Electronic Communications of the EASST
Volume 58 (2013)

Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013)

Monitor-Oriented Compensation Programming Through Compensating Automata

Christian Colombo and Gordon J. Pace

12 pages

Guest Editors: Matthias Tichy, Leila Ribeiro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Monitor-Oriented Compensation Programming Through Compensating Automata

Christian Colombo[1] **and Gordon J. Pace**[2]

[1] christian.colombo@um.edu.mt [2] gordon.pace@um.edu.mt
Dept. of Computer Science, University of Malta, Malta

Abstract: Compensations have been used for decades in areas such as flow management systems, long-lived transactions and more recently in the service-oriented architecture. Since compensations enable the logical reversal of past actions, by their nature they crosscut other programming concerns. Thus, intertwining compensations with the rest of the system not only makes programs less well-structured, but also limits the expressivity of compensations due to the tight coupling with the system's behaviour.

To separate compensation concerns from the normal system behaviour, we propose compensating automata, a graphical specification language dedicated to compensation programming. Compensating automata are subsequently employed in a monitor-oriented fashion to program compensations without cluttering the actual system implementation. This approach is shown applicable to a complex case study which existing compensation approaches have difficulty handling.

Keywords: compensations, runtime verification, monitor-oriented programming

1 Introduction

Computer systems have been growing in size and complexity for decades, making it virtually impossible for such systems to be faultless. On the other hand, their role in sensitive human activities have put higher pressures on their correctness. Consequently, fault tolerance techniques such as backward error recovery [RLT78] started being incorporated so that systems could withstand failures. Backward recovery (*e.g.*, rollback in traditional databases) has the advantage of being automatable but cannot be used when a system interacts with real-life processes (*e.g.*, a bank transfer) — such processes cannot be simply undone and forgotten. In such cases, instead of undoing some actions, one might actually need to execute further "counter-actions", better known as *compensations*. For example in the case of a bank account transfer, one might have to add a processing fee over and above the return of funds to the original account. Compensations have thus become particularly useful in areas which program real-life processes such as in flow management systems, long-lived transactions, and more recently web services. To facilitate programming such interactions, several approaches have been proposed along the years with the current de facto industry standard being the Business Process Execution Language (BPEL) [AAB+07]. Moreover, extensive research [CP12] has been conducted in the area, particularly by suggesting formal models of compensation [BF04, BHF04, LMMT08] and defining BPEL semantics (*e.g.*, [HZWL08]) in which compensations play a crucial role.

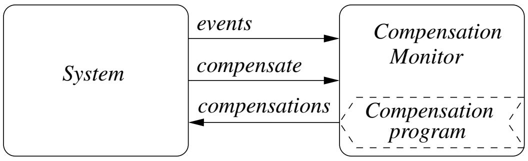
Since compensations enable the logical reversal of past actions and are, by their nature, history-based, while typical recovery is usually programmed statically (*e.g.*, *try-catch-block*), compensation programming has to be programmed dynamically to take into consideration the current execution path (which is to be reversed). Programming compensations statically is possible but limits their expressivity [LVF10]. On the other hand, programming compensations dynamically using traditional means of code organisation results in unstructured code which has to continuously keep track of the history. A recurring approach for compensation programming [CP12] involves associating compensation blocks to corresponding system blocks in a *try-catch-block* fashion which cannot handle complex compensation logic [GFJK03].

A natural way of programming dynamic elements while preserving separation of concerns is runtime verification [BGR07, MJG+12]. The main contribution of this paper is an instantiation of the monitor-oriented programming (MOP) paradigm [MJG+12] for compensation programming: monitor-oriented compensation programming (MOCP) (§ 2) in which compensations are programmed separately from the system, and incorporated within a compensation monitor which listens for relevant system events and programs compensations accordingly. A central aspect of MOCP is the specification language which programs the monitor. Due to the advantages that automata bring to monitoring, namely that monitoring states are directly programmable/visible to the user, we propose compensating automata (§ 3) — a novel graphical compensation notation. Finally, we show how MOCP can be applied to a sophisticated e-procurement system from the literature (§ 4) which existing compensation notations have difficulty in handling [GFJK03].

2 Programming Compensations

To highlight the challenges of programming compensations we use the e-procurement system (EPS) case study presented in [GFJK03]. Interestingly, programming the EPS when everything "works fine" is relatively straightforward but it quickly becomes complex when handling "unexpected" behaviour *e.g.*, cancellation by a customer, an unavailable third party system, software or hardware crashes, *etc.*. This observation leads us to believe that strictly separating normal behaviour from abnormal behaviour helps to keep the complexity manageable. By extension, we also propose separating abnormal compensations from other recovery code which is not compensation-based since there is a significant difference in their programming. To this effect, using a strict interpretation of "compensations", *i.e.*, that they are logical reverses of corresponding activities, a substantial number of the failure handling aspects of the EPS do not fall within the remit of compensations (*e.g.*, stopping or pausing a business process or trying an alternative transport company). On the other hand, giving a refund, and returning shipped goods are compensations. In fact, by our definition, compensation features of the EPS can be described in three points: *(i)* If a user cancellation is received after goods have been reserved and transportation arranged but before an invoice has been sent and the goods shipped, then the order can be cancelled by running compensations in reverse chronological order for those activities that have successfully executed; *(ii)* If an order is cancelled, then the cancellation fee is dependent on the state of the delivery: if there is a fee for the cancellation of delivery, then the costs are passed onto the customer; if an invoice has not been sent, then an invoice for the cancellation fee is sent to the customer; instead, if an invoice has been sent and payment has been received, then a partial refund is sent to the customer; *(iii)* If goods were delivered to the wrong address, the shipment should be returned.

To facilitate programming of compensations we propose a complete separation of concerns with the system not keeping track of *how* and *what* to compensate but simply knows *when* compensations are to be triggered. Such a design pattern necessitates a compensation monitor which listens for system activities and programs compensations accordingly. As soon as the system communicates with the compensation monitor that it needs to start compensating, the compensation monitor takes over and communicates to the system which (compensating) actions need to be executed. The figure below shows the overall setup with the system communicating events to the compensation monitor and the latter communicating compensations if it receives a signal on the *compensate* line.



In order to program the compensation monitor, we propose a dedicated formalism which is solely concerned with programming compensations. Based on the literature [CP12], a compensating formalism should support two main activities: that of programmatically collating compensations and that of activating them. Collating compensations usually involves installing compensations with the possibility of replacing previously installed compensations. Once compensations are activated, the mechanism should allow for an indication that the compensations have been applied, and that the system can resume execution (and the compensation monitor to revert back to collating compensations). In what follows we informally introduce and motivate the constructs of the proposed compensation notation, *compensating automata*:

Basic compensation installations The formalism should allow actions to be designated as compensations for other actions. Subsequently, upon completion of an action, the corresponding compensation is pushed onto a stack. If compensation is invoked, the actions on the stack are executed in the reverse order of their counterparts. In the EPS, this corresponds to a number of examples such as "unreserving" previously reserved goods. This is depicted in Fig. 1(a)[left][1] where the automaton transitions upon the *ReserveGoods* event while installing the compensation *UnreserveGoods*. Some activities such as sending a quotation might not have a compensation (*e.g.*, Fig. 1(a)[middle]). Similarly, we allow automatic installation of compensations with no associated event by annotating the forward arrow with τ (see Fig. 1(a)[right]).

Replacing compensations Compensations may have to be replaced at some point and therefore it should be possible to delimit compensation patterns which upon being matched are replaced by another compensation. For example, as soon as the goods are shipped, then it no longer makes sense to cancel the transport arrangement. Instead, this is replaced by the shipment back of the goods once they reach their destination. Compensation replacement is depicted in Fig. 1(b) where an automaton monitoring transport arrangement is scoped so that when the goods are shipped, any accumulated compensations (*CancelA* or *CancelB*) are discarded and replaced by the *ReturnGoods*.

[1] We use the following abbreviations: trans (transport), canc (cancellation), addr (address), unv (unavailable), notf (notification), rec (receive), gds (goods), delv (delivery), ack (acknowledgement), pay (payment).

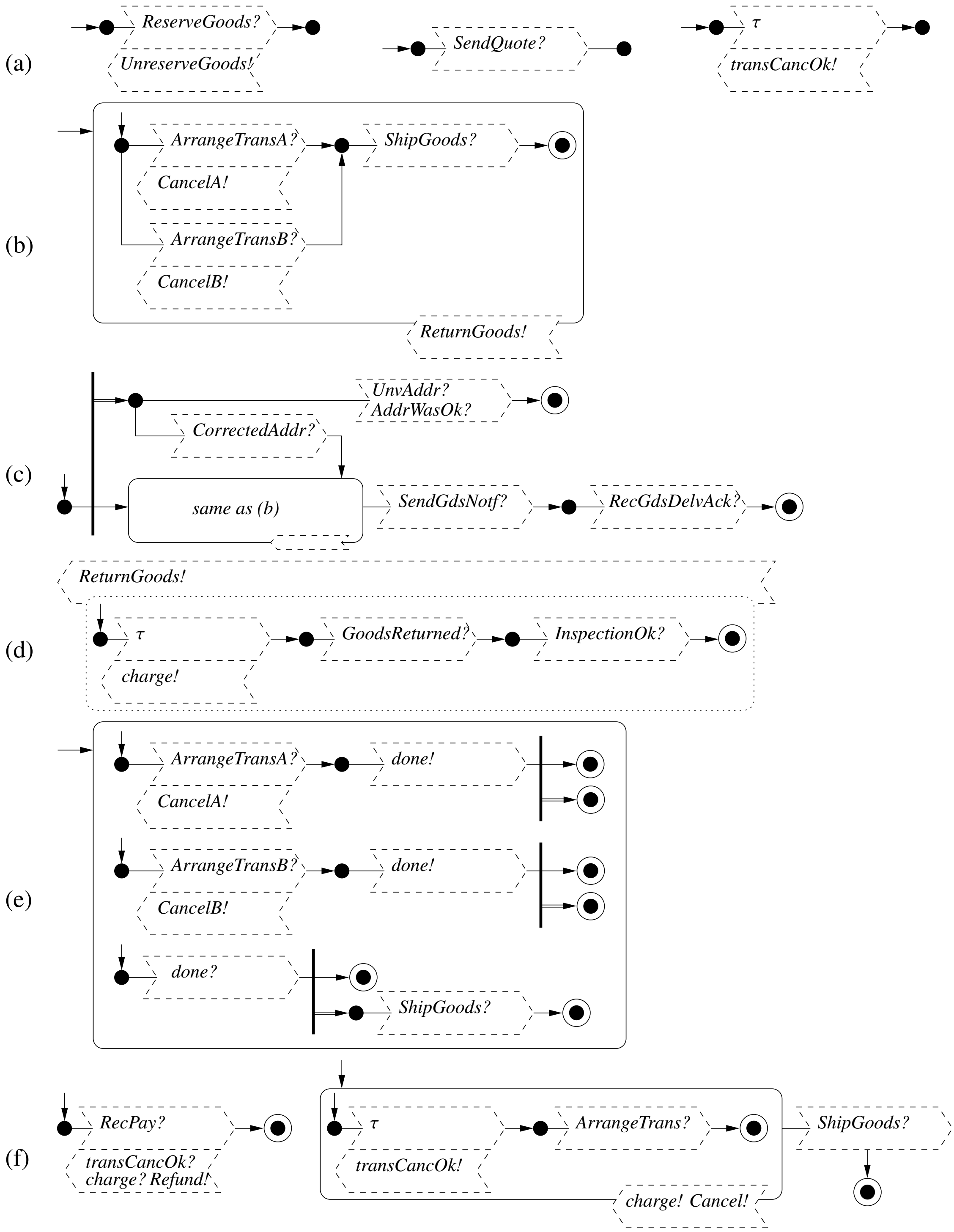


Figure 1: Examples of different compensation constructs

Stopping compensation activation Sometimes a business process should not be reversed completely. For example if goods have been shipped, but returned (*e.g.*, the goods notification remains unacknowledged), the system checks whether the reason is a wrong address. If it is the case, the compensation should be temporarily suspended — *deviated* to another state — until the system attempts to verify the address was correct. If the address was correct then the system signals the monitor to continue compensating. Otherwise, the system should continue by re-attempting shipping to the corrected address while the compensation monitor continues to collate compensations from the "deviated-to" state onwards. A deviation is shown in Fig. 1(c) as a bold line with two outgoing arrows: a plain arrow which is taken on the first traversal and a double arrow taken on a deviation.

Compensations having compensations Compensation actions may have compensations themselves. For example, while goods are being shipped back (the compensation in the previous point), compensations might still be needed since the process might also fail at some stage. As depicted in Fig. 1(d), the *ReturnGoods* action is expected to give rise to a number of system events including the shipment of the goods back to the supplier and inspecting the goods to ensure that they are as expected. Note that if inspection fails, then the compensation of the shipment would involve a charge to the customer.

Concurrent communicating compensation handlers To enable better decomposition of compensations, concurrent compensation handlers can synchronise. For example a more efficient way of booking transport is to start the booking process with the two shipping companies and then cancel one as soon as the other is confirmed. Known as *speculative choice*, this can be encoded by communicating automata as shown in Fig. 1(e). Communication can also be used during compensation execution. In our example, the refund operation has to wait for the transport cancellation (if this has taken place) so that any fees incurred can be passed on to the user. Fig. 1(f) shows the payment and the transport automata where the *Refund* has to wait for either *charge* or *transCancOk* (signifying transport cancellation charge and no charge respectively).

Following from this informal introduction, the next section gives the formal syntax and semantics of compensating automata.

3 Formalising Compensating Automata

A compensating automaton (CA) is intended to enable the user to program the compensation monitor so that depending on the sequence of system events, compensations are collated and possibly later executed if the system signals *compensate*. As such a CA should not only be aware of, but also able to affect system activities.

Definition 1 A CA event I is a non-empty subset of activities in an alphabet Σ, $I \subseteq \Sigma$, and triggers if one of the activities $i \in I$ is observed by the monitor. We take τ to be a special event which immediately triggers without external stimulus, and use Σ_τ to denote $\Sigma \cup \{\tau\}$. A CA action O is a set of monitor-executable activities, $O \subseteq \Sigma$, which are carried out concurrently. CAs can run concurrently and may need to communicate. This is achieved by distinguishing between the set of system activities Σ_S and the set of local activities Σ_L (where $\Sigma_S \cap \Sigma_L = \emptyset$ and $\Sigma_S \cup \Sigma_L = \Sigma$).

Once a system event is detected by the monitor, a CA takes transitions to move through the

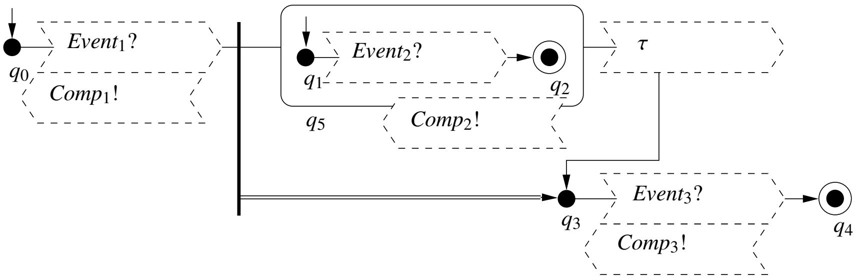


Figure 2: An example of a compensating automaton

automaton states. For such an event, a transition defines a compensation which should be installed to compensate for it. To enable local communication, transitions can also trigger over local events and can dually trigger local events themselves. Moreover, transitions may sometimes specify a third state to support deviation. CAs' states may be of two types: basic or nested. Nested states support compensation scoping, *i.e.*, compensations which expire and are thus discarded and replaced with a specified compensation. Finally, a compensation in the context of CAs is an action, which may include both system and local activities, and may itself have programmed compensations in terms of CAs.[2]

Definition 2 A CA is a quintuple $A = (\Sigma_\tau, Q, \delta, q_0, F)$: an alphabet Σ_τ, a set of states Q, a set of transitions δ, an initial state $q_0 \in Q$, and a set of final states $F \subseteq Q$. We use $\mathcal{A}$ to represent the set of CAs and $\widehat{\mathcal{A}}$ for the set of vectors of CAs. We will use variables $A, A' \in \mathcal{A}$ to range over CAs and $\widehat{A}, \widehat{A'} \in \widehat{\mathcal{A}}$ to range over vectors of CAs. Furthermore, $Q = B \cup N$ where N and B represent nested states and basic states respectively. $q \in N$ is a tuple $\widehat{\mathcal{A}} \times C$ where C is the compensation used for replacing the compensations of the nested vector.

A transition $t \in \delta$ is a quintuple: a source state, an event-action tuple together with their compensation, and a destination state — $t \in (Q \times 2^{\Sigma_\tau} \times 2^{\Sigma_L} \times C \times Q)$. A deviating transition t' is a sextuple consisting of a source state, an event-action tuple together with their compensation, a deviation state, and a destination state — $t' \in (Q \times 2^{\Sigma_\tau} \times 2^{\Sigma_L} \times C \times Q \times Q)$. To simplify the presentation of the semantics, we represent non-deviating transitions as deviating transitions with blank state $\circ$. Using $\mathcal{D} = Q \cup \{\circ\}$, the set of transitions δ is a subset of the Cartesian product $(Q \times 2^{\Sigma_\tau} \times 2^{\Sigma_L} \times C \times \mathcal{D} \times Q)$. We will write *event*$(t)$ and *src*(t) to respectively refer to the event and source state of a transition t. Finally, a compensation $c \in C$ is an element of: $(2^{\Sigma_{L\tau}} \times 2^{\Sigma} \times \widehat{\mathcal{A}})$.

Example 1 *As an example of a CA we consider Fig. 2 where state q_5 contains a nested automaton with initial state q_1, final state q_2 and a single transition with event* Event2*, no action, and no compensation. The parent automaton has three basic states, q_0, q_3, and q_4 and a nested state. Furthermore, it has three transitions one of which is a deviation connecting q_0 to the nested state and q_3 as the deviating state. Note that compensation* Comp_1 *is installed upon receiving* Event_1, Comp_2 *upon completion of the nested automaton, and* Comp_3 *upon receiving* Event_3.

Since a CA may have nested vectors of automata, its configuration should be correspond-

[2] For a more detailed presentation the reader is referred to [Col12]

ingly nested by a vector of configurations. Each basic (non-nested) configuration has to keep track of: (*i*) the automaton and the state it is in; (*ii*) the execution "direction" — whether it is currently collating compensations (conceptually the system state is progressing *forward*) or activating compensations (conceptually the system state is progressing *backwards*); and (*iii*) the collated compensation in terms of a stack, including the installed compensations and deviations. The execution direction is represented by $\mathcal{R} \overset{\text{def}}{=} \{\ogreaterthan, \olessthan\}$, signifying forward and backward execution respectively, while a stack $S \in \mathcal{S}$ is a sequence of stack elements $\mathcal{S} \overset{\text{def}}{=} \mathit{Stack}^*$ where each element of type *Stack* can either be a compensation or a deviation, $\mathit{Stack} ::= \mathcal{C} \mid \mathcal{D}$.

Definition 3 A configuration is defined as: $\mathit{BConf} ::= \mathit{Basic}(\mathcal{A} \times Q \times \mathcal{R} \times \mathcal{S})$

$$\mathit{NConf} ::= \mathit{BConf} \;\mid\; \mathit{Nest}(\mathit{BConf}, \mathit{Conf}) \qquad\qquad \mathit{Conf} ::= \mathit{Vect}(\mathit{seq}\ \mathit{NConf})$$

To abbreviate we drop A when clear from the context and use $(q,S)^r$ to denote $\mathit{Basic}(A,q,r,S)$ where $r \in \mathcal{R}$ (*e.g.*, if $r = \ogreaterthan$, $(q,S)^{\ogreaterthan}$ is a forward-executing configuration); and $(q,S)^r_{\mathrm{cf}}$ to denote $\mathit{Nest}(\mathit{Basic}(A,q,r,S), \mathit{cf})$. The initial configuration of a vector of CAs is given by the *in* function which starts each automaton from its respective initial state:

$$\mathit{in}(\widehat{A}) \overset{\text{def}}{=} [(q_{0 1},[])^{\ogreaterthan}_{\mathit{in}(q_{0 1})}, (q_{0 2},[])^{\ogreaterthan}_{\mathit{in}(q_{0 2})}, \ldots, (q_{0 n},[])^{\ogreaterthan}_{\mathit{in}(q_{0 n})}]$$

Note that to handle state nesting we call the *in* function overloaded to states: if the state is nested, the configuration of the nested vector of automata is obtained by calling *in* once more, otherwise *in* returns an empty configuration.

$$\mathit{in}(q) \quad \overset{\text{def}}{=} \quad \begin{cases} \mathit{in}(\widehat{A'}) & \text{if } q = (\widehat{A'}, c) \\ [] & \text{otherwise} \end{cases}$$

A configuration reaches a point where it cannot proceed further when either the automaton has reached a final state during forward execution, *i.e.*, no further installations can occur, or all the compensations have been activated during backward execution.

Definition 4 A basic configuration cf is said to be *terminated*, written $\boxdot(\mathit{cf})$, if and only if it has reached a final state and it is in forward direction: $\boxdot((q,S)^r) \overset{\text{def}}{=} q \in F \;\wedge r = \ogreaterthan$. A basic configuration cf is said to be *compensated*, written $\boxtimes(\mathit{cf})$, if and only if it has an empty stack and it is in backward direction: $\boxtimes((q,S)^r) \overset{\text{def}}{=} S\ = [] \;\wedge r = \olessthan$. Nested configurations are neither terminated nor compensated since execution continues with the parent. A vector configuration is said to have *terminated* if all sub-configurations are either terminated or compensated and at least one has terminated[3]:

$$\boxdot([\mathit{cf}_1, \mathit{cf}_2, \ldots, \mathit{cf}_n]) \overset{\text{def}}{=} \exists i \in 1..n \cdot \boxdot(\mathit{cf}_i) \wedge \forall j \in 1..n \cdot \boxdot(\mathit{cf}_j) \vee \boxtimes(\mathit{cf}_j)$$

On the other hand, a vector configuration is said to have *compensated* if all sub-configurations have compensated: $\boxtimes([\mathit{cf}_1, \mathit{cf}_2, \ldots, \mathit{cf}_n]) \overset{\text{def}}{=} \forall i \in 1..n \cdot \boxtimes(\mathit{cf}_i)$.

Definition 5 The push operation on stacks, denoted $s \fatsemi S$, adds an element s onto the top of stack S where s is either a compensation $c \in \mathcal{C}$ or a deviation $d \in \mathcal{D}$:

[3] The decision to consider a vector configuration as terminated even if only one of the sub-configurations has terminated, is based on the fact that frequently, in various typical examples (such as the speculative choice in Fig. 1(e)) one would still want to continue programming compensations if some of the branches have compensated.

$$c \fatsemi S \overset{\text{def}}{=} c : S \qquad\qquad d \fatsemi S \overset{\text{def}}{=} \begin{cases} S & \text{if } d = \circ \\ d : S & \text{otherwise} \end{cases}$$

To ensure monitoring remains cheap, CAs should be deterministic.

Definition 6 A CA, $(\Sigma, Q, \delta, q_0, F)$, is said to be deterministic if and only if: (i) if a state has an outgoing τ transition, then it may have no other outgoing transitions:

$$\forall t, t' \in \delta \cdot (t \neq t' \wedge src(t) = src(t')) \implies \tau \notin event(t)$$

(ii) there are no outgoing transitions with shared labels from the same state:

$$\forall t, t' \in \delta \cdot (t \neq t' \wedge src(t) = src(t')) \implies event(t) \cap event(t') = \emptyset$$

(iii) once a final state is reached, no further transitions may be taken: $\forall t \in \delta \cdot src(t) \notin F$.

Furthermore, a CA is said to be well-formed if it contains no loops made up of transitions with labels over $\Sigma_L \cup \tau$.

3.1 Semantics

We give the semantics of CAs in SOS style [Plo81] in terms of a labelled transition system where states are configurations.

Definition 7 The semantics of a vector of CAs is given by the least transition relation $\xrightarrow{x}$ (with $\xrightarrow{x} \subseteq Conf \times Trace \times Conf$) satisfying the rules in Fig. 3 (using $+\!\!+$ to denote vector concatenation) where an element of *Trace* is either (i) an automaton transition, *i.e.*, an activity triggering local action in $\Sigma_\tau \times 2^{\Sigma_L}$; ($ii$) a compensation action, *i.e.*, a local activity triggering an action in $\Sigma_{L\tau} \times 2^{\Sigma}$; ($iii$) a compensate signal $\Lsh$ which switches the automaton from collating to activating compensations; and (iv) a silent transition, τ, representing the rest of the automaton activities. Thus, we define *Trace* as: $Trace ::= Forw(\Sigma_\tau \times 2^{\Sigma_L}) \mid Back(\Sigma_{L\tau} \times 2^{\Sigma}) \mid \Lsh \mid \tau$.

The first rule, Suc, deals with the basic automaton transitions such that if the transition event triggers, then the action is carried out and the compensation and deviation are pushed onto the stack. If the destination state (q') is nested, then the corresponding nested initial configuration is given by calling the *in* function. When a CA receives the *compensate* signal, denoted $\Lsh$, rule Fail turns the execution mode of the automaton from forward to backwards. Once the execution direction is backwards, if the topmost stack element is a compensation, then rule Comp pops it from the stack and activates it. Recall that each compensation action has an associated (possibly empty) vector of automata as compensation. Thus the resulting configuration is a nested configuration including the configuration for the compensation of the compensation. If the topmost stack element is a deviation, then the automaton should stop activating compensations and resume collating them. Rule Dev deviates the execution by reverting the configuration direction from backwards to forwards and sets the deviation state as the new configuration's state. Since this state may be nested, the resulting configuration is nested.

Upon successful completion of a nested vector of automata, *i.e.*, its configuration is terminated, rule NestSuc is responsible to pass control back to the parent by discarding the corresponding configuration and installing the programmed compensation. In case the nested configuration is compensated, then the parent configuration starts compensating itself. This scenario is handled by the NestFail by discarding the nested configuration and changing the parent's direction to

$$\textsc{Suc}\ \frac{}{(q,S)^{\rhd} \xrightarrow{i_O} (q',d \fatsemi c \fatsemi S)^{\rhd}_{in(q')}}\ \begin{array}{l}(q,I,O,c,d,q') \in \delta\\ i \in I\end{array} \qquad \textsc{Fail}\ \frac{}{(q,S)^{\rhd} \xrightarrow{\Lsh} (q,S)^{\lhd}}$$

$$\textsc{Comp}\ \frac{}{(q,c \fatsemi S)^{\lhd} \xrightarrow{{}_iO} (q,S)^{\lhd}_{in(\widehat{A})}}\ \begin{array}{l}c = I,O,\widehat{A}\\ i \in I\end{array} \qquad \textsc{Dev}\ \frac{}{(q,d \fatsemi S)^{\lhd} \xrightarrow{\tau} (d,S)^{\rhd}_{in(d)}}$$

$$\textsc{NestSuc}\ \frac{}{(q,S)^{\rhd}_{cf} \xrightarrow{\tau} (q,c \fatsemi S)^{\rhd}}\ \begin{array}{l}q = \widehat{A},c\\ \boxdot(cf)\end{array} \qquad \textsc{NestFail}\ \frac{}{(q,S)^{\rhd}_{cf} \xrightarrow{\tau} (q,S)^{\lhd}}\ \boxtimes(cf)$$

$$\textsc{NestComp}\ \frac{}{(q,S)^{\lhd}_{cf} \xrightarrow{\tau} (q,S)^{\lhd}}\ \boxdot(cf) \vee \boxtimes(cf) \qquad \textsc{Nest}\ \frac{cf \xrightarrow{x} cf'}{(q,S)^{r}_{cf} \xrightarrow{x} (q,S)^{r}_{cf'}}$$

$$\textsc{Vect}\ \frac{cf \xrightarrow{x} cf'}{\alpha +\!\!+ [cf] +\!\!+ \beta \xrightarrow{x} \alpha +\!\!+ [cf'] +\!\!+ \beta}$$

Figure 3: The semantic rules for compensating automata

backwards. When the parent configuration has a backward direction and the nested configuration is terminated or compensated, rule NestComp triggers and passes control back to the parent so that the latter continues with its compensation. Whenever a nested configuration is reached, the Nest rule is required to enable nested configurations to progress. Similarly, the Vect rule enables individual configurations within a vector to progress independently (α,β : *seq Conf*).

Example 2 *Referring back to the previous example, upon receiving* Event1*, the transition from* q_0 *proceeds to* q_1 *through rule* Suc *which pushes the compensation and deviation onto the stack and invokes the* in *function on state* q_5*. More interestingly, upon reaching* q_2 *with configuration* $(q_5,[q_3,(\{\tau\},\{\text{Comp1}\},[])])^{\rhd}_{(q_1,[(\{\tau\},\emptyset,[])])^{\rhd}}$*, rule* NestSuc *triggers and installs compensation* Comp2 *while discarding the nested configuration. This leads to configuration* $(q_5,[(\{\tau\},\{\text{Comp2}\},[]),q_3,(\{\tau\},\{\text{Comp1}\},[])])^{\rhd}$.

An interesting change occurs if the failure event occurs, triggering the Fail *rule. In this case the configuration direction turns backwards to* $(q_5,[(\{\tau\},\{\text{Comp2}\},[]),q_3,(\{\tau\},\{\text{Comp1}\},[])])^{\lhd}$ *and subsequently rule* Comp *begins consuming and triggering the compensations on the stack. After triggering compensation* Comp2*, the resulting topmost stack element is a deviation. This causes rule* Dev *to trigger and the configuration is turned into forward direction once more:* $(q_3,[(\{\tau\},\{\text{Comp1}\},[])])^{\rhd}$.

Upon observing system activities, the compensating monitoring system relays them to the compensating automata which potentially perform a number of steps based on the rules presented above. Importantly, these steps should respect three properties which are crucial in the context of compensation programming and runtime monitoring: (i) self-cancellation — a standard way of checking that a compensation formalism is sane [BHF04], by ensuring that the monitor-triggered compensations are the correct compensation for the actions carried out by the system, *i.e.*, assuming perfect compensations the outcome of the system behaviour plus the execution of compensations would be logically equivalent to performing no action[4]; (*ii*) stability — it is crucial that the monitor does not contain live loops which would result in keeping the system waiting forever; and (*iii*) determinism — deterministic monitoring ensures that the monitor produces the same result under equivalent observations. While the proofs for these properties cannot be included here due to space limitations, these have been carried out in full [Col12].

4 Programming with Compensating Automata

The compensation logic for the EPS has been programmed as a vector of five automata whose monitoring logic applied to the EPS would satisfy all the compensation features as described in [GFJK03]. Here we only present the part of the EPS which deals with transporting the order while giving the full detail in [Col12]. The compensation logic modelled in the compensating automaton shown in Fig. 4 can be summarised as: (*i*) if the process fails before the goods have been shipped, then the transport arrangement is simply cancelled; (*ii*) if the process fails after the goods have been shipped, then the goods are returned (using compensation scoping); (*iii*) if the return of the goods fails, then *returnFailed* communicates with the compensating automaton handling the order so that the action which restores the goods back to stock is discarded (through the compensation of the compensation); (*iv*) if the return of the goods fails, then *charge!* communicates with the compensating automaton handling the payment (not shown here) so that the customer would still pay for the items (through the compensation of the compensation); and (*v*) if the failure occurred due to an incorrect address which is later corrected, then the transport is reattempted without affecting the procurement further.

5 Related Work and Conclusion

Compensation concerns often crosscut other programming concerns and thus attempting to program compensations within the main flow of a program would clutter the program and also limit the expressivity of compensation programming. In this paper, we have presented an alternative approach to compensation programming through the use of runtime monitoring techniques — monitoring system activities to instantiate compensations. The contributions of this paper include: (*i*) an instantiation of the MOP framework to compensations, advocating complete separation of compensation concerns; (*ii*) a compensation programming notation, CAs, which includes a new compensation construct, the deviation, used to redirect compensation; (*iii*) formalisation of the syntax, semantics, and self-cancellation of these automata; and (*iv*) programming compensations for an EPS — showing CAs to be useful for handling non-trivial compensation logic.

Our work is mainly inspired by the MOP framework [MJG$^+$12] which has been proposed as a means of separating programming concerns. In this paper, we have instantiate MOP to compensation programming, *i.e.*, monitor-oriented compensation programming.

[4] Note that this definition can be considered at various levels of abstraction. In our case, we assume that the user supplies a correct list of actions and their corresponding counteractions and use this as the basis of our reasoning.

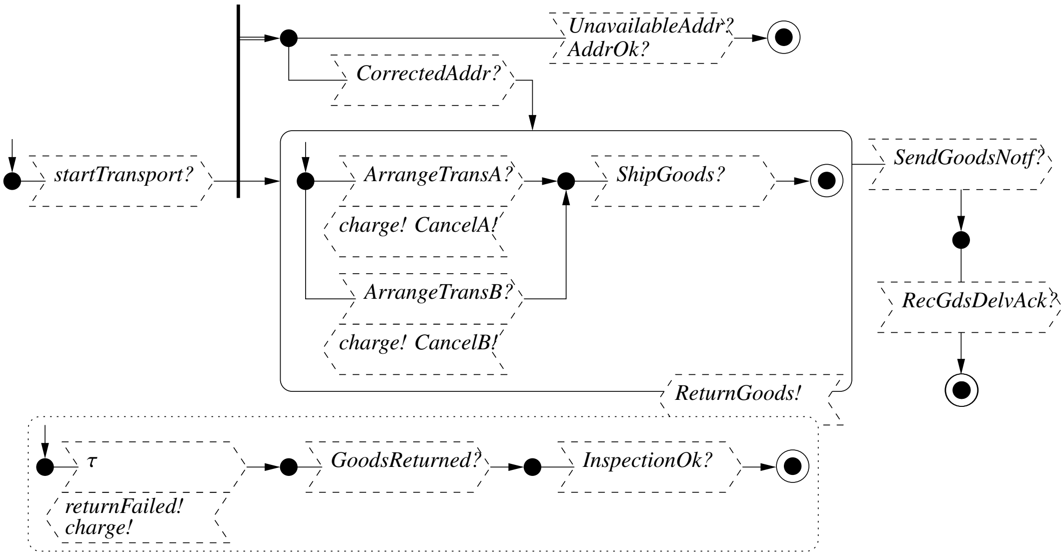


Figure 4: The compensating automaton for the transportation aspect of the EPS

Monitoring and steering of loosely coupled systems is not new [ACM+07, BG11] and has been mostly motivated by the need for autonomous adaptation and self-healing in the context of the highly dynamic service-oriented architecture. Our work differs in that while all these approaches support monitor-triggered compensations, they do not separate compensations from other concerns such as exception handling. Our strict separation of concerns allows the use of more dedicated compensation constructs such as the *deviation*, enabling a fine interplay of normal and compensation execution.

There are numerous formalisations and notations [bpm08, BF04, BHF04, HZWL08, LMMT08] which support compensations including some which are pictorial (*e.g.*, Petri net-based formalisms [HZWL08], communicating hierarchical transaction-based timed automata (CHTTA) [LMMT08], and BMPN [bpm08]). None of these formalisms are designed to support monitoring-orientation, nor do they support a notion similar to *deviation*.

A limitation of the current approach is that the system has a single feedback line through which it can signal compensations. This is a limitation when there are several different potential compensations from which the system can choose (for example see [BF04]). In the future we aim to lift this limitation by using monitoring techniques to decide which compensation should be run in the given runtime context.

Bibliography

[AAB+07] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, A. Yiu. Web Services Business Process Execution Language Version 2.0. 2007. OASIS Standard.

[ACM+07] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, P. Plebani. PAWS: A Framework for Executing Adaptive Web-Service Processes. *IEEE Software* 24:39–46, 2007.

[BF04] M. J. Butler, C. Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In *COORDINATION*. LNCS 2949, pp. 87–104. 2004.

[BG11] L. Baresi, S. Guinea. Self-Supervising BPEL Processes. *Software Engineering, IEEE Transactions on* 37(2):247 –263, 2011.

[BGR07] H. Barringer, D. Gabbay, D. Rydeheard. From Runtime Verification to Evolvable Systems. In *RV*. LNCS 4839, pp. 97–110. 2007.

[BHF04] M. J. Butler, C. A. R. Hoare, C. Ferreira. A Trace Semantics for Long-Running Transactions. In *25 Years CSP*. LNCS 3525, pp. 133–150. 2004.

[bpm08] Business Process Modeling Notation, v1.1. 2008. http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf (Last accessed: 2010-02-17).

[Col12] C. Colombo. *Runtime Verification and Compensations*. PhD thesis, Dept. of Computer Science, University of Malta, 2012.
http://staff.um.edu.mt/__data/assets/pdf_file/0013/173020/cc_phd.pdf

[CP12] C. Colombo, G. Pace. Recovery within Long Running Transactions. *ACM Computing Surveys* 45, 2012. To appear.

[GFJK03] P. Greenfield, A. Fekete, J. Jang, D. Kuo. Compensation is Not Enough. In *EDOC*. Pp. 232–239. 2003.

[HZWL08] Y. He, L. Zhao, Z. Wu, F. Li. Formal Modeling of Transaction Behavior in WS-BPEL. In *CSSE*. Pp. 490–494. 2008.

[LMMT08] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, A. Troina. Design and verification of long-running transactions in a timed framework. *Sci. Comput. Program.* 73:76–94, 2008.

[LVF10] I. Lanese, C. Vaz, C. Ferreira. On the Expressive Power of Primitives for Compensation Handling. In *ESOP*. LNCS 6012, pp. 366–386. 2010.

[MJG+12] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu. An Overview of the MOP Runtime Verification Framework. *STTT* 14:249–289, 2012.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical report, Dept. of Computer Science, Aarchus University, Denmark, 1981. DAIMI FM-19.

[RLT78] B. Randell, P. Lee, P. C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys* 10:123–165, 1978.