Compensation-Aware Runtime Monitoring*

Christian Colombo[1], Gordon J. Pace[1], and Patrick Abela[2]

[1] Dept. of Computer Science, University of Malta, Malta
[2] Ixaris Ltd, Malta
{christian.colombo, gordon.pace}@um.edu.mt, patrick.abela@ixaris.com

Abstract. To avoid large overheads induced by runtime monitoring, the use of asynchronous log-based monitoring is sometimes adopted — even though this implies 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 may perform unauthorized actions. Since stopping such actions is not feasible, in this paper we investigate the use of compensations to enable the undoing of actions, thus enriching asynchronous monitoring with the ability to restore the system to the original state in which the anomaly occurred. Furthermore, we show how allowing the monitor to adaptively synchronise and desynchronise with the system is also possible and report on the use of the approach on an industrial case study of a financial transaction system.

1 Introduction

The need for correctness of systems has driven research in different validation and verification techniques. One of the more attractive approaches is the use of monitors on systems to verify their correctness at runtime. The main advantage in the use of runtime verification over other approaches, is that it is a relatively lightweight approach and scales up to large systems — guaranteeing the observation of abnormal behaviour.

Even though monitoring of properties is usually computationally cheap when compared to the actual computation taking place, 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 can cripple the user-experience at peak times of execution. One approach usually 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, 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 [14] — 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. To solve the

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

problem, typically one defines *compensations*, to undo partially executed transactions if discovered to be infeasible half way 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. We 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. 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. Thus, we propose an architecture 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.

In this paper, we present a framework to enable compensation-aware monitoring — and prove that the compensation triggering mechanism works as expected, resulting in similar behaviour as though we had run the monitor synchronously. Furthermore, we show that enabling the monitor to synchronise (and desynchronise) at will with the system does not change the behaviour. We have investigated the use of this approach on an industrial case study — dealing with financial transactions, and for which a compensation-based implementation was already in place.

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. An architecture implementing this mode of monitoring is presented in section 4, and we illustrate its use on an industrial case study in section 5. Finally we discuss related work in section 6.

2 Compensations

Two major changes occurred which rendered traditional databases inadequate in certain circumstances [14, 13]: on the one hand there was the advent of the Internet, facilitating the participation of heterogeneous systems in a single transaction, and on the other hand, 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. These issues rendered the traditional mechanism of resource locking for the whole duration of the transaction impractical since it may cause severe availability problems, and 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 [14, 13] 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 compen-

sations have been proposed [5, 4, 3, 15, 21], little work [5, 6] has been done to provide a mathematical basis for compensations. For simplicity, 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 correctness 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 equal, *written $v =_\tau w$, if their external manifestation is identical: $v^{-\tau} = w^{-\tau}$. This notion is extended to sets of strings.*

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

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*[3]*. Extending compensations to an alphabet enriched with the internal action τ, we assume that $\overline{\tau} = \tau$.*

[3] 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*.

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: $\overline{\varepsilon} \stackrel{def}{=} \varepsilon$ and $\overline{aw} \stackrel{def}{=} \overline{w}\,\overline{a}$. 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 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':*

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

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

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

Proposition 1. *The catenation 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

We start by characterising synchronous and asynchronous monitoring strategies. 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. Based on the asynchronous semantics, 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. Finally we show how enabling synchronisation and desynchronisation at will leaves the results intact.

3.1 Synchronous and Asynchronous Monitoring

We will assume a labelled transition system semantics over alphabet Σ for both systems and monitors. 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. $(m, a, m') \in \longrightarrow_{mon}$). For any transition relation $\xrightarrow{a}_X$ ($a \in \Sigma$), we will write $\overset{w}{\Longrightarrow}_X$ ($w \in \Sigma^*$) to denote its reflexive transitive closure.

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. For example consider a simple 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 for any other input. The synchronous composition of such system and monitor takes a step if both the system and the monitor can take a step independently 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 messages 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* Async_S *allows progress of the system adding the events to the intermediate buffer, while rule* 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: $(P, \varepsilon, A) \overset{b}{\longrightarrow}_{|||} (P', b, A)$. Subsequently, the system may either continue further, or the monitor can consume the event from the buffer and fail: $(P', b, A) \overset{\tau}{\longrightarrow}_{|||} (P', \varepsilon, \otimes)$. At this stage the system can still progress further until it is stopped by the rule AsyncErr.

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 $(\sigma, ww', m) \overset{\tau^*}{\Longrightarrow}_{|||} (\sigma, w', m')$.*

Synchronous Monitoring

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

Asynchronous Monitoring

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

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

Compensation-Aware Monitoring

$$\textsc{Comp}\ \frac{}{(\odot, wa, \otimes) \overset{\overline{a}}{\longrightarrow}_{C} (\odot, w, \otimes)}$$

Adaptive Monitoring

$$\textsc{ReSync}\ \frac{}{(\sigma, \varepsilon, m) \overset{\tau}{\longrightarrow}_{A} (\sigma, m)} \qquad \textsc{DeSync}\ \frac{}{(\sigma, m) \overset{\tau}{\longrightarrow}_{A} (\sigma, \varepsilon, m)}$$

Fig. 1. Semantics of different monitoring schemas

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 uses the asynchronous monitoring rules, together with an additional one* Comp *which performs a compensation action of actions 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.

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:

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

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 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 the derivation 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 $m = \otimes$. By taking $w_1 = w_2 = \varepsilon$, 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*:* $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', v, \otimes) \xrightarrow{\tau}_C (\odot, v, \otimes)$.
(b) Using the rule CompB*:* $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\odot, va, \otimes) \xrightarrow{\overline{a}}_C (\odot, v, \otimes)$.
(c) Using the rule Async_S*:* $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma'', v, m') \xrightarrow{a}_C (\sigma', va, m')$.
(d) Using the rule Async_M*:* $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', av, m'') \xrightarrow{\tau}_C (\sigma', v, m')$.

The proofs of the four possibilities proceed similarly. Consider the possibility (b):

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

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:*

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

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

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 steps by the compensation-aware transitions* $(\sigma', v, m) \overset{a\tau}{\Longrightarrow}_C (\sigma'', v, m'')$*, 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 (*$\sigma' = \odot$*):*

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

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

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

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 Desynchronising and Resynchronising

Despite compensation-awareness, in some systems it may be desirable to run monitoring synchronously with the system during critical sections of the code, only to desynchronise the system from the monitor again once control leaves the critical code section. 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 two additional (over and above synchronous and asynchronous monitoring) rules 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 $\mathit{traces}_A(\sigma, m)$, is defined as follows:

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

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

Theorem 2. *Asynchronous and adaptive monitoring are indistinguishable up to traces:* $\mathsf{traces}_{A}(\sigma, m) = \mathsf{traces}_{|||}(\sigma, m)$*. Compensation-aware adaptive monitoring is also indistinguishable from compensation-aware monitoring up to traces:* $\mathsf{traces}_{AC}(\sigma, m) = \mathsf{traces}_{C}(\sigma, m)$*.*

The theorems can be easily proved based on transition-relation inclusion. An immediate corollary of this last result, 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 A Compensation-Aware Monitoring Architecture

LARVA [9] is a synchronous runtime verification architecture supporting DATEs [8] 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, 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 directly to rules ASYNCERR and COMP respectively. Therefore, the actual time of stopping and how the indicated actions are compensated are left for the system to decide.

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 these actions for compensation. It is important to point out that the monitor can only compensate for actions of which it is aware — the monitor can never alert the system to compensate actions which have not been logged.

To support switching between synchrony and asynchrony, 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

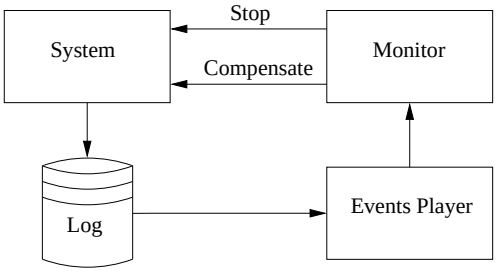

Fig. 2. The asynchronous architecture with compensations cLarva.

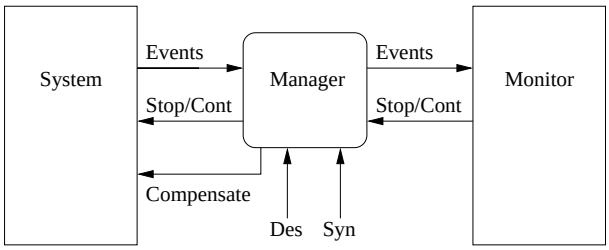

Fig. 3. The asynchronous architecture with synchronisation and desynchronisation controls.

from the manager. The following code snippet shows the logic of the synchronisation manager:

```
c = ok                    ;set default control to ok
while (c != stop)
  if (synch_mode)
    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
```

The behaviour in which this architecture differs from cLarva is that it can operate in both synchronous and asynchronous modes and can switch between modes. Switching from synchronous to asynchronous is trivial. The opposite requires that the manager waits for the monitor to consume all the events in the buffer and then allowing the system to proceed further. So far this has not been implemented, but we aim to implement it in the future as an improvement on cLarva.

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 parts of the system to ensure that only the relevant parts of the system are stopped. For example, consider the case where a transaction is carried out without necessary rights. In such a case, the

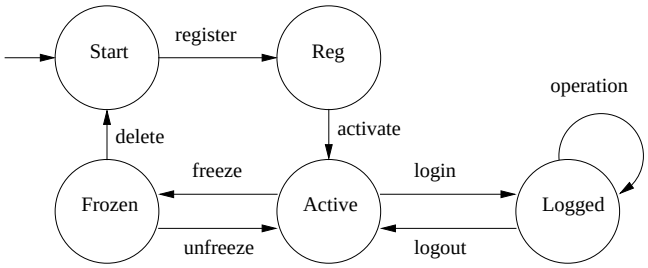

Fig. 4. The lifecycle property.

transaction should be stopped and compensated. However, if a user has managed to illegally login and start a session, then user operations during that session should be stopped and compensated.

5 Case Study

We have applied cLarva 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.

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. 4 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.

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 is compensated

[4] `www.ixaris.com`

and the user 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 database of 300,000 users with around a million credit cards. A number of issues have been detected through the monitoring system: (i) certain logs were missing; (ii) some users were found to be in a wrong state, eg. should be in a frozen state but still active; (iii) the limit of the amount of money a user can spend was in some cases exceeded. Monitoring of the logs performed asynchronously ensured the identification of issues, and through the compensation mechanism, identification of actions to be taken to rollback the system to the point where the violation occurred. At that point, one can then either notify the operator of the issue, or trigger the system's own exception handling mechanism.

Although the current properties being monitored on Entropay are relatively lightweight and monitoring can be done relatively seamlessly, due to security issues, running the monitor synchronously is not an option — avoiding changes in the architecture of Entropay. The monitors are linked to the database of log entries to enable asynchronous monitoring, but giving feedback and compensation actions upon discovering issues.

6 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 [19] 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 [2, 7, 1, 19, 20, 12, 16, 11] which support asynchronous monitoring — sometimes also known as trace checking or offline monitoring. A number of these tools and algorithms [2, 7, 1, 19] support only asynchrony unlike our approach which supports both synchronous and asynchronous approaches. Furthermore, although a number of approaches [12, 16, 20, 11] 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 in the area of distributed games [17, 18, 10]. 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 simplified version of the distributed game problem with only 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.

7 Conclusions and Future Work

In this paper, we have presented an adaptive compensation-aware monitoring architecture, and an implementation cLarva. Combined with the notion of compensations where actions of a system can be 'undone' to somewhat restore a previous state, we reduce the effect of errors detected late (due to asynchronous monitoring) by compensating for additional events which the system may have performed in the meantime. We have demonstrated the use of this approach on a financial transaction handling software. The advantage of this case study is that compensations were already a well-defined concept from the developers perspective.

At the moment we are investigating the use of heuristics for desynchronisation and resynchronisation of the system and monitor. At the simplest level, one can simply trigger asynchronous monitoring when the system load reaches a certain level, and switch back to synchronous monitoring when it falls below the threshold. It would be interesting to explore further the development of smarter heuristics for this purpose — taking into account other issues, such as the trust in (or lack thereof) parties involved in the transaction and its monetary value.

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, or commit a series of actions such that they cannot be compensated. In the future, we aim to lift this limitation by introducing a structured approach to compensations.

References

1. J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 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)*, 2009.

3. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, 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*, pages 133–150, 2004.
6. L. Caires, C. Ferreira, and H. T. Vieira. A process calculus analysis of compensations. In *Trustworthy Global Computing (TGC)*, volume 5474 of *LNCS*, 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, 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, L'Aquila, Italy, 2008.
9. 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 Computer Society, 2009.
10. E. Cronin, A. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools Appl.*, 23(1), 2004.
11. 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'05)*. IEEE Computer Society Press, 2005.
12. 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, 1995.
13. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, 1987.
14. J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154, 1981.
15. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *Service-Oriented Computing (ICSOC)*, 2006.
16. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems 02*, 2002.
17. D. Jefferson. Virtual time. In *International Conference on Parallel Processing (ICPP)*, pages 384–394, 1983.
18. 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.
19. G. Roşu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, 2001.
20. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Eng.*, 12(2):151–197, 2005.
21. C. Vaz, C. Ferreira, and A. Ravara. Dynamic recovering of long running transactions. *Trustworthy Global Computing: 4th International Symposium (TGC)*, 2008.