Comprehensive Monitor-Oriented Compensation Programming

Christian Colombo Gordon J. Pace
Department of Computer Science
University of Malta
{christian.colombo | gordon.pace}@um.edu.mt

Compensation programming is typically used in the programming of web service compositions whose correct implementation is crucial due to their handling of security-critical activities such as financial transactions. While traditional exception handling depends on the state of the system at the moment of failure, compensation programming is significantly more challenging and dynamic because it is dependent on the runtime execution flow — with the history of behaviour of the system at the moment of failure affecting how to apply compensation. To address this dynamic element, we propose the use of runtime monitors to facilitate compensation programming, with monitors enabling the modeller to be able to implicitly reason in terms of the runtime control flow, thus separating the concerns of system building and compensation modelling. Our approach is instantiated into an architecture and shown to be applicable to a case study.

1 Introduction

With the advent of long-lived transactions, particularly in the context of web service compositions, compensation programming has gained considerable attention from both industry and academia. While the *de facto* standard of programming business processes having compensations is BPEL [3], the literature (see [11] for an overview) is inundated with varying proposals of formal approaches to compensation programming. The main challenge which these attempt to address is the dynamic aspect of compensations; compensations are chosen depending on the control-path being taken, i.e. a snapshot of the history of the system at the moment of failure, which is unknown at compile time. For example the cancellation of an online transaction depends on whether the payment has *previously* been affected or not, whether the transport arrangement has *earlier* been confirmed or not, etc. This is significantly different from traditional exception handling in the *try-catch* fashion since the latter is generally static, i.e. depends on the state of the system at the moment of failure, e.g., on an invalid user input, or corrupt data being read from a file.

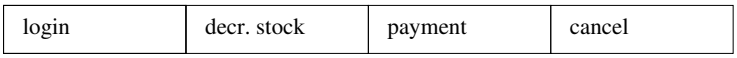
Yet, even vanilla exception handling may need to reason about history snapshots — something which typical programming languages do not support well. For this reason, software monitoring techniques such as monitor-oriented programming (MOP) [25], have been specifically proposed to support reasoning about control paths by detecting events such as method calls and matching event patterns to logic formulae.

Note that while compensations historically have roots in error recovery in highly concurrent contexts [16, 28], nowadays compensations are also commonly used to model exceptional case handling as part of the normal flow of a system. This is also the case with MOP; it originates from a background of error recovery techniques but can be used to trigger any functionality through monitoring. Similarly, as the examples we use in this paper, we do not confine ourselves to the error handling aspect of compensation programming and indeed our approach can be used alongside exception handling techniques.

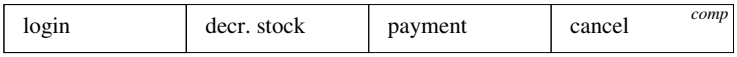
Submitted to:
FESCA 2014

When MOP is used to support error handling, it can be used to help the programmer detect exceptions, but it does little to support their reparation. Put differently, MOP is able to detect *when* an exception has occurred but does not directly support *how* to handle the exception. This is not surprising in the context of exception handling because the programmer is typically given complete freedom as to the choice of the recovery code. However, this is not the case with compensation programming [11] because it exploits the execution pattern of a program to deduce the appropriate recovery action: (i) knowledge of what has been executed enables the demarcation of actions which have to be compensated for; while (ii) knowledge of the execution pattern enables the composition of the corresponding compensation actions (typically in reverse order of their original execution). This has the advantage of partially automating the *how* of compensation programming while giving rise to two further questions: *what* to compensate for, and *which* compensation strategy to use if there is more than one way of compensating an earlier behaviour. More concretely, the four questions of *when*, *what*, *how* and *which* can be understood in terms of the following example:

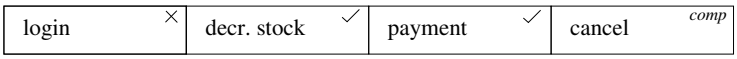
Consider an e-procurement system through which users are able to login, order, and pay for bought items which are subsequently shipped to the customer. During the transaction, the user can also cancel the order. A possible trace of actions is:



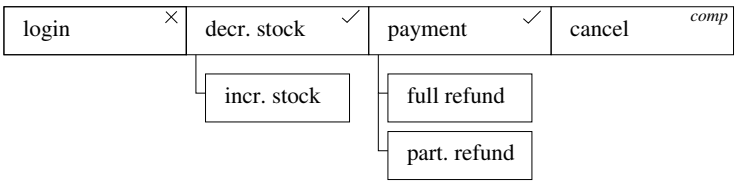
***When* to trigger compensations.** At each step of a process, compensations can potentially be triggered. In the case of the example trace, compensations may be triggered *when* the user cancels the purchase (shown below by *comp* in the top right corner). Note that while compensations are typically triggered upon an exception throw (e.g., payment fails) as an error handling mechanism, they may also be used for programming normal business logic as in the case of this example.



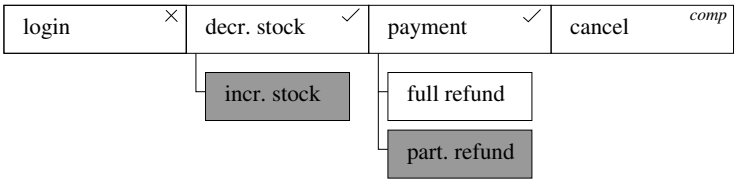
***What* to compensate for.** When compensating, due to the correspondence between actions and their compensations, one would have to deduce *what* actions to compensate for. Compensations by definition compensate for previously completed actions (e.g., a completed payment). However, not all previous actions might need to be compensated for. For example the login action (see figure below) does not need to be compensated for when cancelling a purchase (the user would typically still want to remain logged in even though a purchase has been cancelled).



***How* to compensate for actions.** Building upon the example, for each activity earmarked for compensation, an action has to be selected for the job. As shown in the figure below there may be one or more ways in which an activity may be compensated for. To compensate for a decrement in stock, one would simply increment the stock while a payment is compensated in full or in part depending on the context.



***Which* compensation strategy to use.** The next step is to choose *which* of the possible compensation options are to be executed. In the example (depicted below) the choice of the type of refund depends on whether the user is a privileged user or a standard user at the time of refund (which might have changed since the time of payment). In this case, we have opted for the partial refund, assuming the user is standard. Once the compensation actions are outlined, these are typically executed in the reverse order of the corresponding actions. A similar issue is that of the order in which compensations are to be performed. To be true to the notion of compensations, sequential behaviour is typically compensated for in reverse order, but concurrent behaviour may be compensatable in a concurrent manner. These decisions merit abstraction and careful consideration when programming compensations.



The simple example above highlights the number of questions involved in compensation programming. While we have only considered the choices on a per-action basis, in real life scenarios the decisions might need to be taken on sub-sequences of actions. For example a sub-sequence of actions such as the four actions in the above scenario might have a specific single compensation, logout, if the user is detected to be fraudulent. Furthermore, coming up with answers to the questions may involve arbitrary complex logic considering various issues including the control flow, system performance, fraud, and any relevant attributes of the entities involved in the transaction. These intricacies motivate the need to abstract these questions and use an appropriate design to handle their interaction.

Given the use of runtime monitors for the detection of complex patterns of system behaviours as in MOP, applying monitoring to these questions seems to be a natural solution. In previous work [10], we have proposed a monitoring approach, monitor-oriented compensation programming (MOCP) to the programming of the *what* and *how* questions through a compensation manager residing on a monitoring component receiving live updates from the system. Our contribution in this paper is to extend the architecture so as to enable the programming of all four issues in compensation programming. This is done by combining standard runtime monitoring to seamlessly interact with the compensation manager, instructing it when and which compensations to trigger — signals, which in previous work, had to be provided by the system itself. Such an arrangement enables the specification of compensations to be done at a higher level of abstraction — in the monitoring language — while also providing more fine-grained separation of concerns, alleviating the system of compensation programming.

In the rest of the paper, we start by providing the preliminaries in Section 2 and go on to introduce the proposed architecture in Section 3. Subsequently, we show how our approach is applicable to a case

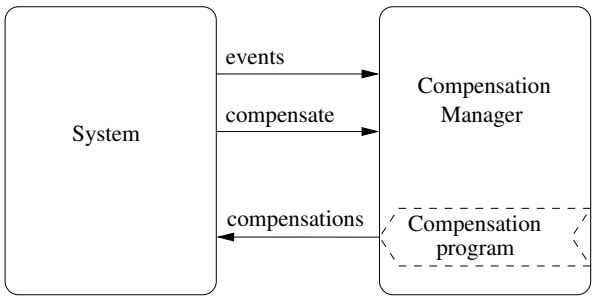


Figure 1: The architecture with a compensation-aware system

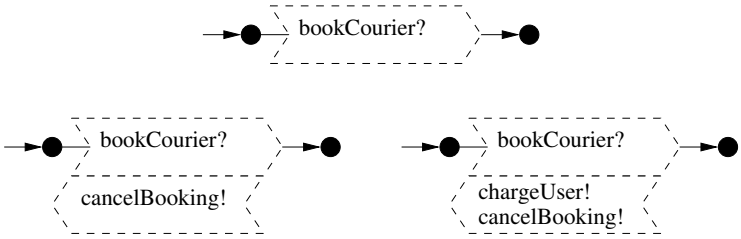


Figure 2: Basics of compensating automata

study in Section 4 followed by a more practical account of how the architecture can be applied to real-life scenarios in Section 5. Finally, we compare our proposal to related work in Section 6, whilst concluding in the final section.

2 Preliminaries

In previous work [10] we have proposed compensating automata — a notation for specifying compensation logic. Through a case study, compensating automata have been shown to be able to express complex compensation logic alleviating the system from handling compensations. Instead, the system simply indicates the need to be compensated and the compensation manager triggers compensations. Figure 1 depicts monitor-oriented compensation programming as previously proposed, where the system communicates all the relevant events to the compensation manager while the latter decides *what* events to compensate for and *how* — collating the applicable compensations. If the system reaches a state where it needs to execute compensations, the system signals the compensation manager which in turns starts executing the collated compensations. Upon executing all the relevant compensations, the compensation manager returns control back to the system.

To assist the user in programming the compensation manager, we have proposed compensating automata — essentially state machines which listen to system events and compose compensations accordingly. A basic transition of a compensating automaton listens for an event and installs no compensation (Figure 2 [top]). To program a compensation installation upon an event, one or more compensation instructions can be specified as shown in Figure 2 [bottom].

Furthermore, transitions can be composed in sequence as shown in Figure 3 [top], signifying that subsequent compensations are installed on a stack and executed in reverse order if activated. In particular cases, the user might need to purge the stack and this can be programmed in terms of the box (shown in Figure 3 [bottom]) such that when execution reaches the end of the box, the stack with the compensations

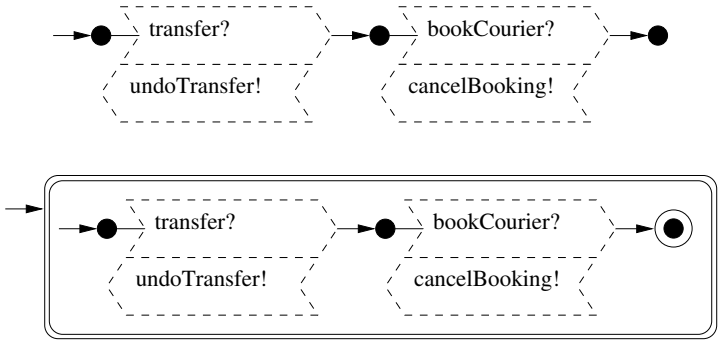


Figure 3: Sequencing and scoping in compensating automata

collected during the box's execution is purged and discarded. Note that the box is double edged in this case because it is also acting as the final (and only) state of the parent automaton.

While compensating automata provide further advanced features for compensation programming, this short introduction suffices to help the reader understand the case study presented in Section 4. The next section introduces the modified MOCP architecture, extending the role of monitoring further from our previous work [10]. Note that in the rest of the paper MOCP will refer to the extended architecture.

3 A Compensation-Handling Architecture

To support comprehensive compensation programming — alleviating a system from compensation programming up to the point where it need not be aware of compensations — the proposed architecture is entirely based on monitoring. We decompose this architecture into two parts: (i) one which deals with compensation aspects such as compensation installation and discarding, addressing the questions of *what* and *how* to compensate, and (ii) another which deals with triggering compensations at particular points in time, addressing *when* and *which* compensation strategy to execute. Note that the former is involved directly with compensation programming while the latter simply triggers compensation strategies of choice at particular points in time. The first component of such an architecture can be programmed using compensating automata which have been specifically devised for such programming. On the other hand, encoding the *when* and *which* logic of compensations can be done by any specification notation which can be used for runtime verification and supports the triggering of reparatory actions. For this job we choose Dynamic Automata with Timers and Events (DATEs) [13] because they are somewhat natural to integrate with compensating automata — they are also automaton-based and support channel communication. Note that the proposed architecture can also be instantiated using any other suitable specification notation.

In the proposed architecture, as shown in Figure 4, the monitoring components may interact with the system through three connections: one which enables the system to communicate events, another enabling the monitor to signal the system to continue, and a third one on which the compensation manager component instructs the system regarding what compensations are to be executed. Furthermore, the architecture is event-driven and each event the system emits triggers a series of steps as follows:

1. The system emits an event and waits for the *continue* signal from the compensation framework.
2. The monitor receives the system event and processes it, emitting a *compensate* signal to a particular compensation manager [answering the questions of *when* and *which* compensation to trigger] or a

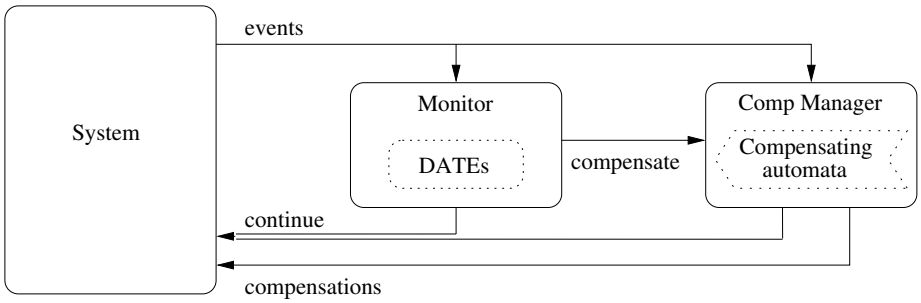


Figure 4: The monitor-oriented compensation programming architecture

continue signal (but not both).

3. The role of the compensation manager is the following:
   (a) It receives the system event and processes it, deciding whether to compensate for the event or not [answering the question of *what* to compensate for] and in case of compensation, storing an appropriate compensating action [deciding the *how* of compensations].
   (b) It then checks whether a *compensate* signal has been received from the monitor. If a *compensate* signal has been received:
      i. The compensation manager starts emitting compensations to the system.
      ii. While the system is executing compensation instructions, the monitor and compensation manager are still potentially receiving and processing system events.
      iii. When the compensation manager has sent all compensation instructions to the system (the stack is exhausted), it checks once more for the *compensate* signal and if present repeats from Point 3(b)i. Otherwise, execution continues as below.
   (c) The compensation manager issues a *continue* signal.
4. Once the system detects two *continue* signals it continues normally.

While the architecture seems to show a single *compensate* line (Figure 4), in fact this line is multiplexed to enable the monitor to choose whichever compensation strategy is applicable. Similarly, although we show a single DATEs box, in effect there would typically be several DATEs and several compensating automata. Finally, we note that although the architecture is not formally defined beyond the algorithm given above, the components of the architecture, i.e. compensating automata and DATEs, have been fully formalised in previous work.

The following section aims to showcase the benefits of this highly modular architecture through a case study. More concrete details about the architecture are subsequently provided in Section 5.

4 An eProcurement Case Study

The use of MOCP as proposed in this paper is particularly useful when the decision to activate a compensation strategy is not straightforward. One example where this is the case is an implementation of an e-procurement system[1] which handles payments and shipments of goods. The e-procurement system allows users to create virtual credit cards and subsequently (the emphasised words are system events observable by the monitors):

[1] Inspired from the e-procurement system presented in [20] and the Entropay system presented in [12].

- *Load* money onto the virtual credit cards from their personal bank accounts.
- *Transfer* money across virtual credit cards.
- *Order* goods from a third party using the e-procurement system as an intermediary. This involves concurrently *paying* the third party through the bank and *booking a courier* (either courier *A* or *B* depending on availability).

In this case study, we assume that the system as described above is already implemented and that the relevant events can be detected by a monitor when they occur. Note that there is no mention of compensations in the above description since all compensation logic will be handled by the monitor in a MOCP fashion. In the rest of this section we thus elaborate on the monitor logic which handles compensation programming and triggering. Finally, we conclude the section with a discussion of the case study.

4.1 Compensation Logic for the eProcurement Case Study

During the normal activities of the eProcurement system described above, a number of possible failures might occur (elaborated further below) causing the transaction to fail. Cancelling a courier or reversing bank transactions usually incur a charge and under certain circumstances it might not be possible to cancel such operations at all, *e.g.*, when the shipment has already left, (corresponding to *what* to compensate for). Furthermore, assuming that compensation is possible, there are at least three parties who might incur the charge (corresponding to the *how* of compensations): the user, the third party involved (*e.g.*, the courier or the bank), or the intermediary, i.e. the e-procurement system. This means that for one courier booking transaction, there are at least three possible compensation strategies (corresponding to *C1*, *C2*, and *C3* in Figure 4) whose choice can only take place after the error actually occurs (as it depends on the kind of error). Note that in all cases compensations are discarded with no replacement once the shipment of goods occurs, *i.e.*, the booking cannot be cancelled after shipment.

Similarly, for the banking transactions there are at least three possible compensations plus a fourth one which keeps track of the credit cards used so that these can be blocked in case of blacklisted users (corresponding to *B1*, *B2*, *B3*, and *B4* in Figure 4). Note that in this case, once payment succeeds, we assume that the money loads and transfers are not to be undone.

To decide *which* compensation strategy to trigger *when*, the e-procurement system classifies users and errors so that the charge is incurred by different parties under different circumstances:

Classification of errors The e-procurement system policy broadly considers three kinds of errors: a bank error, a courier error, or a user cancellation. In the case of a user cancellation it is always the user who should pay for cancellation charges. However, in case of a banking error, the cancellation charges for the banking transactions are incurred by the bank while the cancellation charges for the courier transaction are incurred by the user or the e-procurement system depending on the type of user (see next point). A similar approach is taken in case of a courier error. Failure detection is handled by the two (simplified) monitors depicted in Figure 5(a) and (b). These monitors listen for system events such as *payment* or *bookCourier* (more insight into event capturing is given in the next section). If after a number of retries the system fails to perform the successful event, the monitor communicates *bankError* or *courierError* to the main monitor shown in (d).

Classification of users Users are classified as whitelisted, greylisted, or blacklisted. Blacklisted users are suspicious users who cannot be trusted. For this reason these users are not automatically given back their money in case of a failure. Instead, their credit cards are blocked till after human investigation. On the other hand, whitelisted users are trusted customers for whom certain allowances are made such

as paying cancellation charges on their behalf. Greylisted users are users who are neither blacklisted nor whitelisted. Each user starts off as greylisted and at a particular point in time a designated monitor (partly shown in Figure 5(c)) might classify the user as blacklisted or whitelisted.

Note that such an e-procurement strategy for handling cancellation includes nine different compensation strategies depending on three user types and three kinds of failure. Clearly, deciding the compensation strategy for such a scenario is not straightforward. By separating the different aspects of compensation programming, the decision can be taken using monitors as depicted in Figure 5(d) where compensation strategies (shown in square brackets) can be composed in parallel or sequentially depending on the case. For example, in case a blacklisted user cancels the transaction, the payment of the relevant charges should strictly occur before the credit cards are blocked.

4.2 Discussion

The challenge of expressing non-trivial compensation logic in a modular and understandable fashion has been particularly highlighted by Greenfield et al. [20], arguing that "compensation is not enough". As a solution the authors [27] propose an approach which is the complete opposite of ours in terms of a model which does not differentiate between normal and exceptional behaviour, and abstracts away from the notion of compensations. They claim that this approach, which is based on a guarded-command language, simplifies the specification of the system. Admittedly, the examples given in [27] are very readable since the model is somewhat similar to a textual specification with many statements of the form: "If this happens, then this should happen". The disadvantage of this approach is that the "compensation view" is lost, i.e. the relationship between an action and its compensation, the ordering of compensation execution, etc., is not visible from the model. If such a view is not necessary for the programmer, then, indeed, this model should be preferred. However, the compensation view has its advantages: (i) the correspondence between actions and their compensations is useful since one would usually require part of the system state at the time of executing the action to be available during the execution of the compensation; (ii) compensation notations usually clearly encode sequential constraints amongst actions and compensations, information which is useful for ensuring temporal properties. If these advantages are important in a given context, then employing compensating automata might provide the right balance between the compensation view and flexibility.

Using MOCP we have successfully shown how non-trivial compensation logic, comparable to that presented in [20], can be programmed transparently to the underlying system in a modular fashion.

5 Using the Architecture in Practice

While the previous section describes an account of how the architecture is applicable to a case study, in this section we elaborate on the practical steps which one would have to take to adopt the approach. Although the architecture has not been fully implemented, the monitor component has been fully implemented and has been previously applied independently to real-life financial transaction systems [13, 12] written in Java, while the compensation manager is currently being implemented. In what follows we take the current implementation into consideration and comment on how it can be extended to make it fully functional.

Programming the compensation code Due to the nature of compensations, which are not necessarily the exact reversal of the action being compensated for, the compensation code has to be coded by a programmer. However, the advantage of the proposed architecture is that the compensation code can be programmed in separate modules (see Figure 6) and eventually executed by the compensation manager.

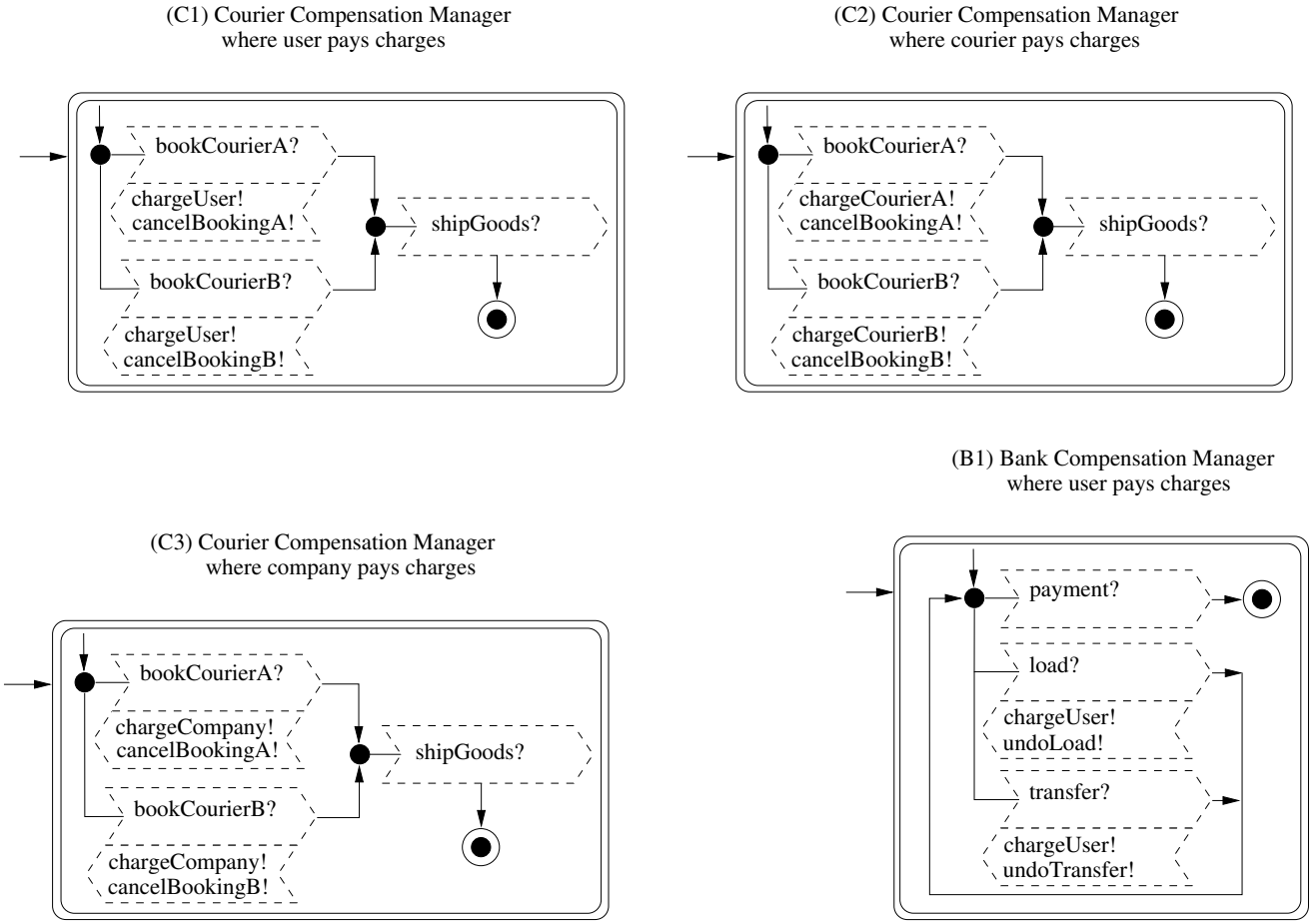


Figure 5: Managing compensations for an e-procurement system (continued on the next page)

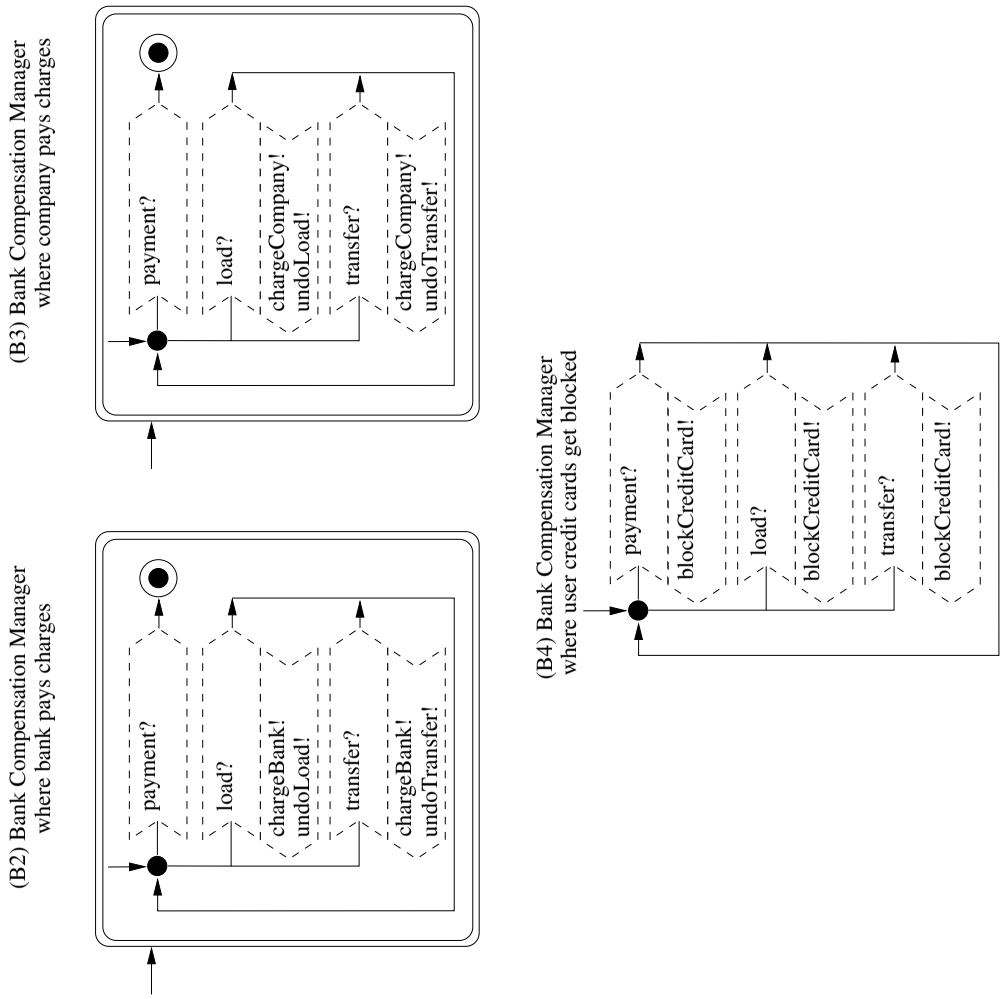


Figure 4: Managing compensations for an e-procurement system

Capturing events from system execution As a first step, the compensation manager has to be aware of the system's behaviour so that it can react accordingly. In the context of Java systems, the system's behaviour is captured through aspect-oriented programming; with the behaviour typically consisting of method calls, method returns, and exception throws. Different events may be captured depending on the type of system.

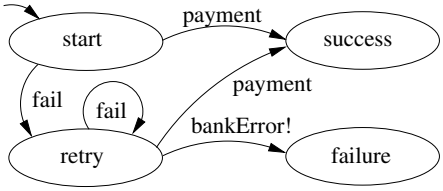
Passing of relevant system state to the compensation module Since the compensation would generally execute on a different (future) system state than the action it is meant to compensate for, it is crucial that any relevant state is passed on for future reference. For example in the case of a payment this might include the account balance at the time of payment (e.g., if a different refund fee applies depending on the balance), and the amount paid. In practical terms, such state would be stored as part of the monitor state and passed on as parameters when invoking the compensation code.

Enabling the automatic invocation of compensation code While the architecture depiction in Figure 4 simply shows a channel output to represent compensation code invocation, an implementation would require a means by which the compensation manager can take over the system execution. To this extent, our standalone monitoring tool LARVA [14] utilises aspect-oriented programming to inject custom code in a Java system. The same approach can be applied in the implementation of the proposed architecture.

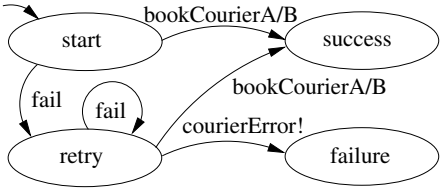
6 Related Work

Monitor-oriented programming (MOP) [25] has been proposed as a programming paradigm advocating separation of concerns through monitoring. Somewhat similar to aspect-oriented programming in principle, it differs in that matching occurs through satisfaction of some formal logic rather than source code pattern matching. Inspired by MOP, we have proposed monitor-oriented compensation programming (MOCP) where monitors are used for a specific kind of programming, *i.e.,* compensation programming. Recall that compensation programming can be split in the programming of four elements: (i) *when* — when to start compensating; (ii) *what* — what system actions to compensate for; (iii) *how* — how to compensate for the designated system actions; and (iv) *which* — which compensation strategy is chosen if there are more than one way of compensating an action or a sequence thereof. Thus, a fundamental difference between MOP and MOCP is that while MOP matches a pattern to decide *when* to execute a particular logic, in our case MOCP is concerned not only with the *when* but also with the other questions since compensations are programmed on-the-fly while monitoring and are based on the control flow. MOCP achieves this added expressivity by combining two automata specifications: compensating automata providing the *what* and *how* aspects, and DATEs providing the *when* and *which* elements.

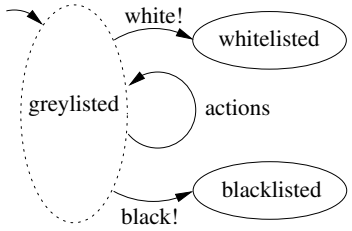
In the areas of autonomous adaptation and self-healing, one finds much related work [2, 4, 8, 15, 18, 22, 26, 29, 30] which has been done in the context of the service-oriented architectures. Of these approaches, only a subset [2, 4, 8, 29, 30] support the possibility of executing compensations. These works (with the exception of AO4BPEL [8]) provide a policy language which is able to separate exception handling (and compensation) concerns from the normal business logic but the policy language itself does not go into the details of compensation programming. In other words, the policy language enables a user to specify *what* and *when* to compensate but it does not provide explicit support for specifying *how* and *which*. Instead, they assume there is a single default compensation for every action. These approaches are referred to as 'policy languages' in Table 1. On the other hand, AO4BPEL provides an aspect-oriented framework which enables the programming of crosscutting concerns such as compensation programming. However, the language does not provide explicit support for programming



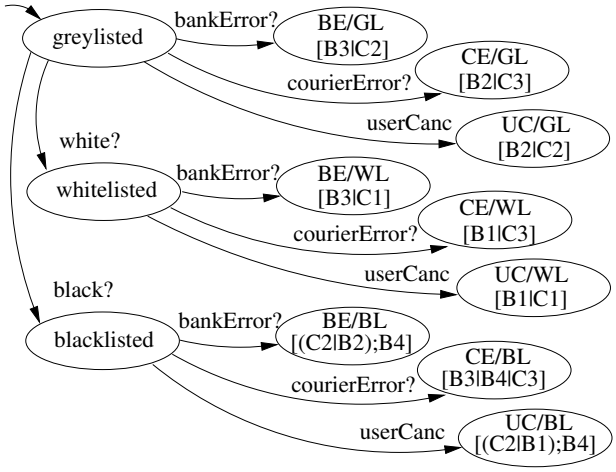
(a) Monitoring for bank errors



(b) Monitoring for courier errors



(c) Monitor for whitelisting and blacklisting users



(d) Receiving signals for other monitors and triggering compensation strategies

Figure 5: Monitoring for triggering compensations for an e-procurement system

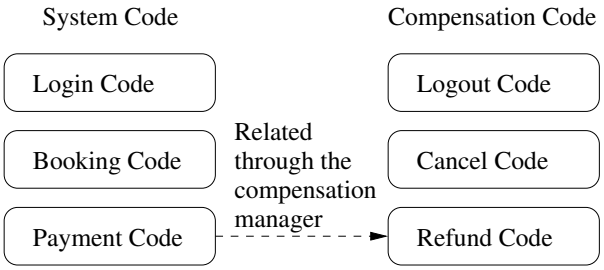


Figure 6: The relation between system code and compensation code through the compensation manager

compensations and is thus left out of Table 1.

Most theoretical frameworks which support the specification of compensations [1, 5, 9, 17, 19, 21, 23, 24] (referred to as 'theoretical frameworks' in Table 1), enable the user to specify *what* and *how* to compensate while the compensation is activated automatically upon an external failure/signal. On the contrary, other approaches including BPEL and cCSP [3, 7] go further to also enable the user to programmatically invoke the compensation activation, *i.e.*, deciding *when* to execute the installed compensations. For example in the case of BPEL, compensation is optionally invoked from the exception handler enabling full user control of whether or not to execute compensations. However, all these approaches ignore the *which* question because they assume that there is a single possible compensation strategy.

StAC$_i$ [6], on the other hand, gives full control to the user and several compensation strategies can be programmed concurrently (each with an individual compensation stack) and subsequently the user is allowed to program *when* to run *which* compensation stack. This means that StAC$_i$ supports all four aspects of compensation programming which are supported by MOCP. However, MOCP differs from StAC$_i$ in two fundamental ways:

- MOCP clearly separates compensation specification from compensation activation concerns: with compensating automata able to specify *what* and *how* while DATEs can be used for specifying *when* and *which*. We believe that this separation, together with the higher level of abstraction provided through both automaton-based notations, makes compensation programming more manageable.
- Furthermore, MOCP is a monitor-based approach and thus separates compensation concerns from the other concerns. On the other hand, StAC$_i$ requires the programmer to not only program all four aspects of compensation programming but also program the rest of the system concerns including exception handling.

Table 1 summarises the capabilities of the reviewed compensation programming approaches.

7 Conclusions

Whilst compensation programming is frequently understood to answer the questions of what and how to compensate, in more complex scenarios, it is also not straightforward to decide when to trigger compensation strategies, and if a system has more than one compensation strategy, which one to choose. Thus the problem of programming complex compensations can be split into two main aspects: the *what and how* of compensations — dealing directly with programming what actions to use as compensations — and the *when and which* of compensations which manages the interplay of normal execution and compensation execution.

	System	Compensation			
		when	what	how	which
Compensating Automata			✓	✓	
DATEs		✓			✓
MOCP		✓	✓	✓	✓
Policy Languages		✓	✓		
Theoretical Frameworks	✓		✓	✓	
BPEL, cCSP	✓	✓	✓	✓	
$StAC_i$	✓	✓	✓	✓	✓

Table 1: Compensation specification approaches vs. expressivity

We are currently implementing compensation automata as part of the runtime verification suite Larva, over which we will be implementing our approach and applying it to the case study discussed in this paper where a number of different compensation strategies would need to be maintained concurrently and executed under different contexts. A possible optimisation is to discard compensation strategies which will surely not be used. Taking the example of the case study, as soon as a user is classified as whitelisted, compensation strategies *B2*, *B4*, and *C2* can be discarded, saving the memory and the time to maintain them.

References

[1] (2008): *Business Process Modeling Notation, v1.1*. `http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf` (Last accessed: 2010-02-17).

[2] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici & Pierluigi Plebani (2007): *PAWS: A Framework for Executing Adaptive Web-Service Processes*. *IEEE Software* 24, pp. 39–46.

[3] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri & A. Yiu (2007): *Web Services Business Process Execution Language Version 2.0*. OASIS Standard. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (Last accessed: 2010-02-17).

[4] L. Baresi & S. Guinea (2011): *Self-Supervising BPEL Processes*. *Software Engineering, IEEE Transactions on* 37(2), pp. 247 –263.

[5] Roberto Bruni, Hernán Melgratti & Ugo Montanari (2005): *Theoretical foundations for compensations in flow composition languages*. In: *POPL*, ACM, pp. 209–220.

[6] Michael J. Butler & Carla Ferreira (2004): *An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions*. In: *COORDINATION, LNCS* 2949, Springer, pp. 87–104.

[7] Michael J. Butler, C. A. R. Hoare & Carla Ferreira (2004): *A Trace Semantics for Long-Running Transactions*. In: *25 Years Communicating Sequential Processes*, LNCS, Springer, pp. 133–150.

[8] Anis Charfi & Mira Mezini (2007): *AO4BPEL: An Aspect-oriented Extension to BPEL*. *World Wide Web* 10(3), pp. 309–344.

[9] M. Chessell, C. Griffin, D. Vines, M. Butler, C. Ferreira & P. Henderson (2002): *Extending the concept of transaction compensation*. *IBM Systems Journal* 41(4), pp. 743–758.

[10] Christian Colombo & Gordon Pace (2013): *Monitor-Oriented Compensation Programming Through Compensating Automata*. *ECEASST* 58.

[11] Christian Colombo & Gordon Pace (2013): *Recovery within Long Running Transactions*. *ACM Computing Surveys* 45.

[12] Christian Colombo, Gordon Pace & Patrick Abela (2012): *Safer asynchronous runtime monitoring using compensations*. Formal Methods in System Design 41(3), pp. 269–294.

[13] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2008): *Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties*. In: *FMICS*, *LNCS* 5596, pp. 135–149.

[14] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*. In: *SEFM*, IEEE, pp. 33–37.

[15] Massimiliano Colombo, Elisabetta Di Nitto & Marco Mauri (2006): *SCENE: a service composition execution environment supporting dynamic changes disciplined through rules*. In: *ICSOC*, Springer, pp. 191–202.

[16] Charles T. Davies, Jr. (1973): *Recovery semantics for a DB/DC system*. In: *ACM annual conference*, ACM, pp. 136–141.

[17] Christian Eisentraut & David Spieler (2008): *Fault, Compensation and Termination in WS-BPEL 2.0 - A Comparative Analysis*. In: *WS-FM*, *LNCS* 5387, Springer, pp. 107–126.

[18] Abdelkarim Erradi, Piyush Maheshwari & Vladimir Tosic (2007): *WS-Policy based Monitoring of Composite Web Services*. In: *ECOWS*, IEEE, pp. 99–108.

[19] Dirk Fahland & Wolfgang Reisig (2005): *ASM-based Semantics for BPEL: The Negative Control Flow*. In: *ASM*, pp. 131–152.

[20] Paul Greenfield, Alan Fekete, Julian Jang & Dean Kuo (2003): *Compensation is Not Enough*. In: *EDOC*, IEEE, pp. 232–239.

[21] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi & Gianluigi Zavattaro (2006): *SOCK: A Calculus for Service Oriented Computing*. In: *ICSOC*, *LNCS* 4294, Springer, pp. 327–338.

[22] Sam Guinea, Gabor Kecskemeti, Annapaola Marconi & Branimir Wetzstein (2011): *Multi-layered Monitoring and Adaptation*. In: *Service-Oriented Computing*, *LNCS* 7084, Springer, pp. 359–373.

[23] Yanxiang He, Liang Zhao, Zhao Wu & Fei Li (2008): *Formal Modeling of Transaction Behavior in WS-BPEL*. In: *CSSE*, IEEE, pp. 490–494.

[24] Alessandro Lapadula, Rosario Pugliese & Francesco Tiezzi (2008): *A Formal Account of WS-BPEL*. In: *COORDINATION*, *LNCS* 5052, Springer, pp. 199–215.

[25] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen & Grigore Roşu (2012): *An Overview of the MOP Runtime Verification Framework*. Journal on Software Tools for Technology Transfer 14(3).

[26] Oliver Moser, Florian Rosenberg & Schahram Dustdar (2010): *Event driven monitoring for service composition infrastructures*. In: *WISE*, Springer, pp. 38–51.

[27] Surya Nepal, Alan Fekete, Paul Greenfield, Julian Jang, Dean Kuo & Tony Shi (2005): *A service-oriented workflow language for robust interacting applications*. In: *On the Move to Meaningful Internet Systems - Part I*, Springer, pp. 40–58.

[28] B. Randell, P. Lee & P. C. Treleaven (1978): *Reliability Issues in Computing System Design*. ACM Computing Surveys 10, pp. 123–165.

[29] Michael Schäfer, Peter Dolog & Wolfgang Nejdl (2007): *Engineering compensations in web service environment*. In: *ICWE*, Springer, pp. 32–46.

[30] Nick Amirreza Tahamtan & WS-Diamond team (2007): *WS-DIAMOND: Web Services - DIAgnosability, MONitoring and Diagnosis*. In: *E. di Nitto, A. Sassen, P.Traverso and A. Zwegers (Eds), At your service, Chapter 9*, MIT Press.