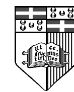
TECHNICAL REPORT

Report No. CS2012-01
Date: April 2012

Christian Colombo
Gordon J. Pace



Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
http://www.cs.um.edu.mt

Christian Colombo
Department of Computer Science
University of Malta
Msida, Malta
christian.colombo@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

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 cross-cut 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 forward behaviour of the system, we propose a novel design paradigm in which compensations are programmed separately from the system, and incorporated within a compensation manager following relevant system events and manages compensations. If the system signals the need to be compensated, the manager triggers the execution of compensations on behalf of the system and subsequently returns control to the system. We show that this approach can be used to program a sophisticated real-life case study which existing compensation approaches have difficulty in handling.

Separating Compensation Concerns and Programming them with Compensating Automata

Christian Colombo
Department of Computer Science
University of Malta
Msida, Malta
christian.colombo@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

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 cross-cut 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 forward behaviour of the system, we propose a novel design paradigm in which compensations are programmed separately from the system, and incorporated within a compensation manager following relevant system events and manages compensations. If the system signals the need to be compensated, the manager triggers the execution of compensations on behalf of the system and subsequently returns control to the system. We show that this approach can be used to program a sophisticated real-life case study which existing compensation approaches have difficulty in handling.

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 error recovery [RLT78] started being incorporated so that systems could withstand failures. In this context, two main techniques were proposed: backward recovery which backtracks to an earlier (correct) state of the system and forward recovery which attempts to repair the current system state. While forward recovery is ideal in that it does not lose any recent state modifications, reparation might not always be possible. For example, when a system crashes, the only way to recover would generally involve loading the last persisted state. Backward recovery, on the other hand, can be automated as in the case of a rollback in traditional databases. However, backward recovery cannot be used when some processes with which the system interacts do not support backward error recovery. Typical examples are real-life processes such as bank account transfers, shipping, etc. Such processes cannot be simply undone and forgotten. In such cases, instead of undoing some actions, one might actually need to execute further "counteractions", 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, while in the case of shipping one might need to ship some items back. Note that while a compensation is conceptually a form of backward recovery, at a lower level of abstraction it is actually forward recovery since the action and the counteraction do not cancel each other out (as in other backward recovery techniques such as a rollback), rather the system would keep record of both the action and the counteraction.

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, enabling loosely-coupled interactions across entities. To facilitate programming such interactions, several notations and architectures have been proposed along the years with the current de facto industry standard being the Business Process Execution Language (BPEL) [AAB+07]. From an academic point of view, extensive research [CP12] has been conducted in the area, particularly by suggesting different formal models of compensation [BMM05, BF04, BHF04, LMSMT08, GLG+06] and defining formal semantics for BPEL [ES08, FR05, HZWL08, LPT08] in which compensations play a crucial role.

Since compensations enable the logical reversal of past actions, by their nature they cross-cut other programming concerns. For example consider a payment which should be refunded free of charge if the customer has earlier bought some items but against a

charge if not. Programming such a compensation from basic principles would require some form of record of the customer's history and a mechanism through which the refund action is associated to the payment action as its compensation. Such additions clutter the code and intertwine the programming of the system actions with their compensations.

A recurrent approach for structuring compensation logic [CP12] involves associating a compensation blocks to corresponding system blocks in a *try-catch-block* fashion. However, as with the try-catch-block for exception handling, this approach still has difficulty in expressing highly cross-cutting concerns such as compensations spanning different modules (see [GFJK03] for an example). In the case of exception handling, this limitation has been overcome by technologies and approaches such as aspect-oriented programming [Kic05] and monitoring-oriented programming [MJG$^+$11]. In the case of compensation programming, we propose a novel design paradigm (Section 2) in which compensations are programmed separately from the system, and incorporated within a compensation manager which listens for relevant system the collated compensations. If the system signals the need to be compensated, the manager triggers the execution of the collated compensations on behalf of the system and subsequently returns control to the system. Furthermore, we propose compensating automata (Section 3) as a means of programming compensations and show how these can be used to program compensations for a sophisticated e-procurement system (Section 4) adapted from [GFJK03] which existing compensation notations have difficulty in handling. We conclude (Section 5) with a discussion on the implementation and security concerns of our approach.

2 Programming Compensations

Attempting to program complex compensations using standard compensation formalisation has been shown to be impractical for particular case studies [GFJK03]. On the other hand, programming without compensations is significantly more straightforward than programming with compensations. As an example we refer to the e-procurement scenario adapted from [GFJK03]. In the first subsection we show that programming the e-procurement scenario without handling compensation concerns is straightforward. Next, we describe a novel design paradigm which enables compensations to be programmed separately — leaving the system code uncluttered.

2.1 An E-Procurement Case Study

The bare logic of a procurement starts by receiving a quote request from a customer. The merchant then checks that the customer is a valid customer — that is, registered and with no overdue payments. For invalid customers, a message informing the customer that he or she is no longer a valid customer is sent and the business process terminates. For valid customers, a quote is calculated and sent to the customer. If the customer chooses to proceed with the order, the customer sends a purchase order to the merchant. Upon its receipt, the merchant reserves the ordered goods and concurrently initiates the payment and delivery processes. The payment process consists of the merchant sending an invoice, receiving payment and issuing a receipt. The delivery process consists of arranging transportation, shipment of goods, sending notification to the customer that ordered goods are now in transit and receiving an acknowledgement that goods have been received by the customer. The transaction is considered complete once the delivery and payment processes have completed. The the vanilla version of the e-procurement system (S) may be defined in terms of three programs: reserve goods (R), payment (P), transport (T) such that the parallel composition of P and T follows R, written $S = R; (P \mid T)$, and each program may be expressed as follows:

Program R

```
RecQuoteReq
If !checkCustomer
Then
  sendInvalidCustomerMsg
  Return
Calculate Price
Send Quote
RecPurchaseOrder
ReserveGoods
```

Program P

```
SendInvoice
RecPayment
SendReceipt
```

Program T

```
ArrangeTransport
ShipGoods
SendGoodsNot
RecGoodsDeliveredAck
```

However, there are numerous ways in which the business process may diverge from the expected behaviour. This may happen for several reasons e.g. explicit cancellation by the customer, software or hardware crashes, loss of communication, third-party system failure, etc. Some such divergences one would like to handle are listed below:

1. If the customer decides to cancel the order before the merchant has reserved the goods, the business process can be simply terminated.

2. 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.

3. If ordered goods have already been shipped then a cancellation process will require the invocation of a return goods process, i.e. it would arrange the delivery for the unwanted goods back from the customer. It would also involve an inspection to make sure that the goods returned were the ones originally delivered. Notice that the return goods process may itself fail and this needs to be appropriately handled by charging the customer an extra fee if the delivery or inspection fails.

4. 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 for 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; if an invoice has been sent and payment has been received, then a partial refund is sent to the customer.

5. The merchant can choose whether to accept or reject the user-initiated cancellation requests. During the time the merchant needs to decide, the transaction should be paused to avoid race conditions.

6. If one transportation company cannot deliver the order then the merchant can find an alternative.

7. An activity may fail due to a network or remote server failure. In this case, the most practical handling of such a failure (such as to affect payment) is to periodically retry the activity until it succeeds.

8. If the delivery of goods were sent to the wrong address. The shipment is returned, the merchant attempts to determine the correct address, and if successful, resends the shipment.

9. When a merchant cannot provide all the goods at the time of delivery, the merchant ships the available goods and later it arranges transport of the other goods when they become available. Consequently, the invoice to the customer is not for the full amount but only for goods that have been shipped. A later invoice is sent when the unavailable goods are shipped.

In view of the strict interpretation of compensation programming we advocate — that compensations are logical reverses of corresponding activities, a substantial number of the features of the e-procurement system do not fall within the realm of compensations. For example stopping or pausing the business process, trying an alternative transport company, or retrying an activity are not compensation operations. On the other hand, giving a refund, and returning shipped goods are compensations. In fact,

only items 2–4 and partially item 8 (from the above list) give any information about compensations. The rest of the items should be reflected in the system code.

Considering component R, we rename it to R' and add a loop to handle partial shipments of the order (item 9). Updating component P to P', we add a loop to retry payment receipt when this fails (item 7). Similarly, we update program T to T' to handle transport alternatives (item 6) and reshipping in case of a wrong address (item 8).

Program R'

```
RecQuoteReq
If (!checkCustomer)
Then
  SendInvalidCustomerMsg
  Return
CalculatePrice
SendQuote
RecPurchaseOrder
ReserveGoods
While (!allGoodsAvailable)
  (P | T)(partialOrder)
(P | T)(fullOrder)
```

Program P'

```
SendInvoice
RecPayment
While (!RecPayment
  && (NetworkIsDown
    || BankIsDown))
  Wait
  RecPayment
SendReceipt
```

Program T'

```
If (!ArrangeTransport(A))
Then ArrangeTransport(B)
ShipGoods
SendGoodsNot
RecGoodsDeliveredAck
If (!RecGoodsDeliveredAck)
VerifyAddress
  If (UnavailableAddress
       || AddressOk)
    Compensate
  Else If (CorrectedAddress)
    Run(T')
```

Thus, we have effectively defined a new system, $S' = R'; P' \mid T'$, which however, still does not address all the features. Particularly, the system does not pause to process user cancellations and stop if the cancellation is approved (items 1 and 5). To address this limitation, we add a process U which receives user cancellations and is able to pause and resume the system.

Program U

```
If (UserCancel)
  Pause
  VerifyCancelValidity
  If (cancelValid) then
    Compensate
  Else
    Continue
```

By combining component U with S', we get an e-procurement system, $S'' = U \mid S'$, which knows when compensation is required but does not know what or how to compensate. We propose to program compensations separately through the proposed architecture on which we elaborate in the following subsection.

2.2 Proposed Compensation Design Paradigm

To facilitate programming of compensations we propose a complete separation of concerns with the system being completely unaware of *how* and *what* to compensate

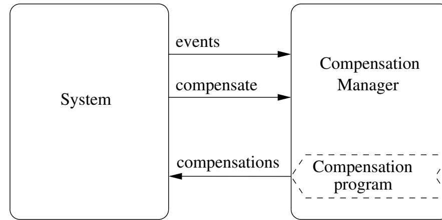


Figure 1: The architecture with programmed compensations.

but it only knows *when* there is need to start compensating. Such a design pattern necessitates a compensation manager which listens for system activities and collates compensations accordingly. As soon as the system communicates with the compensation manager that it needs to start compensating, then the compensation manager takes over and communicates to the system which actions need to be executed. Fig. 1 shows the overall setup with the system communicating events to the compensation manager and the latter communicating compensations if it receives a signal on the *compensate* line.

In order to program the compensation manager, we propose a dedicated formalism which is solely concerned with constructing compensations. Based on the literature [CP12], a compensating formalism should allow 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 should resume execution (and the compensation manager to revert back to collating compensations). Furthermore, the user can also collate compensations during the execution of compensations so that failed compensations can be compensated. Finally, the user might need to collate/execute compensations concurrently. Of all these features, the system is responsible for signalling the switch from collating compensations to activating compensations. The rest of the features should be programmable through the formalism proposed as explained below with examples:

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 in the stack are executed in the reverse order of their counterparts. In the e-procurement scenario, this corresponds to a number of examples such as “unreserving” previously reserved goods. This is depicted in Fig. 2(a)[left] where the automaton transitions upon the *ReserveGoods* event while installing the compen-

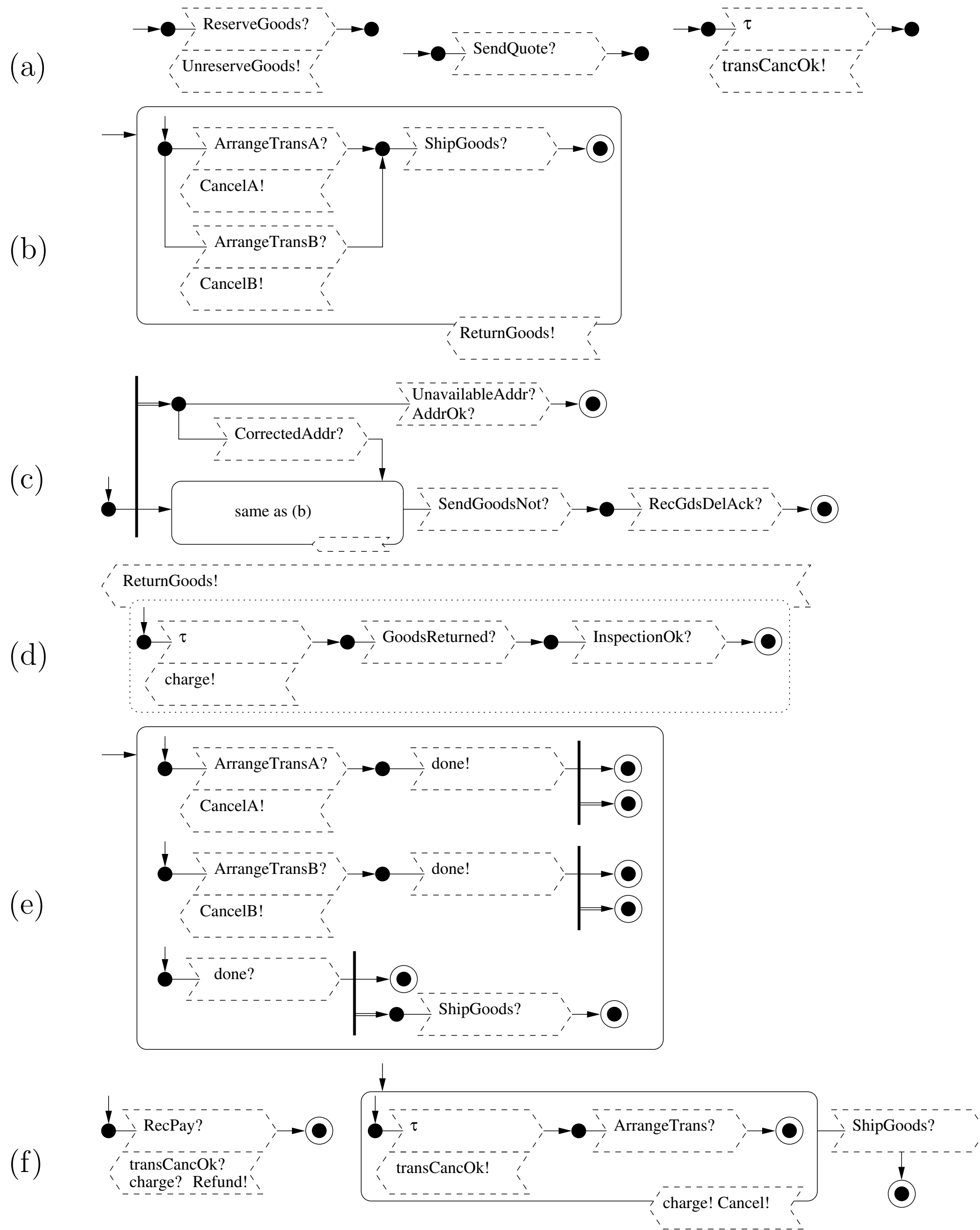


Figure 2: Examples of different compensation constructs.

sation *UnreserveGoods*. Some activities such as sending a quotation might not have a compensation. These are depicted as in Fig. 2(a)[middle]. Similarly, we allow for the automatic installation of compensations with no associated event. The forward arrow in such cases is annotated with τ — see e.g. Fig. 2(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 should be 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. 2(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 compensation *ReturnGoods*.

Stopping compensation activation Sometimes a business process should not be reversed completely. For example if the 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 compensation to continue. Otherwise, the system should continue by re-attempting shipping to the corrected address while the compensation manager continues to collate compensations from the deviated state onwards. A deviation is shown in Fig. 2(c) as a bold line with two outgoing arrows: a plain arrow which is taken on the first traversal and a double arrow which points to the deviation state.

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. 2(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 the expected goods in the expected condition. Note that if inspection fails, then the compensation of the shipment would involve a charge to the customer's credit card.

Concurrent communicating compensation handlers To enable better decomposition of compensation management, particularly for specifying compensation sequences which should run in parallel, it is more convenient to have multiple concurrent compensation handlers which can communicate. For example a more efficient way of arranging for transport might be to start the booking process with the two shipping companies and then cancel one as soon as the other is confirmed. Better known

as a *speculative choice*, this can be encoded by communicating automata as shown in Fig. 2(e). Communication can also be used to synchronise 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. 2(f) shows the payment automaton and the transport automaton where the *Refund* compensation has to wait for either the *charge* signal or the *transCancOk* signal signifying a transport cancellation charge and no charge respectively. Note that we use labels starting in lower caps for local communication.

In the next section we formalise compensating automata, giving their syntax and semantics.

3 Formalising Compensating Automata

A compensating automaton is intended to enable the user to program the compensation manager so that depending on the sequence of occurring system activities, compensations are collated and possibly later executed if the system signals compensation. As such a compensating automaton should not only be aware of, but also able to carry out system activities.

Definition 1 *A compensating automaton event is defined in terms of a set of system activities Σ and triggers if one of the system activities is received. We take τ to be a special event which immediately triggers and use Σ_τ to denote $\Sigma \cup \{\tau\}$. A compensating automaton event I is thus a non-empty disjunction of system activities, $I \in 2^{\Sigma_\tau}$. I is said to trigger upon a system activity i, if and only if $i \in I$.*

A compensating automaton action O is a set of system activities, $O \in 2^{\Sigma}$, which the automaton can instruct the system to carry out concurrently.

Compensating automata 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 (assuming that $\Sigma_S \cap \Sigma_L = \emptyset$ and $\Sigma_S \cup \Sigma_L = \Sigma$), with the latter only used internally across automata.

Once an event triggers, a compensating automaton takes transitions to move through the automaton states.

Definition 2 *A compensating automaton is composed of 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 $F \subseteq Q$. Thus, a compensating automaton is a quintuple $A = (\Sigma_\tau, Q, \delta, q_0, F)$. We*

use $\mathcal{A}$ to represent the type of compensating automata and $\widehat{\mathcal{A}}$ for the type of vectors of compensating automata. We will use variables $A,\ A' \in \mathcal{A}$ to range over compensating automata and $\hat{A},\ \hat{A}' \in \widehat{\mathcal{A}}$ to range over vectors of automata.

To support compensation scoping, each state $q \in Q$ may be nested with compensating automata. When the nested automata complete, their accumulated compensations are discarded and replaced. Thus, q is a tuple $\widehat{\mathcal{A}} \times \mathcal{C}$ where $\mathcal{C}$ is the compensation used for replacing the compensations of the vector. A state with no nested automata is called a basic state. We use N to represent nested states and B to represent basic states such that $Q = B \cup N$.

A transition across compensating automata states defines an event upon which the transition is taken and what compensation should be installed for that event. To enable local communication, a transition event can be a local event and can also trigger a local action. Thus a transition t is a quintuple: a source state, an event-action tuple together with their compensation, and a destination state — $t \in (Q \times 2^{\Sigma_\tau} \times 2^{\Sigma_L} \times \mathcal{C} \times Q)$.

Furthermore, compensating automata allow activated compensations to be interrupted so that the process is allowed to attempt a deviation. Thus, some transitions specify a third state from where the automaton continues after stopping the compensation process. 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 — $t' \in (Q \times 2^{\Sigma_\tau} \times 2^{\Sigma_L} \times \mathcal{C} \times Q \times Q)$.

To simplify semantics, we represent non-deviating transitions as deviating transitions with a blank state, $\circ$. Using $\mathcal{D} = Q \cup \{\circ\}$, the set of transitions, δ, is a subset of the Cartesian product $(Q \times 2^{\Sigma_\tau} \times 2^{\Sigma_L} \times \mathcal{C} \times \mathcal{D} \times Q)$.

We will write $event(t)$ and $src(t)$ to respectively refer to the event and source state appearing in a transition t.

Next, we define what compensations are in the context of compensating automata.

Definition 3 *A compensation is an action which may include both system and local activities. To enable concurrent compensations to synchronise, the action is guarded by a local event. Furthermore, the compensation may itself have programmed compensations in terms of a vector of compensating automata which collate compensations while the compensation executes. More formally a compensation c of type $\mathcal{C}$ is an element of the Cartesian product: $(2^{\Sigma_{L\tau}} \times 2^{\Sigma} \times \widehat{\mathcal{A}})$.*

Since a compensating automaton may have nested vectors of automata, its configuration should be correspondingly nested by a vector of configurations. Each basic configuration, i.e. a non-nested configuration, has to keep track of: (i) the state it is

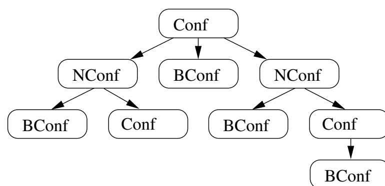


Figure 3: An example of a structure or configurations.

in; (ii) 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, including the installed compensations and the points at which compensation activation should be deviated.

Definition 4 *A configuration of a compensating automaton is either (i) a basic configuration composed of an automaton of type $\mathcal{A}$, a state of type Q, a (forward or backward) direction of type $\mathcal{R}$, and a stack of type $\mathcal{S}$; (ii) a nested configuration consisting of a basic configuration and the configuration of the nested automata vector; or (iii) a vector configuration encoded as a sequence of nested configurations. This can be summarised as:*

$$\begin{aligned} \text{BConf} &::= \text{Basic}(\mathcal{A} \times Q \times \mathcal{R} \times \mathcal{S}) \\ \text{NConf} &::= \text{BConf} \mid \text{Nest}(\text{BConf}, \mathit{Conf}) \\ \mathit{Conf} &::= \text{Vect}(\text{seq NConf}) \end{aligned}$$

A compensating automaton is either executing forward, or backward. Thus, an automaton execution direction, is of type $\mathcal{R} \stackrel{def}{=} \{\rightharpoonup, \leftharpoonup\}$ signifying forward and backward execution respectively.

A stack $S : \mathcal{S}$ is a sequence of stack elements $\mathcal{S} \stackrel{def}{=} \text{seq}\ Stack$ where each element of type Stack can either be a compensation or a deviation $Stack ::= Comp\ \mathcal{C} \mid Dev\ \mathcal{D}$.

An example of a structure of configurations is shown in Fig. 3 including a vector of three configurations with two of them being nested configurations. The first nested configuration consists of basic configuration and an empty vector while the second consists of a basic configuration and a vector containing a basic configuration: *[(BConf,[]), BConf, (BConf,[BConf])]*. As abbreviations for configurations we use $\tilde{A}[q]^S$ to denote $Basic(A, q, \sim, S)$ where $\sim \in \mathcal{R}$ (for instance if $\sim = \rightharpoonup$, $\vec{A}[q]^S$ signifies a forward executing configuration); and $\tilde{A}[q]^S_{cnf}$ to denote $Nest(\tilde{A}[q]^S, cnf)$.

Definition 5 *The push operation, denoted $s \rhd S$, adds an element s onto the top of*

stack S where s is either a compensation $c : \mathcal{C}$ or a deviation $d : \mathcal{D}$:

$$c \rhd S \stackrel{def}{=} (Comp\ c) : S$$
$$d \rhd S \stackrel{def}{=} \begin{cases} S & \text{if } d = \circ \\ (Dev\ d) : S & \text{otherwise} \end{cases}$$

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 6 *A basic configuration cnf is said to be* terminated*, written $\boxdot(cnf)$, if and only if it has reached a final state and it is in forward direction: $\boxdot(\tilde{A}[q]^S) \stackrel{def}{=} q \in F \wedge \sim = \rightharpoonup$.*

A basic configuration cnf is said to be compensated*, written $\boxtimes(cnf)$, if and only if it has an empty stack and it is in backward direction: $\boxtimes(\tilde{A}[q]^S) \stackrel{def}{=} S = [] \wedge \sim = \leftharpoonup$.*

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:*
$\boxdot([cnf_1, cnf_2, \ldots, cnf_n]) \stackrel{def}{=} \exists i \in 1..n \cdot \boxdot(cnf_i) \wedge \forall j \in 1..n \cdot j \neq i \Rightarrow \boxdot(cnf_j) \vee \boxtimes(cnf_j)$

On the other hand, a vector configuration is said to have compensated *if all sub-configurations have compensated: $\boxtimes([cnf_1, cnf_2, \ldots, cnf_n]) \stackrel{def}{=} \forall i \in 1..n \cdot \boxtimes(cnf_i)$.*

Definition 7 *The initial configuration of a vector of compensating automata is given by the* init *function which starts each automaton from its respective initial state with an empty stack: $init(\hat{A}) \stackrel{def}{=} [goto(A_1, q_{01}, []), goto(A_2, q_{02}, []), \ldots, goto(A_n, q_{0n}, [])]$*

Function goto *returns the configuration of a compensating automaton, starting at a particular state with a particular stack. If the function* goto *is called on a basic state, then the configuration returned is a basic configuration. Otherwise, if the state is nested, the configuration of the nested vector of automata is obtained through a recursive use of* init*:*

$$goto(A, q, S) \stackrel{def}{=} \begin{cases} \vec{A}[q]^S & \text{if } q \in B \\ \vec{A}[(\hat{A}', c)]^S_{\text{init}(\hat{A}')} & \text{if } q = (\hat{A}', c) \in P \end{cases}$$

Since compensating automata are intended for monitoring systems, determinism is crucial to ensure that monitoring remains cheap.

Definition 8 *A compensating automaton* $(\Sigma, Q, \delta, q_0, F)$ *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) 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 compensating automaton 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 compensating automata in SOS style [Plo81] in terms of a labelled transition system where states are configurations.

Definition 9 *The semantics of a vector of compensating automata is the least transition relation* $\xrightarrow{x}$: *Conf* $\times$ *Trace* $\times$ *Conf satisfying the rules in Fig. 4(top) where an element of type trace is either (i) an automaton transition, i.e. an activity triggering local action, of type* $\mathcal{A} \times \Sigma_\tau \times 2^{\Sigma_L}$*; (ii) a compensation action, i.e. a local activity triggering an action, of type* $\mathcal{A} \times \Sigma_{L\tau} \times 2^{\Sigma}$*; (iii) a compensate signal* $\urcorner$ *which switches the automaton from collating to activating compensations; and (iv) a silent transition,* τ*, representing the rest of the automata activities. Thus, we define the type Trace as Trace* $::= \text{Forw}(\mathcal{A} \times \Sigma_\tau \times 2^{\Sigma_L}) \mid \text{Back}(\mathcal{A} \times \Sigma_{L\tau} \times 2^{\Sigma}) \mid \urcorner \mid \tau$*. As abbreviation, we write* $\text{Forw}(A, i, O)$ *as* $(i_O)_A$ *and* $\text{Back}(A, i, O)$ *as* $({}_iO)_A$ *(subscripting the local symbols).*

The first rule, Suc*, deals with the basic automaton transitions such that if the transition event triggers, then the transition action is carried out and the compensation and deviation are pushed onto the stack. If the destination state (*q'*) has a nested automaton, then the corresponding nested initial configuration is given by calling the* goto *function on automaton* A *with* q' *as the state.*

When the compensating automaton receives the compensate *signal, denoted by* $\urcorner$*, 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 programmed compensation. Thus the resulting configuration is a nested configuration including the configuration for the programmed compensation. If the topmost stack element is a deviation, then this signifies that 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 have nested automata, the new configuration is obtained through the goto *function.*

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 changes the parent's direction to backwards. When the parent configuration has a backward direction, when 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 ($\alpha, \beta : \text{seq } Conf$).*

Upon receiving a system activity or the compensate signal, a vector of compensating automata triggers the relevant transitions followed by other non-(directly-)system-triggered transitions. All the steps triggered through a single system signal (directly or not) are collectively called a *big step*. To give the semantics of communication within compensating automata, the big step is split into small steps where each step allows one transition to occur, taking into account local communication.

Definition 10 *Small steps are specified on small-step configurations ssConf which are each composed of a vector configuration plus a set of local symbols: $ssConf \in (Conf \times 2^{\Sigma_L})$. The set of local symbols is used to keep track of the local communication. There are five kinds of small steps (each described by a rule in Fig. 4(bottom)): two which are directly triggered as a result of system signal (rules* Sys *and* Fal*) while there are three kinds of transitions which are not directly system-triggered, namely: (i) silent transitions (rule* Sil*); and (ii) local- or (iii) tau-triggered communication (rules* Loc *and* Tau *respectively). Note that local symbols are added to the configuration whenever local communication triggers (rules* Sys*,* Tau*, and* Loc*). Furthermore, local communication (rule* Loc*) can only trigger if a corresponding symbol is present in the configuration set of symbols.*

Grouping small steps into big steps, we use the notion of an *exhaustive transitive closure* of a relation $\rightarrow$. Denoted $\twoheadrightarrow\!\!\bullet$, the exhaustive transitive closure is defined to be the maximal transitive closure with no further possible compositions; more formally, $\twoheadrightarrow\!\!\bullet \stackrel{\text{def}}{=} \{(\alpha, \beta) \in \rightarrow^* \mid \beta \notin dom(\rightarrow)\}$.

Basic Configurations

$$\textsc{Suc}\ \frac{}{\overrightarrow{A}[q]^{S} \xrightarrow{(i_O)_A} goto(A, q', d \rhd c \rhd S)} \quad \begin{array}{l}(q, I, O, c, d, q') \in \delta \\ i \in I\end{array}$$

$$\textsc{Fail}\ \frac{}{\overrightarrow{A}[q]^{S} \xrightarrow{\Lsh} \overleftarrow{A}[q]^{S}} \qquad \textsc{Comp}\ \frac{}{\overleftarrow{A}[q]^{((I,O),\widehat{A}) \rhd S} \xrightarrow{({}_iO)_A} \overleftarrow{A}[q]^{S}_{Init(\widehat{A})}}\ i \in I$$

$$\textsc{Dev}\ \frac{}{\overleftarrow{A}[q]^{d \rhd S} \xrightarrow{\tau} goto(A, d, S)}$$

Nested Configurations

$$\textsc{NestSuc}\ \frac{}{\overrightarrow{A}[(\widehat{A}, c)]^{S}_{cnf} \xrightarrow{\tau} \overrightarrow{A}[(\widehat{A}, c)]^{c \rhd S}}\ \boxdot(cnf) \qquad \textsc{NestFail}\ \frac{}{\overrightarrow{A}[q]^{S}_{cnf} \xrightarrow{\tau} \overleftarrow{A}[q]^{S}}\ \boxtimes(cnf)$$

$$\textsc{NestComp}\ \frac{}{\overleftarrow{A}[q]^{S}_{cnf} \xrightarrow{\tau} \overleftarrow{A}[q]^{S}}\ \boxdot(cnf) \vee \boxtimes(cnf) \qquad \textsc{Nest}\ \frac{cnf \xrightarrow{x} cnf'}{\tilde{A}[q]^{S}_{cnf} \xrightarrow{x} \tilde{A}[q]^{S}_{cnf'}}$$

Vector Configurations

$$\textsc{Vect}\ \frac{cnf \xrightarrow{x} cnf'}{\alpha \mathbin{++} [cnf] \mathbin{++} \beta \xrightarrow{x} \alpha \mathbin{++} [cnf'] \mathbin{++} \beta}$$

Small Steps

$$\textsc{Sys}\ \frac{cnf \xrightarrow{(i_O)_A} cnf'}{cnf_{[L]} \xhookrightarrow{(iO)_A} cnf'_{[L \cup O]}}\ i \in \Sigma_S \qquad \textsc{Fal}\ \frac{cnf \xrightarrow{\Lsh} cnf'}{cnf_{[L]} \xhookrightarrow{\Lsh} cnf'_{[L]}} \qquad \textsc{Sil}\ \frac{cnf \xrightarrow{\tau} cnf'}{cnf_{[L]} \xhookrightarrow{\tau} cnf'_{[L]}}$$

$$\textsc{Tau}\ \frac{cnf \xrightarrow{(\tau_O)_A} cnf' \ \ or\ \ cnf \xrightarrow{({}_\tau O)_A} cnf'}{cnf_{[L]} \xhookrightarrow{(\tau O)_A} cnf'_{[L \cup (O \cap \Sigma_L)]}} \qquad \textsc{Loc}\ \frac{cnf \xrightarrow{(l_O)_A} cnf' \ \ or\ \ cnf \xrightarrow{({}_lO)_A} cnf'}{cnf_{[L]} \xhookrightarrow{(lO)_A} cnf'_{[L \cup (O \cap \Sigma_L)]}}\ l \in L$$

Figure 4: Basic semantic rules (top), small steps (bottom)

Definition 11 *Given a sequence $e : E$ of system activities and failure signals, $e \in \Sigma \cup \{\urcorner\}$, the behaviour of a compensation manager programmed with vector of automata $\hat{A}$, is characterised by two phases.*

1. *Starting from the initial configuration $init(\hat{A})$, the compensation manager performs all possible non-system-triggered small steps. More formally, the first phase is characterised by $\overset{\tau}{\Rightarrow}\bullet$ where $\overset{\tau}{\Rightarrow}$ is one of the three non-system-triggered small steps.*

$$
\begin{aligned}
cnf_{[L]} \overset{\tau}{\Rightarrow} cnf'_{[L']} \quad \overset{def}{=} \quad & cnf_{[L]} \overset{\tau}{\hookrightarrow} cnf'_{[L]} \\
\vee \quad & cnf_{[L]} \xhookrightarrow{(\tau O)_A} cnf'_{[L \cup O]} \\
\vee \quad & cnf_{[L]} \xhookrightarrow{(lO)_A} cnf'_{[L \cup O]}
\end{aligned}
$$

2. *Execution continues by repeatedly consuming elements from $e : E$ until either the sequence of system activities and failures is fully consumed or the automata vector cannot proceed further. Processing each e element, denoted by an e-big-step involves triggering all possible transitions waiting on e and collecting all the triggered local actions in the configuration set of symbols — $\xhookrightarrow{(eO_1)_{A_1}(eO_2)_{A_2}\dots(eO_n)_{A_n}}\bullet$ or $\xhookrightarrow{\urcorner\urcorner\dots\urcorner}\bullet$ depending on e. This is followed by a $\overset{\tau}{\Rightarrow}\bullet$ so that the automata vector exhausts all the internal small steps. Note that to ensure that each automaton makes only one step, each automaton $A_1 \dots A_n$ must be unique.*

$$
\begin{aligned}
cnf_{[\emptyset]} \overset{e}{\Rightarrow} cnf'_{[L]} \quad \overset{def}{=} \quad & cnf_{[\emptyset]} \xhookrightarrow{(eO_1)_{A_1}(eO_2)_{A_2}\dots(eO_n)_{A_n}}\bullet \overset{\tau}{\Rightarrow}\bullet cnf'_{[L]} \\
& (\forall i, j : \mathbb{N} \cdot i \neq j \implies A_i \neq A_j) \\
\vee \quad & cnf_{[\emptyset]} \xhookrightarrow{\urcorner\urcorner\dots\urcorner}\bullet \overset{\tau}{\Rightarrow}\bullet cnf'_{[L]}
\end{aligned}
$$

Putting the two phases together, the overall behaviour can be summarised as $\overset{\tau}{\Rightarrow}\bullet \overset{e:E}{\Longrightarrow}\bullet$.

Proposition 1 *Given a deterministic and well-formed compensating automaton vector (Definition 8) $\hat{A}$, the semantics are deterministic, i.e. only one configuration can be reached over a given a sequence $e : E$:*
$\nexists cnf, cnf' \cdot init(\hat{A}) \overset{\tau}{\Rightarrow}\bullet \overset{e:E}{\Longrightarrow}\bullet cnf \wedge init(\hat{A}) \overset{\tau}{\Rightarrow}\bullet \overset{e:E}{\Longrightarrow}\bullet cnf'$

Example 1 *Consider the communication example depicted in Fig. 5(f) and let $[A_1, A_2]$ represent the vector composed of the two automata (with $A_1 = (\Sigma_1, Q_1, q_{01}, F_1)$ and $A_2 = (\Sigma_2, Q_2, q_{02}, F_2)$ representing the left and right automaton respectively). The*

initial configuration, given by init$([A_1, A_2])$ *is* $[\vec{A}_1[q_{01}]^{[]}, \vec{A}_2[q_{02}]^{[]}]$. *Performing the initial phase* ($\overset{\tau}{\Rightarrow}\bullet$) *would result in the installation of compensation* transCancOk*,* $[\vec{A}_1[q_{01}]^{[]}, \vec{A}_2[q_{12}]^{[(\tau,\text{transCancOk},[])]}]$*, through rules* Suc*,* Vect*, and* Tau*.*

Next we consider two scenarios for the second phase: (i) *payment succeeds followed immediately by failure (*RecPay*,*$\urcorner$*), and* (ii) *all actions succeed except for* ShipGoods*, i.e. (*RecPay*,*ArrangeTrans*,*$\urcorner$*).*

Under scenario (i)*, upon* RecPay*, the configuration proceeds to* $[\vec{A}_1[q_{11}]^{[(\text{transCancOk, charge,Refund},[])]}, \vec{A}_2[q_{12}]^{[(\tau,\text{transCancOk},[])]}]$ *through rules* Suc*,* Nest*,* Vect*, and* Sys*. Upon failure, execution direction turns backwards for both automata and local communication takes place,* $[\overleftarrow{A}_1[q_{11}]^{[]}, \overleftarrow{A}_2[q_{12}]^{[]}]$*, first through rules* Comp*,* Nest*,* Vect*, and* Tau *(storing* transCancOk *in the configuration), and then through rules* Comp*,* Nest*,* Vect*, and* Loc*.*

Under scenario (ii)*, upon* ArrangeTrans*, first through rules* Suc*,* Nest*,* Vect*, and* Sys*, and then by rules* NestSuc*,* Vect*, and* Sil *the configuration evolves to* $[\vec{A}_1[q_{11}]^{[(\text{transCancOk, charge,Refund},[])]}, \vec{A}_2[q_{22}]^{[(\tau,\text{charge, Cancel},[])]}]$*. Upon failure, execution direction turns backwards for both automata and local communication takes place, first through rules* Comp*,* Vect*, and* Tau *(storing* charge, Cancel *in the configuration), and then through rules* Comp*,* Vect*, and* Loc*, triggering the* Refund *action.*

In what follows we define the sanity of compensating automata — i.e. events are correctly compensated, and we prove that compensating automata are indeed sane.

3.2 Self-Cancellation in Compensating Automata

A standard way of checking that a compensation formalism is sane [BHF04] is to show that for any compensation program, if it has completely and successfully compensated, assuming perfect compensations, then the outcome would be *self-cancellation*, i.e. logically equivalent to performing no action.

Definition 12 *Following the same idea of [BHF04], we assume the existence of a total injective function* comp*,* $\bar{\cdot}$*, which returns a non-empty unique compensation action* $O \in 2^\Sigma$ *for an activity* $i \in \Sigma_\tau$ *of which the former is the perfect compensation — intuitively meaning that executing i following by O is equivalent to executing nothing. Overloading* $\bar{\cdot}$ *to strings,* $\overline{a_1\, a_2\, \ldots\, a_n} = \overline{a_n}\, \ldots\, \overline{a_2}\, \overline{a_1}$*. Furthermore,* $\overline{\overline{a}} = a$ *and* $\overline{\tau} = \tau$*.*

In order for compensating automata to be perfectly self-cancelling, it must be ensures that installed compensations correspond to the event triggering the installation on

every transition. Furthermore, note that when replacing compensations, fine-grained compensations are discarded, and hence lose the one-to-one correspondence of events and compensations. For this reason, to prove self-cancellation, one must ignore both the actions of nested automata (whose compensations are discarded) and also the replacing compensation. Rule Nest already silences actions of nested automata to the parent. Thus, what remains is to assume that the replacing compensation is an empty action.

Definition 13 *A perfectly-compensating automaton A^- is an automaton whose transitions $(q, I, O, (I', O', \hat{A}), d, q') \in \delta$ such that events and compensation actions must correspond, i.e. $\forall i \in I \cdot O' = \overline{i}$.*

Next, since we lose event-compensation correspondence when discarding/replacing compensations, we assume that all nested states of self-cancelling automata $(\hat{A}, (I, O, \hat{A})) \in N$ should have an empty compensation, i.e. $O = \emptyset$.

We use $\mathcal{A}^-$ to refer to the set of perfect automata.

Since we are concerned with using compensating automata for programming a system's compensations, we ignore the strictly local components in the trace, and silent actions. Furthermore, traces are projected for a particular automaton, leaving out any activities related to other automata.

Definition 14 *Given trace elements of the form $(i_O)_A$ we drop O (which are local actions) and given trace elements of the form $({}_iO)_A$ we drop i (which is a local event). Silent trace elements $\Lsh$ and τ are also dropped and for an automaton A, all elements tagged with $A' \neq A$ are dropped. Thus, given a trace w : $Trace^*$, we define w_A^- : $Trace^{-*}$ such that*

$$\begin{array}{rcl} Trace^- & ::= & \Sigma_\tau \mid 2^{\Sigma_\tau} \\ (a : w)_A^- & = & \begin{cases} i(w_A^-) & \text{if } a = (i_O)_A \\ O(w_A^-) & \text{if } a = ({}_iO)_A \\ w_A^- & \text{otherwise} \end{cases} \end{array}$$

Next we define what it means for a compensating automaton to be self-cancelling.

Definition 15 *Given a projected trace w_A^- of a compensating automaton A, the function cancOut removes consecutive corresponding event-compensation pairs:*

$$
\begin{array}{rcll}
cancOut(w\,x\,x'\,w') & = & cancOut(w\,w') & \text{if } x' = \overline{x} \\
cancOut(w) & = & w & \text{otherwise}
\end{array}
$$

A string w for which $cancOut(w^{-}_{A}) = \varepsilon$ is said to be self-cancelling*. Using the above cancelling function we now go on to define what it means of a compensating automaton to be* self-cancelling.

A compensating automaton A is said to be self-cancelling *if and only if all traces originating from an initial configuration and ending in a terminated configuration are self-cancelling:*
$\forall w : Trace^{*} \cdot goto(A, q_0, []) \xrightarrow{w} \overleftarrow{A}[q]^{[]} \Rightarrow cancOut(w^{-}_{A}) = \varepsilon$.

A vector of compensating automata $\hat{A}$ is said to be self-cancelling *if and only if all traces originating from an initial configuration and ending in a terminated configuration are self-cancelling:*
$\forall w : Trace^{*} \cdot init(\hat{A}) \xrightarrow{w} cnf \wedge \boxtimes(cnf) \Rightarrow \forall A \in \hat{A} \cdot cancOut(w^{-}_{A}) = \varepsilon$.

To facilitate reasoning about stacks, we define a function which returns the string representation of the stack dropping any stack elements which do not contribute to the trace.

Definition 16 *Given a stack S, the function* behaviour*, $S_{\#}$, returns a sequence of actions such that each character represents a compensation on the stack, with the head being the bottom element of the stack. Since deviations only contribute a τ action to the trace, we ignore deviation elements on the stack. Similarly, empty actions are ignored.*

$$
\begin{array}{rcl}
((I, O, \hat{A}) \rhd S)_{\#} & = & \begin{cases} S_{\#} & \text{if } O = \emptyset \\ S_{\#}\,O & \text{otherwise} \end{cases} \\
(d \rhd S)_{\#} & = & S_{\#} \\
\bot_{\#} & = & \varepsilon
\end{array}
$$

Proposition 2 *Given a string w composed solely of compensations $w \in (2^{\Sigma})^{*}$, then $\overline{w}$ returns a string composed solely of activities, $\overline{w} \in \Sigma^{*}$ and hence applying cancOut on $\overline{w}$ yields no change: if $w \in (2^{\Sigma})^{*}$, then $cancOut(\overline{w}) = \overline{w}$.*

Proof *Only the third case of Definition 15 can be applicable on $\overline{w}$ and hence no reduction can take place.*

Proposition 3 *Given an activity* $i \in \Sigma$ *and a string* $w \in Trace^{-*}$*, then appending* i *to* w *cannot contribute to cancellations which are not present in* w*. More formally,* $cancOut(w)\, i = cancOut(w\, i)$

Proof *By Definition 15,* i *can only cause a reduction if* $i\,\bar{i}$ *is a sequence of the string. Thus,* $cancOut(w\, i)$ *would return the same cancellation result as* $cancOut(w)\, i$ *since nothing follows* i*.*

Proposition 4 *Given a compensation* $\bar{i}$ *and a string* $s = w\ j$ $(i \neq j)$*, then appending* i *to* s *cannot contribute to cancellations which are not present in* w*. More formally,* $cancOut(s)\, \bar{i} = cancOut(s\, \bar{i})$

Proof *By Definition 15,* $\bar{i}$ *can only cause a reduction if* $i\,\bar{i}$ *is a sequence of the string. By Proposition 3,* $cancOut(s)\, \bar{i} = cancOut(w)\, j\, \bar{i}$ *and since* $j\, \bar{i}$ *by definition of* $cancOut$ *and uniqueness of compensations do not cancel out, then* $cancOut(w)\, j\, \bar{i} = cancOut(s\, \bar{i})$*.*

Proposition 5 *Multiple applications of* $cancOut$ *is the same as one application:* $cancOut(cancOut(w)) = cancOut(w)$

Proof *The proof follows by string induction on* w*.*

The base case for $w = \varepsilon$ *follows from Definition 15.*

The inductive case $w = k\, x$ *is split into the following cases:*

Case 1: $x = i$*,* $(i \in \Sigma)$

$$\begin{aligned} & cancOut(k\, x) \\ & \{ \text{ By Proposition 3 } \} \\ = \; & cancOut(k)\, x \\ & \{ \text{ By the inductive hypothesis } \} \\ = \; & cancOut(cancOut(k))\, x \\ & \{ \text{ By Proposition 3 twice } \} \\ = \; & cancOut(cancOut(k\, x)) \end{aligned}$$

Case 2a: $x = \bar{i}$ *assuming* $k = k'\, j$ $(i \neq j)$

$$
\begin{array}{rl}
& cancOut(k\,\bar{i}) \\
& \{ \text{ By Proposition 3 } \} \\
= & cancOut(k)\,\bar{i} \\
& \{ \text{ By the inductive hypothesis } \} \\
= & cancOut(cancOut(k))\,\bar{i} \\
& \{ \text{ By Proposition 4 twice } \} \\
= & cancOut(cancOut(k\,\bar{i}))
\end{array}
$$

Case 2b: $x = \bar{i}$ *and* $k = k'\,i$

$$
\begin{array}{rl}
& \{ \text{ By the inductive hypothesis } \} \\
& cancOut(k'\,i) = cancOut(cancOut(k'\,i)) \\
& \{ \text{ By Proposition 3 trice } \} \\
\Rightarrow & cancOut(k')\,i = cancOut(cancOut(k'))\,i \\
& \{ \text{ By removing } i \text{ both sides } \} \\
\Rightarrow & cancOut(k') = cancOut(cancOut(k')) \\
& \{ \text{ By Definition 15 twice } \} \\
\Rightarrow & cancOut(k\,i\,\bar{i}) = cancOut(cancOut(k\,i\,\bar{i}))
\end{array}
$$

Proposition 6 *Applying cancOut on a string* $w\,\bar{i}$ *is the same as cancelling out* w *and then cancelling the result appended with* $\bar{i}$*: for any* $w, w' : Trace^{-*}$, $cancOut(w\,\bar{i}) = cancOut(cancOut(w)\,\bar{i})$.

Proof *The proof follows by string induction on* w.

The base case for $w = \varepsilon$ *follows by Definition 15.*

The inductive case $w = k\,x$ *follows from the following cases:*

Case 1: $x = j$, $(j \neq i)$

$$
\begin{array}{rl}
& cancOut(k\,b\,\bar{i}) \\
& \{ \text{ By Definition 15 } \} \\
= & cancOut(cancOut(k\,j\,\bar{i})) \\
& \{ \text{ By Proposition 4 twice } \} \\
= & cancOut(cancOut(k\,j)\,\bar{i})
\end{array}
$$

Case 2: $x = a$

$$
\begin{array}{ll}
& cancOut(k\,a\,\overline{a}) \\
& \{ \text{ By Definition 15 } \} \\
= & cancOut(k) \\
& \{ \text{ By Proposition 5 } \} \\
= & cancOut(cancOut(k)) \\
& \{ \text{ By Definition 15 } \} \\
= & cancOut(cancOut(k)\,a\,\overline{a}) \\
& \{ \text{ by Proposition 3 } \} \\
= & cancOut(cancOut(k\,a)\,\overline{a})
\end{array}
$$

Proposition 7 *If* $cancOut(w) = w'\,a$ *then* $cancOut(w\,\overline{a}) = cancOut(w')$.

Proof *The result is proved as follows:*

$$
\begin{array}{ll}
& \{ \text{ By Proposition 6 } \} \\
\Rightarrow & cancOut(w\,\overline{a}) = cancOut(cancOut(w)\,\overline{a}) \\
& \{ \text{ By substitution } \} \\
\Rightarrow & cancOut(w\,\overline{a}) = cancOut(w'\,a\,\overline{a}) \\
& \{ \text{ By Definition 15 } \} \\
\Rightarrow & cancOut(w\,\overline{a}) = cancOut(w')
\end{array}
$$

Recall that the stack of a configuration stores a compensation for each activity which occurred. When a compensation is activated, it is removed from the stack and executed. Note that for each activity or compensating action, a corresponding modification occurs on the stack. Thus, we can define the resulting stack of a configuration step in terms of the original stack and the activity/action which occurred. Consider the stack $S = \overline{c} \triangleright \overline{b} \triangleright \overline{a} \triangleright S$ which contains the compensation of three consecutive activities $a\,b\,c$. Note that $\overline{S_\#} = \overline{\overline{c}\,\overline{b}\,\overline{a}} = \overline{\overline{a\,b\,c}} = a\,b\,c$. Thus the compensation stack acts as a record of past activities. If another activity d occurs, its compensation would be pushed onto the stack resulting in a stack $S' = \overline{d} \triangleright S$. As shown before $\overline{S'_\#} = a\,b\,c\,d = \overline{S_\#}\,d$. If the compensation $\overline{d}$ is executed, then the resulting stack S'' would satisfy $\overline{({}_\# S'')} = a\,b\,c$. Furthermore, note that this is also equal to $cancOut(\overline{S'_\#}\,\overline{d})$. This is expressed in the following lemma.

Lemma 1 *Generalising these observations, given a configuration* $\tilde{A}[q]^S$ *(where* $A : \mathcal{A}^-$*) which reaches* $\tilde{A}[q']^{S'}$ *on string* w*, the resulting stack* S' *can be expressed as a function of the original stack and the generated string such that if* $\tilde{A}[q]^S \stackrel{w}{\Rightarrow} \tilde{A}[q']^{S'}$,

then $S'_\# = \overline{cancOut(\overline{S_\#} w_A^-)}$*. The same can be said if the initial or end configuration (or both) is a nested configuration.*

Proof *The proof proceeds by string induction on* w*.*

Base case: $w = \varepsilon$ *and thus* $S = S'$*.*

$$\begin{aligned} & \{ \text{ By application of Definition 16 } \} \\ \Rightarrow\ & \overline{S_\#} = \overline{S'_\#} \\ & \{ \text{ By Definition 16 and by Proposition 2 } \} \\ \Rightarrow\ & \overline{S_\#} = cancOut(\overline{S'_\#}) \\ & \{ \text{ By Definition 12 and } w_A^- = \varepsilon \text{ } \} \\ \Rightarrow\ & S_\# = \overline{cancOut(\overline{S'_\#} w_A^-)} \end{aligned}$$

Inductive case: $w = k\,x$ *— The proof proceeds by a rule-by-rule analysis starting by the rules which trigger on forward configurations,* $\vec{A}[q]^S$ *or* $\vec{A}[q]^S_{\text{cnf}}$*.*

Case 1: Rule Suc*,* $w = k\,(i_{\bar{i}})_A$

$$\begin{aligned} & \{ \text{ By application of } \textsc{Suc} \text{ } \} \\ \Rightarrow\ & S'' = ((i', \bar{i}), \hat{A}) \triangleright S' \\ & \{ \text{ By Definition 16 } \} \\ \Rightarrow\ & S''_\# = \bar{i} S'_\# \\ & \{ \text{ By inductive hypothesis } \} \\ \Rightarrow\ & S''_\# = \bar{i}\, \overline{cancOut(\overline{S_\#}\, k_A^-)} \\ & \{ \text{ By Definition 12 } \} \\ \Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, k_A^-)\, i} \\ & \{ \text{ By Proposition 3 and Definition 14 } \} \\ \Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, (k\, i)_A^-)} \end{aligned}$$

Case 2: Rule Fail*,* $w = k\, \urcorner$*, and* $S'' = S'$

$$\begin{aligned} & \{ \text{ By application of } \textsc{Fail} \text{ and Definition 16 } \} \\ \Rightarrow\ & S''_\# = S'_\# \\ & \{ \text{ By inductive hypothesis } \} \\ \Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, k_A^-)} \\ & \{ \text{ By Definition 14 } \} \\ \Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, (k\, \urcorner)_A^-)} \end{aligned}$$

Case 3: Rule NESTSUC, $w = k\,\tau$

$$
\begin{aligned}
& \{\ \textit{By application of}\ \textsc{NestSuc}\ \textit{and Definition 16}\ \} \\
\Rightarrow\ & S''_\# = S'_\# \\
& \{\ \textit{By inductive hypothesis}\ \} \\
\Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, k^-_A)} \\
& \{\ \textit{By Definition 14}\ \} \\
\Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, (k\,\tau)^-_A)}
\end{aligned}
$$

Case 4: Rule NESTFAIL, $w = k\,\urcorner$

$$
\begin{aligned}
& \{\ \textit{By application of}\ \textsc{NestFail}\ \textit{and Definition 16}\ \} \\
\Rightarrow\ & S''_\# = S'_\# \\
& \{\ \textit{By inductive hypothesis}\ \} \\
\Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, k^-_A)} \\
& \{\ \textit{By Definition 14}\ \} \\
\Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, (k\,\urcorner)^-_A)}
\end{aligned}
$$

Case 5: Rule NEST, $w = k\,x$
$(x \in \{(i_O)_{A'}, ({}_iO)_{A'} \mid i \in \Sigma \wedge O \in 2^\Sigma \wedge A' \neq A\} \cup \{\tau, \urcorner\})$

$$
\begin{aligned}
& \{\ \textit{By application of}\ \textsc{Nest}\ \textit{and Definition 16}\ \} \\
\Rightarrow\ & S''_\# = S'_\# \\
& \{\ \textit{By inductive hypothesis}\ \} \\
\Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, k^-_A)} \\
& \{\ \textit{By Definition 14}\ \} \\
\Rightarrow\ & S''_\# = \overline{cancOut(\overline{S_\#}\, (k\,x)^-_A)}
\end{aligned}
$$

Next, the rules which trigger on backward configurations, $\overleftarrow{A}[q]^S$ or $\overleftarrow{A}[q]^S_{\text{cnf}}$ are considered.

Case 1: Rule COMP, $w = k\,(i,\bar{i})_A$

$$
\begin{array}{ll}
 & \{ \textit{ By application of } \text{COMP} \ \} \\
\Rightarrow & S' = ((i',\bar{i}),\hat{A}) \rhd S'' \\
 & \{ \textit{ By Definition 16 } \} \\
\Rightarrow & S'_{\#} = \bar{i}\, S''_{\#} \\
 & \{ \textit{ By inductive hypothesis } \} \\
\Rightarrow & \overline{cancOut(\overline{S_{\#}}\, k^{-}_{A})} = \bar{i}\, S''_{\#} \\
 & \{ \textit{ By Definition 12 twice } \} \\
\Rightarrow & cancOut(\overline{S_{\#}}\, k^{-}_{A}) = \overline{S''_{\#}}\, i \\
 & \{ \textit{ By Proposition 7 and Definition 14 } \} \\
\Rightarrow & cancOut(\overline{S_{\#}}\, (k\,\bar{i})^{-}_{A}) = cancOut(\overline{S''_{\#}}) \\
 & \{ \textit{ By Proposition 2 } \} \\
\Rightarrow & cancOut(\overline{S_{\#}}\, (k\,\bar{i})^{-}_{A}) = \overline{S''_{\#}} \\
 & \{ \textit{ By Definition 12 } \} \\
\Rightarrow & \overline{cancOut(\overline{S_{\#}}\, (k\,\bar{i})^{-}_{A})} = S''_{\#}
\end{array}
$$

Case 2: Rule DEV, $w = k\,\tau$

$$
\begin{array}{ll}
 & \{ \textit{ By application of } \text{DEV} \ \} \\
\Rightarrow & S' = d \rhd S'' \\
 & \{ \textit{ By Definition 16 } \} \\
\Rightarrow & S'_{\#} = S''_{\#} \\
 & \{ \textit{ By inductive hypothesis } \} \\
\Rightarrow & \overline{cancOut(\overline{S_{\#}}\, k^{-}_{A})} = S''_{\#} \\
 & \{ \textit{ By Definition 14 } \} \\
\Rightarrow & \overline{cancOut(\overline{S_{\#}}\, (k\,\tau)^{-}_{A})} = S''_{\#}
\end{array}
$$

Case 3: *Rule* NESTCOMP, $w = k\,\tau$

$$\begin{aligned}
& \{ \textit{ By application of } \textsc{NestComp} \textit{ and Definition 16 } \} \\
\Rightarrow\ & S'_{\#} = S''_{\#} \\
& \{ \textit{ By inductive hypothesis } \} \\
\Rightarrow\ & \overline{cancOut(\overline{S_{\#}}\, k^{-}_{A})} = S''_{\#} \\
& \{ \textit{ By Definition 14 } \} \\
\Rightarrow\ & \overline{cancOut(\overline{S_{\#}}\, (k\,\tau)^{-}_{A})} = S''_{\#}
\end{aligned}$$

Case 4: *Rule* NEST, $w = k\,x$
$(x \in \{(i_O)_{A'}, ({}_iO)_{A'} \mid i \in \Sigma \wedge O \in 2^{\Sigma} \wedge A' \neq A\} \cup \{\tau, \urcorner\})$

$$\begin{aligned}
& \{ \textit{ By application of } \textsc{Nest} \textit{ and Definition 16 } \} \\
\Rightarrow\ & S''_{\#} = S'_{\#} \\
& \{ \textit{ By inductive hypothesis } \} \\
\Rightarrow\ & S''_{\#} = \overline{cancOut(\overline{S_{\#}}\, k^{-}_{A})} \\
& \{ \textit{ By Definition 14 } \} \\
\Rightarrow\ & S''_{\#} = \overline{cancOut(\overline{S_{\#}}\, (k\,x)^{-}_{A})}
\end{aligned}$$

Theorem 1 *A compensating automaton is self-cancelling, i.e.*
$\forall w : \mathit{Trace}^{*} \cdot goto(A, q_0, []) \overset{w}{\Rightarrow} \overleftarrow{A}[q]^{\boxdot} \implies cancOut(w^{-}_{A}) = \varepsilon.$

Proof *By applying Definition 7 and Lemma 1 on the premise of the implication, then it follows that* $[]_{\#} = \overline{cancOut(\overline{[]_{\#}}\, w^{-}_{A})}$. *Since* $[]_{\#} = \varepsilon$*, this gets simplified to* $\varepsilon = \overline{cancOut(w^{-}_{A})}$. *Finally, by applying* comp *on both sides and by Definition 12, we get* $\varepsilon = cancOut(w^{-}_{A})$ *as required.*

Note that the theorem above proves self-cancellation of a compensating automaton A based on the definition of *cancOut* (Definition 15) which ignores symbols from nested automata. If a nested automaton B is also a sibling of A in an automata vector, then this would interfer with self-cancellation. Thus, without loss of generality, in the corollary below it is assumed that nested automata are labelled differently from sibling automata.

Corollary 1 *A vector of compensating automata is self-cancelling, i.e.*
$\forall w : \mathit{Trace}^{*} \cdot init(\hat{A}) \overset{w}{\Rightarrow} cnf \wedge \boxtimes(cnf) \implies \forall A \in \hat{A} \cdot cancOut(w^{-}_{A}) = \varepsilon.$

Proof *The proof follows by induction on the number of automata in the vector. Self-cancellation straightforwardly holds on an empty vector. The inductive case holds*

by considering rule VECT*, Definition 14, which drops all activities relating to other automata, and Theorem 1 which states that each trace of a compensating automaton is self-cancelling.*

4 Programming with Compensating Automata

The compensation logic for the e-procurement system has been programmed as a vector of five automata shown in Fig. 5. Fig. 5(a) listens for program R' events up to the point where it is confirmed that at least some of the goods are available. Subsequently, Fig. 5(b) is triggered and installs the compensation *UnreserveGoods* upon the event *ReserveGoods*. Next, payment and transport for the goods available are triggered through *startPayment* and *startTransport* respectively. If only some of the goods are initially available and a partial shipment is going to take place, then the Fig. 5(a) iterates till all the goods are available.[1] One might argue that Fig. 5(a) is useless since it is not installing any compensations. However, note that all the following compensations are only installed if the pattern of its events is matched. Through such event patterns, one may distinguish the kind of compensation required in different contexts.

Fig. 5(c) depicts the automaton which compensates for payment. Initially a *PayProc* compensation is installed so that if the process fails, any transport costs incurred (hence the local communication guards) can be collected. Note that *PayProc* has programmed compensation so that if the payment fails, an operator is notified. If normal payment succeeds then the *PayProc* compensation is replaced by a *Refund* which also takes into consideration the progress of the transport arrangement.

Fig. 5(d) depicts the automaton which compensates for transport arrangement. In particular, if the process fails after the goods have been shipped, then the goods are returned. If the return of the goods fails, then the *returnFailed* signal communicates with Fig. 5(a) and the *UnreserveGoods* compensation is discarded. Furthermore, Fig. 5(d) also includes a deviation which have been explained in Section 2.

Finally, Fig. 5(e) is responsible for installing the *transCancOk* signal and discarding it whenever it is not longer required, i.e. whenever a *charge* compensation is installed. This ensures that actions listening on event *transCancOk, charge* get triggered and that only one of the elements is enabled.

The overall behaviour of programming the compensation manager with these automata and applying it to system S'' (assuming S'' knows the logic corresponding

[1] To provide compensations for all iterations we use parametrised events and dynamically trigger copies of the automata in the spirit of the LARVA framework [CPS08].

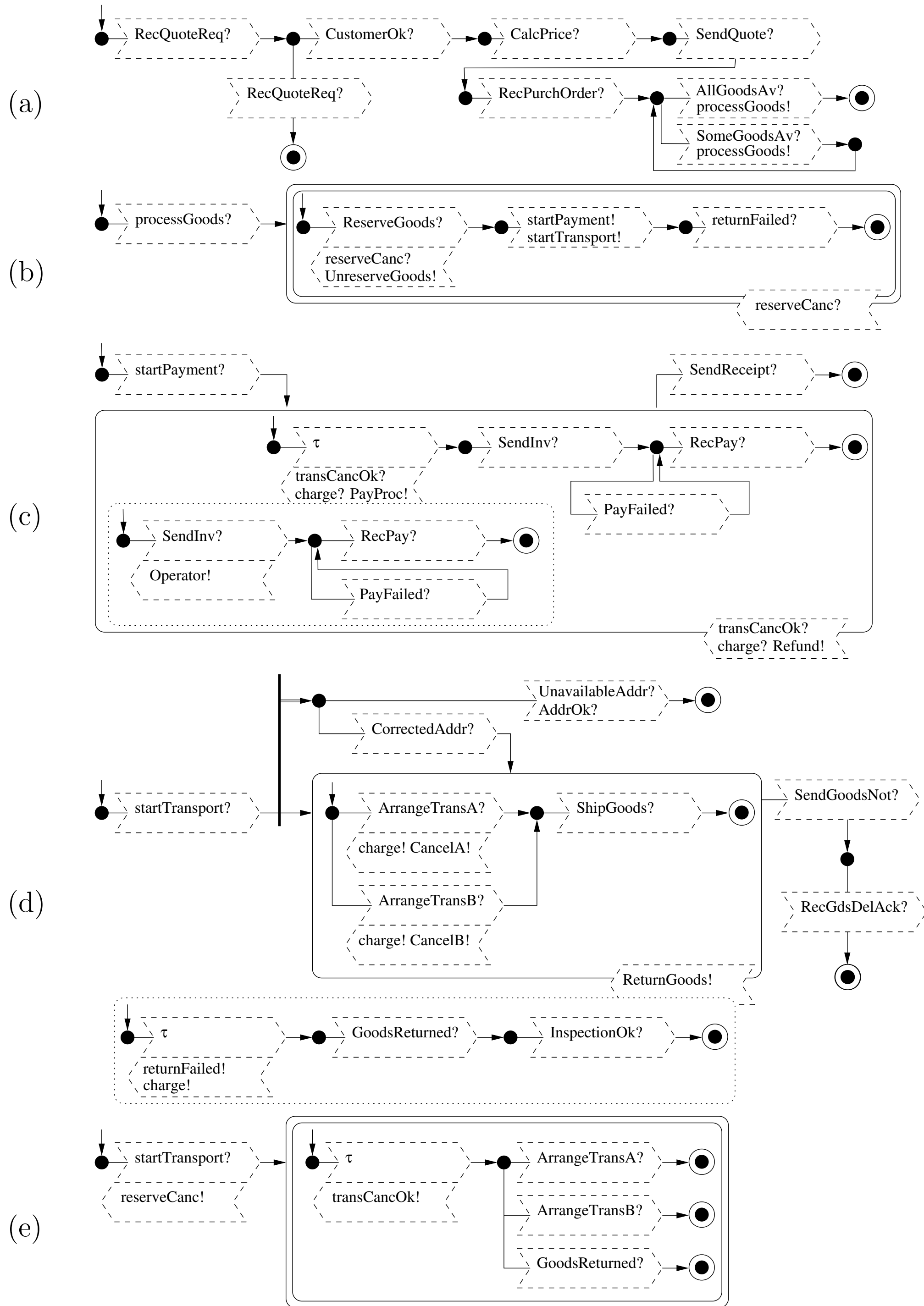


Figure 5: The compensating automata vector for the e-procurement system.

to *UnreserveGoods*, *RecPay*, *Refund*, etc) would satisfy all the features of the e-procurement system.

5 Discussion

A concern for the take up of our approach would probably be security and privacy concerns for the communication of system events to the compensation manager. However, while it is true that we have logically separated the system from its compensations, this is not necessarily so at the execution level. For example a preliminary implementation on Java uses aspect-oriented programming [Kic05] which preserves the logical separation even at the Java code level but at the same time does not expose sensitive information to external entities (at least not any more than the system already exposes).

There are numerous formalisations and notations [bpm08, BMM05, BF04, BHF04, ES08, FR05, GLG+06, HZWL08, LMSMT08, LPT08, SDN07] which support compensations including some which are pictorial (e.g. petrinet-based formalisms [HZWL08], communicating hierarchical transaction-based timed automata (CHTTAs) [LMSMT08], and BMPN [bpm08]). There are three main features distinguishing are work. Firstly, do to the proposed compensation design paradigm, compensating automata do not have operators which include forward recovery such as alternative forwarding and speculative choice (provided for example in [BMM05, BHF04]). Secondly, compensating automata provide a set of compensation-dedicated operators which do not require the user to hand code frequently occurring patterns in compensation design (for example compensating automata provide explicit compensation replacement as opposed to for example [BF04, GLG+06] which provide generic stack operations). Thirdly, compensating automata support the concept of a deviation which enables a business process to be partially compensated. To the best of our knowledge this has not been proposed before in the literature. The formalism which is most similar to ours is that of CHTTAs. However, in CHTTAs compensations are not first-class operators and are instead wired in terms of communicating channels. Furthermore, compensations in CHTTAs are programmed in terms of patterns which lose the unstructured nature of automata.

We are aware of two other approaches which offer a solution to the compensation issues highlight through the e-procurement scenario. The first, by Nepal et al. [NFG+05], moves towards the complete opposite of our approach and proposes a model which does not differentiate between normal and exceptional behaviour, and abstracts away from the notion of compensation. They claim that this approach simplifies the specification of the system. The second approach, by Schäfel et al. [SDN07], is similar to

ours in that it proposes the separation of compensations from functionality, proposing a manager which manages compensations for a particular web service functionality and a compensation language which specifies the corresponding compensations. However, although conceptually similar, our approach is fundamentally different due to their lose understanding of compensations. By our strict definition of compensation, their approach does not separate compensation concerns from other concerns such as repeating or retrying activities.

6 Conclusions

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 monitoring of system events using a compensation manager. The contributions of this paper include: (i) A novel compensation design paradigm which advocates complete separation of compensation concerns; (ii) A compensation programming notation — compensating automata, 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 a real-life e-procurement system — showing compensating automata to be useful for handling non-trivial compensation logic.

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.

References

[AAB+07] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0, 2007. OASIS Standard. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (Last accessed: 2010-02-17).

[BF04] Michael J. Butler and Carla Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *Coordination Models and Languages (COORDINATION)*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2004.

[BHF04] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, Lecture Notes in Computer Science, pages 133–150. Springer, 2004.

[BMM05] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Principles of programming languages (POPL)*, pages 209–220. ACM, 2005.

[bpm08] Business process modeling notation, v1.1, 2008. `http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf` (Last accessed: 2010-02-17).

[CP12] Christian Colombo and Gordon Pace. Recovery within long running transactions. *ACM Computing Surveys*, 2012. to appear.

[CPS08] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2008.

[ES08] Christian Eisentraut and David Spieler. Fault, compensation and termination in WS-BPEL 2.0 - a comparative analysis. In *Web Services and Formal Methods (WS-FM)*, volume 5387 of *Lecture Notes in Computer Science*, pages 107–126. Springer, 2008.

[FR05] Dirk Fahland and Wolfgang Reisig. ASM-based semantics for BPEL: The negative control flow. In *Abstract State Machines (ASM)*, pages 131–152, 2005.

[GFJK03] Paul Greenfield, Alan Fekete, Julian Jang, and Dean Kuo. Compensation is not enough. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 232–239. IEEE, 2003.

[GLG+06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi 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.

[HZWL08] Yanxiang He, Liang Zhao, Zhao Wu, and Fei Li. Formal modeling of transaction behavior in WS-BPEL. In *Computer Science and Software Engineering (CSSE)*, pages 490–494. IEEE, 2008.

[Kic05] Gregor Kiczales. Aspect-oriented programming. In *International Conference on Software Engineering (ICSE)*, page 730. ACM, 2005.

[LMSMT08] Ruggero Lanotte, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. Design and verification of long-running transactions in a timed framework. *Sci. Comput. Program.*, 73:76–94, 2008.

[LPT08] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A formal account of WS-BPEL. In *Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008.

[MJG+11] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. To appear; http://dx.doi.org/10.1007/s10009-011-0198-6.

[NFG+05] Surya Nepal, Alan Fekete, Paul Greenfield, Julian Jang, Dean Kuo, and Tony Shi. A service-oriented workflow language for robust interacting applications. In *On the Move to Meaningful Internet Systems - Part I*, pages 40–58. Springer, 2005.

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

[RLT78] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10:123–165, 1978.

[SDN07] Michael Schäfer, Peter Dolog, and Wolfgang Nejdl. Engineering compensations in web service environment. In *International Conference on Web Engineering (ICWE)*, pages 32–46. Springer, 2007.