrightarrow}_C (\odot,v,\otimes)$*, it follows immediately that* $\sigma = \odot$*,* $v = \varepsilon$ *and* $m = \otimes$*, and all three statements follow immediately.*
- *Alternatively, if* $(\sigma,\varepsilon,m) \overset{\varepsilon}{\Longrightarrow}_C (\sigma',v,m')$*, it follows immediately that* $\sigma = \sigma'$*,* $v = \varepsilon$ *and* $m = m'$*. By taking* $w_1 = \varepsilon$*, all three statements follow immediately.*

Assume the property holds for a string w, we proceed to prove that it holds for a string wa. By analysis of the transition rules, there are four possible ways in which the final transition can be produced:

(a) Using the rule AsyncErr*:* $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\sigma',v,\otimes) \overset{\tau}{\longrightarrow}_C (\odot,v,\otimes)$*.*
(b) Using the rule CompB*:* $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\odot,va,\otimes) \overset{\overline{a}}{\longrightarrow}_C (\odot,v,\otimes)$*.*
(c) Using the rule Async_S*:* $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\sigma'',v,m') \overset{a}{\longrightarrow}_C (\sigma',va,m')$*.*
(d) Using the rule Async_M*:* $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\sigma',av,m'') \overset{\tau}{\longrightarrow}_C (\sigma',v,m')$*.*

The proofs of the four possibilities proceed similarly:

Possibility (a):

$$(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\sigma',v,\otimes) \overset{\tau}{\longrightarrow}_C (\odot,v,\otimes)$$

By the inductive hypothesis, it follows that there exists w'_1 *such that (i)* $w =_\tau w'_1 v$*; (ii)* $m \overset{w'_1}{\Longrightarrow}_{mon} \otimes$*; (iii)* $\sigma \overset{w'_1 v}{\Longrightarrow}_{sys} \sigma'$*.*
We require to prove that there exist w_1 *and* w_2 *such that: (i)* $w\tau =_\tau w_1 v w_2 \overline{w}'_2$*; (ii)* $m \overset{w_1}{\Longrightarrow}_{mon} \otimes$*; (iii)* $\exists \sigma'' \cdot \sigma \overset{w_1 v w_2}{\Longrightarrow}_{sys} \sigma''$*.*
Taking $w_1 = w'_1$ *and* $w_2 = \varepsilon$*, statement (i) can be proved as follows:*

$$\begin{array}{ll} & w\tau \\ =_\tau & \{\ \textit{by statement (i) of the inductive hypothesis and } =_\tau\ \} \\ & w'_1 v \\ = & \{\ \textit{by definition of compensation of strings}\ \} \\ & w'_1 v \varepsilon \overline{\varepsilon} \\ = & \{\ \textit{by choice of } w_1 \textit{ and } w_2\ \} \\ & w_1 v w_2 \overline{w}_2 \end{array}$$

Statement (ii) follows immediately from statement (ii) of the inductive hypothesis and the fact that $w_1 = w'_1$*. Similarly, from statement (iii) of the inductive hypothesis,* $\sigma \overset{w'_1 v}{\Longrightarrow}_{sys} \sigma'$*, it follows by definition of* w_1 *and* w_2*, that* $\sigma \overset{w_1 v w_2}{\Longrightarrow}_{sys} \sigma''$*.*

Possibility (b):

$$(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_C (\odot,va,\otimes) \overset{\overline{a}}{\longrightarrow}_C (\odot,v,\otimes)$$

By the inductive hypothesis, it follows that there exist w'_1 *and* w'_2 *such that (i)* $w =_\tau w'_1 v a w'_2 \overline{w}'_2$*; (ii)* $m \overset{w'_1}{\Longrightarrow}_{mon} \otimes$*; (iii)* $\exists \sigma'' \cdot \sigma \overset{w'_1 v a w'_2}{\Longrightarrow}_{sys} \sigma''$*.*
We require to prove that there exist w_1 *and* w_2 *such that: (i)* $w\overline{a} =_\tau w_1 v w_2 \overline{w}_2$*; (ii)* $m \overset{w_1}{\Longrightarrow}_{mon} \otimes$*; (iii)* $\exists \sigma'' \cdot \sigma \overset{w_1 v a w_2}{\Longrightarrow}_{sys} \sigma''$*.*
Taking $w_1 = w'_1$ *and* $w_2 = a w'_2$*, statement (i) can be proved as follows:*

$$
\begin{array}{ll}
 & w\overline{a} \\
=_\tau & \{\ \textit{by statement (i) of the inductive hypothesis}\ \} \\
 & w'_1 v a w'_2 \overline{w'_2}\,\overline{a} \\
= & \{\ \textit{by definition of compensation of strings}\ \} \\
 & w'_1 v a w'_2 \overline{a w'_2} \\
= & \{\ \textit{by choice of } w_1 \textit{ and } w_2\ \} \\
 & w_1 v w_2 \overline{w}_2
\end{array}
$$

Statement (ii) follows immediately from statement (ii) of the inductive hypothesis and the fact that $w_1 = w'_1$. *Similarly, from statement (iii) of the inductive hypothesis,* $\sigma \overset{w'_1 v a w'_2}{\Longrightarrow}_{sys} \sigma'$, *it follows by definition of* w_1 *and* w_2, *that* $\sigma \overset{w_1 v w_2}{\Longrightarrow}_{sys} \sigma'$.

Possibility (c):

$$(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma'', v, m') \xrightarrow{a}_C (\sigma', va, m')$$

By the inductive hypothesis, it follows that there exist w'_1 *such that (i)* $w =_\tau w'_1 v$; *(ii)* $m \overset{w'_1}{\Longrightarrow}_{mon} m'$; *(iii)* $\sigma \overset{w'_1 v}{\Longrightarrow}_{sys} \sigma''$.

We require to prove that there exist w_1 *such that: (i)* $wa =_\tau w_1 va$; *(ii)* $m \overset{w_1}{\Longrightarrow}_{mon} m'$; *(iii)* $\sigma \overset{w_1 va}{\Longrightarrow}_{sys} \sigma'$.

Taking $w_1 = w'_1$, *statement (i) can be proved as follows:*

$$
\begin{array}{ll}
 & wa \\
=_\tau & \{\ \textit{by statement (i) of the inductive hypothesis}\ \} \\
 & w'_1 va \\
= & \{\ \textit{by choice of } w_1\ \} \\
 & w_1 va
\end{array}
$$

Statement (ii) follows immediately from statement (ii) of the inductive hypothesis and the fact that $w_1 = w'_1$. *Similarly, from statement (iii) of the inductive hypothesis,* $\sigma \overset{w'_1 v}{\Longrightarrow}_{sys} \sigma''$, *it follows by definition of* w_1 *and the application of rule* $\textsc{Async}_S$, *that* $\sigma \overset{w_1 va}{\Longrightarrow}_{sys} \sigma'$.

Possibility (d):

$$(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', av, m'') \xrightarrow{\tau}_C (\sigma', v, m')$$

By the inductive hypothesis, it follows that there exists w'_1 *such that (i)* $w =_\tau w'_1 av$; *(ii)* $m \overset{w'_1}{\Longrightarrow}_{mon} m''$; *(iii)* $\sigma \overset{w'_1 av}{\Longrightarrow}_{sys} \sigma'$.

We require to prove that there exist w_1 *such that: (i)* $w\tau =_\tau w_1 v$; *(ii)* $m \overset{w_1}{\Longrightarrow}_{mon} m'$; *(iii)* $\sigma \overset{w_1 v}{\Longrightarrow}_{sys} \sigma'$.

Taking $w_1 = w'_1 a$, *statement (i) can be proved as follows:*

$$
\begin{array}{ll}
 & w\tau \\
=_\tau & \{\ \textit{by statement (i) of the inductive hypothesis}\ \} \\
 & w'_1 av \\
= & \{\ \textit{by choice of } w_1\ \} \\
 & w_1 v
\end{array}
$$

Statement (ii) follows from statement (ii) of the inductive hypothesis, the application of rule $\textsc{Async}_M$, *and the fact that* $w_1 = w'_1 a$.

Statement (iii) follows immediately from statement (iii) of the inductive hypothesis, $\sigma \overset{w_1' a v}{\Longrightarrow}_{sys} \sigma'$*, and the fact that* $w_1 = w_1' a$*.*

□

We can now prove that synchronous monitoring is equivalent to compensation-aware monitoring with perfect compensations. This result ensures the sanity of compensation triggering as defined in the semantics.

Theorem 1 *Given a sane system and monitor pair* (σ, m)*, the set of traces produced by synchronous monitoring is cancellation-equivalent to the set of traces produced through compensation-aware monitoring:* $\mathsf{traces}_{\|}(\sigma, m) =_c \mathsf{traces}_C(\sigma, m)$*.*

Proof To prove that $\mathsf{traces}_{\|}(\sigma, m) \subseteq_c \mathsf{traces}_C(\sigma, m)$*, we note that every synchronous transition* $(\sigma', m') \xrightarrow{a}_{\|} (\sigma'', m'')$*, can be emulated in two or three steps by the compensation-aware transitions (three are required when the monitor fails)* $(\sigma', v, m') \overset{a\tau^*}{\Longrightarrow}_C (\sigma'', v, m'')$*, leaving the buffer intact. Using this fact, and induction on string w, one can show that if* $(\sigma, m) \overset{w}{\Longrightarrow}_{\|} (\sigma', m')$*, then* $(\sigma, \varepsilon, m) \overset{v}{\Longrightarrow}_C (\sigma', \varepsilon, m')$*, with* $w = v^{-\tau}$*. Hence,* $\mathsf{traces}_{\|}(\sigma, m) \subseteq_c \mathsf{traces}_C(\sigma, m)$*.*
Proving it in the opposite direction ($\mathsf{traces}_C(\sigma, m) \subseteq_c \mathsf{traces}_{\|}(\sigma, m)$*) is more intricate.*
By definition, if $w \in \mathsf{traces}_C(\sigma, m)$*, then* $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', \varepsilon, m')$*. We separately consider the two cases of (i)* $\sigma' = \odot$ *and (ii)* $\sigma' \neq \odot$*.*

– *When the final state is suspended (*$\sigma' = \odot$*):*

$$
\begin{array}{ll}
 & (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\odot, \varepsilon, m') \\
\Longrightarrow & \{ \textit{ by sanity of initial state and proposition 4 } \} \\
 & (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\odot, \varepsilon, \otimes) \\
\Longrightarrow & \{ \textit{ by lemma 1 } \} \\
 & \exists w_1, w_2 \cdot w =_\tau w_1 w_2 \overline{w}_2 \wedge m \overset{w_1}{\Longrightarrow}_{mon} \otimes' \wedge \exists \sigma'' \cdot \sigma \overset{w_1}{\Longrightarrow}_{sys} \sigma'' \\
\Longrightarrow & \{ \textit{ by proposition 2 } \} \\
 & \exists w_1, w_2 \cdot w =_\tau w_1 w_2 \overline{w}_2 \wedge \exists \sigma'' \cdot (\sigma, m) \overset{w_1}{\Longrightarrow}_{\|} (\sigma'', \otimes) \\
\Longrightarrow & \{ \textit{ by definition of } \mathsf{traces}_{\|} \} \\
 & \exists w_1, w_2 \cdot w =_\tau w_1 w_2 \overline{w}_2 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m) \\
\Longrightarrow & \{ \textit{ by proposition 1 } \} \\
 & \exists w_1 \cdot w =_c w_1 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m)
\end{array}
$$

– *When the final state is not suspended (*$\sigma' \neq \odot$*):*

$$
\begin{array}{ll}
 & (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', \varepsilon, m') \\
\Longrightarrow & \{ \textit{ by lemma 1 } \} \\
 & \exists w_1 \cdot w =_\tau w_1 \wedge m \overset{w_1}{\Longrightarrow}_{mon} m' \wedge \sigma \overset{w_1}{\Longrightarrow}_{sys} \sigma' \\
\Longrightarrow & \{ \textit{ by proposition 2 } \} \\
 & \exists w_1 \cdot w =_\tau w_1 \wedge (\sigma, m) \overset{w_1}{\Longrightarrow}_{\|} (\sigma', m') \\
\Longrightarrow & \{ \textit{ by definition of } \mathsf{traces}_{\|} \} \\
 & \exists w_1 \cdot w =_\tau w_1 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m) \\
\Longrightarrow & \{ \textit{ by the alphabet of synchronous monitoring } \} \\
 & \exists w_1 \cdot w =_c w_1 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m)
\end{array}
$$

Hence, in both cases it follows that:

$$w \in \mathit{traces}_C(\sigma, m) \implies \exists w_1 \cdot w =_c w_1 \wedge w_1 \in \mathit{traces}_{\|}(\sigma, m)$$

From which we can conclude that:

$$\mathit{traces}_C(\sigma, m) \subseteq_c \mathit{traces}_{\|}(\sigma, m)$$

□

3.3 Desynchronisation and Resynchronisation

Despite compensation-awareness, in some systems it may be desirable to run monitoring synchronously with the system for operations considered risky, only to desynchronise the system from the monitor again once control leaves the risky operation. In this section, we investigate a monitoring strategy which can run both synchronously or asynchronously in a non-deterministic manner. Any heuristic used to decide when to switch between modes corresponds to a refinement of this approach.

Definition 8 The adaptive monitoring of a system, is defined in terms of the synchronous and asynchronous monitoring rules and two additional ones (given in Fig. 1). Rule ReSync allows the system to synchronise once the buffer is empty, while rule DeSync allows the monitor to be released asynchronously. By also including the compensation rule Comp, we obtain adaptive compensation-aware monitoring ($\longrightarrow_{AC}$).

The set of traces generated through the adaptive composition of system σ and monitor m, written $\mathsf{traces}_A(\sigma, m)$, is defined as follows:

$$\mathsf{traces}_A(\sigma, m) \stackrel{\text{def}}{=} \{w \mid \exists \sigma', w', m' \cdot (\sigma, m) \stackrel{w}{\Longrightarrow}_A (\sigma', w', m') \vee (\sigma, m) \stackrel{w}{\Longrightarrow}_A (\sigma', m')\}$$

The traces for compensation-aware adaptive composition $\mathsf{traces}_{AC}(\sigma, m)$ can be similarly defined.

Theorem 2 *Asynchronous and adaptive monitoring are observationally indistinguishable:* $\mathit{traces}_A(\sigma, m) =_\tau \mathit{traces}_{|||}(\sigma, m)$.

Proof *Proving that* $\mathit{traces}_{|||}(\sigma, m) \subseteq_\tau \mathit{traces}_A(\sigma, m)$ *is trivial since all the rules which can be used to generate traces in* $\mathit{traces}_{|||}(\sigma, m)$ *are also available for traces in* $\mathit{traces}_A(\sigma, m)$.

Proving that $\mathit{traces}_A(\sigma, m) \subseteq_\tau \mathit{traces}_{|||}(\sigma, m)$ *is also easy and can be done by showing that both* ReSync *and* DeSync *do not affect traces. In fact both rules either introduce or consume an empty buffer while adding a* τ *to the trace — all actions which clearly leave no effect on traces.*

□

Theorem 3 *Compensation-aware adaptive monitoring is also indistinguishable from compensation-aware monitoring up to traces:* $\mathit{traces}_{AC}(\sigma, m) =_\tau \mathit{traces}_C(\sigma, m)$.

Proof *The proof is similar to that of the previous theorem.* □

An immediate corollary of these results is that compensation-aware adaptive monitoring is cancellation-equivalent to synchronous monitoring.

It is important to note that the results hold about trace equivalence. In the case of adaptive monitoring, we are increasing the set of diverging configurations since every state can diverge through repeatedly desynchronising and resynchronising. One would be required to enforce fairness constraints on desynchronising and resynchronising rules to ensure achieving progress in the monitored systems.

4 Compensation Scopes

The compensations we have used till now in the paper undo all extra actions taken *after* an error has occurred. To avoid additional complexity arising from compensation resolution, we assumed that each action has a unique compensation, independent of its context. Two issues arise from this limitation: (i) an action may be used in different ways, thus necessitating different compensations in different parts of the program e.g. a transfer of funds can be either a withdrawal or a deposit, for which different charges would apply when undoing; and (ii) an action may have different compensations, depending on what occurred before or after the action e.g. if an account has been closed after a transfer, then its compensation should not attempt to transfer back the funds. The first issue can be handled by our framework by viewing the different uses as different actions with different compensations. The second is much more involved — to solve the problem in general, and allowing the user to program context-sensitive compensations incurs a substantial increase in program complexity and hence the possibility of errors. However, one scenario frequently occurring in compensations is that actions which form part of a transaction become locked and impossible to compensate for once it is closed. For example, an online order may be split into a series of transfers of funds between accounts involving the buyer, the seller, the courier company and possibly different banks. Failure during the transaction should lead to the previous actions to be undone. However, once the full order is processed, none of the subparts of the transaction should be undone. To address this issue, we develop an extension of compensation-aware monitoring to handle compensation-scoping.

To handle compensation scopes, we will allow the system to perform two special actions: $\blacktriangleleft$ to open a scope, and $\blacktriangleright$ to close the most recently opened one. These two symbols will be considered as part of the alphabet Σ and we will assume that the system will always produce proper scope markers — at no point will it have produced more $\blacktriangleright$ than $\blacktriangleleft$.

Definition 9 We say that a string over such an alphabet including scope markers is well-scoped if every prefix has no more close scope markers than open scope ones. A string s is said to be balanced, written $balanced(s)$, if it contains an equal number of open and close scope markers, and all prefixes are well-scoped.

Example 4 To illustrate the use of scopes with compensation-aware monitoring, we look at different system traces with errors captured on prefixes by the monitor, indicating the expected behaviour of the recovery mechanism upon error discovery:

1. If the system performed $ab \blacktriangleleft cd \blacktriangleright e$ (with each single letter indicating an action) by the time the monitor discovered a problem after executing action a, the compensation mechanism must compensate for $b \blacktriangleleft cd \blacktriangleright e$. However, the scope $\blacktriangleleft cd \blacktriangleright$ cannot be undone, meaning that we will compensate by performing $\overline{e}\overline{b}$.
2. If the system has, however, performed $ab \blacktriangleleft cd$ by the time the monitor discovered a problem after executing action a, the compensation mechanism will compensate for $b \blacktriangleleft cd$ by performing $\overline{d}\overline{c}\overline{b}$.
3. To look at the use of subscopes, if the system's behaviour at the point in time when the monitor discovers an error is $ab \blacktriangleleft cd \blacktriangleleft ef \blacktriangleright g$, the behaviour within the subscope $\blacktriangleleft ef \blacktriangleright$ will not be compensated for since the context is closed. However, the outer scope, which is not yet closed, will allow for compensation of actions a, b, c, d and g, depending on the point of the tract where the error is discovered.
4. Now consider a prefix trace $ab \blacktriangleleft cd \blacktriangleright ef \blacktriangleleft g$ of the system's behaviour the moment an error is discovered by the monitor after consuming $ab \blacktriangleleft c$. The actions left in the buffer

which have to be compensated for are $d \blacktriangleright e\,f \blacktriangleleft g$. Since the second scope has not been closed, we will compensate for $e\,f\,g$ by performing $\overline{g}\,\overline{f}\,\overline{e}$. Should action d be compensated for? If we compensate by executing $\overline{d}$, to reverse the system to the point of error we may run into problems since the scope closure indicates that resources may no longer be available. The compensation for d, should thus not be triggered, since it appeared within a closed scope.

Definition 10 Given a trace of actions t, we define $strip(t)$ to be the same trace but removing away all actions occuring within a scope and any remaining open scope markers. We define $strip(w)$ to be the shortest string for which there are no further reductions of the form $strip(w_1 \blacktriangleleft w \blacktriangleright w_2) = strip(w_1 w_2)$ (where $\blacktriangleleft$ and $\blacktriangleright$ do not appear in w), $strip(w_1 \blacktriangleleft w_2) = strip(w_1 w_2)$ (where $\blacktriangleright$ does not appear in w_2) and $strip(w_1 \blacktriangleright w_2) = strip(w_2)$ (where $\blacktriangleleft$ does not appear in w_1).

Strings w and w' are said to be *scope-cancellation-equivalent*, written $w =_{sc} w'$, if they reduce via compensation cancellation and scope stripping to the same string: $strip(w) =_c strip(w')$. As before, we define what it means for a set of strings to be *included in set W' up-to-scope-cancellation*, written $W \subseteq_{sc} W'$, and set *equality up-to-scope-cancellation*, written $W =_{sc} W'$.

Scope stripping is well-defined and cancels with compensation:

Proposition 5 *Scope stripping* $\underline{\text{strip}}$ *is a well-defined function over the domain of well-scoped strings. Furthermore,* $w\,\underline{\text{strip}}(w) =_{sc} \varepsilon$.

To handle scopes in compensations, we adopt the addition of three rules to the compensation-aware monitoring semantics. In the first two cases, whenever a scope closure $\blacktriangleright$ is found in the buffer, the whole scope is removed before proceeding (considering separately whether or not the scope was opened before the error was discovered):

$$\textsc{CloseScope}_M \ \frac{}{(\odot, w, \otimes) \xrightarrow{\tau}_{SC} (\odot, w', \otimes)}\ w = w' \blacktriangleleft w'' \blacktriangleright \text{ WITH } balanced(w'')$$

$$\textsc{CloseScope} \ \frac{}{(\odot, w, \otimes) \xrightarrow{\tau}_{SC} (\odot, \varepsilon, \otimes)}\ w = w' \blacktriangleright, \blacktriangleleft \text{ DOES NOT APPEAR IN } w'$$

Whenever a scope open symbol $\blacktriangleleft$ is found, it is simply discarded, since it represents a scope which was opened but not closed by the time the error was identified:

$$\textsc{CompOpenScope} \ \frac{}{(\odot, w \blacktriangleleft, \otimes) \xrightarrow{\tau}_{SC} (\odot, w, \otimes)}$$

The state sanity preservation result of proposition 4, also holds for scope compensation-aware monitoring.

Proposition 6 *States reachable through scope compensation-aware monitoring from a sane state are themselves sane. Similarly, for suspended and faulty states.*

To prove that the modified system is still correct, we need a stronger version of lemma 1, which caters for complete contexts which will be discarded when compensations are triggered:

Lemma 2 *For an unsuspended state (σ,ε,m), if $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{SC} (\odot,v,\otimes)$, then there exist some $w_1, w_2 \in \Sigma^*$ such that the following three properties hold: (i) $w =_\tau w_1 v w_2 \overline{\text{strip}(w_2)}$; (ii) $m \overset{w_1}{\Longrightarrow}_{mon} \otimes$; (iii) $\exists \sigma'' \cdot \sigma \overset{w_1 v w_2}{\Longrightarrow}_{sys} \sigma''$.*

Similarly, for an unsuspended state (σ,ε,m), if $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{C} (\sigma',v,m')$ (with $\sigma' \neq \odot$), then there exists $w_1 \in \Sigma^*$ such that the following three properties hold: (i) $w =_\tau w_1 v$; (ii) $m \overset{w_1}{\Longrightarrow}_{mon} m'$; (iii) $\sigma \overset{w_1 v}{\Longrightarrow}_{sys} \sigma'$.

□

The proof of lemma 2 is almost identical to that of lemma 1, but taking into account the additional buffer reduction rules. This allows us to prove the stronger theorem stating the correctness of scoped compensation-aware monitoring:

Theorem 4 *Synchronous and compensation-aware monitoring with scopes behave in equivalent manner. Given a sane system and monitor pair (σ,m):*

(i) A trace accepted by synchronous monitoring is also accepted by compensation-aware monitoring with scopes: $\mathsf{traces}_{\|}(\sigma,m) \subseteq_{sc} \mathsf{traces}_{SC}(\sigma,m)$.

(ii) A trace accepted by compensation-aware monitoring with scopes can be split into two parts, the first of which is accepted by synchronous monitoring, and the second of which is cancellation-equivalent to doing nothing:
$w \in \mathsf{traces}_{SC}(\sigma,m) \implies \exists w_1,\ w_2 \cdot w = w_1 w_2 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma,m) \wedge w_2 =_{sc} \varepsilon$

Proof The correctness of the first part (i) follows from the fact that every synchronous transition can be emulated by two or three scoped compensation-aware rules. This ensures forward language inclusion.

As in the proof of theorem 1, the proof of (ii) for $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{SC} (\sigma,\varepsilon,m')$ takes into consideration two cases: (a) $\sigma' = \odot$; and (b) $\sigma' \neq \odot$. Case (b) is identical to the proof of the equivalent case in theorem 1. Case (a) can be proved as follows:

$$
\begin{array}{ll}
 & (\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{SC} (\odot,\varepsilon,m') \\
\Longrightarrow & \{\ \textit{by sanity of initial state and proposition 6}\ \} \\
 & (\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{SC} (\odot,\varepsilon,\otimes) \\
\Longrightarrow & \{\ \textit{by lemma 2}\ \} \\
 & \exists w_1, w_1' \cdot w =_\tau w_1 w_1' \overline{\text{strip}(w_1')} \wedge m \overset{w_1}{\Longrightarrow}_{mon} \otimes' \wedge \exists \sigma'' \cdot \sigma \overset{w_1}{\Longrightarrow}_{sys} \sigma'' \\
\Longrightarrow & \{\ \textit{by proposition 2}\ \} \\
 & \exists w_1, w_1' \cdot w =_\tau w_1 w_1' \overline{\text{strip}(w_1')} \wedge \exists \sigma'' \cdot (\sigma,m) \overset{w_1}{\Longrightarrow}_{\|} (\sigma'',\otimes) \\
\Longrightarrow & \{\ \textit{by definition of } \mathsf{traces}_{\|}\ \} \\
 & \exists w_1, w_1' \cdot w =_\tau w_1 w_1' \overline{\text{strip}(w_1')} \wedge w_1 \in \mathsf{traces}_{\|}(\sigma,m) \\
\Longrightarrow & \{\ \textit{adding variable } w_2 = w_1' \overline{\text{strip}(w_1)}\ \} \\
 & \exists w_1, w_1', w_2 \cdot w =_\tau w_1 w_2 \wedge w_2 = w_1' \overline{\text{strip}(w_1')} \wedge w_1 \in \mathsf{traces}_{\|}(\sigma,m) \\
\Longrightarrow & \{\ \textit{proposition 5}\ \} \\
 & \exists w_1, w_2 \cdot w = w_1 w_2 \wedge w_2 =_{sc} \varepsilon \wedge w_1 \in \mathsf{traces}_{\|}(\sigma,m)
\end{array}
$$

This completes the proof.

□

Using this result, we can show that scoping still keeps monitoring correct up to ignoring of scope content and compensations. The semantics given to scope compensation monitoring gather the scopes in the buffer and only discard them while emptying the buffer. The

advantage of this approach, is that the decision of how to handle scopes is left until the compensation triggering phase. Alternatively, one could have discarded actions in the buffer as soon as the system closes a scope, which is less flexible, but may result in smaller buffers being used.

5 A Compensation-Aware Monitoring Architecture

LARVA [11] is a synchronous runtime verification architecture supporting DATEs [10] as a specification language. A user wishing to monitor a system using LARVA must supply a system (a Java program) and a set of specifications in the form of a LARVA script — a textual representation of DATEs. Using the LARVA compiler, the specification is transformed into the equivalent monitoring code together with a number of aspects which extract events from the system. Aspects are generated in AspectJ, an aspect-oriented implementation for Java, enabling automatic code injection without directly altering the actual code of the system. When a system is monitored by LARVA generated code, the system waits for the monitor before continuing further execution.

We propose an asynchronous compensation-aware monitoring architecture and implementation, cLARVA, with a controlled synchronous element. In cLARVA, control is continually under the jurisdiction of the system — never of the monitor. However, the system exposes two interfaces to the monitor: (i) an interface for the monitor to communicate the fact that a problem has been detected and the system should stop; and (ii) an interface for the monitor to indicate which actions should be compensated. Note that these correspond precisely to rules ASYNCERR and COMP respectively. Furthermore, the actual time of stopping and how the indicated actions are compensated for are decisions left up to the system.

Fig. 2 shows the four components of cLARVA and the communication links between them. The monitor receives system events through the events player from the log, while the system can continue unhindered. If the monitor detects a fault, it communicates with the system so that the latter stops. Depending on the actions the system carried out since the actual occurrence of the fault, the monitor indicates the actions to be compensated for.

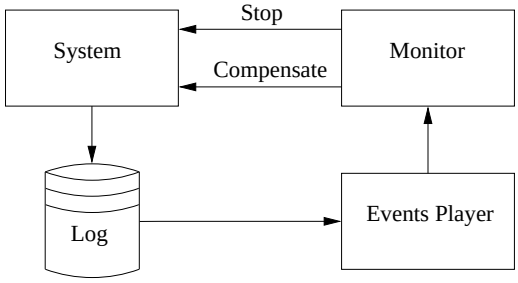

Fig. 2 The asynchronous architecture with compensations cLARVA.

To support switching between synchronous and asynchronous monitoring, a *synchronisation manager* component is added as shown in Fig. 3. All connectors in the diagram are synchronous with the system not proceeding after relaying an event until it receives control from the manager. The following shows the logic of the synchronisation manager:

```
c = proceed                ; set default control to proceed
```

```
while (c != stop)
  if (monitoring_mode == SYNC)
    e = in_event()          ; read event from system
    c = out_event(e)        ; forward to monitor and get its resulting state
    out_control(c)          ; relay control to system
  else
    par                     ; parallel execution
      e1 = in_event()       ; read from system
      addToBuffer(e1)       ; store in buffer
      out_control(c)        ; return control to system
    with
      e2 = readFromBuffer() ; read from buffer
      c = out_event(e2)     ; forward to monitor and get its resulting state
end
```

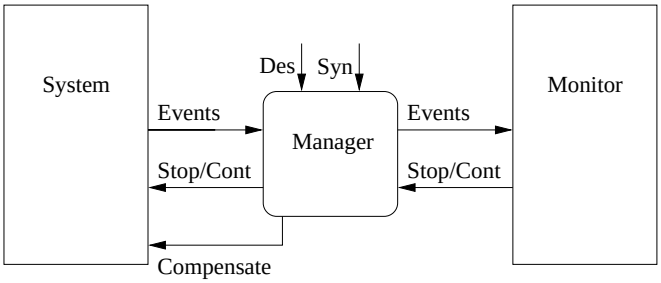

Fig. 3 The asynchronous architecture with synchronisation and desynchronisation controls.

In real-life scenarios it is usually undesirable to stop a whole system if an error is found. However, in many cases it is not difficult to delineate components of the system to ensure that only the relevant parts of the system are stopped. For example, when a transaction is carried out without necessary rights, it should be stopped and compensated for. Similarly, if a user has managed to illegally login and start a session, then only user operations during that session should be stopped and compensated for.

This approach of system decomposition into relatively independent parts can be extended further to simultaneously allow synchronous and asynchronous monitoring. This is further discussed in the next section.

6 Extending the Architecture with Automatic Synchronisation and Desynchronisation Heuristics

Synchronisation guarantees immediate identification and possible reparation of problems, making it desirable for parts of the system where higher dependability is required. For instance, if a particular transaction is considered high-risk, it would be desirable to synchronise monitoring during the transaction, only to desynchronise once again when a less risky part of the system is reached. Having an architecture which allows switching between synchronous and asynchronous modes of monitoring requires a mechanism to appropriately select the active mode. Although the switching between synchronous and asynchronous monitoring can be done manually, it is much more useful to have an automatic mechanism which handles this feature.

The issue is how to assess risk associated with particular states and actions, thus ensuring that high risk actions are always monitored synchronously. There are various ways in which

this can be achieved: (i) keep track of the activities of each user and use pattern matching and statistics to deduce the risks associated with individual users; or (ii) classify transactions according to the risk they involve, e.g. a transfer between a user's own accounts might be considered as safe but spending a large sum of money might not. If a transaction has an associated high risk factor and/or is being carried out by a user tagged as risky, then one might decide that during this action the monitor should switch to synchronous mode.

In this section, we propose possible heuristics which can be used to automate such a mechanism, effectively determining the switching between synchronous and asynchronous monitoring. We also discuss implementation considerations and how these heuristics can be incorporated in cLarva.

6.1 Adding Heuristics to the Monitoring Architecture

An important design issue is where to decide de/synchronisation: either within the system itself, at the manager, or at the monitor. Leaving the decision up to the system has the advantage that the system would always be in control of the monitoring mode. On the other hand, this would add an overhead to the system; something which the whole architecture is meant to avoid. If heuristics are executed by the manager, both synchronised and desynchronised modes are possible. However, useful information for deciding de/synchronisation (such as user risk factor) might be already available within the monitor; making it wasteful to recalculate at the system or manager.

In our case study we opt for a monitor-side, asynchronous de/synchronisation decision where the heuristics are themselves implemented as monitors. This strategy avoids any duplication between monitors and heuristics while also avoiding the introduction of additional overheads to the system. Although this might lead to a de/synchronisation decision to be taken late, the problem is minimised by the scheduling strategy discussed in section 6.2.

Therefore, updating the architectural view of the system would involve adding a heuristics component (to the architecture shown in Fig. 3) which is in charge of executing heuristics and signalling the manager to switch between synchronous and asynchronous monitoring. Such a component requires the following connections: (i) a connection to the incoming system events — supplying the required information for executing heuristics; and (ii) connections to the de/synchronisation signals entering the synchronisation manager. The updated architecture with these modifications is shown in Fig. 4.

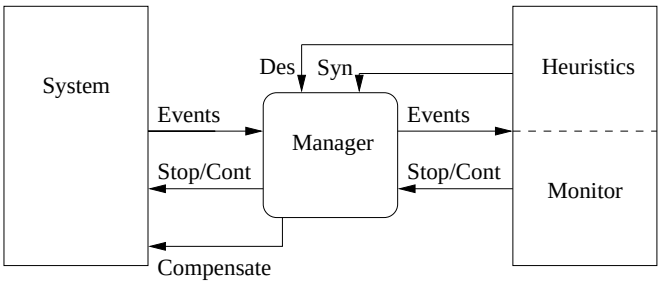

Fig. 4 The monitoring architecture with heuristics.

In practice, this approach requires that the monitoring system can handle parametrised monitoring of transactions/users. Furthermore, it requires a way of decomposing the system into independent components which would simultaneously allow synchronous and asynchronous monitoring. In the kind of financial systems we are considering (such as the case study in the next section), such components are usually transactions or users.

In fact, the monitor-side (including the manager) typically consists of multiple parametrised monitors, each with its own buffer, synchronisation manager and heuristics component. This is expanded further in the following subsection.

6.2 Monitor Demultiplexing and Scheduling

Monitors in cLarva are dynamically instantiated for each monitored object. Thus, a system's monitor is in fact composed of many sub-monitors. For example if we are monitoring a number of properties regarding a number of transactions for each logged-in user, then a monitor would be created for each property, for each transaction, for each user. For this reason, although at a high level we have shown the architecture as having one buffer, in actual fact it has a buffer for each sub-monitor as illustrated in Fig. 5.

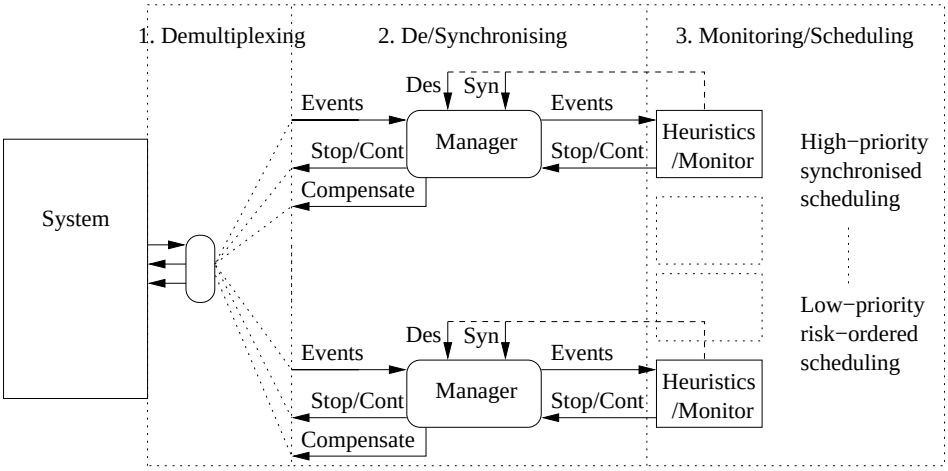

Fig. 5 The monitoring architecture with heuristics, demultiplexing, and scheduling.

To coordinate the execution of the sub-monitors, upon the receipt of a system event, the following steps are carried out:

1. The event is replicated to all the relevant sub-monitor buffers. For example, a transaction event would be copied to all buffers pertaining to sub-monitors of that particular transaction.
2. Subsequently, if the sub-monitor is in asynchronous mode, control is immediately passed back to the system. Otherwise the manager first forwards the event to the heuristics and the monitor components and waits for their response before allowing the system to proceed further.

3. Given the potentially substantial number of sub-monitors, the choice of a scheduling strategy among sub-monitors might be crucial to detect problems as early as possible. A sensible scheduling strategy would be to associate a scheduling priority according to the corresponding risk; the higher the risk, the higher the scheduling priority. Naturally, this is over and above the priority that synchronised monitors should have over asynchronous monitors; to ensure minimal disruption to the system, asynchronous monitors should only be allowed to run when no synchronous monitors are running.

Note that by following the above steps, the priority of heuristics execution is the same as that of the corresponding sub-monitor. Given that heuristics are normally based on the history of system events, heuristics can in fact be implemented as monitors as discussed in the next subsection.

6.3 Implementing Heuristics as DATEs

In practice, one expects there to be a substantial overlap between monitors and heuristics. For example to monitor for fraudulent behaviour, one would usually try to measure how risky a particular pattern of activities is. Such a measure can lend itself useful to decide for or against synchronous monitoring. In this subsection we will show how DATEs can be used for this purpose.

The following features of DATEs are particularly useful for implementing heuristics: (i) it is easy to integrate such heuristics with cLarva which is DATE-based; (ii) the compositionality of DATEs — different heuristics can be implemented as separate DATEs and then connected together to form a single DATE through channel communication; (iii) LarvaStat [8] builds upon Larva, extending DATEs to support statistical properties; and (iv) it is easy to keep track of multiple objects at a time through the dynamic mechanism which replicates monitors — one for each object being monitored.

Using these features and considering our case study (see next section) we opt to implement the following heuristics for each user: (i) the monitor uses statistics to calculate the risk factor depending on the series of activities which the user performs; and (ii) if the risk factor exceeds a particular threshold, then the monitor is forced to synchronise before the end of a scope (Note that the end of a scope can be time-based, e.g. a purchase can only be compensated within 24 hours. In such a case the end of the scope is signalled earlier so that there would be ample time for synchronisation and compensation if necessary).

Thus, we will split the implementation of the heuristics into the following parts: (i) a DATE which keeps track of whether a user is currently monitored synchronously or asynchronously; (ii) a DATE which keeps track of the risk factor of a user; and (iii) a DATE which decides whether a user should be monitored synchronously or asynchronously upon detecting a close-scope event based on the risk factor (and communicates the decision to (i)). In what follows, we give the definition of these DATEs:

1. Fig. 6 shows the main DATE which listens on two channels: *sync*, signifying that the monitor should be synchronised, and *async* to signal that the monitor need no longer remain synchronised.
   As soon as the DATE receives either of these messages (from other monitors assessing whether it would be advisable to synchronise or switch to asynchronous mode), it relays the change to the synchronisation manager. Note that since we would like to apply the heuristics on a per-user basis, the DATE has to be parametrised for each user. The limitation of this approach is that properties which span over multiple users have to be very

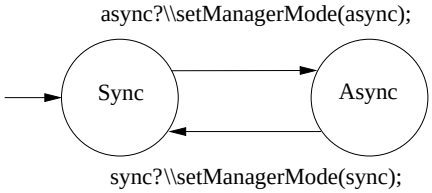

Fig. 6 The DATE which listens to other DATEs for messages to synchronise or desynchronise the monitor from the system.

carefully devised as the monitor states of different users might reflect different synchronisation levels. For this reason communication across DATEs should only occur through channel communication and not through global variables.

2. The LARVA framework has an extension to directly support the specification of statistical properties called LarvaStat [8]. One of the case studies carried out with LarvaStat involved an intrusion detection system on top of an ftp server, assessing each user, and assigning him or her a risk factor. The risk factor was calculated using two main techniques: (i) a Markov chain analysing the user's command sequence, with each ftp command being related to a risk factor, and marking the user as suspicious if the command sequence exceeds a threshold; and (ii) the use of statistical moments for the characterisation of abnormal user behaviour, monitoring each user's download and upload behaviour patterns, and assuming a statistically predictable pattern. Similar techniques can be used as heuristics which increase or decrease the perceived user risk factor: for example users who use the money for a purchase are not considered as risky, while users who load money a number of consecutive times, perform several transfers and withdraw the money are considered highly suspicious. This logic is encoded as a Markov chain shown in Fig. 7(a).

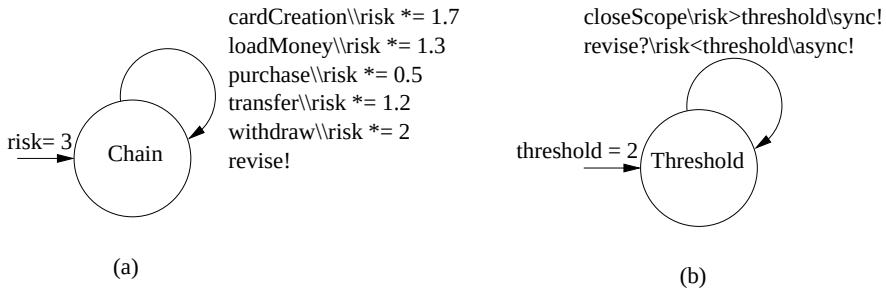

Fig. 7 The DATEs which track the user risk factor and notifies of a risk change over channel *revise* (a) and decides whether to synchronise or desynchronise (b).

3. Fig. 7(b) illustrates the DATE which would force a monitor to synchronise (by sending a signal to the main DATE (Fig. 6)) in case a closing scope action is detected and the risk factor of the corresponding user is higher than the threshold. On the other hand, when the risk goes below the threshold, the monitor is desynchronised from the system. Note that new users are considered risky and thus their risk factor is initialised to a higher value than the threshold. Such users are only considered safe after carrying out a pattern of non-risky transactions.

In what follows, we describe a real-life case study where DATE heuristics similar to the ones described above have been used to prioritise the monitoring of users with high perceived risk.

7 Case Study

We have applied cLarva on Entropay, an online prepaid payment service offered by Ixaris Systems Ltd[2]. 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.

The advantage of applying the proposed architecture to EntroPay is that the latter already incorporates compensations in its implementation. The case study is further simplified by the fact that properties are not monitored globally but rather on a per user or per credit card basis. Therefore, when a problem is found with a particular user or card, only the compensations for that particular entity need to be triggered.

The case study implementation closely follows the architecture described above with two control connections: one with an interface for stopping EntroPay with respect to a particular user and another to the compensation interface of EntroPay, through which the monitor can cause the system to execute compensations.

In what follows, we give a classification of properties which were monitored successfully and how these are compensated in case of a violation detection.

Life cycle: A lot of properties in Entropay depend on which phase of the life-cycle an entity is in. Fig. 8 is an illustration of the user life-cycle, starting with registration and activation, allowing the user to login and logout (possibly carrying out a series of operations in between), and finally, the possibility of freezing/unfreezing/deleting a user in case of inactivity.

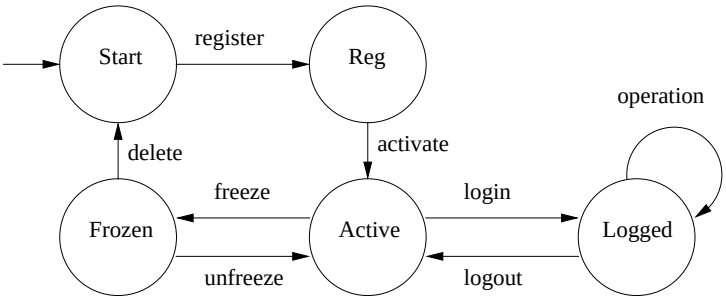

Fig. 8 The lifecycle property.

Implicitly, such a property checks that for a user to perform a particular operation and reach a particular state, the user must be in an appropriate state. If a life cycle property is violated, the user actions carried out after the violation are compensated and the user

[2] http://www.ixaris.com

state is corrected. For example, if a user did not login and managed to carry out a transfer, then as soon as the monitor detects the violation, any ongoing user operations are stopped and the illegal transfer is compensated.

Real-time: Several properties in Entropay, have a real-time element. For example, a user account which is inactive for more than six months is frozen. If freezing does not take place, then, upon detection, the monitor issues a compensation for any actions carried out after the expected freezing and freezes the user account.

Rights: User rights are a very important aspect of Entropay's security. A number of transactions require the user to have the appropriate rights before a transaction is permitted. If a transaction is carried out without the necessary rights, it is compensated.

Amounts: There are various limits (for security reasons) on the frequency of certain transactions and the total amount of money which these transactions constitute. If a user is found to have carried out more transactions than allowed, then the excess transactions are compensated. Similarly, transaction amounts which go beyond the allowed threshold are compensated for.

The case study was successfully executed on a sanitized[3] database of 300,000 users with around a million virtual cards. A number of issues have been detected through the monitoring system: (i) not all system activities were recorded consistently; (ii) some system state was found to be inconsistent, e.g. certain cards which were marked as inactive were still found to be active; (iii) in some exceptional cases, system limits did not tally with the overall balance of the transactions monitored.

Although the current properties being monitored on Entropay are relatively light-weight, due to security and performance considerations, it is not desirable to run the monitor synchronously when there is no clear evidence of specification violation. Users which pose little or no risk to the system and which have been using the system for a number of years should not suffer any service deterioration. The conciliatory approach of cLarva would guarantee added security with the cost of logging under normal execution while incurring overhead only when there is convincing evidence that something is wrong.

7.1 Real-Life Traces

To demonstrate the results of our case study, in this subsection we give three anonymised[4] system traces in which problems were discovered.

In the following traces, we assume that any transaction starting with > signifies normal behaviour while those starting with < represent the corresponding compensation. A transaction with neither symbols does not have a compensation. Furthermore, a number of transactions do not have a closing scope, meaning that they can be compensated at any time after their occurrence. These include user login and virtual credit card creation. On the other hand, for the rest of the compensable activities, the closing scope is time-bound. For simplicity we assume that the time limit always occurs one hour after the completion of the activity. This approach to scoping might seem at odds with the theory presented earlier where scopes cannot intersect (except by inclusion). However, in practice, the cLarva would have a monitor for each transaction and thus the scopes would never intersect locally.

The following are excerpts from the system log merged with monitor actions:

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

[4] Due to privacy considerations the data in certain fields cannot be exposed.

No rights issue: After encountering traces such as the one presented below, the monitor reported that some actions were carried out without the necessary rights. For example a user requires a special right to be allowed to login, to create a virtual credit card and also to load money onto the credit card. The following trace was found:

```
Timestamp:            User:     Transaction:                              Amount:
 13:00:35 5-5-2010     user1     account registration                      n/a
  ... a monitor is dynamically created for user1 with default risk factor 3 ...
 13:05:41 5-5-2010     user1    >account activation                        n/a
 13:05:45 5-5-2010     user1    >logged in                                 n/a
  ... user1 logged in without having the right ...
 13:07:10 5-5-2010     user1    >created virtual credit card               n/a
  ... risk factor for user1 increases to 5.1 ...
 13:12:06 5-5-2010     user1    >loaded money onto virtual credit card     £100
  ... risk factor for user1 increases to 6.63 ...
  ... monitor detects rights violation ...
  ... monitor initiates compensation ...
 13:12:52 5-5-2010     user1    <withdraw money from virtual credit card   £100
 13:12:05 5-5-2010     user1    <delete virtual credit card                n/a
 13:12:10 5-5-2010     user1    <logged out                                n/a
```

In this trace the monitor was run asynchronously with a high priority scheduling. Once the monitor detected the violation, the transactions which occurred after the illegal login were compensated (see last three trace entries).

Late freezing of user accounts: According to the system specification, after six months of user monetary inactivity, i.e. no transactions involving money are carried out, the user account is frozen. Nonetheless, traces such as the following one were discovered:

```
Timestamp:            User:     Transaction:                              Amount:
 15:00:38 8-6-2010     user2     account registration                      n/a
  ... a monitor is dynamically created for user2 with default risk factor 3 ...
 15:15:31 8-6-2010     user2    >account activation                        n/a
 15:15:33 8-6-2010     user2    >granted login, card creation rights       n/a
 15:15:45 8-6-2010     user2    >logged in                                 n/a
 15:35:45 8-6-2010     user2     logged out                                n/a
 18:12:14 5-9-2010     user2    >logged in                                 n/a
 18:42:55 5-9-2010     user2     logged out                                n/a
  ... by now the account of user2 should have been frozen ...
 17:52:21 8-12-2010    user2    >logged in                                 n/a
 17:55:50 8-12-2010    user2    >created virtual credit card               n/a
  ... risk factor increases to 5.1 ...
  ... monitor detects unfrozen account ...
  ... monitor initiates compensation ...
 18:00:12 8-12-2010    user2    <delete virtual credit card                n/a
 18:00:15 8-12-2010    user2    <logged out                                n/a
 18:00:20 8-12-2010    user2     account frozen                            n/a
```

In the above trace, there are two compensating activities which have been suggested by the monitor upon the time of detection (third and second trace entries from below). The last activity is the correction which is carried out as correction after the synchronisation (of the system and the monitor) is complete. Note that although the risk factor for this user was relatively high (ensuring favourable scheduling), this could not lead to synchronisation since no scope closes were encountered.

Excessive money loading to credit cards: The system's business logic imposes limits on the amount of money which can be loaded onto a virtual credit card each day, each week and each month. However, two traces similar to the following were discovered where the limit for *user3* for a day was £2000.

```
Timestamp:            User:     Transaction:                              Amount:
  ... risk factor for user3 is 1.6 ...
 11:05:15 7-10-2010    user3    >logged in                                 n/a
 11:12:16 7-10-2010    user3    >loaded money onto virtual credit card     £1000
```

```
 ... risk factor for user3 increases to 2.08 ...
11:25:44 7-10-2010     user3     logged out                                n/a
 ... monitor synchronises at 12:13:20
13:15:35 7-10-2010     user3    >logged in                                 n/a
 ... user3 attempts to load £1500 onto virtual credit card ...
 ... risk factor for user3 increases to 2.704 ...
 ... monitor detects violation and stops the activity ...
```

In this case, the risk associated with *user3* exceeded the threshold and thus upon the closing scope of the money load, the monitor was synchronised. Note that for this reason the system was immediately stopped when attempting to allow the user to load money which exceeded the limit.

Notwithstanding rigorous testing, unexpected behaviour still occurs in complex systems such as Entropay. Although the issues detected and shown in the above traces are minor issues, having monitors in place with automated compensating mechanisms has thus been shown to provide an extra security layer to the benefit of the business and the clients.

8 Related Work

In principle, any algorithm used for synchronous monitoring can be used for asynchronous monitoring as long as all the information available at runtime is still available asynchronously to the monitor through some form of buffer. The inverse, however, is not always true because monitoring algorithms such as [21] require that the complete trace is available at the time of checking. In our case, this was not an option since our monitor has to support desynchronisation and resynchronisation at any time during the processing of the trace.

There are numerous algorithms and tools [1,2,7,13,14,18,21,22] which support asynchronous monitoring — sometimes also known as trace checking or offline monitoring. A number of these tools and algorithms [1,2,7,21] support only asynchrony unlike our approach which supports both synchronous and asynchronous approaches. Furthermore, although a number of approaches [13,14,18,22] support both synchronous and asynchronous monitoring, no monitoring approach of which we are aware is able to switch between synchronous and asynchronous monitoring during a single execution.

Although the idea of using rollbacks (or perfect compensations) as a means of synchronisation might be new in the area of runtime verification, this is not the case other areas, such as distributed games [12,19,20]. The problem of distributed games is to minimise the effects on the playing experience due to network latencies. Two general approaches taken are pessimistic and optimistic synchronisation mechanisms. The former waits for all parties to be ready before anyone can progress while the latter allows each party to progress and resolve any conflicts later through rollbacks.

The problem which we have addressed in this work is a variant of the distributed game problem with two players: the system and the monitor. In a similar fashion to game synchronisation algorithms, the system rolls-back (or compensates) to revert to a state which is consistent with the monitor.

9 Conclusions and Future Work

Notwithstanding the rigorous testing which critical system undergo, problems still arise particularly due to the unpredictable environment under which such systems operate. This scenario motivates the need for monitoring such systems during normal use where the occurrence of an error might imply serious repercussions. However, the problem with monitoring

is that it adds overhead to the system which might already be under pressure during peak hours of usage. This motivates the use of asynchronous monitoring which minimises the overhead to logging system events. The problem with asynchronous monitoring is that upon the detection of a problem, it might be too late to take any corrective measures. Adaptive compensation-aware monitoring is a conciliatory approach which allows the monitoring architecture to switch between synchronous and asynchronous monitoring, using compensations to restore the system state in case an error is detected late.

In this paper we extended compensation-aware monitoring to handle scopes such that when a compensation terminates, it is no longer compensable. Furthermore, we have extended the architecture to include heuristics which are able to automatically steer the monitoring system from synchrony to asynchrony and vice-versa. As a heuristic example we have demonstrated the use of perceived user risk for switching to monitor synchrony upon scope closure detection.

A significant limitation of our work is the assumption that compensations are associated to individual actions. Apart from the fact that this might not always be the case, this approach is highly inflexible as one cannot simultaneously compensate for several actions. In the future, we aim to lift this limitation by introducing a more structured approach to compensation handling.

Another possible direction for future work is to support compensating monitoring which is not constrained to compensate only for buffered actions. This can be useful for example in the case of fraud detection where violation can be detected late even when monitoring synchronously, and in which case one may desire to undo certain actions which took place before the violation.

References

1. J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, 2003.
2. H. Barringer, A. Groce, K. Havelund, and M. Smith. An entry point for formal methods: Specification and analysis of event logs. In *Formal Methods in Aerospace (FMA)*. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2009.
3. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Principles of Programming Languages (POPL)*, pages 209–220. ACM, 2005.
4. M. J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104, 2004.
5. M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2004.
6. L. Caires, C. Ferreira, and H. T. Vieira. A process calculus analysis of compensations. In *Trustworthy Global Computing (TGC)*, volume 5474 of *Lecture Notes in Computer Science*. Springer, 2008.
7. F. Chang and J. Ren. Validating system properties exhibited in execution traces. In *Automated Software Engineering (ASE)*, pages 517–520. ACM, 2007.
8. C. Colombo, A. Gauci, and G. J. Pace. Larvastat: Monitoring of statistical properties. In *Runtime Verification (RV)*, volume 6418 of *Lecture Notes in Computer Science*, pages 480–484. Springer, 2010.
9. C. Colombo, G. J. Pace, and P. Abela. Compensation-aware runtime monitoring. In *Runtime Verification (RV)*, volume 6418 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2010.
10. C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2008.
11. C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE, 2009.
12. E. Cronin, A. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1):7–30, 2004.

13. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning (TIME)*. IEEE, 2005.
14. S. A. Ezust and G. V. Bochmann. An automatic trace analysis tool generator for estelle specifications. In *Applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 175–184. ACM, 1995.
15. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD international conference on Management of data*, pages 249–259. ACM, 1987.
16. J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.
17. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *Service-Oriented Computing (ICSOC)*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.
18. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.
19. D. Jefferson. Virtual time. In *International Conference on Parallel Processing (ICPP)*, pages 384–394. IEEE, 1983.
20. M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.
21. G. Roşu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, RIACS, 2001.
22. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
23. C. Vaz, C. Ferreira, and A. Ravara. Dynamic recovering of long running transactions. *Trustworthy Global Computing (TGC)*, 5474:201–215, 2009.