Programming Compensations for System-Monitor Synchronisation

Christian Colombo
Dept. of Computer Science
University of Malta
christian.colombo@um.edu.mt

Gordon J. Pace
Dept. of Computer Science
University of Malta
gordon.pace@um.edu.mt

ABSTRACT

In security-critical systems such as online establishments, runtime analysis is crucial to detect and handle any unexpected behaviour. Due to resource-intensive operations carried out by such systems, particularly during peak times, synchronous monitoring is not always an option. Asynchronous monitoring, on the other hand, would not compete for system resources but might detect anomalies when the system has progressed further and it is already too late to apply a remedy. A conciliatory approach is to apply asynchronous monitoring but synchronising when there is a high risk of a problem arising. Although this does not solve the issue of problems arising when in asynchronous mode, compensations have been shown to be useful to restore the system to a sane state when this occurs. In this paper we propose a novel notation, compensating automata, which enables the user to program the compensation logic within the monitor, extending our earlier results by allowing for richer compensation structures. This approach moves the compensation closer to the violation information while simultaneously relieving the system of the extra burden.

1. INTRODUCTION AND BACKGROUND

Online systems handling large volumes of financial transactions have become common: online betting and gaming, payment gateways, online shopping, online banking, etc. Although such systems go through heavy testing regimes, runtime analysis remains crucial for detecting malicious user behaviour such as fraud. Once users are detected to be fraudulent, they can be stopped immediately from completing illegal activities. The downside is that detecting such behavioural patterns is usually resource intensive. While adding computing resources to meet the overheads might be an option, we explore approaches which make better use of available resources. On the one hand, online systems are designed to cope with peak-time loads and thus valuable resources might be lying idle during the rest of the time. On the other hand, not all users pose the same risk level, meaning that priority can be given for the analysis of high-risk users over the rest of the users. To keep a close watch on high-risk users while at the same time incurring as little overhead penalty as possible, we have proposed a monitoring architecture, cLarva[5, 6], which is able to synchronise and desynchronise monitors depending on the perceived risk. Thus, while monitors with associated high risk are executed synchronously, consuming system resources, the majority of the normal-risk monitors are run asynchronously — when resources are otherwise lying idle.

The disadvantage of asynchronous monitoring is that by the time the monitor detects a violation, the system would have already progressed further. Considering the case of a fraudulent user as an example, by the time the monitor detects a fraud, the user might have already performed a number of transactions. To this end we built means through which the system can be resynchronised to the point at which the monitor identified the violation. In cLarva, we have proposed the use of *compensations* — actions which logically undo particular actions when executed. For example, if a user transfers money from a third-party bank account onto his credit card, the compensation would entail the reverse transfer minus a transfer charge. Note that the compensation need not be the exact inverse of the action it is meant to "undo". If the system monitor is running asynchronously, and by the time the monitor identifies that the user is fraudulent the transfer has taken place, the compensation can be used to restore the system to the state which occurred at the time of violation. The advantage of this approach is that in the context of financial transactions, the compensations are part-and-parcel of programming such transactions, commonly referred to as *long-lived transactions.*

Figure 1 depicts the current cLarva architecture where the system sends events to the manager. If the manager is working in synchronous mode (depending on the outcome of dedicated heuristics), the manager forwards the event to the monitor and waits for monitor feedback which is relayed to the system. On the other hand, in case of asynchronous monitoring, control is passed back immediately to the system, keeping the event in a buffer for later consumption by the monitor. If the monitor detects a violation, then the system is stopped and the manager initiates the compensation of the events remaining in the buffer, i.e. the events which the system had carried out since the violation had occurred.

To illustrate what happens when the monitor detects a violation in asynchronous mode, we use an example where the system has sent the events a,b,c,d,e,f to the manager while the monitor has only processed a,b,c and upon processing d, it detects a problem. Note that at this point e and f are still waiting to be processed in the manager buffer. To synchronise the monitor and the system, e and f have to be compensated for; reverting back the system to the state it was in when the problem occurred. To this end the system is stopped and $\overline{f},\overline{e}$ are executed, representing the compensations of f and e respectively. Once the system is back to exactly after d occurred, any necessary corrective action

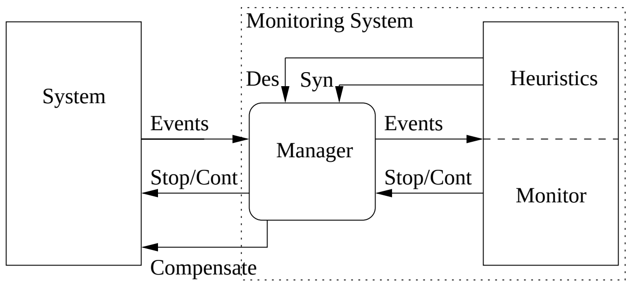

Figure 1: The asynchronous architecture with compensations cLarva.

may take place.

While this architecture incorporates basic synchronisation through compensations, it has a major shortcoming — it assumes that all actions are compensable and that the compensation of a particular action is the same no matter the context. Clearly, this is not always the case. For example a money transfer may be cancelled free of charge in the context of a promotional offer while a percentage charge may be incurred in the context of a normal transfer. Furthermore, it might even be the case that under particular contexts a particular action is not reversible (e.g. if the transfer took place in the context of a shipped purchase order). As an improvement over this naïve approach, we introduced the notion of compensation scopes [5] such that once a scope ends, activities within the scope can no longer be compensated. While scopes make the architecture more realistic, it still remains highly inflexible. For example there is no way of expressing *programmable compensations* which enable the definition of a compensation for a whole scope; e.g. if a purchase order has been shipped, a possible programmed compensation would be to specify a new transaction which ships back the order once it reaches its destination.

Another limitation of the existing approach is that it is not possible to compensate for activities which occurred before the point of violation detection. Note that even in the case of synchronous monitoring, sometimes it is necessary to compensate for activities which were allowed before a pattern could be detected. For example in the case of a fraud, the fraud might not have been detectable until the third suspicious money transfer. In this case it would make sense to reverse the previous two transfers as well once the user has been detected as fraudulent. Referring back to the example, this would correspond to the compensation of activities a,b,c.

Moreover, in cLarva compensations are programmed and executed by the system, introducing extra programming within the system and potentially introducing bugs. Furthermore, the system is not aware of why compensation is required and thus eliminates the possibility of adapting compensations according to the context of the detected violation.

In this paper, we propose an improved cLarva architecture which uses a flexible notation (see Section 3) enabling programmers to easily express compensations for activities in an asynchronous monitoring setting. The new architecture (see next section) allows programmed compensations and the compensation of activities which occurred before the violation. A case study is presented in Section 4 to show how the monitor can be programmed with compensation logic in realistic scenarios.

2. ENRICHING COMPENSATIONS FOR MONITORING

The existing cLarva architecture is shown in Figure 1. It has can signal the system to compensate for a sequence of actions (the actions remaining in the buffer less scoped actions). To allow more flexibility as regards the choice of compensations, an architecture which has an additional compensation management component within the monitoring system is shown in Figure 2. This component receives the system's events (at the same time as the monitor), based on which it dynamically manages which compensations are to be executed if (and when) the monitor detects a problem. When the monitor identifies a problem, it signals the compensation manager with a *compensate signal* (*Comp*) followed by the events remaining in the buffer. Once the compensation manager has received and processed all the buffer events, i.e. it receives the *end-of-buffer signal* (*EoB*), it starts signalling the system to carry out the appropriate compensations.

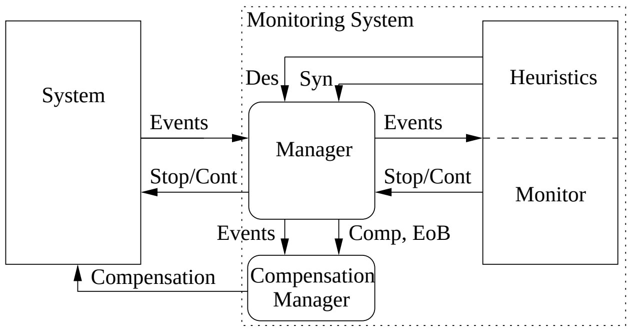

Figure 2: The asynchronous architecture with the compensation manager.

Note that unlike the previous architecture where there is a fixed way to compensate for activities (the monitor simply indicated which activities had to be compensated), the new architecture enables an enriched notion of compensations by delegating the decision to the compensation manager. Still, the compensation manager requires a specification of how to manage the compensations for a particular system. To this end, we propose a specialised programming notation which is focused on expressing automatic management of compensations. Based on our experience with asynchronous monitoring of real-life financial transactions [6], there are a number of characteristics which the notation should support:

Explicit correspondence The notation should make the correspondence between activities and their compensations clear.

Automatic management Upon reaching an error state, not only should compensations have been stored, but it should also be clear which compensations should be run and in which order. Thus, compensations should be automatically stored during execution and automatically activated upon reaching an error state.

Context-awareness The compensation of an activity varies depending on the context in which the activity occurs. For example if we are compensating a money transfer, a fee is charged to the user account to cover the bank charges involved. However, if the transfer originally occurred in the context of a promotional offer, then such a fee would be charged to the company account.

Support complex dependencies Properties of transaction-based systems frequently revolve around life-cycles for entities such as users and credit cards. Such life-cycles usually involve complex intertwined logic which is very naturally expressed in terms of unstructured notations such as automata.

Scope management The current cLarva discards compensations of scoped activities as soon as the end of scope is reached. While it is true that various activities can no longer be compensated once the end of scope is reached, it is usually the case that the transaction as a whole might be compensated for in a different way. For example, once a purchase has been carried out, the activities might not be individually compensable (e.g. shipping cannot be reversed). However, the purchase as a whole might be compensable in terms of another transaction which ships the purchased goods back to the seller.

Alternative Another desirable characteristic of a compensation notation is to be able to specify alternative behaviour in case some behaviour fails. For example consider a failed money transfer in the context of a purchase. If the transfer fails, it might be appropriate to retry it some time later rather than proceeding to compensate for the whole purchase transaction. Thus, the "alternative" mechanism would provide the possibility of stopping compensation execution (e.g. of the purchase) and continuing with "normal" behaviour (e.g. retrying the transfer).

There is much literature on notations supporting compensations [4] including flow languages [2, 3], process algebras [7, 9], automata [8] and pictorial representations [1]. Considering the desirable characteristic of an unstructured notation only the latter two (automata [8] and BPMN [1]) are potential candidates. However, in [8], the unstructured nature of automata is only used to encode compensation patterns (indeed, compensations are not first class elements of these automata) while the programmer is limited to using a number of patterns to program transactions with compensations. As regards BPMN, the language is a generic language for business modelling which goes way beyond our need of a concise compensation-centred notation.

Based on these observations, we propose *compensating automata* in which compensations are first class elements and provide automatic activation of compensations. The next section gives examples showing how compensating automata can be used to model realistic scenarios involving compensations.

3. COMPENSATING AUTOMATA

To illustrate the suitability of compensating automata for expressing systems with compensations, we use examples based on our experience on real-life financial transactions control systems. Consider a simplified system which allows users to login, load money from their bank accounts onto their cards, transfer money from one card to another, and use their money to pay third parties for purchases. These activities (except logging in) must all have corresponding banking transactions — unless the actions succeed at the bank (externally), the system cannot proceed (internally). In what follows we express a number of scenarios using compensating automata:

Money load transaction Consider the *load* procedure (Figure 3), consisting of three activities which are expected to be carried out in sequence: decrease the external source account balance, increase card balance, increase internal actual and available balances. Each of these activities have a corresponding compensation (the second label on each transition). If all the activities succeed, then the success state is reached (the state with the dot). Otherwise, if something goes wrong along the way, the error state is reached (the state with a star). In this case, we must also reverse the successful parts of the load. This is automatically handled by the compensating automaton which keeps track of the sequence of compensating activities of traversed transitions. For example, if attempting to increase the internal balances fails, then the external card balance should be decreased to its original amount and the external account balance should be increased, charging a fee to cover bank charges. Through this example we show that compensating automata provide a clear correspondence between activities and their compensations while they automatically maintain and activate compensations.

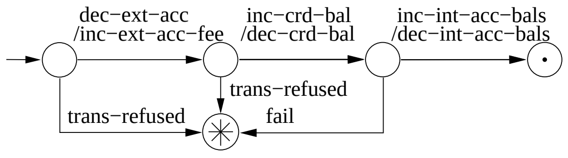

Figure 3: A compensating automaton modelling a *load* transaction.

Purchase transaction Recall that identical activities may have different compensations depending on their context. For example in a purchase transaction (see Figure 4), the compensation for decreasing the external available balance does not entail a fee as in the case of the load. The rationale is that for a purchase the user would have already paid a commission and the cancellation of the purchase might arise from the seller's and not the buyer's side. Furthermore, note that after the settlement is received, control goes through a compensation *deviation*. Intuitively, this means that an alternative path of execution is provided in case of reaching the error state later on. In this case, if a failure occurs after the settlement had been received (a highly unlikely event), then, instead of compensating further, the operator is notified of the issue. Therefore, if an error state is reached after successfully decreasing the actual card balance, the latter is compensated but subsequently, instead of executing other compensations, the operator is notified to rectify the situation.

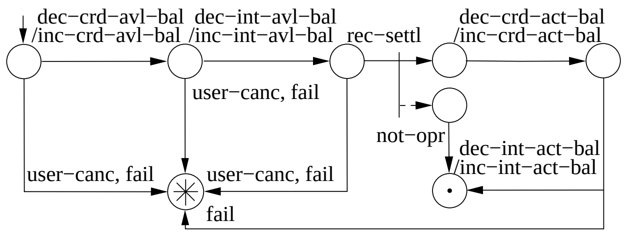

Figure 4: A compensating automaton modelling a *purchase* transaction.

User life-cycle Life-cycle properties are common in transaction systems. In this case, we are essentially monitoring a user session cycle from login till logout. Figure 5 shows a property which also includes the previous two example properties as nested automata (not fully shown in the diagram). Note that each nested automaton may have another automaton as its compensation. This is depicted as an automaton in the bottom half of a state with nested automata. For example the compensation of money loading is to freeze the card involved. The reason for choosing not to reverse the load is that the involved money might be the result of a fraud and would be needed for investigation purposes. The approach is similar for money transfer but two (source and destination) cards have to be frozen in this case. For a purchase, we assume that there is no compensation to carry out once it has completed.

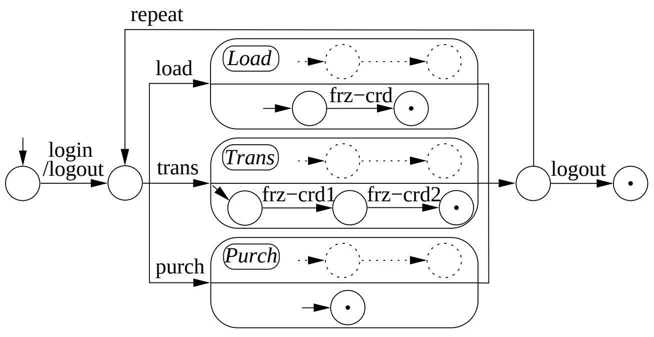

Figure 5: A compensating automaton modelling the user life-cycle.

While expressing compensations as shown above can be useful for specifying systems with compensations, our main aim is to use these specifications to synchronise a system with a monitor upon a late violation detection. The next section explains how compensating automata can be incorporated within cLarva.

4. INTEGRATING COMPENSATING AUTOMATA IN CLARVA

As shown in the proposed architecture (Figure 2), the compensation manager receives two dedicated signals from the main manager (apart from events): *compensate* and *end-of-buffer* (*EoB*). The former indicates that the monitor has detected a problem while the latter indicates that the system has stopped (i.e. no more system events will be forthcoming) and is waiting to receive compensation instructions. These two signals are crucial in deciding which compensations should be executed. In what follows, we give two examples to show how these signals can be used within compensating automata to program the compensation manager.

Excessive loads Figure 6 shows a compensating automaton which accumulates compensations for a sequence of money loads following the receipt of the *comp* signal from the monitor. Note that the automaton is essentially a part of the user life-cycle automaton (of Figure 5) which ignores all activities except money loads. When the monitor detects that the limit of money loading has been violated, the compensation manager would receive the *comp* signal and starts accumulating compensation while consuming the events remaining in the buffer. Upon receiving the *EoB* signal, i.e. all events in the buffer have been processed, the compensating automaton would reach an error state (note that all states lead to an error state upon an *EoB* signal). At this point, the compensation manager starts executing the accumulated compensation — reversing all the loads which contributed to the excess. It is significant that the compensation in this case is the reverse of a load while in the life-cycle example it consisted of freezing the card. The reason for this choice is that the life-cycle example is intended to accumulate compensations for fraud detection while in this example we are interested in simply reversing any loads through which a user exceeded the stipulated limit.

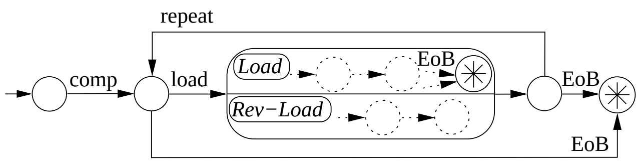

Figure 6: A *load* compensation automaton which listens for the *comp* and *EoB* signals.

User life-cycle violation Figure 7 shows how the payment gateway life-cycle example can be incorporated into the compensation manager. There are three main modifications from the specification shown in Figure 5:

- While going through each life-cycle activity, the automaton listens for the *EoB* signal. If such a signal is received then the automaton reaches an error state and starts compensating.
- After user logout, the compensation monitor does not go into a successful final state, but instead waits for a possible *EoB* signal. The reason for this decision is that since we are trying to detect fraudulent users, then, any activity carried out by such user should be compensated whether or

not such activities occurred after the monitor detected that the user is fraudulent. For this reason, the *compensate* signal is ignored in this example since the point at which the monitor detects the problem is irrelevant to the resulting compensation.

- Upon taking the logout transition, a deviation is traversed which denotes that compensation should be stopped at that point. Through this feature the compensating automaton would suggest that, since the user has logged out, instead of going through all of the compensations, it would be sufficient to freeze the user's account.

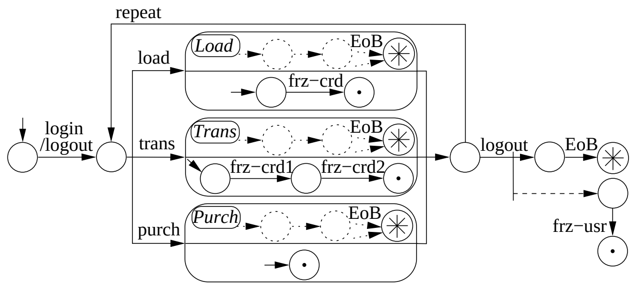

Figure 7: A modified user life-cycle automaton which listens for the *EoB* signal.

5. CONCLUSIONS AND FUTURE WORK

Monitoring asynchronously is particularly convenient for checking systems with peak-load periods during which resources cannot be compromised. However, asynchronous monitoring would usually delay the detection of violations and thus limits the possibility of the monitor taking any corrective action. To this end, we have presented compensating automata which enable a user to specify to the monitoring architecture how to effectively synchronise the system with the monitor. This approach enables the monitor to generally run asynchronously whilst switching to synchrony when there is a high probability of violation detection. Compensating automata facilitate the specification of rich compensation notions such as context-sensitive compensations, programmable scope compensations and the specification of alternative behaviour in case of failure.

To showcase compensating automata, we have presented realistic examples of a system handling financial transactions whereupon the monitor detecting a violation, the compensation manager automatically executes appropriate compensation. Admittedly, we have only discussed the semantics of compensating automata informally in this paper. In view that such automata will be used for monitoring purposes, we are currently looking at their formal semantics and proving them sound.

Using compensating automata, we have delegated the compensation logic for system-monitor synchronisation to the monitor (rather than the system). In the future, we aim to take this approach further and explore *monitoring-oriented compensation programming* where compensations (not only those used for monitor synchronisation) can be programmed independently of a system's logic, delegating the choice of compensations to a monitor. This would alleviate the system code from catering for compensations, while gaining more confidence that compensations are programmed correctly due to the use of a formal notation.

6. REFERENCES

[1] Business process model and notation, v2.0, 2011. Available at: http://www.omg.org/spec/BPMN/2.0/PDF/ (Last accessed: 2012-01-7).

[2] 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.

[3] M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, Lecture Notes in Computer Science, pages 133–150. Springer, 2004.

[4] C. Colombo and G. Pace. Recovery within long running transactions. *ACM Computing Surveys*, 2012. to appear.

[5] C. Colombo and G. J. Pace. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 2012. to appear.

[6] 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.

[7] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *International Conference on Service-Oriented Computing (ICSOC)*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[8] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. Modeling long-running transactions with communicating hierarchical timed automata. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2006.

[9] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Programming Languages and Systems (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.