Runtime Verification and Compensations

Christian Colombo

Supervisor: Gordon J. Pace

Faculty of ICT

University of Malta

October 2012

Submitted for the degree of Doctor of Philosophy

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Runtime Verification and Compensations

submitted is my work, except where acknowledged and referenced.

Christian Colombo

October 2012

Acknowledgements

I would like to sincerely thank Gordon J. Pace for his continuous support and enthusiasm particularly through frequent meetings and discussions which served as a major inspiration throughout the past years.

I would like to thank my colleagues, particularly Adrian Francalanza and Mark Micallef, and my family and friends, particularly Jennifer Fenech, for their presence and continued support during my studies.

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

Abstract

As systems grow more complex, it becomes increasingly difficult to ensure their correctness. One approach for added assurance is to monitor a system's execution so that if a specification violation occurs, it is detected and potentially rectified at runtime. Known as *runtime verification*, this technique has been gaining popularity with its main drawback being that it uses system resources to perform the checking. An effective way of minimising the intrusiveness of runtime verification is to desynchronise the system from the monitor, meaning that the system and the monitor progress independently. The problem with this approach is that the monitor may fall significantly behind the system and by the time the monitor detects a violation, it might be too late to make a correction. To tackle this issue we propose a monitoring architecture, cLarva, providing fine control over the system-monitor relationship, enabling the monitor to be synchronised to the system by fast-forwarding through monitoring states, and the system to synchronise with the monitor by reverting it to an earlier state.

Going back through system states is generally hard to automate since not all actions are reversible; reverse actions may expire, the reverse of an action may be context-dependent and so on. This subject, known as *compensations*, has been studied in the context of transactions where the reversal of incomplete transactions is used to ensure that either the transaction succeeds completely or it leaves no effect on the system. Although a lot of work has been done on compensations, the literature still presents challenges to compensation programming. We show how these limitations can be alleviated by separating compensation programming concerns from other concerns. Inspired by monitor-oriented programming — a way of using runtime verification to trigger functionality — we propose a novel monitor-oriented notation, *compensating automata*, for compensation programming. Integrated within a monitoring framework which we call *monitor-oriented compensation programming*, this notation enables a programmer to program and trigger compensation execution completely through monitoring with the system being unaware of compensations.

Finally, we show how compensating automata can be used for programming the synchronisation between the system and the monitor in cLarva, enabling complex compensation logic to be seamlessly programmed. To evaluate our approach, we applied it to an industrial case study based on a financial system handling virtual credit cards, consisting of thousands of users and transactions. The results show that the architecture has been successful in both synchronising the monitor to the system by fast-forwarding the monitor, and also in synchronising the system to the monitor using compensations to reverse the system state — achieving a virtually non-intrusive runtime monitoring architecture.

Contents

List of Figures

List of Tables

1. Introduction

Computer systems have been growing in size and complexity for decades, making it virtually impossible for such systems to be faultless. Besides, their role in sensitive human activities have put higher pressures on their correctness. While exhaustive verification techniques such as model checking are available, they do not scale well to typical industry-scale software systems. On the other hand, testing is scalable and is heavily used for bug detection, but it often cannot ensure the absence of bugs since it lacks coverage.

Although checking all of a system's potential behaviour is surely a way of avoiding the occurrence of incorrect behaviour, an alternative is to check the behaviour which *actually* occurs. *Runtime verification* is a lightweight formal methods technique which captures a live system's events and detects any divergence from the specification. Naturally, runtime checking consumes system resources with potential repercussions on the monitored system's behaviour such as slowing it down. One way of significantly reducing the impact is to desynchronise runtime verification from the system, enabling the system to progress independently while the monitor processes the events at its pace. The downside of asynchronous runtime verification is that violation detection may occur significantly later than its occurrence, meaning that the system state could have evolved since.

However, synchronous and asynchronous runtime verification are only two extremes of the synchronisation spectrum: synchronous runtime verification gives full control to the monitor while in the asynchronous version control never leaves the system. A potentially better solution seems to lie in between — finer-grained control over the system-monitor interaction can be achieved if it is possible to switch between synchrony and asynchrony at runtime. In this way, synchronous monitoring can be used when security concerns are high such as in the case of an impending violation, while asynchronous monitoring can be used when efficiency concerns are high such as during a period of heavy load. This combination can be useful for example in the case of a financial system where users pose different levels of risks to the system, and where the load level typically tends to be very high during short periods of times.

Switching between synchrony and asynchrony is however challenging: while going from synchrony to asynchrony is trivial since synchrony can be considered as a special case of asynchrony with zero delay, the other direction is not as straightforward. In general, one way this can be done is by pausing the system to wait for the monitor to reach the same execution point as the system. However, if the monitor has already detected a violation, it is useless for the system to wait since the monitor cannot proceed forward. Instead, the only option would be to somewhat revert the system state back to the point where the violation had occurred.

Although reversing a system's state may seem far-fetched, the topic has long been studied; not in the context of runtime verification, but in the context of undoing completed parts of failed transactions — giving the impression that no part of the transaction had ever occurred. The main problem is that undoing activities whose results might have been used by other activities can potentially leave a system in an inconsistent state. A traditional solution is to isolate a transaction so that no other transaction can use intermediate results. However, in real-life scenarios this is not always possible since intermediate results may have consequences outside the realm of the computer system, *e.g.*, reversing a bank account transfer might involve a fee. Thus, reversing such actions cannot be done by “undoing” but rather by executing “counteractions”, better known as *compensations, e.g.,* executing the reverse bank transfer and charging the fee. Compensations have thus become particularly useful in programming transaction-like interactions which cannot be isolated during their execution such as web service compositions which typically multi-party. The increasing interest in compensation programming has led to several attempts of formalising the essence of compensations with significantly different notations being proposed. Still, the state-of-the-art approach to compensation programming approaches has been shown to be limited with respect to particular realistic scenarios.

A main reason behind the challenges of compensation programming is that since they enable the logical reversal of past actions, by their nature they *crosscut* 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 fee if not. Programming such a compensation from basic principles would require some form of record-keeping 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 programming of system actions with their compensations. The current solutions essentially provide standard patterns which can be conveniently instantiated for programming compensations. However, the problem is that frequently real-life scenarios diverge from standard patterns, requiring more flexibility. Unfortunately, while flexible approaches have been proposed, they quickly become very complex or leave a lot of effort to the programmer. A better compromise between rigid patterns and flexible programming seems to be lacking.

1.1 Aims and Achievements

Runtime verification provides an effective way of detecting specification violations but leaves the programming of violation correction completely up to the programmer. On the other hand, compensations provide a means of structuring error recovery actions. While the two tackle the detection and recovery aspects of violations respectively, the question of how these two could be combined remains largely unexplored. One obvious way of combining runtime verification and compensations is to verify behavioural properties of compensating programs using runtime verification. Another possibility is to use compensations to structure the runtime-verification-triggered recovery actions. While these may be useful, such applications yield clearcut relationships with no real cross-fertilisation of one research area to the other. Our aim is precisely to explore how the two can be intertwined together for mutual enrichment:

- A main concern of adopting runtime verification is the overheads it induces on the system being monitored. One way of doing away with this overhead is by desynchronising the system from the monitor, allowing the system to proceed independently from the monitor. However, by the time an asynchronous monitor detects a problem, the system would have usually progressed further. The consequence is that the recovery which would have been appropriate at the point of violation, the *violation-time recovery*, might no longer be appropriate at the time of detection because the system state would have changed. The solution is to either use a tailor-made recovery which takes into consideration what happened between violation and detection, or to somehow revert the system state to the point of violation and use the violation-time recovery. Naturally, the latter is preferred since it is able to use a single recovery method for numerous scenarios. An aim of this work is to use compensations as a means of reverting a system detection-time state to the violation-time state so that violation-time recovery can be applied. However, under certain practical scenarios, a large discrepancy between the violation and its detection may not be compensable since compensations may expire. We aim to tackle this problem in two ways: on the one hand compensations should be automatically discarded when they are no longer applicable, and on the other hand, monitors should be synchronisable at runtime when required — executing compensations before it is too late. The former is useful when compensations are time-bound or not applicable after the occurrence of a particular action, *e.g.,* once an order has been concluded and the delivery has started, the order cannot be compensated. The latter is useful for monitoring sensitive elements of a system where the risk of late error detection is not acceptable, *e.g.,* in the case of untrusted users, the delivery should not start unless the transaction has been verified.

- Runtime verification has been used as a means of separating programming concerns through the monitor-oriented programming paradigm. This allows different concerns to be programmed independently in terms of monitor-triggered "reparation" code. Using this approach, we aim to facilitate compensation programming by delegating compensation programming to runtime monitors. This approach relieves the clutter of programming compensations with the rest of the programming concerns. Monitors, not only provide separation of concerns as in approaches such as aspect-oriented programming, but also improve the quality of the code since monitors are synthesised from formal specifications. In this regards, we aim to provide a formal notation for programming compensations in the context of runtime verification. Such a notation should be able to both process the input from the monitored system while simultaneously constructing compensation programs accordingly. When the system signals the need to execute compensations, the monitor-constructed compensations are executed.

To address the first of the above aims, *i.e.,* of managing runtime overheads, we have devised a compensation-based architecture which enables finer control of runtime verification overheads. As regards facilitating compensation programming, we propose a dedicated automata-based notation and show how this can be integrated with runtime monitoring. More details describing our main contributions are given below:

Compensation-based system-monitor synchronisation To reduce monitoring intrusiveness on the monitored system we propose a practical architecture supported by a theoretical framework which manages the system-monitor interaction. Figure 1.1 depicts the architecture with the system producing a trace and the monitor processing it, possibly falling behind if it cannot keep up with the system. To manage their relationship, we provide a number of operations (metaphorically related to typical cassette deck actions), some of which are applied to the system while others are applied to the monitor: a running system can be conceptually stopped (■), reversed (◀), fast-reversed (◀◀), paused (❚❚), or unpaused (▯▯) while the monitor can be fast-forwarded (▶▶) to keep up with the system by ignoring irrelevant intermediate monitoring states. Stopping the system (or part thereof) is useful to perform corrective actions when the monitor detects a violation while reversing the system is useful when the system has progressed more than the monitor and corrective actions need to be carried out. Pausing and unpausing the system can be used to allow the monitor to synchronise with the system.

In this architecture compensations play a central role since they are able to reverse the system state to a past state. Using the mathematical framework which underpins the architecture, we prove that under certain assumptions, employing compensations to synchronise the system with the monitor is equivalent to using synchronous monitors. Furthermore, since compensations may not be applicable beyond a certain time bound or af-

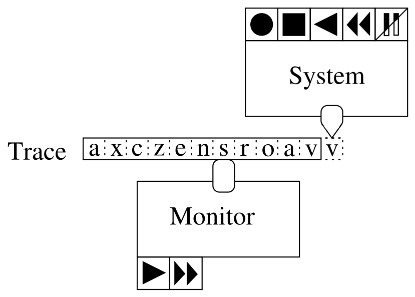


Figure 1.1: The complete system-monitor architecture

ter the execution of a particular action, the architecture allows the user to discard expired compensations by specifying compensation scopes — by the end of which compensations are discarded. Depicted as a fast-rewind, this feature enables the architecture to cater for more realistic scenarios.

Moreover, synchronous and asynchronous monitoring may sometimes be needed for a single system's behaviour and potentially during a single run. A typical example is that at peak times of system load, monitoring is switched to asynchronous to be able to maintain a better quality of service, while monitoring untrusted users might require the opposite. For this reason, the architecture supports compensation-based synchronisation by enabling on-the-fly synchronisation and desynchronisation of the system and the monitor, depicted in Figure 1.1 as the *pause* symbol on a cassette deck. Again, we prove that on-the-fly synchronisation and desynchronisation yields the same logical behaviour as synchronous monitoring.

Compensating automata To facilitate compensation programming through runtime verification we propose an automata-based notation, compensating automata — from which monitors can be automatically synthesised for managing complex compensations. This notation is dedicated to compensation programming, providing specialised constructs which are particularly suited for programming the combination of the normal system execution and the monitor's contribution of compensation execution.

Monitor-oriented compensation programming Monitor-oriented programming (MOP) is a programming paradigm advocating separation of concerns through runtime verification where additional functionality is triggered via monitors. In our instantiation to compensations, monitor-oriented compensation programming (MOCP), we extend MOP to not only trigger functionality but is also able to configure the compensation functionality while monitoring. With this addition, compensations are configured on-the-fly based on monitored events and consequently executed upon being triggered by designated monitors. Exploiting specialised compensation constructs of compensating automata, MOCP enables a fine interplay between normal system execution and compensation execution.

Compensating automata for system-monitor synchronisation Finally, we combined the previous achievements and used compensating automata to enable the user to program the system-monitor compensation-based synchronisation. This approach delegates the management of monitor synchronisation to a dedicated module, leaving the system free from compensation synchronisation concerns.

To test the above contributions, we have applied the theory to two significant case studies:

E-procurement case study The first case study is an e-procurement system taken from the literature [57] which has been used to demonstrate that "compensations are not enough". The main problem of existing approaches of compensation programming is that the distinction between system programming and compensation programming is not clearly outlined. This has several consequences such as lack of modularity and inflexibility when it comes to programming complex scenarios. The case study has been successfully programmed using compensating automata, completely separating compensation concerns from others and clearly outlining the complex interplay between system execution and compensation execution.

Industrial financial case study The theoretical and practical framework revolving around system-monitor synchronisation has been motivated by an industrial case study from Ixaris Ltd based on a financial system called Entropay. Using Entropay, users can use virtual credit cards to affect payment/purchases to/from third parties. Although such a system has many security-critical aspects which can benefit from runtime monitoring, synchronous runtime verification was not possible due to performance concerns. The proposed compensation-based asynchronous monitoring architecture has been successfully applied to Entropay with encouraging results.

1.2 Overview of Subsequent Chapters

The document is organised into four main parts starting with the background, followed by the two main parts — tackling how runtime verification can be useful for compensations and how compensations can be useful for runtime verification respectively — and concluded by the last part. In more detail:

Part I: Background This part provides the necessary background for later parts. Given that this work explores the relationship between runtime verification and compensations, this part is divided into two corresponding sections:

Chapter 2: Runtime Verification This chapter gives a brief overview of runtime verification by presenting a number of instances including logic-based and automata-based runtime verification, and the monitoring-oriented programming paradigm which enables functionality to be programmed in terms of monitors.

Chapter 3: Compensations The third chapter introduces the basics of compensations including the main stages of compensation programming and the notions of compensation correctness. Furthermore, we review a number of significant compensation formalisms, highlighting the distinctions amongst them.

Part II: Runtime Verification for Compensations This part focuses on how runtime verification can be used for compensations, particularly for programming compensations in a modular fashion — separated from other programming concerns. In more detail:

Chapter 4: Compensating Automata This chapter presents a novel automata-based compensation programming notation — compensating automata — and proves that these have desirable compensation and monitoring attributes. Next, a case study from the literature is encoded in compensating automata to highlight their expressivity and finally they are compared to other notations supporting compensations.

Chapter 5: Monitor-Oriented Compensation Programming In this chapter we integrate compensating automata with runtime monitoring to form an architecture which is able to support compensation programming in a completely transparent way to the system.

Part III: Compensations for Runtime Verification This part explores how compensations can be useful for runtime verification, particularly how monitors can be synchronised and desynchronised effectively from the system being monitored. The approach is then evaluated over an industrial case study. This is described in the following chapters:

Chapter 6: Compensation-Based System-to-Monitor Synchronisation This chapter presents a monitoring architecture which supports both synchronous and asynchronous runtime monitoring, possibly switching across the modes at runtime, using compensations as a synchronisation technique.

Chapter 7: Monitor-to-System Synchronisation Taking an opposite approach to synchronisation, in this chapter we investigate the possibility of synchronising the monitor to the system rather than the opposite. This is achieved by conceptually fast-forwarding the monitor through monitoring states to synchronise it with the system.

Chapter 8: System-Monitor Synchronisation with Compensating Automata This chapter combines aspects from previous chapters of this work, presenting the use of compensating automata (presented in Chapter 4) for programming elaborate system-to-monitor synchronisations (the architecture of which is presented in Chapter 6).

Part IV: Conclusions The last part concludes the work:

Chapter 9: Conclusion This chapter gives a summary of the work followed by thoughts on possible directions of future work and concludes with some final remarks.

Part I

Background

The section which provides the background for the rest of the work is divided into two chapters: one dealing with runtime verification and another dealing with compensations.

2. Runtime Verification

Correctness remains a major concern in the development and deployment of software systems. To date, testing is commonly used to check software dependability which, despite its effectiveness, is not exhaustive and thus does not guarantee the absence of bugs. On the other hand, techniques such as model checking aim to exhaustively verify all execution paths of a system against a set of formally specified properties. While this approach has been successfully used to check small pieces of software such as device drivers, it is usually impractical for checking typical software systems.

A midway approach is that of runtime verification [12, 36, 47, 77] which checks an execution trace at runtime, thus covering all encountered traces while remaining scalable since checking a single trace is relatively cheap. In its basic form runtime verification is similar to assertion checking which is frequently used for debugging purposes. However, runtime verification introduces further powerful features:

- Assertions are inserted manually in particular points in the execution while runtime monitors are typically weaved automatically.
- Assertions are evaluated on a particular point is a system's execution, while runtime monitors are usually evaluated over an execution path.
- Assertions are typically expressed in the host language, while runtime

monitors are typically specified in formal notation from which monitors are automatically synthesised.

Applying runtime verification usually involves the formal specification of the acceptable runtime behaviour which is then compiled and automatically instrumented into the target system. If the observed runtime behaviour diverges from the formal specification, runtime verification can be used to either issue a meaningful error message, or execute corrective actions to restore the system to a sane state. In more detail, the process of applying runtime verification can be loosely described through five phases (depicted in Figure 2.1):

① **Specification** The first phase includes the specification of the system's properties in some kind of formal notation. The choice of the notation mostly depends on the domain of the problem with typical notations being temporal logics (*e.g.,* JavaMOP [84]), regular expressions (*e.g.,* Java-MaC [94]), and automata (*e.g.,* LARVA [41]).

② **Synthesis** Once the system properties are specified, they are usually synthesised into some form of executable verification code or an intermediate representation which is then manipulated at runtime. For example in the case of LARVA [41] the automata-based properties are directly synthesised into code while in the case of JavaMOP [84], logic-based specifications are transformed into automata and then into code. On the other hand, Java-MaC [67] and Hawk [44] manipulate a representation of the specification at runtime. Approaches which generate executable code are generally preferred since compilation takes place before runtime, avoiding runtime overheads.

③ **Event extraction** To monitor a system, the monitor has to be somehow connected to the system so that the former is aware of relevant events. To automate the event extraction process, thus minimising the chances of errors, techniques such as tracing and instrumentation are typically used *e.g.,* Hawk [44], JavaMOP [84], Larva [41], tracematches [18] and Java-MaC [67] all use automated instrumentation to elicit event, with the first four using aspect-oriented programming [66] which automatically instruments code (source or binary) into a system by pattern matching.

④ **Monitoring** Subsequently, the actual monitoring of the system is carried out by running the synthesised monitor or monitoring algorithm concurrently with the system and passing it system events. A major issue of monitor execution is the intrusiveness of the monitor on the system: if the monitor is running on the same address space as the system, then the monitor would be competing for resources while if the monitor is on a separate address space, the system would still have to wait for monitor feedback before progressing.

⑤ **Feedback to system** The next phase would be to react to any detected runtime violations. Such reactions may vary from simply raising an exception, stopping the offending system, or initiating an appropriate fault-handling mechanism. Several runtime verification tools (including Hawk [44], Java-MOP [84], Larva [41], tracematches [18] and Java-MaC [67]) favour the last option, supporting the execution of a reparation in response to a detected violation.

To substantiate the runtime verification life cycle briefly described above, in this chapter we give a number of instances of runtime verification, starting with the monitoring of the linear-time temporal logic (Section 2.1.1) followed by the synchronous monitoring tool Larva (Section 2.1.2), and a programming paradigm based on runtime verification — monitoring-oriented programming (Section 2.1.3). Finally, before concluding, Section 2.1.4 discusses the limitations of runtime verification.

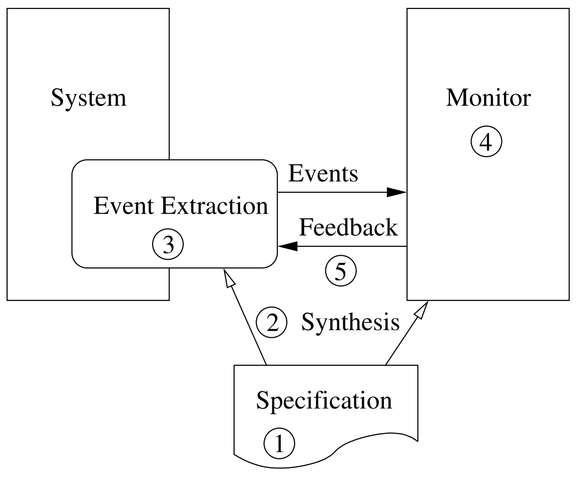


Figure 2.1: The phases of runtime verification

2.1 Instances of Runtime Verification

Along the years numerous approaches to runtime verification have been proposed. A significant body of work revolves around linear-time temporal logic which has been originally used for model checking but has now been adapted for runtime verification. A lot of work has also been done to provide tools (*e.g.,* Hawk [44], JavaMOP [84], Larva [41], tracematches [18] and Java-MaC [67]) which differ not only in the specification languages they support, but have also other significant differences as alluded in the previous section. While refraining from going into the details of each tool, in this section we focus more on the underlying approaches: logic-based runtime verification, automata-based runtime verification, and monitoring-oriented programming.

2.1.1 Logic-Based Monitoring

When introducing runtime verification we noted that a distinguishing factor from assertion checking is that runtime verification supports the checking of trace properties, *i.e.,* properties which can be verified over a system run. One natural way of relating events over a system run is through temporal operators. In fact temporal logics, particularly Linear-time Temporal Logic (LTL) [89], have been heavily used in the area of runtime verification. LTL enables the specification of properties about a program's behaviour over time such as (i) "it is always the case that *something bad* does not happen" (known as *safety* properties), denoted by $G\neg x$, where x is *something bad*; or (ii) "eventually *something good* should happen" (known as *liveness* properties), written $F\,y$, where y is *something good*.

Definition 2.1.1. The full syntax of LTL including both temporal and standard binary operators is given recursively below, assuming a set of propositions ranged over by p:

$$\varphi \quad ::= \quad \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, U \varphi \mid X\varphi$$

with the following abbreviations:

$$\begin{array}{lll} \varphi_1 \wedge \varphi_2 & \overset{\text{def}}{=} & \neg(\neg\varphi_1 \vee \neg\varphi_2) \\ \varphi_1 \implies \varphi_2 & \overset{\text{def}}{=} & \neg\varphi_1 \vee \varphi_2 \\ F\varphi & \overset{\text{def}}{=} & \mathit{true}\, U \varphi \\ G\varphi & \overset{\text{def}}{=} & \neg(\mathit{true}\, U \neg\varphi) \end{array}$$

□

Assuming a sequence of system states, a basic LTL formula is a logical composition of propositions which is satisfied if it holds on the current system state. The two main temporal operators include the *Next* operator (denoted $X\cdot$) which asserts its operand on the next system state, and the *Until* operator (denoted $\cdot U \cdot$) which asserts that its first operand is continually true until the second one is satisfied.

Definition 2.1.2. Let a system be represented by a set of propositions P whose truth values change with time, a system's state, $s \in 2^P$, be the subset of propositions which are true at the time, and a system's behaviour, $w = s_1, s_2, s_3, \ldots$, be an infinite sequence of states with $w^i = s_i, s_{i+1}, s_{i+2}, \ldots$. A behaviour suffix w^i satisfies a temporal formula φ, written $w^i \vDash \varphi$ by the following definition:

$$\begin{array}{lll}
w^i \vDash true & \textit{iff} & true \\
w^i \vDash p & \textit{iff} & p \in s_i \\
w^i \vDash \neg\varphi & \textit{iff} & w^i \nvDash \varphi \\
w^i \vDash \varphi_1 \vee \varphi_2 & \textit{iff} & w^i \vDash \varphi_1 \textit{ or } w^i \vDash \varphi_2 \\
w^i \vDash \varphi_1 \, U \varphi_2 & \textit{iff} & \exists j \geq i \cdot w^j \vDash \varphi_2 \textit{ and } \forall i \leq k < j \cdot w^k \vDash \varphi_1 \\
w^i \vDash X\varphi & \textit{iff} & w^{i+1} \vDash \varphi
\end{array}$$

□

While LTL has long been used in the context of model checking [35], the developed techniques cannot be directly used for runtime verification. There are three main reasons for this [15]:

- Model checking verifies that all the possible behaviours of a system adhere to the specification, *i.e.,* the system-generated language is included in the language satisfying the property, while runtime verification only needs to check that one behaviour (the current one) is included in the language accepted by the property. The latter problem is known to be generally simpler than the former and thus it pays to develop techniques which are specific to runtime verification.

- While model checking is able to check infinite executions due to its exhaustive nature, in runtime verification only finite prefixes of (potentially infinite) traces can be (observed and hence) checked in practise. This prompted the need for an adaptation of LTL semantics from one which reasons about infinite traces to others which handle finite traces.

- Since typically runtime verification occurs during a system's execution, the system trace would only become available gradually. Thus runtime verification techniques can benefit if the checking process in runtime verification occurs incrementally, considering one additional symbol at a time while using the result of the prefix.

In this overview we consider two main approaches to defining LTL semantics for finite traces. The first approach [92] simply assumes that the traces are finite, *i.e.,* the length of the trace is known. The main problem of this approach is when it comes to give a verdict for a temporal property which concerns an (unavailable) extension of the trace. For example this situation arises when attempting to define the semantics of the *next* operator at the end of the trace — the formula has certainly not been satisfied but neither has it technically been violated. In this case the semantics we are considering evaluate the operand on the last element of the trace. A similar situation occurs with the *until* operator. The formal definition of these two operators in the finite trace setting are given below, leaving out the other definitions which are identical to the ones given earlier.

Definition 2.1.3. The suffix w^i of a finite trace satisfies a formula φ if either the formula is satisfied as with the semantics for infinite traces, or else if the end of the trace is reached, the formula is evaluated on the last element available. Mathematically:

$$w^i \vDash X\varphi \quad \overset{\text{def}}{=} \begin{cases} w^i \vDash \varphi & \textit{if } i = \textit{length}(w) \\ w^{i+1} \vDash \varphi & \textit{otherwise} \end{cases}$$

$$w^i \vDash \varphi_1\, U \varphi_2 \quad \overset{\text{def}}{=} \begin{cases} w^i \vDash \varphi_2 & \textit{if } i = \textit{length}(w) \\ w^i \vDash \varphi_2 \textit{ or} & \\ (w^i \vDash \varphi_1 \textit{ and } w^{i+1} \vDash \varphi_1\, U \varphi_2) & \textit{otherwise} \end{cases}$$

□

The disadvantage of this approach is that it is not in line with the idea of incremental traces which are synonymous to runtime verification. Rectifying this issue, another approach to finite semantics [15] considers finite traces as prefixes of infinite traces. A major difference from the previous semantics is that when considering a prefix, it makes more sense not to give a final verdict if the verdict can change later on. For this reason, the semantics enable the possibility of a third verdict apart from the usual *true* or *false*: *inconclusive*, denoted by ?. Thus using the original semantics on infinite traces, the following definitions give the LTL semantics on finite prefixes.

Definition 2.1.4. A finite prefix u is said to satisfy a property φ, written $[u \models \varphi]$, if for any extension σ, the resulting trace still satisfies φ. Similarly, a finite prefix u is said to violate a property φ if for any extension σ, the resulting trace still violates φ. Otherwise, the result is inconclusive. Formally:

$$[u \models \varphi] \overset{\text{def}}{=} \begin{cases} \top & \textit{if } \forall \sigma \in \Sigma^\omega \cdot u\sigma \models \varphi \\ \bot & \textit{if } \forall \sigma \in \Sigma^\omega \cdot u\sigma \not\models \varphi \\ ? & \textit{otherwise} \end{cases}$$

□

Having a specification language and its formal semantics is of little use to runtime verification unless a verification technique is available for checking adherence of runtime behaviour to the specification. In the context of LTL there are two main approaches: one using term rewriting techniques (*e.g.,* [92]) and another by compiling the formula into an automaton (*e.g.,* [15]). The two approaches are intimately linked with the main difference being that in the case of rewriting the formula is expanded in a step-by-step fashion during monitoring, while in the case of automata the expansion occurs upfront before monitoring starts. Their relationship is thus similar to the relationship of interpreted and compiled languages: in the case of rewriting one needs the rewriting engine to be running in the background, applying rewriting rules on the current state of the formula; while in the case of automata, they are directly executable. Naturally, the automata approach has a significant advantage when it comes to keeping the runtime processing overhead of monitoring low. Furthermore, using automata one would know upfront the amount of memory required for monitoring while in the case of rewriting, the memory requirements have to be estimated. On the other hand, this means that using automata, more memory would be required on average to enumerate the whole state space in advance. In what follows we give an example of using term rewriting and automata to monitor a LTL property.

Example 2.1.1. Consider a traffic lights system (inspired from [92]) whose state is defined by the lights which are on/off at a particular moment. A typical trace of the system would be $w = \{green\}, \{yellow\}, \{red\}, \{red, yellow\}, \ldots$. Note that starting the cycle from the *green* light being on, the *red* light should not appear before the *yellow* light. Thus a property which should be true at the start of w is $\neg red\, U yellow$. Taking the rewriting approach, recursive definitions are used to expand the formula as shown below:

$w^1 \vDash \neg red\, U yellow$

{ U rewrite }

$w^1 \vDash yellow \textit{ or } (w^1 \vDash \neg red \textit{ and } w^2 \vDash \neg red\, U yellow)$

{ Left-hand side evaluation and logical rewrite }

$w^1 \vDash \neg red \textit{ and } w^2 \vDash \neg red\, U yellow$

{ Left-hand side evaluation and logical rewrite }

$w^2 \vDash \neg red\, U yellow$

{ U rewrite }

$w^2 \vDash yellow \textit{ or } (w^2 \vDash \neg red \textit{ and } w^3 \vDash \neg red\, U yellow)$

{ Left-hand side evaluation and rewrite }

true

On the other hand, compiling the formula[1] yields the automaton shown in Figure 2.2 which starts at q_0, takes the self loop ($\neg red \wedge \neg yellow$) upon *green*, and then reaches q_1 upon *yellow*. Note that reaching q_1 signifies that the property has been satisfied and that based on the three-valued LTL semantics, the formula cannot be violated by any extension of the trace in the future.

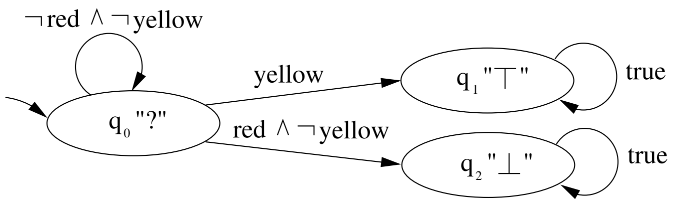


Figure 2.2: The automaton which monitors for the LTL formula $\neg red\, U yellow$

Apart from being useful for logic compilation as explained in the previous subsection, automata may also be useful for directly specifying correctness properties. This is further expanded in the next subsection.

2.1.2 Automata-Based Monitoring

Since automata are directly executable, using them for specifying monitors implies that there is no need for compilation (except for an encoding into a programming language). This means that what the programmer specifies is being monitored directly with a number of favourable aspects:

- The user is directly aware of the memory overheads involved in monitoring; there are no further compilation/expansions which are outside of the user's direct control.
- Avoiding compilation also means that debugging is significantly easier — the connection between the user specification and the point of violation

[1]The compilatin is based on the algorithm given in [15] where the standard LTL to Büchi automata conversion is applied on the formula and its negation followed by a number of automata operations such as composition and determinisation.

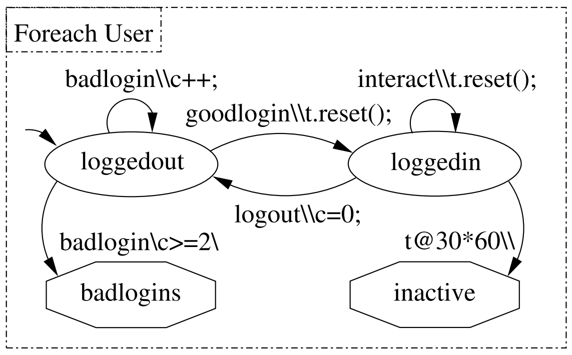
is explicit in the case automata while due to the compilation of a logic, further effort is required to understand why a formula has been violated.

- Due to the proliferate knowledge of automata, users might find it easier to express specifications in terms of automata rather than having to learn a logic. Automata might also be preferred due to their pictorial nature while on the other hand (from our experience with industry) sometimes even the word “logic” puts many potential runtime verification users off.

These significant advantages have led us in past efforts to propose an automata-based runtime verification tool Larva [41], enabling the synchronous monitoring of Java programs against properties specified in terms of Dynamic Automata with Timers and Events (DATEs) [40]. These automata are similar to timed-automata [4] enriched with stopwatches, variables, and channel communication. Using the Larva compiler the specification is transformed into the equivalent monitoring code, together with aspect-oriented programming code that extracts the events from the system.

As an example, consider a Java system where one needs to monitor bad logins and the activity of logged in users. By having access to *badlogin*, *goodlogin* and *interact* events (each of which corresponds to method calls in the Java system), one can keep a successive bad login counter and a clock to measure the time a user is inactive.

Figure 2.3 shows the specification of a property stating that there should be no more than two successive bad logins nor more than 30 minutes of inactivity when logged in, expressed as a DATE automaton. Transitions have three (backslash-separated) labels: (i) the event triggering it; (ii) the condition which is checked before taking it; and (iii) the action performed when it is taken. A total ordering on the transitions is used to ensure determinism. Note that the *foreach* construct causes the property to be monitored for each user who uses the system, *i.e.,* a monitor instance is instantiated upon the monitor detecting an event from a non-monitored user.



```
Global {
 Foreach (User u) {
  Variables {
   int c = 0;
   Clock t;
  }
  Events {
   badlogin() = {User u.badlogin()}
   clk() = {t@30*60}
   ...
  }
  Property authentication {
   States {
     Bad { badlogins inactive }
     Normal { loggedin }
     Starting { loggedout }
   }
   Transitions {
     loggedout -> badlogins [badlogin\c>=2\]
     loggedout -> loggedin  [goodlogin\\t.reset();]
     ...
     loggedout -> loggedout [badlogin\\c++;]
     loggedin -> inactive   [clk\\]
   }
  }
 }
}
```

Figure 2.3: The DATE automaton and Larva code of the bad logins scenario

Events in the context of DATEs can be system methods calls, timer events (*e.g.,* a monitor stopwatch triggers an event after 30 minutes since it was reset), synchronisation events, or a disjunction of events. Since basic events contribute to disjunction events, then at any time instant several events may fire simultaneously.

Definition 2.1.5. A system trace s is a sequence of time instants such that each instant, s_i, is composed of a set of events, $E_i \in 2^{\text{Event}}$ and a timestamp $t_i \in \mathbb{R}_0^+$: $s_i = (E_i, t_i)$. The set of possible instants will be written as Ω: $\Omega \overset{\text{def}}{=} 2^{\text{Event}} \times \mathbb{R}_0^+$. □

A DATE automaton has a set of states Q and transitions $\rightarrow$ which trigger upon receiving a system event given that a condition holds on the system's and timers' state (with Θ ranging over system states and $\mathcal{CT}$ ranging over timer states). A subset of states $B \subseteq Q$ are considered bad; representing a property violation if a bad state is reached. Furthermore, each DATE transition can carry out an action modifying the system or/and timer states.

Definition 2.1.6. A *DATE automaton*, $M \in \mathcal{M}$, running over a system state of type Θ is a quadruple $\langle Q,\ q_0,\ \rightarrow,\ B\rangle$ with set of states Q, initial state $q_0 \in Q$, transition relation $\rightarrow$, and bad states $B \subseteq Q$. Transitions are labelled by (i) an event expression which triggers them; (ii) a condition on the system state and timer configuration which will enable the transition to be taken; (iii) timer actions to perform when taking the transition; (iv) a set of channels upon which to signal an event; and (v) code which may change the state of the underlying system:

$$Q \times \mathit{Event} \times ((\Theta \times \mathcal{CT}) \rightarrow \mathbb{B}) \times \mathcal{TA} \times 2^{\mathit{channel}} \times (\Theta \rightarrow \Theta) \times Q$$

□

A monitor consists of a vector of DATE automata and directives to instantiate new automata dynamically upon receiving certain events. Full details of the the formalism of DATEs can be found in [40]. For the needs of this report, it suffices to identify the configuration of a vector of DATE automata and explain how they form a run depending on the system events observed.

Definition 2.1.7. A configuration $c \in \mathcal{C}$ of a vector of DATE automata, $\overline{M} \in \overline{\mathcal{M}}$, consists of the current system and monitor state, $\theta \in \Theta$, the current state of the stopwatches, $ct \in \mathcal{CT}$, a vector of locations representing the location each automaton is in, $\overline{q} \in \overline{Q}$, and the vector of automata itself, $\overline{M}$: $c = (\theta, ct, \overline{q}, \overline{M})$. □

We can now outline the semantics of a vector of DATEs.

Definition 2.1.8. The semantics of a vector of DATEs, $\overline{M} \in \overline{\mathcal{M}}$, can be given by extracting from $\overline{M}$ a labelled transition system over configurations $\langle \mathcal{C}, c_0, \rightarrow_M \rangle$ — with states $\mathcal{C}$ (the configurations of $\overline{M}$), initial configuration $c_0 \in \mathcal{C}$, and transition function labelled by events and timestamps $\rightarrow_M \in (\mathcal{C} \times \Omega) \rightarrow \mathcal{C}$. We write $c \xrightarrow{a}_M c'$ to refer to a particular $(c, a, c') \in \rightarrow_M$ and $c \overset{w}{\Longrightarrow}_M c'$ (with $w \in \Omega^*$) for the transitive closure of $\rightarrow_M$.

The set of bad configurations, $\mathcal{C}_B \subseteq \mathcal{C}$, corresponds to the configurations arising from the states which are tagged as undesirable in the original DATEs, *i.e.,* one of the DATEs in the vector reaches a bad state. We will assume that the transition system guarantees that recovery from a bad state is not possible — if $c \in \mathcal{C}_B$ and $c \xrightarrow{a}_M c'$, then $c' \in \mathcal{C}_B$.

The set of bad traces starting from a configuration c, written $\mathcal{B}(c)$, are the strings leading to a bad configuration: $\mathcal{B}(c) = \{w \mid \exists c' \in \mathcal{C}_B \cdot c \overset{w}{\Longrightarrow} c'\}$. A configuration c_2 is said to be as strict or stricter than c_1 (written $c_1 \sqsubseteq_M c_2$) if c_2 rejects all traces rejected by c_1 (and possibly more): $\mathcal{B}(c_1) \subseteq \mathcal{B}(c_2)$. We say that they are equivalent if they reject the same traces: $c_1 =_M c_2 \overset{\text{def}}{=} c_1 \sqsubseteq_M c_2 \wedge c_2 \sqsubseteq_M c_1$. □

Upon reaching a bad state, Larva allows the user to execute a reparation to mitigate the violation (which the bad state represents). In this sense, runtime verification is used as a sort of high-level exception handler which performs a double check over and above the native exception handling provided by the system. However, this outlook over runtime verification is not consistent throughout the field. Interestingly, runtime verification has also been proposed as a programming paradigm through which parts of functionality are managed by the runtime verification framework. The next subsection elaborates on this.

2.1.3 Monitor-Oriented Programming

Monitor-oriented programming (MOP) [31, 33, 84] is a paradigm which combines the specification and the implementation of a system. MOP is based on the same principles of runtime verification which allows system behaviour to be matched against formally expressed properties and possibly trigger some action in response. Unlike runtime verification though, it not only specifies properties to detect violations and raise exceptions, but the monitoring mechanism is itself part of the design of the system's functionality. Hence, the monitoring is not simply an extra check on top of the system but an integral part of the system's design. For example, consider a plain vanilla server which does not handle user authentication. Through MOP, the user authentication layer can be implemented without changing the code of the server and keeping functionality concerns separated from authentication concerns (see Figure 2.4).

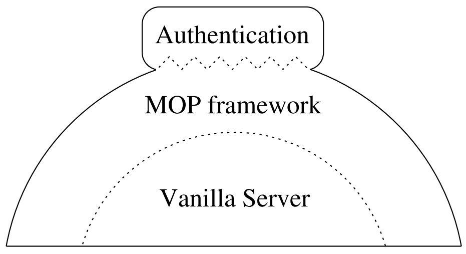
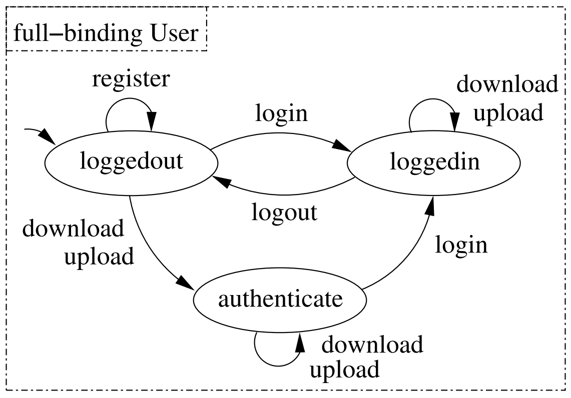


Figure 2.4: MOP example

More concretely, in Figure 2.5 we give an example of how the authentication can be managed through Java-MOP [32] (a tool-supported instantiation of MOP): the user is allowed to register, login, download, and upload files but if the download or upload activities are attempted without being logged in, then the pattern matches the finite state machine specified in MOP. At this point, the authentication process starts and the user is redirected to the login screen. Note that *full-binding* enables the property to match on a per-user basis.

Similar to authentication, other concerns such as tracking user activities, detecting malicious behaviour, managing promotional offers, *etc.*, can be imple-



```
full-binding Authentication(User u) {
 event login after(User u) :
  call(* User.login()) && target(u) {}
 event upload before(User u) :
  call(* User.uploadRequest()) && target(u) {}
 ...

 fsm :
  loggedout [
   login  -> loggedin
   upload -> authenticate
   ...
  ]
  loggedin [
   logout -> loggedout
   ...
  ]
  ...
  alias match = authenticate
  @match { doAuthentication(); }
}
```

Figure 2.5: The MOP specification for the server authentication scenario

mented separately and connected through MOP, enabling the actual implementation to remain uncluttered. In this sense MOP is similar to aspect-oriented programming since it can manage different concerns of a system in a modular fashion *e.g.,* a monitor for handling a graphical user interface, a monitor for handling authentication aspects, another for handling file access, *etc.* The main difference, however, is that while aspect-oriented programming is defined in terms of pattern matching of the program structure, in MOP matching occurs through formally specified properties. This distinction promises to enable

MOP to specify significantly more complex patterns than aspect-oriented programming. Given the popularity of aspect-oriented programming with virtually all major programming languages having an aspect-oriented extension, it seems that (with only a few instances of MOP currently available [84]) the potential of MOP is far from exploited.

2.1.4 Discussion

Whilst each instance of runtime verification presented in the previous subsections define a significantly distinct point in the runtime verification design space, they all suffer from a major problem: *runtime overheads*. Runtime overheads are a major concern for adopting runtime verification in real-life systems. The problem does not only include resource overheads required for running the monitor, which might cause a system to slow down, but also other concerns such as whether the system's behaviour would change in terms of the events generated and their sequence. At the heart of the problem is the *synchrony* between the system and the monitor: the system is not allowed to progress unless the monitor gives the go ahead. While this arrangement may potentially slow down the system considerably, it is crucial for giving timely feedback to the system. One way of considerably minimising the runtime overhead is to verify an execution trace *asynchronously*, *i.e.,* the system can progress without waiting for monitor processing. Asynchrony limits the impact of runtime verification to just logging of events which usually has to take place anyway in real-life security-critical systems. The downside is that asynchronous monitoring cannot reliably give feedback to the system since the system state might change from the time of the violation to the time of detection. Figure 2.6 shows two snapshots of a hypothetical monitoring scenario with the left-hand side showing the setup at the time the system committed the violation while the right-hand side shows the setup when the monitor detects the violation. Note that by the time the monitor detects the violation (marked by $\times$), the system has already progressed and produced two additional events (r and x).

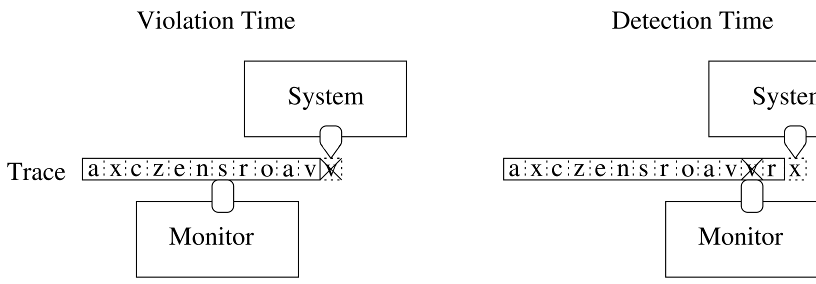


Figure 2.6: Violation-time — when the system just violated the specification [left] and detection-time — the point at which the monitor detects the violation [right]

Architecturally, synchronous and asynchronous monitoring can be depicted as in Figure 2.7 with two major aspects differentiating them:

- Since the monitor might not be able to keep up with the system in asynchronous monitoring, a buffer is required to store the events which the system has produced but which the monitor has not yet processed.
- The feedback line from the monitor to the system is omitted in the case of asynchronous monitoring since the monitor may not have an updated view of the system and any monitor feedback might be outdated.

On one end, synchronous runtime verification leaves control up to the monitor — the monitor is free to take any resources it requires — while on the other hand, asynchronous runtime verification gives absolutely no control to the monitor relegating it to a mere observer. What could be a better solution is a compromise between the two where the user can configure the amount of control which can be afforded to the monitor at any particular time during the execution of the system. This idea is further expanded in Part III of this report.

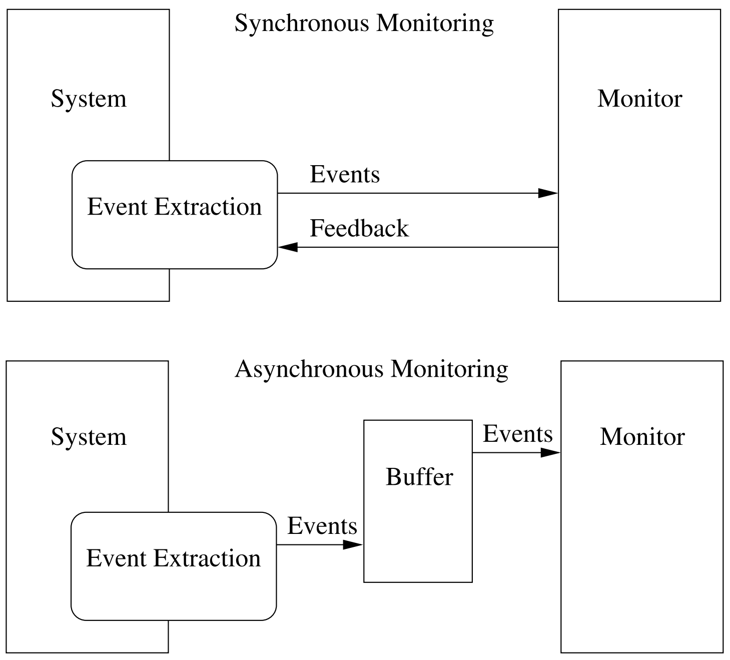


Figure 2.7: Synchronous [top] and asynchronous [bottom] runtime verification

2.2 Conclusion

Runtime verification has evolved as a compromise to the scalability problem of exhaustive verification techniques and the lack of coverage of software testing. Significant work has been carried out in the field with a number of mature tools and various approaches to runtime verification taking root. Nonetheless, runtime verification still has the problem of competing for runtime resources, potentially undesirably impacting the system behaviour and performance. While asynchronous monitor considerably alleviates this problem, it is generally impractical to take corrective actions at the time of error detection due to the lack of synchrony between the system and the monitor. By striking a balance between synchronous and asynchronous monitoring, the system resources can be better managed without compromising the possibility of the monitor taking effective corrective measures.

3. Compensations

Although debugging small and simple systems tends to be manageable, an increase in complexity brings about a disproportionate increase of complexity when detecting and fixing bugs — increasing the possibility of undetected bugs. To this end, fault tolerance techniques started playing a crucial role in software development with the idea that although faults may (or are expected to) happen, a system would be able to cope correctly. A commonly used mechanism consists of attempting to fix the system state upon the discovery of an error to enable normal execution to proceed unhindered. The challenge here is the programming and management of the potentially prohibitive number of distinct fixes — different errors in different contexts typically require different fixes. To address this issue, attempts have been made to homogenise error fixes by having the fix adapt to the context. One way of achieving this is by undoing the most recent actions which led to the error, reaching a previous sane state. While this approach requires the programmer to keep track of the history (or part thereof), it significantly simplifies the task of error recovery since the fix automatically takes the context into consideration.

Unfortunately, history-based recovery where previous actions are undone cannot be used in all circumstances. Typical scenarios would be systems interacting with real-life processes such as bank account transfers, shipping, *etc.* Such processes cannot be simply undone and forgotten. For example in the case of bank account transfers, 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 rather than "undo" the shipping. In such cases, instead of undoing some actions, one might actually need to execute further counteractions, better known as *compensations*, such as the reverse transfer plus a fee or the return of shipped items.

Compensations have become even more relevant with the advent of the Internet which enabled widespread interaction particularly through the use of web services. This phenomenon facilitated interactions across entities which might not even have been aware of each other before the interaction. In such a scenario, with multi-party interactions which are typically long-running, it is usually not possible to control the visibility of the actions involved across all parties. Once actions are visible to the rest of the world, discarding past actions would leave the overall system in an inconsistent state (since the rest of the world might have acted on actions which have been obliterated). To this end, compensations are heavily used to support such interactions, allowing actions to be logically undone, taking into consideration the impact which the action might have left on the rest of the world.

In the rest of the chapter we start by giving a high level overview of error recovery approaches which then enables us to introduce and place compensations into context in Section 3.2. Subsequently (Section 3.3), we delve deeper into the concept of compensations by presenting a variety of compensation formalisations and explain how the correctness of compensations can be specified. Finally, we conclude in Section 3.5.

3.1 Error Recovery Techniques

The higher the reliability expected of a system, the more precautions have to be taken, ironically potentially introducing additional complexity and sources of unreliability. The situation is further aggravated by the many possible sources of faults in a computer system: hardware faults, operating system faults, and faults within the computer system itself. Potentially, faults cause the system to reach an erroneous state which may then lead to failure[1], *i.e.,* making the effects of having reached an erroneous state visible to the outside world.

Given the probability of failure in large systems due to their complexity, the solution is typically not to try to engineer systems which never fail but rather to create safety nets within and around the system so that the system can tolerate faults in such a way that they do not lead to failure. A commonly used fault-tolerance technique [90] is *error recovery* — the task of dealing with an error before it has time to cause a failure.

The various strategies of error recovery [90, 98] are typically broadly classified as *backward* or *forward recovery*. Backward recovery refers to strategies which first revert the current (erroneous) state to a previous (correct) state before attempting to continue execution, while forward recovery attempts to correct the current (erroneous) state and then continues normal execution. The main difference is that backward recovery inherently keeps track of the historic context to fix the problem, while it is not necessarily the case in forward recovery. On one hand, backward error recovery can be considered as a special case of forward recovery, in which additional data structures are used to store the history of execution. On the other hand, forward error recovery can also be considered as an optimisation of backward error recovery [90], in that the recovery path is deduced without keeping a log of the past actions.

Forward error recovery is particularly useful when the failure — the symptom of the error — is sufficient to determine which solution to apply. A simple mechanism to encode forward recovery in most modern programming languages is the use of exception and error event handlers. The recovery code acts as a *reparation*, intended to fix the problem encountered. Note that, unless addi-

[1]The terminology is used in line with that introduced in [83].

tional mechanisms are used to keep a log of what actions the system performed, exception handlers are simply aware that a failure occurred within a particular block of code — no historical data is explicitly available. On the other hand, to make backward recovery possible, one has to keep track of the system's previous states or transitions, which involves recording data or actions previously carried out. One approach to backward recovery is that of backing up the system's data (also called *checkpointing*) so that if an erroneous state is reached, the data can be restored to a past but sane copy. Another approach is that of keeping an audit trail, recording sequences of actions which had been carried out, so that the system state can be restored by reversing the actions previously performed. While the first approach incurs a high overhead in terms of data storage, the latter, commonly known as a *rollback* mechanism, requires knowledge on how to reverse a system's actions.

Rollbacks, being perfect reversals of previous actions, which leave no evidence of either the original action or its reversal, have been heavily used to implement *transaction* frameworks. A transaction is a means of isolating an operation such that it appears (to processes outside the transaction) as an uninterruptable (atomic) action. The main motivation behind transactions is that consistency rules cannot always be maintained on a per-action basis. For example, if a bank system has the following consistency constraint: "unless a deposit or withdrawal takes place, the total sum of money in the bank cannot change", then in a money transfer between accounts, the constraint will be violated after the update of the source account (before the update of the destination account). Thus certain actions have to be grouped into a transaction and appear to other processes as a single action which either succeeds or fails. These principles together with that of *durability* (see below) form the bases of transactions and are referred to as the ACID principles (due to which the transactions we are describing are something referred to as ACID transactions): *atomicity* — ensuring that either the transaction fully succeeds or else it leaves no effect on the system state;

consistency — ensuring that the system state progresses from one correct state to another; *isolation* — ensuring that intermediate results of a transaction are not visible to other transactions, giving the impression that each transaction works in isolation; and *durability* — ensuring that the outcome of a transaction is persisted and never undone. The implementation of ACID transactions often relies on a resource-locking policy whereby transactions operate under the illusion of complete isolation from the rest of the world. Thanks to this strict approach, recovering from an error during a transaction simply requires the application of rollbacks to undo any successful parts of the failed transaction.

For many years the ACID principles have proven to be an adequate way of handling database operations. However, perfect action reversal is not always possible, particularly when a system interacts with other external systems/processes (*e.g.,* a bank system or a shipping agent) which do not support perfect reversal of actions[2]. In this case, rather than a perfect reversal of an action, one would need to execute a counteraction, better known as a *compensation* which semantically undoes the original action as much as possible. So, for example, to compensate for a bank transfer one might need to reverse the involved sum and charge an extra fee. Observe that, even if no fee is incurred, the two transfers (back and forth) would distinguish the account from another one which was never involved in a transfer (although their resulting balances would be identical). Similarly, to compensate for a wrong shipment, one might need to book a reverse shipment followed by shipment to the correct destination. Note the fundamental difference between rollback and compensation in that the latter does not remove evidence of the erroneous action but simply executes a correction.

[2]Even if the external system does support perfect reversal, interactions with external systems are typically long-running, requiring long periods of isolation with repercussions to availability.

3.2 Compensations

Compensations have been around at least since 1973 [46] as a form of forward recovery [90] which attempts to correct the state of a system given some knowledge of the previous actions of the system. For example, consider a bookshop which is processing an order — as long as the bookshop's computer system does not interact with the outside world, say, the shipping agent, then backward recovery would be possible because all processes involved are under the control of the bookshop. However, as soon as the bookshop places the shipping order, an interaction commitment would have taken place as the bookshop does not have access to backward recover the shipping order at the shipping agent's site (and even if the shipping agent was part of the bookshop system, the order might have already started being carried out, *i.e.,* an interaction has taken place with another process outside the control of the system — the physical process). If a client decides to cancel an order, then a special forward recovery, termed a *compensation*, has to be carried out to check whether the shipment is still in time to be cancelled. If the shipment is successfully cancelled, the client is possibly charged a fee and notified of the cancellation. On the other hand, if it is too late to cancel the shipment, an apologetic message is sent to the client explaining the situation. Note that although at a high level of abstraction cancelling the order might be considered as a backward recovery (cancelling the shipping order), in actual terms the order has not been undone but rather a "counter"-transaction took place whether or not cancellation succeeded. Expanding the same concept, [56] presents *compensating transactions,* later as *sagas* [54], as an extra layer on top of ACID transactions. The argument for this addition is two-fold: (i) long-running transactions render ACID transactions impractical as locking resources for long periods of time is infeasible in a highly concurrent system; and (ii) ACID transactions do not support nesting since committed ACID transactions cannot be undone (to ensure the atomicity of the higher-level transaction). Through the notion of compensation, ACID transactions can be composed together to form a saga where some transactions are compensations for others[3].

Later (*e.g.,* [7, 59]), with the advent of transactions across entities over the Internet, compensations became completely independent of ACID transactions. Rather, Internet transactions, better known as *web service compositions*, can be considered as ACID transactions with less stringent constraints on the atomicity and isolation principles, namely that atomicity occurs at a higher level of abstraction — either the transaction happens or it is logically undone — and that isolation is ignored but has to be handled during compensation by also undoing any actions which depended on the failed transaction.

In the rest of this chapter we attempt to focus on the essence of compensations and abstract away from either ACID transactions or web services, and refer to generic *actions*. To pave the way for a deeper understanding of compensations, at this point we present an extended example of an online bookshop scenario which will be used to highlight features and design issues of compensations.

Example 3.2.1. Consider the process a bookshop undertakes upon receiving an order: the bookshop first checks whether the requested books are in stock. If this succeeds, optional promotional offers are presented to the customer and subsequently the bookshop concurrently sends the books to be packed and charges the client's bank account. If both operations are successful, a courier is booked to deliver the order. In the eventuality of a failed activity, any completed activities are compensated by executing the associated compensating activities in order to remove the effects of the transaction. For example a bank charge is compensated by a refund, whereas packing is compensated by unpacking. Note that the compensation need not be the exact reverse of the normal activity; for instance, apart from incrementing the stock as a compensation to a stock decrement, an email is sent to the client as a notification of the transaction failure.

[3]In the rest of this work we use the term *transactions* to refer to compensable transactions.

It is usually important to decide in which order the compensations should be executed. In this case, if for any reason, the unpacking was unsuccessful then it does not make sense to increment the stock level and thus a sequential order would enforce the increment of the stock level to wait for the outcome of the unpacking operation. Generally (as in this case), the order of executing compensations is the reverse order of their normal execution. For example, if the courier booking fails, then both the client charge and the packing need compensating (in parallel) followed by withdrawing the offer and the compensation of the stock decrement.

Figure 3.1 represents the bookshop scenario with the upper half of the boxes representing forward behaviour and those below representing the associated compensations. The arrows with a filled head represent successful execution control flow, while the others represent a fault which triggers compensation execution. Once compensation is triggered, the control flow continues in reverse order of the forward behaviour. For example, failure when processing the client payment will trigger the unpacking of the order (if this has been completed before the processing of the payment failed) followed by withdrawing the offer and sending an email and increasing the stock. If a compensation fails, a human operator is notified (not shown in the diagram). Note that the related backward actions are not intended to fix the related forward actions if it fails halfway through, but rather to undo it if it had previously been successfully completed.

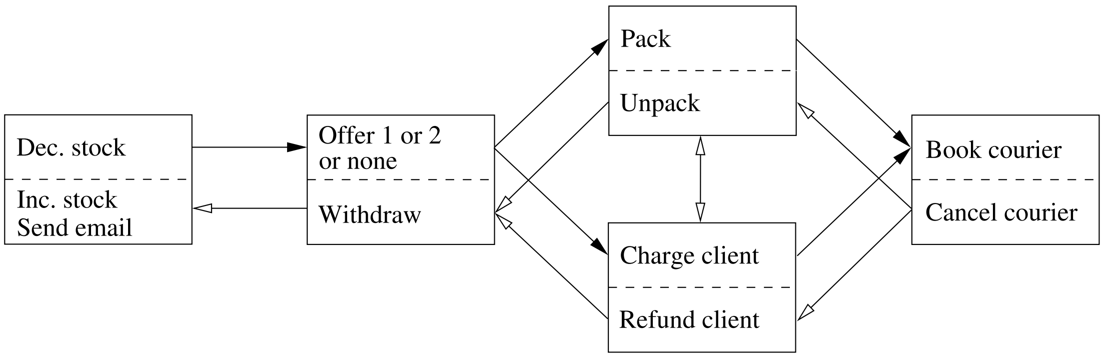


Figure 3.1: A representation of the online bookshop example.

With reference to the bookshop example, we briefly go over the issues involved in programming compensations, introducing the related terminology.

Specification A specification is required to relate compensation actions to their counterparts. At its most basic form this can be a table-like structure such that each action has a specified corresponding action. However, this is rarely sufficient to describe complex compensations where actions require contextual compensations, *i.e.,* the same action may have a different compensation depending on the context. A specification similar to the one shown in Figure 3.1 would suffice to specify contextual compensations. There are yet other complexities which would need to be specified such as the point at which a compensation expires, known as *compensation scoping*, *i.e.,* up to which point of execution a compensation remains valid. Expired compensations may also need to be replaced upon their expiration by another compensation. Compensation specifications may also include other details such as how a failure during compensation execution is to be handled, or how to continue after a successful compensation execution.

Installation During the normal runtime of a system with compensations, the appropriate compensations are stored — more technically *installed* — so that if a failure occurs later, the successful actions may be compensated by executing the stored actions. Compensation installation usually occurs upon the successful completion of an action which has a specified compensation (*e.g.,* upon successfully reducing the stock level, "increase the stock and email the client" is installed as a compensation). Another design issue concerning compensation installation is the order in which actions are installed. Usually these are installed sequentially in a stack-like structure, meaning that eventually they will be executed in reverse order of the corresponding actions.

Discarding/Replacing/Forwarding Compensations Upon completion of a transaction, it might no longer make sense to execute the stored compensations (*e.g.,* if an order has been completed and shipment has left it is usually impractical to reverse the shipment). Considering the completed order as the compensation scope, in such cases compensations are discarded upon termination of the scope. Alternatively, there are cases where it makes sense to replace the discarded compensations by another appropriate compensation (*e.g.,* the sequence of actions handling courier booking cancellation, unpacking, stock increase, *etc.,* is replaced by returning the bought goods). Note that returning the bought goods is in itself another transaction involving a number lower-level actions. For this reason such a compensation (used as replacement) is sometimes referred to as *coarse-grained compensation*. If compensations remain valid at the end of a transaction, *i.e.,* the compensation scope does not correspond to the transaction scope, then stored compensations may need to be forwarded to the parent transaction (if it exists) so that these can be executed if a failure occurs.

Executing Compensations Compensation execution typically occurs upon a failure, triggering the execution of any stored compensations at that point. However, to enable more flexibility (*e.g.,* using compensations to program normal functionality rather than failure handling) sometimes compensations are triggered by the programmer. There are a number of design issues related to compensation execution for example if a process is compensated, are concurrently-running processes also terminated and compensated? And if so, are they simply forcefully terminated, or are some operations protected from forceful termination? Finally, do concurrent processes wait for each other's termination before starting compensating, or can each one start compensating as soon as it has terminated? Another major issue is how to continue after successfully compensating. One option is to retry the failed and compensated transaction or to try out an alternative transaction which achieves a logically equivalent result (*e.g.*, after compensating a failed courier booking, one may try booking with another agency). The latter is known as *alternative forwarding*.

Handling Failure while Compensating As with any other operation, compensations may fail and appropriate action may need to be taken by compensating for the successful part of the failed compensation for instance, or using some exception handling mechanism. This raises the question of whether compensations can have compensations of their own. Subsequently, one would have to decide on how to continue after handling a failed compensation. For example one might continue executing the rest of the program or stop execution altogether, notifying a human operator.

Along the years, a rich literature on compensations has developed and there are numerous formalisations of compensations, each tackling the questions raised above in its own way. In the following section, we review a number of different formalisations with highly diverse answers to the compensation design issues with the aim of giving a fuller picture of the compensation landscape.

3.3 Compensation Formalisations

Given the numerous design aspects of compensation programming, each with a number of possible options, it is not surprising that the literature [38] provides a plethora of formalisms which tackle compensations differently. Interestingly, these formalisms do not only enable the programming of compensations but rather the programming of compensable transactions. There are two main views of compensable transactions: the *orchestration* and the *choreography* view. The former provides a centralised view of how an activity is carried out across a number of participants, while the latter focuses on the interaction each participant is expected to carry out. In more practical terms, an orchestration defines activities (possibly involving a number of participants) while a choreography defines participants (possibly involving a number of activities). Since this work focuses on the monitoring of monolithic systems (as opposed to distributed ones), the natural choice is an orchestration approach. For this reason in this section we review three orchestration approaches and one choreography approach which will be useful for comparing our solution later on[4].

3.3.1 Compensating CSP

Compensating CSP (cCSP) [26, 28] is an extension to CSP [63] with the aim of providing support for compensable transactions. In cCSP, all basic activities succeed and failures are explicitly programmed using a special *throw* activity which always fails. A compensation in cCSP is a process (possibly consisting of a single action) which is associated to another process using the $\div$ operator. Such processes, however, cannot themselves have compensations, *i.e.,* in cCSP one cannot program a compensation for a compensation. Similar to other compensation formalisms Sagas and t-calculus [20, 79], cCSP installs compensations in such a way as to reflect the forward behaviour, *i.e.,* sequential compensations for sequential processes and parallel compensations for parallel processes. A special feature of cCSP is that once a transaction completes, installed compensating actions are automatically discarded. Therefore, by associating a compensation to a transaction one would effectively be replacing the accumulated compensations with a coarser-grained compensation for the whole transaction.

Another particular feature of cCSP is that in case of a process failing within a parallel composition, it allows the programmer to decide at which point the other processes (in the composition) can be interrupted by the failure. This is achieved by using a *yield* operator such that if this operator is not used, it is assumed that the process cannot be interrupted if it is still executing. It is also

[4]For a more complete review we refer the reader to [39].

interesting how fault handling and compensations are intermixed in cCSP: when a failure occurs, the compensation is only triggered if the exception handler (preceded by the $\rhd$ operator) fails (or no exception handler is available). Finally, cCSP offers two choice operators: a normal choice operator ($\Box$) which starts either one of its operand processes, and a speculative choice operator ($\boxtimes$) which starts both operand processes simultaneously but compensates one when the other succeeds. Using these operators the bookshop scenario (from the previous section) can be encoded as follows:

$$\begin{aligned}
Order &\stackrel{\text{def}}{=} Order' \Box throw \\
ReStock &\stackrel{\text{def}}{=} ReStock' \Box throw \\
Transaction &\stackrel{\text{def}}{=} [\ (Order \div (ReStock \parallel Email));\\
&\qquad ((Offer_1 \rhd (Offer_2 \rhd skip)) \div Withdraw);\\
&\qquad ((Pack \div Unpack) \parallel (Credit \div Refund));\\
&\qquad ((Courier_1 \div Cancel_1) \boxtimes (Courier_2 \div Cancel_2))\\
&\quad] \rhd Operator
\end{aligned}$$

To model the fact that all the involved activities can possibly fail, each activity (*e.g., Pack, Credit, Unpack, etc.*) should be defined in a similar fashion to *Order* and *ReStock* where *Order′* and *ReStock′* are some lower level activities. Thus, each activity can non-deterministically fail (including compensating activities). Because cCSP does not offer an explicit construct for alternative forwarding, we have used the exception handling operator which achieves the same result: if $Offer_1$ fails, $Offer_2$ is triggered[5] while if $Offer_2$ fails, the *skip* operation is triggered. Note that due to this workaround the *Withdraw* compensation is installed even if both offers fail. Otherwise the example has been successfully modelled in cCSP, fully adhering to the description, including triggering the *Operator* action if a compensation fails (using square brackets to signify the

[5]Note that this would not have been possible if $Offer_1$ was itself a transaction having programmed compensations since exception handling can only be used on non-compensable processes.

transaction boundaries).

3.3.2 StAC

StAC [23, 24, 34] decouples the compensation mechanism from failure handling and thus compensations can be used freely as any other programming construct. Compensations in StAC are stored in so called *compensation stacks* such that compensations can be installed, executed and discarded through stack operations. This approach provides total freedom to the programmer to use compensations as deemed necessary, enabling the pattern of compensation programming to be used for any context — not necessarily because of a failure.

StAC$_i$ is an extension of StAC supporting concurrent compensation stacks, implying that several compensation scopes can be maintained concurrently during the execution of a process. For example, consider programming a booking process involving multiple sub-bookings whereupon if all provisional sub-bookings succeed they are confirmed, while if one fails, all the sub-bookings are cancelled. In StAC$_i$ this can be conveniently programmed using two compensation stacks: one for storing a confirmation and the other for storing a cancellation for each successful sub-booking. Depending on whether all sub-bookings succeed or not, the appropriate stack is activated. Additionally, StAC$_i$ (but not StAC) provides a mechanism for protecting a process from early termination originating from another process. This guarantees that processes are interrupted only when it is safe to do so.

The example of the bookstore encoded in StAC$_i$ is given in parts, explaining the StAC$_i$ syntax progressively. We start with the definition of the basic actions, using primed names to represent lower level activities:

$$\begin{aligned}
Order &\stackrel{\text{def}}{=} Order' \mathbin{[]} (\boxtimes_0 \mathbin{;} early)\\
Pack &\stackrel{\text{def}}{=} Pack' \mathbin{[]} (\boxtimes_0 \mathbin{;} early)\\
ReStock &\stackrel{\text{def}}{=} ReStock' \mathbin{[]} \odot\\
Offer_1 &\stackrel{\text{def}}{=} Offer_1{}' \mathbin{[]} \odot\\
Offer_2 &\stackrel{\text{def}}{=} Offer_2{}' \mathbin{[]} \odot
\end{aligned}$$

Recall that the example requires that any of the activities might fail. To model such behaviour we use non-deterministic choice (represented by $[]$), which in case of the failure option, any previously successful activities are compensated by activating the appropriate compensation stack (using $\boxtimes_i$ to activate stack i) and terminates execution *early*. This is the case of *Order*, *Pack*, *etc.* On the other hand, if compensating activities fail, we opt to simply signal a failure ($\odot$). Other compensating activities should be defined in a similar fashion to *ReStock*. Note that in the case of the offer activities no compensations are activated since these are optional, *i.e.,* they do not cause the whole transaction to be undone. Rather, their definitions are called from within two nested *TRY* statements which in StAC$_i$ executes the *THEN* clause if the *TRY* clause succeeds or the *ELSE* clause if otherwise.

$$\begin{aligned}
Offers \stackrel{\text{def}}{=} \;& TRY\ Offer_1 \div_0 Withdraw\\
& THEN\ skip\\
& ELSE\ (TRY\ Offer_2 \div_0 Withdraw\ THEN\ skip\ ELSE\ skip)
\end{aligned}$$

Note that if *Offer*$_2$ fails, no failure is signalled to the rest of the transaction since the failure is caught by the *TRY* statement and continues as the inert process *skip*.

The most complex part of the example is the speculative choice of the couriers as shown below:

$$
\begin{aligned}
\mathit{Couriers} \overset{\text{def}}{=} \; & \mathit{TRY}\ \mathit{new}(2).\mathit{new}(3).(\\
& \quad |\mathit{Courier}_1 \div_2 \mathit{Cancel}_1|_{\mathit{true}} \parallel |\mathit{Courier}_2 \div_3 \mathit{Cancel}_2|_{\mathit{true}}); \\
& \mathit{IF}\ \mathit{Ready}_1\ \mathit{THEN}\ \boxtimes_3 ; \{2\} \rhd 0 \\
& \mathit{ELSE}\ (\mathit{IF}\ \mathit{Ready}_2\ \mathit{THEN}\ \boxtimes_2 ; \{3\} \rhd 0\ \mathit{ELSE}\ \boxtimes_0 ; \odot))
\end{aligned}
$$

In order to encode speculative choice, we require two extra boolean variables, Ready_1 and Ready_2, which become true when $\mathit{Courier}_1$ or $\mathit{Courier}_2$ succeed, respectively. Note that the booking of the couriers is put inside a *TRY* block so that failure is contained and furthermore, each booking is placed inside a protected block so that it is protected from the other booking's failure. When the bookings complete (successfully or not), if at least one of the bookings succeeds, its compensation is relayed to the outer compensation (using $\rhd 0$ to push the contents of a stack onto the outermost stack 0) while the other compensation stack is executed. If neither booking succeeds, the compensation is executed and early termination is signalled.

Finally, the overall transaction process is given below:

$$
\begin{aligned}
\mathit{Transaction} \overset{\text{def}}{=} \; & \mathit{TRY}\ (\ (\mathit{Order} \div_0 (\mathit{ReStock} \parallel \mathit{Email})); \\
& \quad \mathit{Offers}; \\
& \quad ((\mathit{Pack} \div_0 \mathit{Unpack}) \parallel (\mathit{Credit} \div_0 \mathit{Refund})); \\
& \quad (\mathit{Couriers}) \\
&)\ \mathit{THEN}\ \mathit{skip} \\
& \mathit{ELSE}\ \mathit{Operator}
\end{aligned}
$$

Note that the transaction is enclosed within a *TRY* block so that if it results in failure, the operator is notified. The StAC$_i$ specification diverges from the example specification in that the speculative choice implementation does not stop executing as soon as one of the alternatives succeeds.

3.3.3 SOCK

SOCK [58, 59, 69] is aimed specifically as a calculus for service-oriented computing. For example, SOCK provides the request-response mode of communication through which a client can request an activity on a server and receive back the output of the activity. For such a scenario, SOCK also offers other notions such as the concept of a location — a process is not only distinguished by its definition but also by the location where it is running. SOCK provides three error handling mechanisms: fault handling (similar to traditional exception handling), termination handling (invoked when the process is externally terminated) and compensation handling — all centred around a process container called a *scope*. The scope associates fault names with fault handlers and the scope's own name is associated with the termination handler. If the scope terminates and the termination handler has not yet been invoked, then the termination handler becomes the compensation handler for that scope (as the scope has been successfully completed). Subsequently, the name of the scope can be used to trigger the compensation handler in case the successful scope needs to be compensated. In SOCK, handlers (any type) can be modified at any point of execution. This provides a high degree of flexibility and does not impose any predefined policy on the programmer. A special feature of SOCK is that it provides a mechanism for distributed compensation, allowing a server to send a failure handler to the client. Thus, if the operation on the server fails, the client is informed by the server how compensation can take place.

SOCK models interaction among processes as channel communication with c and $\overline{c}$ representing input and output on channel c, respectively. Furthermore, compensation and failure handlers are installed as processes attached to handler names and scope names. For example $s \mapsto cH \,\|\, (IncStock \,\|\, Email)$ represents the fact that in parallel with anything already associated with handler s (represented by cH), two additional actions are composed: increasing the stock and sending an email to the client. The example modelled in SOCK is given below in parts:

$$
\begin{aligned}
Order' &\stackrel{\text{def}}{=} order;(\overline{x} \,\|\, (x + (x; throw(st))))\\
Order &\stackrel{\text{def}}{=} \overline{order}([s \mapsto (ReStock \,\|\, Email), st \mapsto throw(f)]) \,\|\, Order'\\
ReStock' &\stackrel{\text{def}}{=} restock;(\overline{x} \,\|\, (x + (x; throw(rs))))\\
ReStock &\stackrel{\text{def}}{=} \overline{restock}([rs \mapsto throw(g)]) \,\|\, ReStock'
\end{aligned}
$$

Modelled in a service-oriented fashion, the *Order* activity contacts the *Store* through channel *order* and if the activity succeeds (modelled as a non-deterministic choice using the + operator), the activity exits, otherwise, it throws fault *st*. This fault is handled by *DecStock* and rethrown as fault f which is in turn handled by *Transaction* (below) — triggering the compensation for scope s. If the compensation (*ReStock*) fails, fault g is triggered instead of f, causing the *Operator* activity to start. The activities which are not given above should be modelled in a similar fashion to *Order* and *ReStock*.

$$
\begin{aligned}
Offer_1 &\stackrel{\text{def}}{=} \overline{offer_1}([failedOffer \mapsto Offer_2])\\
Offer_2 &\stackrel{\text{def}}{=} \overline{offer_2}\\
Offers_1 &\stackrel{\text{def}}{=} (offer_1;(\overline{x} \,\|\, (x; inst([s \mapsto (cH; Withdraw)])))) +\\
&\qquad (x; throw(failedOffer))\\
Offers_2 &\stackrel{\text{def}}{=} offer_2;(\overline{x} \,\|\, (x + (x; inst([s \mapsto (cH; Withdraw)]))))
\end{aligned}
$$

If the first offer fails, the exception *failedOffer* is thrown. However, this is caught by $Offer_1$ and is not propagated further. *failedOffer* is handled by attempting $Offer_2$. If this fails as well, no action is taken. On the other hand, if either of the offers succeeds, the compensation *Withdraw* is installed. Next we consider the courier booking:

$$\begin{aligned}
\mathit{Courier} \overset{\text{def}}{=} \; & \{\overline{\mathit{courier}_1}([c_1 \mapsto \mathit{Cancel}_1, \mathit{failedCourier}_1 \mapsto \overline{\mathit{failed}_1}]);\\
& \quad \overline{\mathit{booked}_1}\}_{c1} \parallel\\
& \{\overline{\mathit{courier}_2}([c_2 \mapsto \mathit{Cancel}_2, \mathit{failedCourier}_2 \mapsto \overline{\mathit{failed}_2}]);\\
& \quad \overline{\mathit{booked}_2}\}_{c_2} \parallel\\
& ((\mathit{booked}_1; (\mathit{failed}_2 + (\mathit{booked}_2; \mathit{comp}(c_2));\\
& \qquad \mathit{inst}([s \mapsto (cH; \mathit{Cancel}_1)]))) +\\
& \quad (\mathit{failed}_1; (\mathit{booked}_2; \mathit{inst}([s \mapsto (cH; \mathit{Cancel}_2)]) +\\
& \qquad (\mathit{failed}_2; \mathit{throw}(f)))))
\end{aligned}$$

For the *Courier* process, the non-deterministic choice has been used to distinguish among the possible outcomes. If both succeed, then the second courier is compensated by running the compensation ($\mathit{comp}(c_2)$) associated to the scope. If either of them fails and the other succeeds, then the appropriate compensation is added to the higher compensation scope s. Finally, if both fail, then the fault f is thrown, causing the whole transaction to fail.
We end this example by giving the topmost transaction which refers to the above defined processes:

$$\begin{aligned}
\mathit{Transaction} \overset{\text{def}}{=} \; & \{\mathit{inst}([f \mapsto \mathit{comp}(s), g \mapsto \mathit{Operator}]) \parallel (\mathit{Offers}_1 \parallel \mathit{Offers}_2) \parallel\\
& \quad (\mathit{Order}; \mathit{Offer}_1; (\mathit{Pack} \parallel \mathit{Credit}); \mathit{Courier})\}_s
\end{aligned}$$

At the transaction level, we compose the previous definitions and install fault handlers f and g, the former for initiating the compensation of scope s and the latter for triggering the *Operator* action if a failure occurs during compensation.

Note that in SOCK we have fully kept to the example specification.

3.3.4 Communicating Hierarchical Transaction-Based Timed Automata

Communicating hierarchical transaction-based timed automata (CHTTAs) [71, 72] are communicating hierarchical machines [5] enriched with time (similarly to timed automata [4]), and with other slight modifications to accommodate the representation of transactions. Two appealing features of CHTTAs (apart from the inherent graphical aspect) is that they support the notion of time and can be reduced to timed automata (making them model-checkable). Long running transactions (LRTs) are defined over and above CHTTAs such that a CHTTA can be specified as the compensation of another CHTTA. Furthermore, LRTs can also be nested or composed in parallel or sequentially. Similar to a number of other approaches such as cCSP, the order of compensation execution in LRTs is in reverse order in case of sequential composition and in parallel in case of a parallel composition. A major limitation of LRTs is that they do not show clearly (graphically) which compensation corresponds to which component. Indeed the main motivation for being automata-based seems to be the benefit of model-checking techniques for time automata rather than clarity for the programmer. Corroborating this claim is the fact the programmer would typically more likely program compensations in LRTs rather than CHTTAs, leaving automata behind the scenes. Thus, although automata-based, programming using LRTs is effectively similar to logic programming, giving up the advantages of programming with automata.

LRTs are defined on compositions of CHTTAs which in turn are defined on top of transaction-based timed automata (TTAs). A CHTTA is the parallel compositions of hierarchically arranged TTAs where a TTA is a timed automaton with a number of additions: (i) a set of communicating channels enabling automata to communicate; (ii) super states to enable hierarchical composition of automata; (iii) two special states $\odot$ and $\otimes$ representing the *commit* state and the *abort* state respectively; and (*iv*) special transitions labelled with $\boxdot$ or $\boxtimes$ which are the only transitions possible from superstates representing success and failure respectively.

Although supporting special states and transitions, hierarchy and communication, note that CHTTAs do not support compensations directly, instead they rely on sophisticated communication arrangements across CHTTAs to encode compensation control flows. To facilitate the programming of such flow, LRTs provide the syntactic sugar for typical basic compensation constructs, namely: (*i*) designating a CHTTA as the compensation of another: given two CHTTAs A and B, an LRT $L = A \Rsh B$ signifies that B is the compensation of A; (*ii*) sequential composition of two LRTs, denoted by $L_1 \cdot L_2$; (*iii*) parallel composition of two LRTs, denoted by $L_1 \,||\, L_2$; and (*iv*) a nested LRT, written $\{L\}$ signifying that the failure of L should not trigger compensation to the parent transaction.

For example, the LRT $L = A_1 \Rsh B_1 \cdot \{A_2 \Rsh B_2\} \cdot A_3 \Rsh B_3$ (where $A_1, A_2, A_3, B_1, B_2, B_3$ are CHTTAs) should (*i*) execute A_1 followed by A_2 followed by A_3; but (*ii*) if A_1 fails, execution stops; while (*iii*) if A_2 fails, execution progresses normally to A_3 since A_2 is nested; (*iv*) if A_3 fails, A_2 has to be compensated (if it has not failed) followed by A_1's compensation; and (*v*) if all of A_1, A_2 and A_3 succeed, then T commits and it can be undone through the compensation B_3, followed by B_2, followed by B_1. Pictorially, the LRT is shown in Figure 3.2[top] while its compensation is shown in Figure 3.2[bottom]. In particular note how the wiring across automata is done through special transitions $\boxdot$ and $\boxtimes$, and communicating channels. For example the nested CHTTA A_2 will always lead to a commit state because its failure should not affect its parent. Furthermore, channels *no* and *yes* record whether the compensation for A_2 needs to be executed if the parent is compensating. Finally note that it is assumed that compensations always succeed, a significant limitation of CHTTAs.

There are a number of limitations in LRTs which do not allow the faithful specification of the bookshop example. Unless one manually adds more op-

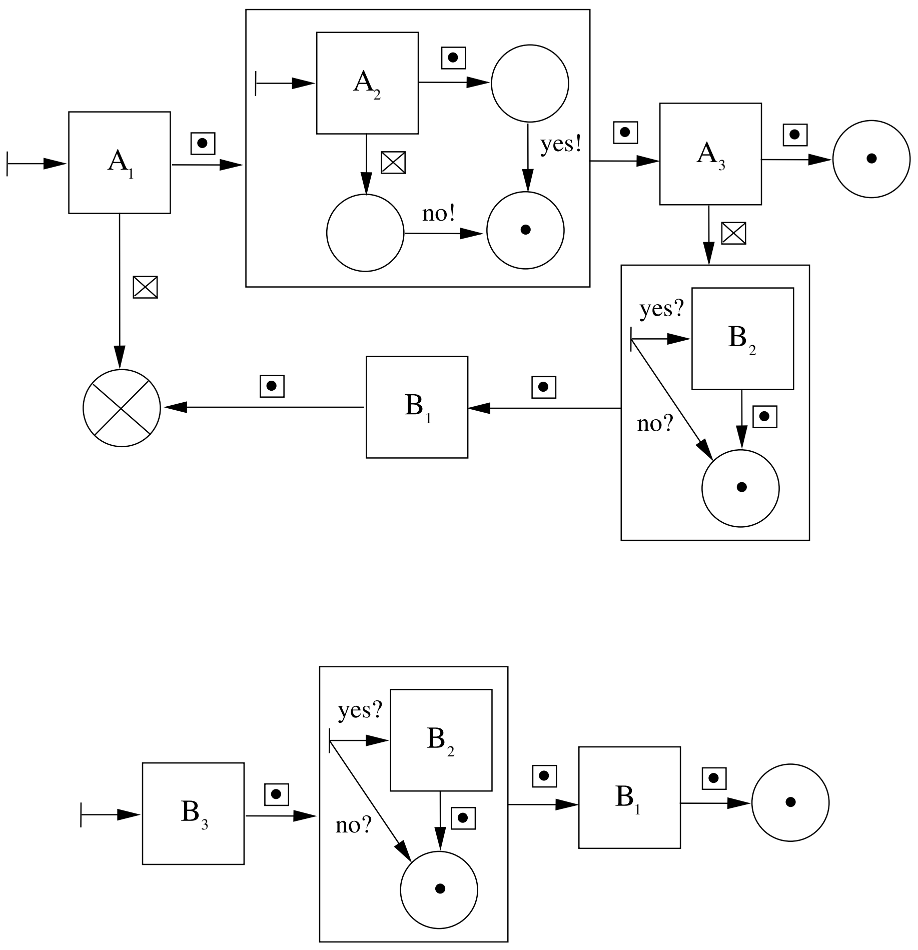


Figure 3.2: A representation of $L = A_1 \Rsh B_1 \cdot \{A_2 \Rsh B_2\} \cdot A_3 \Rsh B_3$ [top] and its compensation [bottom]

erators through channel communication, features such as exception handling, alternative forwarding and speculative choice are not available. Thus the bookshop example in terms of LRTs can be encoded a follows:

$$\begin{aligned}&(\mathit{Order} \Rsh (\mathit{ReStock} \parallel \mathit{Email})) \cdot \{\mathit{Offer}_1 \Rsh \mathit{Withdraw}\} \cdot \\ &\quad ((\mathit{Pack} \Rsh \mathit{Unpack}) \parallel (\mathit{Credit} \Rsh \mathit{Refund})) \cdot (\mathit{Courier}_1 \Rsh \mathit{Cancel}_1)\end{aligned}$$

Note that the following features could not be encoded: (*i*) alternative forwarding to try the second offer if the first one fails; (*ii*) speculative choice among the couriers; and (*iii*) exception handling reporting failure to a human operator.

The basic activities such as *Order* and *ReStock* can be encoded as a CHTTA having channel communication with a third-party which then communicates back on other channels indicating whether the action was successful or not. This is depicted in Figure 3.3.

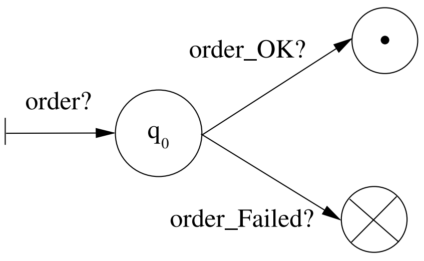


Figure 3.3: A representation of the *Order* CHTTA.

3.3.5 Discussion

The four formalisms presented in this section provide a rich backdrop for comparison due to their diversity in approaching compensations. cCSP is notorious for its succinctness, mainly due to the significant number of dedicated operators provided. However, cCSP is not flexible — attempting to model a non-standard compensation pattern is impossible. If one requires flexibility StAC would be a much better option, giving full control of the stack to the programmer. Naturally, this comes at the cost of longer and less straightforward specifications. Similar comments can be made with respect to SOCK which provides a rich mixture of error handling mechanisms which is fully programmable. Furthermore, SOCK gives visibility to interactions across entities where each action is modelled in terms of channel communication. Admittedly, for the example above, SOCK is not the natural choice since we attempted to model a process at a bookshop rather than a choreography across different entities. This contributed to making the SOCK definitions seem somewhat unintuitive. However, one can still appreciate the advantages of SOCK had it been used to model a business process with a focus on the participant interactions rather than the activities. Finally, we have presented the example in CHTTAs. Unfortunately, while being the only automata-based notation in the compensation literature which we are aware of, the pictorial aspect does not really aid the programmer to write the compensation specifications.

While significantly diverse, the formalisms presented have at least one thing in common: they do not simply program the compensations of a system, they program the system with its compensations. When compensation logic might already be significantly complex, incorporating it with the system logic does not promise to keep things at their simplest. It is not surprising then that the semantics behind the compensation formalisms presented are not straightforward to say the least. This issue is the main motivating factor for Part II of this report.

Another worrying element in the compensation literature is that little work has been done on formalising and proving the soundness of compensation formalisms. In fact of the four formalisms presented only the first one defines a notion of compensation soundness. This aspect is further elaborated in the next section.

3.4 Formalising Compensation Correctness

The complexity of compensation semantics raises the question of the soundness of a given compensation language or approach as to whether it really deals with compensations correctly. Self-cancellation has been proposed for cCSP [26] in an attempt to formalise the correctness criterion of compensation semantics. In essence self-cancellation states that, *a system built from atomic actions, all of which have a perfect compensation*[6] *associated to them, will either successfully complete its*

[6]The strong assumption of perfect compensations is rarely the case in practice, but is only used to ensure that the order of execution of triggered compensations is as expected. How *perfect* a compensation is, depends on the level of abstraction one is viewing the behaviour. For example *delete* is a perfect compensation for *insert* if one views an index at the level of abstraction of an index. It is highly improbable though, that at the bit level the original index is identical to the index following an insert and a compensating delete.

behaviour, or will be aborted but will not leave any side effect on the system state.

The approach adopted to define soundness, is to consider perfect compensation of basic activities, and proving that the formal semantics guarantee that the derived compensation of a transaction (built compositionally from basic activities) is still a perfect compensation of the transaction. The semantics of a compensable process in cCSP is given as sets of pairs of traces — the forward behaviour, and the accumulated backward (compensating) behaviour. The sanity check for compensating programs is thus that the trace semantics of a transaction block may only include: (i) successfully terminating traces; and (ii) traces which after cancellation are equivalent to the inert action *skip*. For the sanity check to hold, two assumptions are crucial: (i) each basic action A and its compensation A' satisfy $A, A' = skip$, *i.e.,* their sequential occurrence is equal to *skip*; and (ii) parallely composed compensation actions commute, *i.e.,* with respect to a formula $(A \div A') \,||\, (B \div B')$, the trace A, B, A', B' is equal to the trace A, B, B', A', enabling actions to cancel with their corresponding compensations[7].

We adopt the same approach in the rest of this work for proving the sanity of our theory. To enable reasoning about system behaviour and compensations, we will be talking about finite strings of events.

Definition 3.4.1. Given an alphabet Σ, we will write Σ^* to represent the set of all finite strings over Σ, with ε denoting the empty string. We will use variables a, b to range over Σ, and v, w to range over Σ^*. We will also assume a subset of actions $\Gamma \subseteq \Sigma$ indicating internal system behaviour, which will be ignored when investigating the externally visible behaviour.

Given a string w over Σ, its *external manifestation,* written w^x, is the same string but dropping instances of Γ elements.

Two strings v and w are said to be *externally (or observationally) equivalent,* written $v =_x w$, if their external manifestation is identical: $v^x = w^x$. We say that a set

[7]Note that associativity — enabling compensations to cancel with their corresponding actions — is implicit as a property of the trace structure.

of strings W is contained in another set W' up to external manifestation, written $W \subseteq_x W'$, if for every string in W, there is an externally equivalent string in W'. Set equality up to external manifestation, $W =_x W'$, is defined as containment in both directions. External equivalence is an equivalence relation, and a congruence up to string concatenation. □

For every event that happens in the system, we will assume that we can automatically deduce a unique[8] perfect compensation.

Definition 3.4.2. Corresponding to every event a in alphabet Σ, its compensation will be denoted by $\overline{a}$ where we assume $\overline{\cdot}$ to be a total injective function. We will write $\overline{\Sigma}$ to denote the set of all compensation actions. For simplicity of presentation, we will assume that the set of events and that of their compensations are disjoint[9].

We also overload the compensation operator to strings over Σ, in such a way that the individual events are individually compensated, but in reverse order: $\overline{\varepsilon} \overset{\text{def}}{=} \varepsilon$ and $\overline{aw} \overset{\text{def}}{=} \overline{w}\,\overline{a}$. For example, $\overline{abc} = \overline{c}\overline{b}\overline{a}$. □

To check for consistency of use of compensations, the approach is typically to consider an ideal setting in which executing a, immediately followed by $\overline{a}$ will be just like doing nothing to the original state. Although not typically the case, this approach checks for sanity of the triggering of compensations.

Definition 3.4.3. The compensation cancellation of a string simplifies its operand by removing actions followed immediately by their compensation. We define $cancel(w)$ to be the shortest string for which, after dropping all internal actions, there are no further reductions of the form $cancel(w_1 a\overline{a} w_2) = cancel(w_1 w_2)$.
□

[8]Uniqueness of compensations ensures that no compensation can cancel out with more than one action.

[9]One may argue that the two could contain common elements — *e.g.*, *deposit* can either be done during the normal forward execution of a system, or to compensate for a *withdraw* action. However, one usually would like to distinguish between actions taken during the normal forward behaviour and ones performed to compensate for errors, and we would thus much rather use *redeposit* as the name of the compensation of *withdraw*, even if it behaves just like *deposit*.

Proposition 3.4.1. Strings may change under cancellation only if they contain symbols from both Σ and $\overline{\Sigma}$. Cancellation reduction is confluent and terminates.

Proof. The proof follows from the fact that the sets of normal and compensation events are disjoint, by the definition of *cancel*(), and by the injectivity property of the $\bar{\cdot}$ function. □

Definition 3.4.4. Two strings w and w' are said to be *cancellation-equivalent*, written $w =_c w'$, if they reduce via compensation cancellation to externally equivalent strings: $cancel(w) =_x cancel(w')$. A string w for which $cancel(w) =_c \varepsilon$ is said to be *self-cancelling*.

A set of strings W is said to be *included in set W' up-to-cancellation*, written $W \subseteq_c W'$, if for every string in W, there is a cancellation-equivalent string in W':

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

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

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

Proposition 3.4.2. The concatenation of a string with its compensation is cancellation equivalent to the empty string: $\forall w \cdot w\overline{w} =_c \varepsilon$.

Proof. The proof follows by string induction on w using Definition 3.4.2. □

These definitions and propositions are the basis of the theory which we present in the coming chapters. While simple, the notion of self-cancellation is at the heart of what it means for a compensation approach to be sane. Indeed, the relaxed notion of atomicity which is synonymous to compensating transactions, relies on self-cancellation — if a failure occurs, executing compensations is as if nothing has been executed. In later chapters, the theory of self-cancellation proves useful to demonstrate the soundness of our approach.

3.5 Conclusion

In complex systems, particularly systems interacting with real-life processes, error recovery is crucial to increase fault tolerance. Whilst closed systems can benefit from automated localised backward error recovery, this is not possible where interaction occurs with non-reversible real-life processes. This motivates compensations — providing a means of execute actions which semantically undo partially successful activities.

Throughout the years since the inception of the idea of compensations, a number of common basic aspects of compensations have emerged. Still, these aspects are distinguishably combined in a plethora of compensation models which can be found in the literature. These divergences [38] with little reflection about the differences and the soundness indicate that the study of compensations is not yet over.

A major drawback of compensation formalisms is that they do not separate compensation programming from programming the rest of the system. This is unfortunate since compensation programming is itself not straightforward. We believe this to be a serious limitation which we address in the next chapter.

Part II

Runtime Verification for Compensations

This part outlines how monitoring can significantly contribute towards compensations, particularly by facilitating their programming in a modular fashion.

4. Compensating Automata

Compensations have become particularly useful in areas which program real-life processes such as in work 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) [9]. From an academic point of view, extensive research [38] has been conducted in the area, particularly by suggesting different formal models of compensations [21, 24, 26, 59, 72] and defining formal semantics for BPEL [49, 52, 62, 75] in which compensations play a crucial role.

Since compensations enable the logical reversal of past actions whose actual ordering would only be known at runtime (as is the case with non-deterministic systems *e.g.,* concurrent systems and user-input-based execution), their programming has to be *dynamic*, *i.e.,* “programmed” at runtime. While programming compensations statically is possible, it limits their expressivity [68] and they cannot affectively mirror the actual runtime behaviour. This contrasts with programming forward recovery which is usually programmed statically in a *try-catch-block* fashion. Thus, it is no surprise that using traditional means of code organisation for programming (dynamic) compensations results in unstructured code: such code would have to continually keep track of the execution history which is crucial for programming two main aspects of compensations: *what* to compensate for, and *how* to compensate for it. For example if a purchase fails, the system has to compensate only for the actions which have been completed — constituting *what* is to be compensated. Furthermore, *how* to compensate would entail issues such as: the purchase payment 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-keeping of the customer's history and a mechanism through which the applicable refund action is associated to the payment action as its compensation. Such additions clutter the code and intertwine programming of system actions with their compensations, making it difficult for the programmer to manage.

A recurrent approach for structuring compensation logic [38] involves associating 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 the interplay of system logic and compensation logic (*e.g.,* a failing transaction might trigger compensation execution but at some stages the system might retry some actions to avoid the costly compensation of the whole transaction), or compensation logic spanning different modules (*e.g.,* refunding a payment only once the transport cancellation has been carried out so that any cancellation costs can be factored out of the refund)[1]. In the case of exception handling, this limitation has been overcome by technologies and approaches such as aspect-oriented programming [66] and monitor-oriented programming [84]. The advantage of these programming paradigms is that they allow the programmer to program cross-cutting concerns in a modular fashion.

Inspired by these solutions, we propose an automata-based monitor-oriented notation, *compensating automata*, solely dedicated for programming the *what* and *how* questions of compensations. The novelty of this notation is that it can

[1]See [57] for more concrete examples.

only be used to program compensations with respect to a given input of system events, *i.e.,* it cannot be used for programming the system itself. This approach enables a separation of concerns with the programmer only focusing on *what* system actions to compensate for and *how* to compensate for such actions. Note that through this arrangement compensation programming can easily cross-cuts all the other system modules since events which reach compensating automata may emanate from any part of the system (potentially even from third party systems).

Through an e-procurement system case study from the literature [57], we present in more depth the problem of programming compensations (Section 4.1). Next, we propose compensating automata (Section 4.2) as a means of programming compensations and show how these can be used to program compensations for the e-procurement system (Section 4.3) which existing compensation notations have difficulty in handling.

4.1 Designing a Compensation Notation

Attempting to program complex compensations using standard compensation formalisation has been shown to be impractical for particular case studies such as the e-procurement scenario presented in [57]. Interestingly, in what follows we show that programming the e-procurement scenario without handling compensation concerns is relatively straightforward. This leads us to expect that separating compensation concerns from other programming concerns would greatly simplify the programming of systems having compensations. To this extent, throughout this section we describe a novel design paradigm which enables compensations to be programmed separately — leaving the system code uncluttered.

4.1.1 An E-Procurement Case Study

A procurement is a business process involving a *merchant* supplying the goods, and a *customer* ordering them. The basic logic of a procurement starts by the merchant receiving a quote request from a customer. The merchant then checks that the customer is a valid one — that is, registered and with no overdue payments. If invalid, a message informing the customer that he or she can no longer place an order 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 vanilla version of the e-procurement system (S) may be defined in terms of three programs: reservation of goods (R), payment (P), and transport (T) such that the parallel composition of P and T follows R, written $S = R;(P \,||\, T)$, and each program may be expressed as follows[2]:

[2]Abbreviations: rec (receive), req (request), msg (message), not (notification), ack (acknowledgement).

Program *R*

```
RecQuoteReq
If !checkCustomer
Then
  sendInvalidCustomerMsg
  Return
CalculatePrice
SendQuote
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.* 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 of delivery, then the costs are passed onto the customer in either of the following ways: 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 (assuming the charge is cheaper than the cost of the order, otherwise the discrepancy is invoiced to the customer).

5. The merchant can choose whether to accept or to reject user-initiated cancellation requests (*e.g.,* some goods cannot be returned). 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 way of handling such a failure (such as a failure to affect payment) is to periodically retry the activity until it succeeds.

8. If the goods were sent to a wrong address, the shipment is first returned to the merchant, and after attempting to determine the correct address, the shipment is retried.

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, i.e. compensations are logical reverses of corresponding activities, a substantial number of the listed divergences 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. Note that this approach is significantly different from that taken in [57] where the difficulty of programming these divergences in terms of compensations is considered as a limitation of the compensation paradigm.

Taking our compensation definition into account, we modify the system to include the non-compensation-related divergences. 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)[3].

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)
VerifyAddr
  If (UnavailableAddr
        || AddrOk)
    Compensate
  Else If (CorrectedAddr)
    Run(T′)
```

[3]Abbreviations: rec (receive), req (request), msg (message), not (notification), addr (address), ack (acknowledgement).

Thus, we have effectively defined a new system, $S' = R'; (P' \,||\, T')$, which however, still does not address all the features. Particularly, the system does not pause to process user cancellations and compensate 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, $S'' = U \,||\, S'$, which knows when compensation is required (denoted by the *compensate* action) but does not know what or how to compensate. We propose to program compensations separately through a novel architecture on which we elaborate in the following subsection.

4.1.2 Proposed Architecture

To support the specification of the e-procurement case study we propose to use a dedicated compensation notation which enables total separation of concerns as pictorially shown in Figure 4.1: a compensating specification component listens to relevant system events[4] and at any point the system may signal the activation of the configured compensations through the signal *compensate*. At this point the compensation specification starts signalling compensations to be executed at the system side while still listening to any relevant events during the system's execution of compensations. When compensation execution stops, control is passed back to the system through the *deviate* signal.

[4]We will be assuming a fully ordered sequence of events.

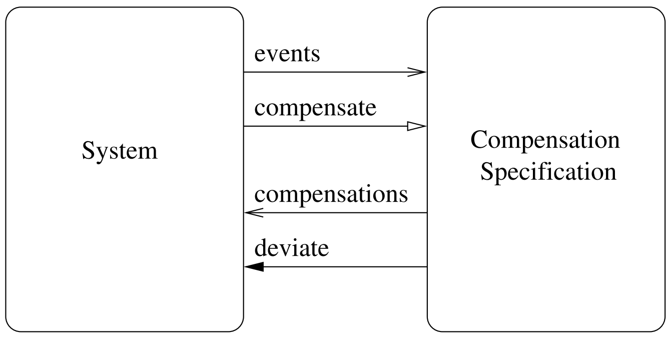


Figure 4.1: The proposed compensation programming architecture

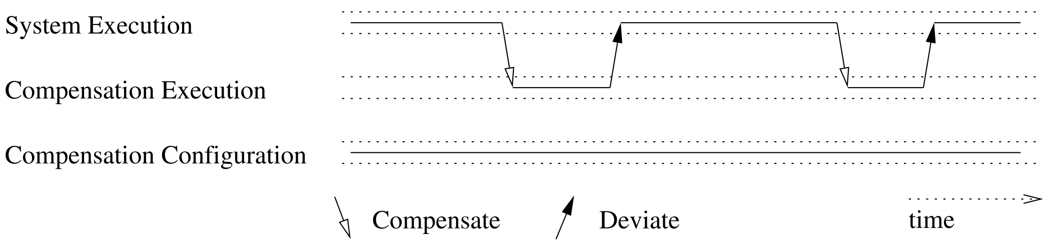


Figure 4.2: The interplay of system and compensation execution

A typical timing diagram of the interplay between system execution and compensation execution could be depicted as shown in Figure 4.2 where control passes back and forth between the system and the compensation specification module. Note that the compensation module continues performing compensation configuration even during compensation execution. This is crucial for configuring compensations for compensations such as for example compensating a failed return of goods in the e-procurement scenario.

In view of the proposed architecture, the compensation module should be able to both listen to the system events for configuring compensations, while also being able to instruct the system what actions to execute and when to continue normal execution. More concretely the two aspects of the architecture involve:

Configuration of compensations Upon designated system events, the compensation module is responsible for installing and scoping compensations with the possibility of replacing compensations upon being discarded at the end of their scope.

Executing compensations Configured compensations should be executable at the system side in programmer-specified orderings and at programmatically specified points, after which the control of execution should be returned back to the system.

Note that the proposed architecture is conceptually similar to monitoring-oriented programming (MOP) which has been introduced in Section 2.1.3 — the compensation module in our architecture is essentially a monitor. However, a significant difference between MOP and our proposal is that MOP has been proposed as a means of detecting *when* a particular reparation is to be executed, i.e. it is the monitor and not the system which triggers reparation. On the other hand, in our approach the monitor's aim is to configure the "reparation" according to the observed sequence of events, effectively deciding *what* is to be compensated and *how*, i.e. it is still the system's responsibility to trigger compensations.

A crucial aspect of MOP is the specification language from which the monitor is synthesised. Accordingly, the proposed architecture requires a specialised compensation specification language which is able to express the elements presented above. In the next subsection, we delve deeper in the design of such a compensation notation by considering the design options in the literature and choosing the best suited approach in view of the e-procurement system. Following this, in Section 4.1.4 we give an informal introduction to the proposed compensation notation showcasing the basic constructs through examples.

4.1.3 Design Options in Compensation Programming

The main challenge of designing a compensation notation which would fit within the architecture presented in the previous subsection is that there is a significant underlying difference from existing notations — the compensation notation we are designing should be purely dedicated to compensation programming. The notations we have reviewed in the previous section and other similar ones in the literature (see [38]) are more concerned with programming the "transaction which has compensations" rather than the "compensations which happen to belong to a transaction". In other words, the former are transaction-centric while our approach is compensation-centric. As such, in our proposed notation we provide no notion which is equivalent to transactions; the only related construct is that of the compensation scope (which frequently coincides with a transaction). Thus, it is no surprise that some aspects related to compensation programming found in other notations have no equivalent in our proposed notation. The most obvious example is that of alternative forwarding: this operator enables the compensation of an action to be followed by the execution of an alternative. Note that while executing compensations for a failed action should be programmable in our notation, choosing an alternative, *i.e.,* a way of continuing after compensating, is the prerogative of the system. Hence, in our strict separation of concerns approach, alternative forwarding can be encoded implicitly through the alternation between system and compensation execution.

In what follows we outline the design decisions involved in the proposed notation in view of the design choices of related work in the literature. The structure used is inspired from the compensation design choices presented in [38] and is organised according to the compensation life cycle stages (see Section 3.2), with each subsection corresponding to a different life cycle stage.

1. **Specification of Compensations** In this point we focus on designing the way compensations can be specified in the novel notation. In particular, we discuss a number of aspects: (i) compensation operators — what operators are supported for specifying compensations; (ii) action compensations versus transaction compensations — whether a compensation is itself a basic action or a transaction; (iii) compensating for failure of compensations — whether compensations can have compensations; and (iv) concurrent compensation scopes — how simultaneously active scopes can be dealt with.

Compensation Operators We have identified four main compensation operators used in the literature [38]: (i) the scope operator, which is responsible for defining the boundaries up to which a compensation remains valid; (ii) the compensation installation operator, which is responsible for associating compensations to a scope — ready to be activated in case of a failure; (iii) the compensation discard operator which discards the installed compensations in the active scope; and (iv) the compensation activation operator which triggers the execution of the currently installed compensations.

Compensation operators, with the exception of the scope operator which is always explicit, can be provided explicitly or implicit (via default behaviour). Unlike a number of notations [22, 70, 72] which install compensations implicitly at the start of a transaction scope, we choose the approach of enabling compensations to be explicitly designated as the compensations for system events (similar to [21, 26, 79]): when the system event occurs, the compensation action is installed. This allows more flexibility to the programmer since compensations are more finely defined and a clear (intuitive) correspondence can be drawn between actions and their compensations.

For the discard operation, several notations [22, 26, 70] implicitly associate it to the termination of a transaction scope. Since our proposed notation does not have the notion of a transaction, we provide a dedicated compensation scope which is dedicated to discarding and replacing compensations. Indeed, when the compensation scope does not coincide with the transaction scope, the approach we propose is the only option. For example consider the concurrent execution of two transactions composed of actions $a;a'$ and $b;b'$ respectively. If the sequence a,b should be compensated by c, there is no way a formalism such as cCSP can encode such a specification, and similarly for any compensation pattern crossing over the transaction boundary. On the other hand, formalisms which give full freedom to the user such as StAC$_i$ are able to model such scenarios but do not provide a structured means of doing this. We believe that the compensation scope suggested for our notation is a good compromise between customisability and adequate support for compensation programming.

As regards compensation activation, our notation is completely dependent on the *compensate* signal from the system. This gives freedom to the system to use the compensation module as required, not necessarily using compensations as a means of failure recovery (similarly to the concept of MOP).

Over and above the operators found in the literature, our compensation notation requires an additional operator, termed *deviation*, which transfers control back to the system. Naturally, this operator is not required in other notations since the system and compensations are implicitly intertwined and they assume that once compensation starts, the whole transaction is compensated. As motivated by the e-procurement scenario, the interplay between the system and its compensations is not always clearcut. For example, if the shipping of an order failed, one reason may be a wrong address. In such a case compensating the whole transaction is an extreme measure when the transaction may be fixed by simply reshipping the order to the correct address.

Basic Action Compensations versus Transaction Compensations

A compensation may either be an action or a whole compensable transaction itself. The argument in favour of having only transactions as compensations [9, 16, 22, 24, 69, 70, 74, 97] is that this provides a uniform approach such that the compensation may itself fail and have its own compensations. On the other hand, a number of formalisms [21, 26, 28, 79] choose to allow basic actions to have actions as compensations while transactions have transactions as compensations, thus providing a clear correspondence between actions and their corresponding compensating actions. The problem with this might be that in reality a compensation for an action may not itself be an action. For example, the compensation for a bank refund might involve the actual money transfer together with an email being sent as an explanation to the client. In the case of our proposed notation, since there is no notion of a transaction, a basic compensation should simply be a place holder for potentially complex logic which is outside the scope of the notation. However, such a compensation might still require compensation programming itself. Thus our approach is a hybrid of the two found in literature: compensations are actions but possibly having compensations.

Concurrent Compensation Scopes All notations except StAC$_i$ [24] enable only one compensation scope to be active at a time, *i.e.,* all compensations are installed within a single scope. The advantage of having multiple active scopes is that the programmer can choose which compensation scope to install to, and eventually activate. In our proposed notation we follow this same approach which is useful when compensating for parallel processes or when system event patterns overlap. Furthermore, since all active compensation scopes are executed in parallel when activated, this feature is also useful for running compensations concurrently. Consider again the execution of $(a;a') \parallel (b;b')$ with compensation c for a followed by b, and c' for a' followed by b'. Having two concurrent scopes would not only enable the potential matching of both patterns, but also the concurrent execution of c and c'.

2. **Installation of Compensations** The first arising issue concerning compensation installation is which points during the execution of a transaction

can be used for compensation installation. Another important question is the policy to be used in the ordering of compensations at the installation location. The problem is particularly complex when there are parallel processes — should the respective compensations also be activated in parallel or in the order in which execution has actually taken place at runtime? This is a delicate question to which one finds a variety of answers in the literature.

Compensation Installation Points Usually, compensation installation occurs upon the completion of an activity [21, 26, 28, 79] since a compensation can be thought of as the reverse of that activity. However, sometimes one would like to compensate for a whole transaction rather than a single activity and for this reason it is also reasonable to install a compensation upon the completion of a transaction [9, 16, 74]. Still there are various other options of allowing compensation installation at particular points in a transaction (*e.g.,* after interactions [97]) or possibly allowing the installation of compensations anywhere during the execution of a transaction [24, 69]. In the case of our notation, since the compensation module is driven by system events, we follow the more flexible approaches and allow the programmer to define what events are and what compensation to be installed upon such events. Having said this, one would generally expect the system events to represent the end of system activities, and in this sense we follow the majority of the literature instances.

Composing Compensations When installing a compensation, this is typically composed in some way to the already accumulated compensations for that particular scope, defining the order in which compensations are executed (if later activated). While any ordering is theoretically possible, there are four main options in the literature: (*i*) install compensations always in parallel [16, 97]; (*ii*) install compensations always in sequence — achieving reverse order of execution [9, 24]; (*iii*) install compensations such that they match the forward behaviour, *i.e.,* parallel compensation for parallel activities and sequential compensation for sequential activities [21, 26, 28, 72, 79]; or (*iv*) user-specific [69]. Since event patterns received from the system are inherently sequential, we opt for option (*ii*), *i.e.,* compensations are always installed sequentially. However, compensations can still be executed in parallel through concurrent scopes.

3. **Replacing and Discarding Compensations** Once the end of a transaction is reached, a decision is required regarding the accumulated compensation: if transactions are considered independent (as opposed to nested), the compensation may either be maintained as the compensation of a past transaction [69, 74], or discarded [22, 70] — implying that completed transactions cannot be compensated; if transactions are considered nested, the compensation of a child transaction may either be forwarded as is to the parent transaction [9, 97], or discarded with a coarser-grained replacement installed as part of the parent transaction's compensation [16, 26]. Considering the e-procurement scenario it is clear that compensations are sometimes only valid in a particular context, *e.g.,* as soon as the ship leaves, it is futile attempting to perform the basic compensations of cancelling the order; instead more complex compensations are required. Thus, compensation replacement is sometimes unavoidable. Given that our proposed notation does not have a notion of a transaction, providing a structured compensation scoping operator seems to be a plausible way of handling compensation replacement. Note that this scope is not necessarily related to a transaction scope and thus effectively we allow the programmer to maintain the compensation beyond a transaction boundary. This level of flexibility lies in between the rigid, transaction-dictated compensation boundaries and the full flexibility allowed in [24] where

compensations can be modified (discarded and replaced) in any way, in any instant. We believe that this approach strikes a good balance between these two extremes since full customisation of compensations leaves all the responsibility on the programmer and requires substantial syntactic overhead in the specification.

4. **Activation of Compensations** Once a compensation is installed, it can be activated and executed to compensate for already completed activities. Since our notation need only be concerned with compensation programming, a number of issues related to compensation activation in other notations are a non-issue for our design. For example in our case it is the role of the system to signal compensation activation and thus the propagation of failure across active transactions (before starting compensating) lies within the system's remit. Similarly, other issues such as how it is ensured that transactions are gracefully terminated are outside of the scope of compensation programming. However, since we have decided that our compensation notation supports several compensation scopes and the special deviation operator, questions arise: Can compensation scopes synchronise? What happens if one of the scopes deviates while the others are still compensating? In this case there are no relevant literature sources from which we can take inspiration. In this scenario, we opt for a flexible solution where the compensation scopes are allowed to synchronise compensation execution. This feature is highly useful when compensation scopes are mostly concurrent but require a few synchronisation points due to dependency issues. For example in the e-procurement scenario, although the transport and payment can be generally compensated concurrently, the refund has to take into consideration the transport cancellation costs if these occur. As regards having a scope deviating while the other scopes are still compensating, we choose not to interrupt compensation execution since this creates other problems such as how to safely

interrupt ongoing execution.

5. **Compensation Execution** While the proposed notation provides a means of specifying compensations, we abstract away from the actual implementation details of compensations. In an implementation, deciding whether the implementation details of a particular compensation lies at the system side or the compensation module side is orthogonal to the design issues being discussed here. The advantage of keeping to this level of abstraction is that the compensation specifications would be technology agnostic, *i.e.,* portable across platform, not brittle to technology improvements, can be maintained by non-technical persons, *etc.*

6. **Post-Compensation Execution** Upon completion of a compensation execution, an important choice is how execution should resume. This obviously has to take into account whether compensation was successful or not. Since the proposed notation should only be concerned with compensations, the choice of how the system should continue after a successful compensation is not within its remit. However, if a compensation fails, it is still the compensation module's responsibility to propose a solution. As mentioned in earlier points, we plan to enable compensations to have compensations which are executed in case a compensation fails.

In the above design discussion of a compensation programming notation, a novel balance emerges between full flexibility — essentially reducing it to stack programming — and the rigidity of strict compensation patterns which are further restricted by the lack of separation of concerns. We believe that this balance supplies the programmer with the necessary operators while still allowing leeway for customisability.

Before going on to present the full account of the notation in terms of formal syntax and semantics, in what follows we distil the above discussion, extracting the salient aspects of the notation and giving an informal introduction to the main constructs.

4.1.4 An Informal Introduction to the Compensation Notation

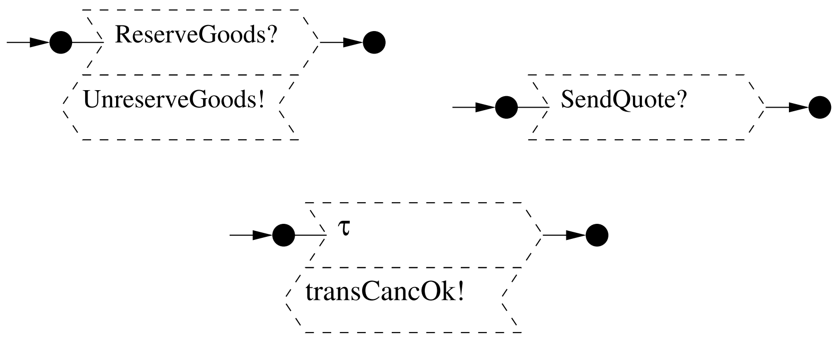
After discussing the design choices of the proposed compensation notation with respect to other similar formalisms, we now move on to give a more tangible feel of the notation. Due to the advantages of automata-based runtime verification (discussed in Section 2.1.2), we choose to concretise the notation in terms of automata. In what follows, we graphically present the main features of compensating automata taking into consideration the design options chosen. These features are further expanded below with examples[5]:

Basic compensation installations The formalism should allow actions to be designated as compensations for system events. Subsequently, upon receipt of an event, the corresponding compensation action is pushed onto a stack. If compensation is activated, the actions in the stack are executed in the reverse order of their installation. In the e-procurement scenario, this corresponds to a number of examples such as releasing previously reserved goods. This is depicted in Figure 4.3(a)[left] where the automaton, with states represented as blobs, transitions upon the *ReserveGoods* event[6] while installing the compensation *UnreserveGoods*. Note that some activities such as sending a quotation might not have a compensation (see Figure 4.3(a)[right]). Similarly, we allow for the automatic installation of compensations with no associated system event. The forward arrow in such cases is annotated with τ — see *e.g.,* Figure 4.3(a)[bottom].

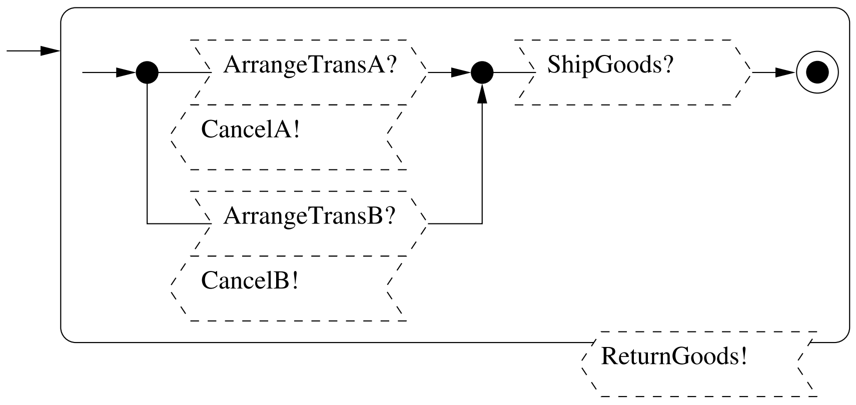
Scoping and replacing compensations Compensations may have to be replaced at some point and therefore it should be possible to delimit compensation patterns whereupon being matched, should be replaced by another

[5]We use the following abbreviations: trans (transport), canc (cancel), addr (address), not (notification), gds (goods), del (delivery), ack (acknowledgement), rec (receive).

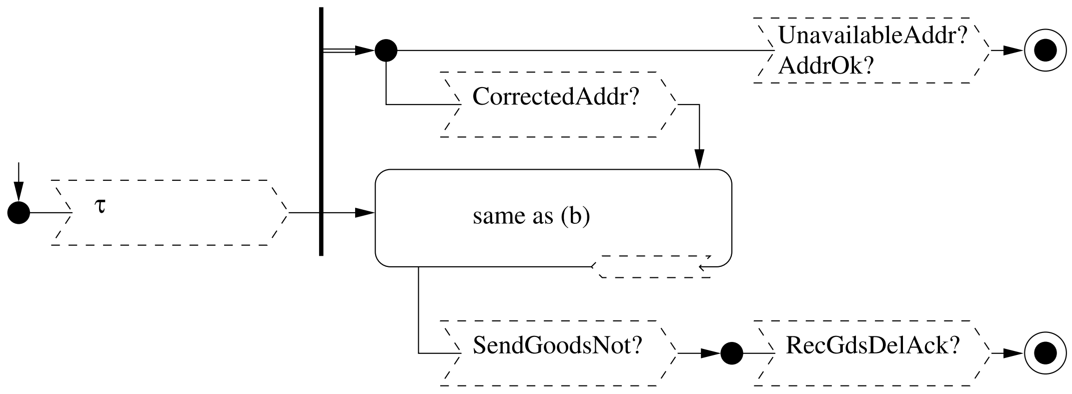
[6]We write labels referring to system events ending in *?* while writing labels referring to compensation actions ending in *!*.



(a) Associating compensations to actions

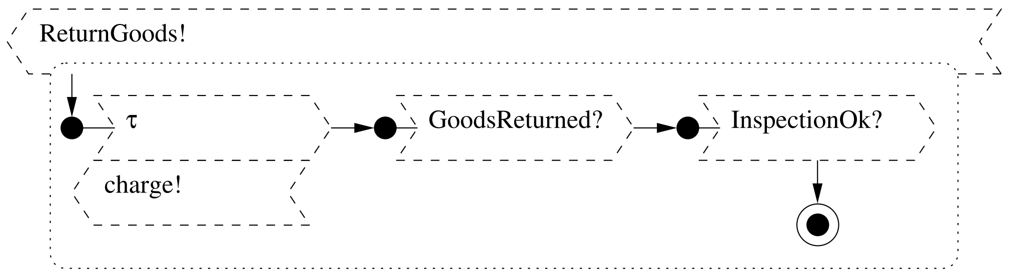


(b) Scoping compensations

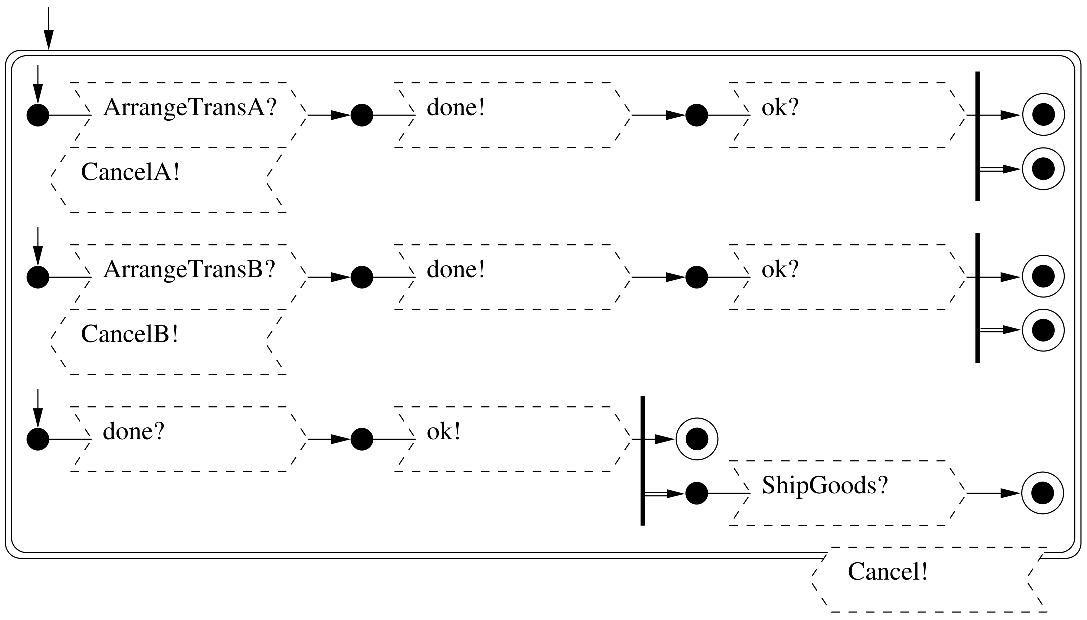


(c) Deviating compensation execution

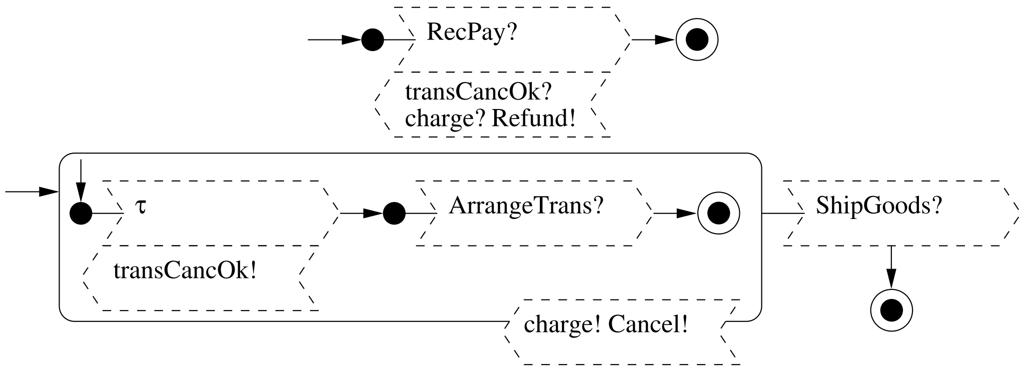
Figure 4.3: Examples of different compensation constructs (cont.)



(d) Defining a compensation's compensation



(e) Speculative choice



(f) Communicating compensating automata

Figure 4.3: Examples of different compensation constructs

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 Figure 4.3(b) where an automaton monitoring transport arrangement is scoped (depicted as a solid-lined box) so that when the goods are shipped (reaching the final state), any accumulated compensations (*CancelA* or *CancelB*) are discarded and replaced by the compensation *ReturnGoods*. Note that if the final state is not reached the compensations will not be discarded.

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), then before cancelling the whole procurement 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. 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 configure compensations from the deviated state onwards. A deviation is shown in Figure 4.3(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 Figure 4.3(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. If inspection fails, then the compensation of the shipment would involve a charge to the customer's credit card. Note that compensations of compensations are enclosed in dotted boxes which partially overlay the compensation action they are meant to compensate for.

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. The alternative of conjuncting multiple automata would result in significantly less understandable compensation specifications.

For example a more efficient way of arranging for transport might be to start the booking process with two shipping companies and then cancel one attempt as soon as the other is confirmed. Better known as a *speculative choice* [27], this can be encoded by communicating automata as shown in Figure 4.3(e)[7]. Most notably, the first arrangement to succeed synchronises on channel *done* and goes past a deviation — ensuring that only one of the bookings can go past the deviation, and the one which does would be safe from the compensation signal to cancel the other. Communication can also be used to synchronise during compensation execution. In our case study, 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. Figure 4.3(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. Although we choose to abstract away from parameters to keep the notation simple, in practice these would be useful to

[7]Note that we use labels starting in lower caps for local communication with labels ending with *?* signifying blocking sinks while those ending in *!* signifying non-blocking sources.

communicate values such as the amount to be refunded.

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

4.2 Compensating Automata

Compensating automata are intended to enable the user to program the compensations so that depending on the sequence of occurring system activities, compensations are configured 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 4.2.1. 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 ($\Sigma_\tau = \Sigma \cup \{\tau\}$) and $\P$ to be the symbol representing failure with $\Gamma = \{\P, \tau\}$ (*cf.* Definition 3.4.1). A compensating automaton event I is thus a non-empty disjunction of system activities, $I \subseteq \Sigma_\tau$, and 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, $O \subseteq \Sigma$, 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 4.2.2. 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 set of compensating automata and $\widehat{\mathcal{A}}$ for the set of vectors of compensating automata. We will use variables A, A' to range over compensating automata and $\widehat{A}, \widehat{A'}$ 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. 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 $Q = B \cup N$. Thus, $q \in N$ is a tuple $\widehat{\mathcal{A}} \times \mathcal{C}$ where $\mathcal{C}$ is the compensation used for replacing the compensations of the vector.

A transition across states within a compensating automaton 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 perform 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 $\mathit{event}(t)$, $\mathit{action}(t)$, $\mathit{src}(t)$, $\mathit{dst}(t)$, and $\mathit{dev}(t)$ to respectively refer to the event, action, source state, destination state, and deviation state appearing on a transition t. □

Example 4.2.1. As an example of a compensating automaton we consider the one in Figure 4.4 where we have intentionally left out compensations for the

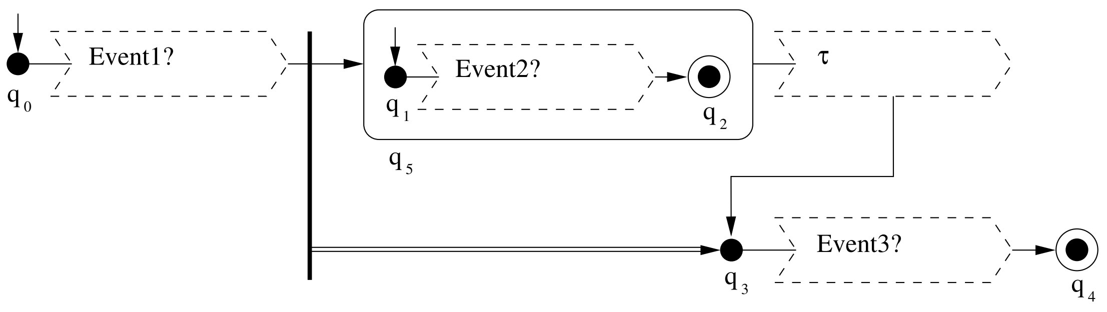


Figure 4.4: An example of a compensating automaton

time being (these will be added later). Instead, we focus on two major features: nesting and deviation. 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 with the nested state and q_3 as the deviating state.

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

Definition 4.2.3. 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 $c \in \mathcal{C}$ is an element of the Cartesian product: $(2^{\Sigma_{L\tau}} \times 2^{\Sigma} \times \widehat{\mathcal{A}})$. □

Example 4.2.2. Adding compensations to the previous example we now consider Figure 4.5 which installs *Comp1* upon receiving *Event1*, *Comp2* upon completion of the nested automaton, and *Comp3* upon receiving *Event3*. Note that had *Comp2* been installed directly upon receiving *Event2*, this would always be discarded at the end of the scope.

Since a compensating automaton may have nested vectors of automata, its configuration should be correspondingly nested by a vector of configurations.

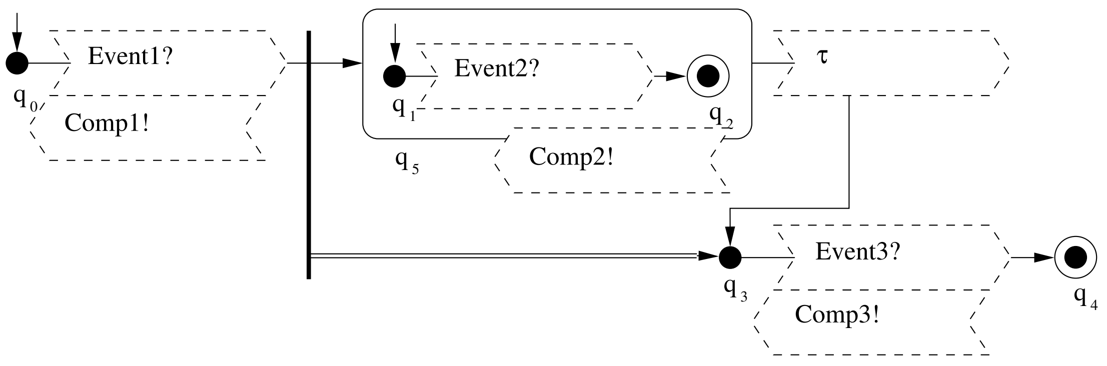


Figure 4.5: An example of a compensating automaton with compensations

We refer to a configuration with no nesting as a basic configuration. Each such basic configuration has to keep track of: (*i*) the state it is in; (*ii*) whether it is currently simply listening for system events (conceptually the system state is progressing *forwards*) or executing compensations (conceptually the system state is progressing *backwards*); and (*iii*) the configured compensation in terms of a stack including the installed compensations and the points at which compensation activation should be deviated.

Definition 4.2.4. A configuration of a compensating automaton is either (*i*) a basic configuration composed of an automaton in set $\mathcal{A}$, a state in Q, a (forward or backward) direction in $\mathcal{R}$ (defined below), and a stack in $\mathcal{S}$ (defined below); (*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:

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

A compensating automaton is either executing forwards, or backwards. Thus, an automaton execution direction, is a member of $\mathcal{R} \stackrel{\text{def}}{=} \{⧁, ⧀\}$ signifying forward and backward execution respectively.

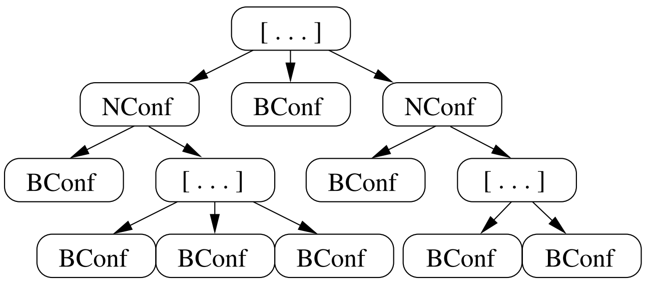


Figure 4.6: An example of a structure or configurations

A stack $S \in \mathcal{S}$ is a sequence of stack elements $\mathcal{S} \overset{\text{def}}{=} Stack^*$ where each element is of type *Stack* and can either be a compensation or a deviation
$Stack ::= \mathcal{C} \mid \mathcal{D}$.

□

An example of a structure of configurations is shown in Figure 4.6 including a vector of three configurations with two of them being nested configurations. The first nested configuration consists of a basic configuration and a vector with three basic configurations, while the second consists of a basic configuration and a nested vector containing two basic configurations:

[(BConf,[BConf,BConf,BConf]), BConf, (BConf,[BConf,BConf])]

As abbreviations for configurations we use $(q,S)^r$ to denote Basic(A,q,r,S) (whenever A is clear from the context) where $r \in \mathcal{R}$ (for instance if $r = \oslash$, $(q,S)^{\oslash}$ signifies a forward executing configuration); and $(q,S)^r_{cf}$ to denote Nest$((q,S)^r,cf)$.

Definition 4.2.5. The push operation, 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 $d \in \mathcal{D}$ (using “:” to represent the *cons* operator) the push operations is defined as follows:

$$c \fatsemi S \overset{\text{def}}{=} c : S$$

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

□

Example 4.2.3. Referring back to Figure 4.5 if execution is in forward mode in state q_0 the configuration would be $(q_0, [])^{⧁}$. However, if execution is in forward mode in state q_1, the configuration would be nested and the compensation *Comp1!* should be installed in the stack together with the deviation to state q_3: $(q_5, [q_3, (\{\tau\}, \{Comp1\}, [])])^{⧁}_{(q_1,[])^{⧁}}$. Because of the stack, it is difficult to give further examples of configurations without first presenting the semantic rules. For this reason more configuration examples will be given in the next section.

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 4.2.6. A basic configuration *cf* is said to be *terminated*, written $\boxdot(cf)$, if and only if it has reached a final state and it is in forward direction:
$\boxdot((q, S)^r) \overset{\text{def}}{=} q \in F \land r = ⧁$.

A basic configuration *cf* is said to be *compensated*, written $\boxtimes(cf)$, if and only if it has an empty stack and it is in backward direction: $\boxtimes((q, S)^r) \overset{\text{def}}{=} S = [] \land r = ⧀$.

Nested configurations are neither terminated nor compensated since execution continues with the parent.

A vector configuration is said to have *terminated* if the vector is empty, or all sub-configurations are either terminated or compensated and at least one has terminated:

$\boxdot([cf_1, cf_2, \ldots, cf_n]) \overset{\text{def}}{=} n = 0 \lor ((\forall j \in 1..n \cdot \boxdot(cf_j) \lor \boxtimes(cf_j)) \land (\exists i \in 1..n \cdot \boxdot(cf_i)))$

On the other hand, a vector configuration is said to have *compensated* if all sub-configurations have compensated:
$\boxtimes([cf_1, cf_2, \ldots, cf_n]) \overset{\text{def}}{=} n > 0 \land \forall i \in 1..n \cdot \boxtimes(cf_i)$. □

We note that a configuration cannot both be terminated and compensated.

Proposition 4.2.1. A terminated configuration implies it is not compensated: $\boxdot(cf) \implies \neg \boxtimes (cf)$. A compensated configuration implies it is not terminated:

$\boxtimes(cf) \implies \neg \boxdot (cf)$. By the law of contrapositive it is enough to show that the former holds.

Proof. The proof follows by considering basic and vector configurations: By Definition 4.2.6 if $\boxdot((q,S)^r)$, then $r = \ogreaterthan$ and thus $r \neq \olessthan$ which implies $\neg \boxtimes (cf)$.

By Definition 4.2.6 if $\boxdot([cf_1, cf_2, \ldots, cf_n])$, then there exists cf_i such that $\boxdot(cf_i)$. Clearly, this violates $\boxtimes([cf_1, cf_2, \ldots, cf_n])$. $\square$

Definition 4.2.7. The initial configuration of a vector of compensating automata is given by the *in* function which starts each automaton from its respective initial state:

$$in(\widehat{A}) \stackrel{\text{def}}{=} [(q_{0_1}, [\,])^{\ogreaterthan}_{in(q_{0_1})}, (q_{0_2}, [\,])^{\ogreaterthan}_{in(q_{0_2})}, \ldots, (q_{0_n}, [\,])^{\ogreaterthan}_{in(q_{0_n})}]$$

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.

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

$\square$

Example 4.2.4. For example applying function *in* on a vector containing the automaton shown in Figure 4.5 one would obtain the configuration $[(q_0, [\,])^{\ogreaterthan}]$ while applying it on nested state q_5 would result in configuration $[(q_1, [\,])^{\ogreaterthan}]$. The latter configuration is useful for obtaining the initial state of a nested automaton during the parent's transition (in this case from q_0 to q_5).

Since compensating automata are intended for monitoring systems, determinism of the specification language is crucial to ensure that monitors behave consistently upon identical input.

Definition 4.2.8. 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) 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) there is at most one outgoing transition from a particular state which is listening for communication on local labels:
$\forall t, t' \in \delta \cdot (t \neq t' \wedge src(t) = src(t')) \implies event(t) \cap \Sigma_L = \emptyset \vee event(t') \cap \Sigma_L = \emptyset$; and

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

Apart from the property of determinism, monitors are typically also expected to terminate, i.e. the monitor eventually stops monitoring and returns control to the system. In more practical terms, this property, sometimes referred to as stability [53], requires us to show that compensating automata cannot enter infinite loops. Thus, compensating automata are not allowed to have τ-loops, i.e. loops which do not involve system events, and, due to the adopted communication model (introduced later), cross-automata communication should not be allowed to result into infinite loops.

Definition 4.2.9. A compensating automaton is said to be well-formed if it contains no τ-loops. Hence, we define a dependency relationship over states which are connected by τ- or local-channel-triggered transitions as follows:

$$\begin{aligned} dep(A) \stackrel{\text{def}}{=} \{(q, q') \mid \exists t \in \delta_A \cdot src(t) = q \wedge dst(t) = q' \\ \wedge\ (\tau \in event(t) \vee event(t) \cap \Sigma_L \neq \emptyset)\} \end{aligned}$$

A compensating automaton A is said to be τ-loop-free if, for any state q, $(q, q) \notin dep(A)^+$.

A vector of compensating automata is said to be well-formed if all its elements are well-formed. □

Note that the above definition constraining local communication is sufficient (see Theorem 4.2.3) to avoid communication loops but not necessary. In general, it is up to the user to ensure that no loops are introduced, but if one sticks to the above restriction, then one is guaranteed to have stability.

Another issue related to looping is that since compensating automata interact with the system by listening for system events and instructing the system to perform compensation actions, a loop can also occur if the system sends repeated events triggered by repeated compensations. Since in our approach we do not model the system behaviour, it is not possible to check for such loops. Furthermore, note that such looping behaviour does not violate our definition of stability since the system would still repeatedly get control back from the monitor.

In what follows when referring to compensating automata we implicitly refer to deterministic, well-formed compensating automata.

4.2.1 Semantics

We give the semantics of compensating automata in SOS style [88] in terms of a labelled transition system where states are configurations. Specifying the semantics of compensating automata poses the issue of whether the semantics should be concurrent or whether the execution of automata in a vector should occur in a predefined order. The latter option is preferred from the point of view of determinism since concurrency introduces various possible behaviour interleavings. On the other hand, opting for a concurrent semantics means that an implementation of the theory will not be bound to be sequential. This is an important advantage in the context of highly concurrent settings such as the service-oriented architecture. Thus, the semantics presented below are concurrent, allowing any order of execution amongst automata running in parallel.

Definition 4.2.10. The semantics of a vector of compensating automata is the least transition relation $\xrightarrow{x}$: $\mathit{Conf} \times \mathit{Trace} \times \mathit{Conf}$ satisfying the rules in Figure 4.7

where x is an element of type *Trace* and is either (i) an automaton transition, *i.e.,* an activity triggering local action, of type[8] $\mathcal{A} \times \Sigma_\tau \times 2^{\Sigma_L}$; (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, ⁋, which signals the automaton to start executing compensations; and (iv) a silent transition, τ, representing the rest of the automata activities. Thus, we define the type *Trace* as

$$\mathit{Trace} ::= \mathrm{Forw}(\mathcal{A} \times \Sigma_\tau \times 2^{\Sigma_L}) \mid \mathrm{Back}(\mathcal{A} \times \Sigma_{L\tau} \times 2^{\Sigma}) \mid ⁋ \mid \tau$$

As abbreviation, we write $\mathrm{Forw}(A,i,O)$ as $(i_O)_A$ and $\mathrm{Back}(A,i,O)$ 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 automata vector, then the corresponding nested initial configuration is given by calling the *in* function on q'. Otherwise, the function call returns *in* returns an empty configuration vector.

When the compensating automaton receives the *compensate* signal, denoted by ⁋, 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. 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 *in* function.

[8]Note that transition trace elements are tagged by the automaton triggering them. This would be useful to distinguish different automata symbols when proving properties about individual automata.

The rules covered above have dealt with the automaton whose configuration is the topmost element of the configuration stack. The following rules deal with what happens when the topmost configuration terminates or compensates. Assuming a forward-executing parent configuration, 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 NestFail by discarding the nested configuration and changing the parent's direction to backwards. If 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 ($\alpha, \beta \in \text{seq } Conf$). □

These rules specify the behaviour of a vector of compensating automata without considering how they interact with a monitored system. In what follows we build further semantics to formally specify how compensating automata behave during monitoring and the subsections which follow prove that these semantics possess the desirable monitoring properties of determinism and stability.

Example 4.2.5. Referring back to configuration examples given earlier with respect to Figure 4.5, it should now be clearer how 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 $(q_5, [q_3, (\{\tau\}, \{Comp1\}, [])])^{\obslash}_{(q_1, [(\{\tau\}, \emptyset, [])])^{\obslash}}$, rule NestSuc triggers and installs compensation *Comp2!* while discarding the nested configuration. This leads to configuration $(q_5, [(\{\tau\}, \{Comp2\}, []), q_3, (\{\tau\}, \{Comp1\}, [])])^{\obslash}$.

An interesting change occurs if the failure event occurs, triggering the Fail rule. In this case the configuration direction turns backwards to $(q_5, [(\{\tau\}, \{\mathit{Comp2}\}, []), q_3, (\{\tau\}, \{\mathit{Comp1}\}, [])])^{\olessthan}$ 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: $(q_3, [(\{\tau\}, \{\mathit{Comp1}\}, [])])^{\ogreaterthan}$.

A different scenario would have arisen if the failure occurred when running the nested automaton. In this case only the nested configuration would have had its direction turned backwards: $(q_5, [q_3, (\{\tau\}, \{\mathit{Comp1}\}, [])])^{\ogreaterthan}_{(q_1, [(\{\tau\}, \emptyset, [])])^{\olessthan}}$. Due to the empty stack in the nested configuration, rule NestFail would trigger, discarding the nested configuration and changing the parent's direction to backwards: $(q_5, [q_3, (\{\tau\}, \{\mathit{Comp1}\}, [])])^{\olessthan}$. At this point, execution continues with rule Dev as before.

A significant issue in the design of compensating automata relates to the way automata in a vector interact with each other. While rule Vect above enables each automaton to progress, it does not specify how synchronisation can take place. This is discussed and defined in the following.

4.2.2 Communication Amongst Compensating Automata

Motivated by instances from the e-procurement case study, we enable compensating automata to communicate. Yet, there are various communication patterns which could be adopted. The first choice is whether to go for a point-to-point or a broadcast system. We opt for a broadcast system to mimic the way system events reach monitors, i.e. system events are broadcast to all monitors, providing a consistent event mechanism. Another design issue is whether or not events are consumed by the receiver, and whether the receiver is blocking. Again, for consistency and to avoid race conditions, events are not consumed and receiving is blocking. Moreover, just as the system does not block if no monitor listens for a particular event, local communication senders do not block upon sending.

Basic Configurations

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

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

Nested Configurations

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

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

Vector Configurations

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

Figure 4.7: Basic semantic rules

This communication model also conforms to other similar models of monitoring under concurrency such as [55]. In the rest of the section, we formally specify this communication mechanism and explain how the system and the compensation monitors interact.

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 signal are collectively called a *compound step*. To give the semantics of communication within compensating automata, the compound step is split into basic steps which allow one transition to occur while taking into account local communication.

Definition 4.2.11. Basic steps are specified on basic-step configurations *bsConf* which are each composed of a vector configuration plus a set of local symbols: $bsConf \in (Conf \times 2^{\Sigma_L})$. The set of local symbols is used to keep track of the local communication, *i.e.,* the symbols which have been triggered. There are five kinds of basic steps (each described by a rule in Figure 4.9): two which are directly triggered as a result of a 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) compensation-triggered actions (rules Loc and Cmp respectively). Note that local symbols are added to the configuration whenever local communication triggers (rules Sys, Cmp, and Loc). Furthermore, local communication (rule Loc and Cmp) can only trigger if a corresponding symbol is present in the configuration set of symbols. □

Example 4.2.6. To help illustrate the communication mechanism formalised in basic steps, we refer to Figure 4.8 where three automata communicate through local channels. Assuming all automata are initially active at their first state with no previous local communication, automata A_2 and A_3 cannot transition since the side condition of basic step Loc is not satisfied ($L = \emptyset$). On the other hand,

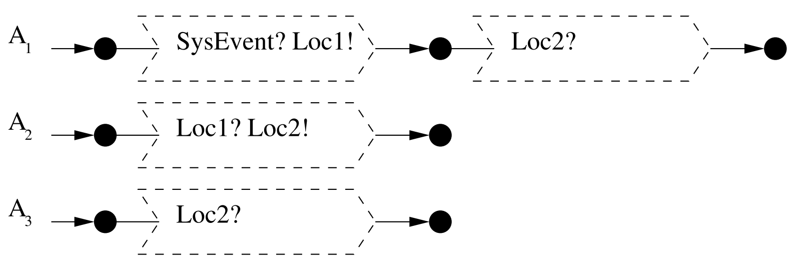


Figure 4.8: Basic steps example

Basic Steps

$$\textsc{Sys}\ \frac{cf \xrightarrow{(i_O)_A} cf'}{cf_{[L]} \overset{i_A}{\hookrightarrow} cf'_{[L\cup O]}}\ i\in\Sigma_S \qquad \textsc{Fal}\ \frac{cf \overset{\Lsh}{\rightarrow} cf'}{cf_{[L]} \overset{\Lsh}{\hookrightarrow} cf'_{[L]}} \qquad \textsc{Sil}\ \frac{cf \overset{\tau}{\rightarrow} cf'}{cf_{[L]} \overset{\tau}{\hookrightarrow} cf'_{[L]}}$$

$$\textsc{Loc}\ \frac{cf \xrightarrow{(\tau_O)_A} cf' \text{ or } cf \xrightarrow{(l_O)_A} cf'}{cf_{[L]} \overset{\tau}{\hookrightarrow} cf'_{[L\cup O]}}\ l\in L \qquad \textsc{Cmp}\ \frac{cf \xrightarrow{({}_\tau O)_A} cf' \text{ or } cf \xrightarrow{({}_l O)_A} cf'}{cf_{[L]} \overset{\tau}{\hookrightarrow} cf'_{[L\cup(O\cap\Sigma_L)]}}\ l\in L$$

Figure 4.9: Basic steps

the first transition of automaton A_1 triggers immediately over basic step Sys (note the only side condition is that the event is an element of Σ_S), contributing *Loc1* to the set of enabled local channels ($L = \{Loc1\}$). Subsequently, A_2 may transition over *Loc1* due to the satisfaction of the side condition, contributing *Loc2* to the set of enabled local channels ($L = \{Loc1, Loc2\}$). Finally, both A_1 and A_3 can transition since $Loc2 \in L$.

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

Definition 4.2.12. Given a sequence $e\!:\!es$ of system activities and failure signals, $e \in \Sigma \cup \{\Lsh\}$, the behaviour of a vector of automata $\widehat{A}$, is characterised by two phases.

1. Starting from the initial configuration $in(\widehat{A})$, the compensation manager

performs all possible non-system-triggered basic steps. More formally, the first phase is characterised by a τ-compound-step (denoted by $\overset{\tau}{\rightsquigarrow}$) and is composed of sequences of non-system triggered basic steps ($\overset{\tau}{\hookrightarrow}$). Note that L' is discarded once all the internal steps have been carried out.

$$cf_{[L]} \overset{\tau}{\rightsquigarrow} cf'_{[\emptyset]} \quad \overset{\text{def}}{=} \quad cf_{[L]} \xhookrightarrow{\tau\tau\ldots\tau}\!\bullet cf'_{[L']}$$

2. Execution continues by repeatedly consuming elements from $\mathtt{e:es}$ 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-compound-step* involves triggering all possible transitions waiting on e and collecting all the triggered local actions in the configuration set of symbols — $\xhookrightarrow{e_{A_1}e_{A_2}\ldots e_{A_n}}\!\bullet$ or $\xhookrightarrow{⁋⁋\ldots⁋}\!\bullet$ depending on whether e is a normal event or a compensate signal. Note that as per rule SYS the former labels are subscripted with the automaton triggering the event. This enables us to limit each automaton to take only one step over a particular event by ensuring that each automaton $A_1 \ldots A_n$ is unique.

$$cf_{[L]} \overset{e}{\rightsquigarrow} cf'_{[L']} \quad \overset{\text{def}}{=} \quad \begin{cases} cf_{[L]} \xhookrightarrow{⁋⁋\ldots⁋}\!\bullet cf'_{[L]} & \text{if } e = ⁋ \\ cf_{[L]} \xhookrightarrow{e_{A_1}e_{A_2}\ldots e_{A_n}}\!\bullet cf'_{[L']} & \text{otherwise} \end{cases}$$
$$\text{where } \forall i,j \in 1..n \cdot i \neq j \implies A_i \neq A_j$$

Putting the two phases together, the overall behaviour can be summarised as $\overset{\tau}{\rightsquigarrow}(\overset{e}{\rightsquigarrow}\overset{\tau}{\rightsquigarrow})^*$. Note that each system-triggered step follows and is followed by a $\overset{\tau}{\rightsquigarrow}$ so that the automata vector exhausts all the internal basic steps before considering other signals from the system. For abbreviation we use
$\overset{es}{\rightsquigarrow}\!{*} \overset{\text{def}}{=} \overset{\tau}{\rightsquigarrow}\overset{e_1}{\rightsquigarrow}\overset{\tau}{\rightsquigarrow}\overset{e_2}{\rightsquigarrow}\overset{\tau}{\rightsquigarrow}\ldots\overset{e_n}{\rightsquigarrow}\overset{\tau}{\rightsquigarrow}$ where $es = e_1e_2\ldots e_n$.

□

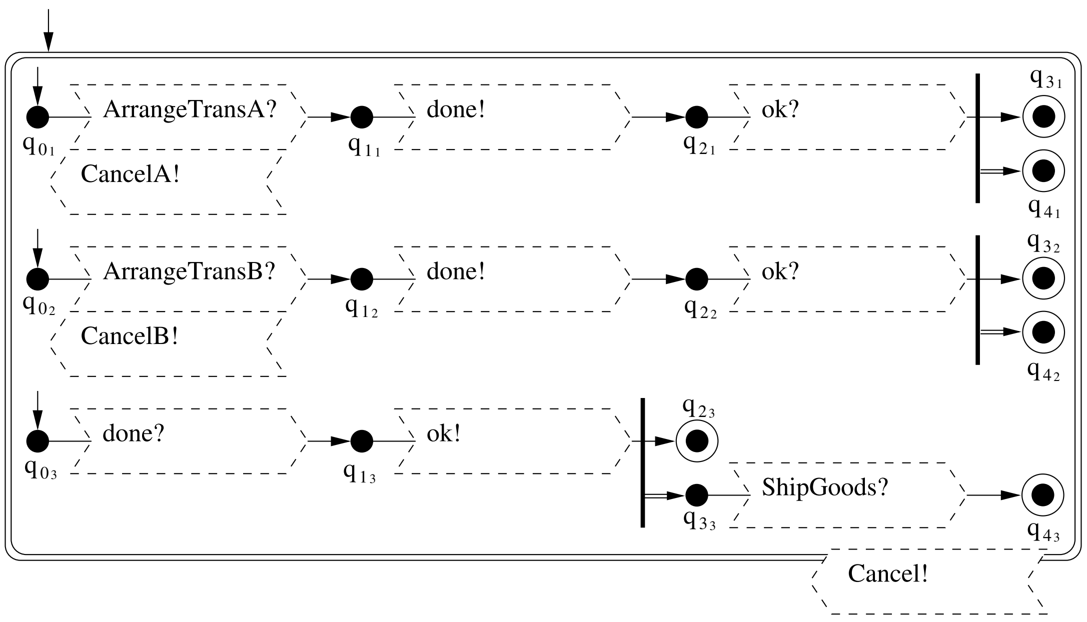


Figure 4.10: Speculative choice

To better illustrate the semantics of compensating automata, we give two examples: one having synchronisation during forward execution and another with synchronising compensations.

Example 4.2.7. Consider the speculative choice example given in Figure 4.3(e), reproduced once more in Figure 4.10, and let $[A_1, A_2, A_3]$ represent the vector composed of the three automata with $A_i = (\Sigma_i, Q_i, q_{0_i}, F_i)$ and basic states being numbered from left to right, top to bottom (as they appear in the diagram), starting from zero[9]. The initial configuration, given by $in([A_1, A_2, A_3])$ is $[(q_{0_1}, [\,])^{\circledcirc}, (q_{0_2}, [\,])^{\circledcirc}, (q_{0_3}, [\,])^{\circledcirc}]$. Performing the initial phase ($\overset{\tau}{\rightsquigarrow}$) would not contribute any change since no local communication or τ events can trigger.

Following the initial stage, we assume that the system emits the following events: (*ArrangeTransB*, *ArrangeTransA*, ↰, *ShipGoods*). Intuitively, this stream of events means that initially the booking of transport B was successful but before the booking with transport A got cancelled, it got confirmed as well. In such a scenario, transport A should be cancelled and subsequently the shipping of the

[9]For simplicity we are ignoring the parent automaton in whose initial state the three automata reside.

goods should be carried out by transport B. In what follows we explain step by step how the compensating automaton processes each of the four system events:

1. Processing the first event changes the configuration into

$$\begin{aligned}&[(q_{0_1},[])^{\ogreaterthan},\\&(q_{1_2},[(\{\tau\},\{CancelB\},[])])^{\ogreaterthan},\\&(q_{0_3},[])^{\ogreaterthan}]\end{aligned}$$

   through rules Suc, Vect, and Sys. Following this, the τ-compound-step constitutes two basic steps (triggering rules Suc, Vect, and Loc) on local channels *done* and *ok* yielding the following configuration:

$$\begin{aligned}&[(q_{0_1},[])^{\ogreaterthan},\\&(q_{3_2},[q_{4_2},(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[]),(\{\tau\},\{CancelB\},[])])^{\ogreaterthan},\\&(q_{2_3},[q_{3_3},(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[])])^{\ogreaterthan}]\end{aligned}$$

   In particular, note that the topmost element in the stacks of the second and third configurations is a deviation. This means that these configurations will simply deviate to another state of the automaton rather than execute compensations from the stack. In this example, this is crucial to protect the first booking from being cancelled when cancelling the second.

2. The second event, *ArrangeTransA*, is similar to the first but fails to progress to the final state since no synchronisation occurs on *ok*. The resulting configuration would be as follows:

$$\begin{aligned}&[(q_{2_1},[(\{\tau\},\emptyset,[]),(\{\tau\},\{CancelA\},[])])^{\ogreaterthan},\\&(q_{3_2},[q_{4_2},(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[]),(\{\tau\},\{CancelB\},[])])^{\ogreaterthan},\\&(q_{2_3},[q_{3_3},(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[])])^{\ogreaterthan}]\end{aligned}$$

3. At this point the system detects that at least a booking has succeeded and thus signals the need for compensation through event ↰. This changes the configuration direction of all configurations to backwards, triggering rules Fail, Vect, and Fal. However, as soon as the second and third configurations start compensating during the τ-compound-step, the deviation (through rules Dev, Vect, and Sil) changes the direction back to forwards:

$$\begin{aligned}&[(q_{2_1},[(\{\tau\},\emptyset,[]),(\{\tau\},\{\mathit{CancelA}\},[])])^{\olessthan},\\&\ (q_{4_2},[(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[]),(\{\tau\},\{\mathit{CancelB}\},[])])^{\ogreaterthan},\\&\ (q_{3_3},[(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[])])^{\ogreaterthan}]\end{aligned}$$

Next, through rules Comp, Vect, and Cmp, the compensation stack of the first configuration is activated and the system compensates the booking of transport A leading to configuration:

$$\begin{aligned}&[(q_{2_1},[])^{\olessthan},\\&\ (q_{4_2},[(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[]),(\{\tau\},\{\mathit{CancelB}\},[])])^{\ogreaterthan},\\&\ (q_{3_3},[(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[])])^{\ogreaterthan}]\end{aligned}$$

4. Finally, on event *ShipGoods*, the third automaton reaches the final state:

$$\begin{aligned}&[(q_{2_1},[])^{\olessthan},\\&\ (q_{4_2},[(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[]),(\{\tau\},\{\mathit{CancelB}\},[])])^{\ogreaterthan},\\&\ (q_{4_3},[(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[]),(\{\tau\},\emptyset,[])])^{\ogreaterthan}]\end{aligned}$$

Example 4.2.8. Consider the communication example given in Figure 4.3(f), reproduced once more in Figure 4.11, and let $[A_1, A_2]$ represent the vector composed of the two automata (with $A_1 = (\Sigma_1, Q_1, q_{0_1}, F_1)$ and $A_2 = (\Sigma_2, Q_2, q_{0_2}, F_2)$ representing the top and bottom automaton respectively). The initial configuration, given by $in([A_1, A_2])$ is $[(q_{0_1},[])^{\ogreaterthan},(q_{0_2},[])^{\ogreaterthan}]$. Performing the initial phase

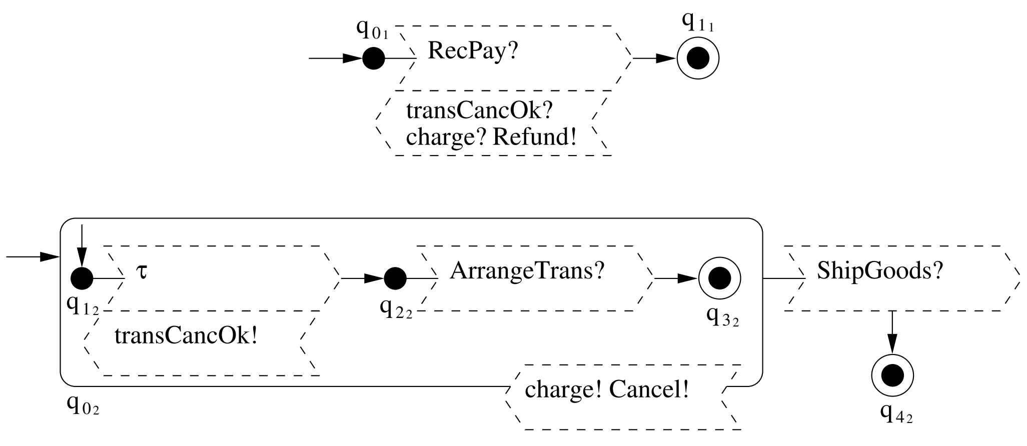


Figure 4.11: Communicating compensating automata

$(\overset{\tau}{\rightsquigarrow})$ would result in the installation of compensation *transCancOk*, $[(q_{0_1},[])^{\smiley},(q_{1_2},[(\{\tau\},\{transCancOk\},[])])^{\smiley}]$, through rules Suc, Vect, and Loc.

Next we consider two scenarios for the second phase:

1. In the first scenario we assume the payment succeeds followed immediately by failure $(RecPay, \Lsh)$. In this case, upon *RecPay*, the configuration proceeds to

$$[(q_{1_1},[(\{transCancOk,charge\},\{Refund\},[])])^{\smiley},$$
$$(q_{1_2},[(\{\tau\},\{transCancOk\},[])])^{\smiley}]$$

   through rules Suc, Nest, Vect, and Sys. Upon failure, execution direction turns backwards for both automata and local communication takes place, $[(q_{1_1},[])^{\frownie},(q_{1_2},[])^{\frownie}]$, first through rules Comp, Nest, Vect, and Cmp (storing *transCancOk* in the basic-step configuration's set of local channels), and again through the same rules triggering the *Refund* compensation.

2. As a second scenario we assume that all actions succeed except for *ShipGoods*, *i.e.*, $(RecPay, ArrangeTrans, \Lsh)$. In this case, upon *ArrangeTrans*, first through rules Suc, Nest, Vect, and Sys, and then by rules NestSuc, Vect, and Sil the configuration evolves to

$$[(q_{11}, [(\{transCancOk, charge\}, \{Refund\}, [])])^{\circleddash},$$
$$(q_{22}, [(\{\tau\}, \{charge, Cancel\}, [])])^{\circleddash}]$$

Upon failure, execution direction turns backwards for both automata and local communication takes place, first through rules Comp, Vect, and Cmp (storing *charge* in the basic-step configuration's set of local channels), and then again through the same rules, Comp, triggering the *Refund* compensation.

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.

4.2.3 Self-Cancellation in Compensating Automata

In order for compensating automata to be perfectly self-cancelling, it must be ensured that installed compensations perfectly 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 and also the replaced compensations.[10]

Definition 4.2.13. A perfectly-compensating automaton is an automaton whose transitions $(q, I, O, (I', O', \widehat{A}), d, q') \in \delta$ are such that their events and compensations correspond perfectly, *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
$(\widehat{A}, (I, O, \widehat{A})) \in N$ should have an empty compensation, *i.e.,* $O = \emptyset$.

[10]Note that in what follows we will be freely quoting definitions and proofs from Chapter 3.

In the rest of this section we use compensating automata to refer to perfectly-compensating automata. □

To analyse the externally visible behaviour of a particular compensating automaton we ignore the strictly local components in the trace, and we leave out any activities related to other automata.

Definition 4.2.14. 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 constitutes local events). Furthermore, for an automaton A, all elements tagged with A' ($A' \neq A$) are dropped, ignoring any activities originating from nested automata or compensations of compensations. Thus, given a trace $w : \mathit{Trace}^*$ and automaton $A \in \mathcal{A}$, we define $\cdot_A : \mathit{Trace}^* \to \mathit{Trace}_x{}^*$ where

$$\begin{aligned}
\mathit{Trace}_x &::= \Sigma_\tau \mid 2^{\Sigma_\tau} \\
\varepsilon_A &\stackrel{\text{def}}{=} \varepsilon \\
(a\,w)_A &\stackrel{\text{def}}{=} \begin{cases} i\,w_A & \text{if } a = (i_O)_A \\ O\,w_A & \text{if } a = ({}_iO)_A \\ w_A & \text{otherwise} \end{cases}
\end{aligned}$$

□

Using the cancelling function definition (Definition 3.4.4) we now go on to define what it means for a compensating automaton to be *self-cancelling*.

Definition 4.2.15. 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:

$$\forall w : \mathit{Trace}^* \cdot in(q_0) \overset{w}{\Longrightarrow} cf \wedge \boxtimes(cf) \implies w_A =_c \varepsilon$$

A vector of compensating automata $\widehat{A}$ 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:

$$\forall w : Trace^* \cdot in(\widehat{A}) \overset{w}{\Longrightarrow} cf \wedge \boxtimes(cf) \implies \forall A \in \widehat{A} \cdot w_A =_c \varepsilon$$

□

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 4.2.16. Given a stack S, the function *behaviour*, denoted $\cdot_\#$, 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.

$$\begin{aligned} ((I, O, \widehat{A}) \fatsemi S)_\# &\overset{\text{def}}{=} \begin{cases} S_\# & \text{if } O = \emptyset \\ O\, S_\# & \text{otherwise} \end{cases} \\ (d \fatsemi S)_\# &\overset{\text{def}}{=} S_\# \\ []_\# &\overset{\text{def}}{=} \varepsilon \end{aligned}$$

□

The following propositions will be used later to prove the main result that compensating automata are self-cancelling.

Proposition 4.2.2. Given an activity $i \in \Sigma$ and a string $w \in Trace_x^*$, then appending i to w cannot contribute to cancellations which are not present in w. More formally, $cancel(w)\, i = cancel(w\, i)$

Proof. By Definition 3.4.4, i can only cause a reduction if $i\, \overline{i}$ is a sequence of the string. Thus, $cancel(w\, i)$ would return the same cancellation result as $cancel(w)\, i$ since nothing follows i. □

Proposition 4.2.3. Given a compensation $\overline{i}$ 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, $cancel(s)\, \overline{i} = cancel(s\, \overline{i})$

Proof. By Definition 3.4.4, $\overline{i}$ can only cause a reduction if $i\,\overline{i}$ is a sequence of the string. By Proposition 4.2.2, $cancel(s)\,\overline{i} = cancel(w)\,j\,\overline{i}$, and since $j\,\overline{i}$ by definition of *cancel* and injectivity of the compensation function do not cancel out, then $cancel(w)\,j\,\overline{i} = cancel(s\,\overline{i})$. □

Proposition 4.2.4. Function *cancel* is idempotent: $cancel(cancel(w)) = cancel(w)$

Proof. The proof follows by string induction on w.

The base case for $w = \varepsilon$ follows from Definition 3.4.4.

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

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

$$\begin{aligned}
& cancel(k\,x) \\
& \quad \{ \text{ By Proposition 4.2.2 } \} \\
= \; & cancel(k)\,x \\
& \quad \{ \text{ By the inductive hypothesis } \} \\
= \; & cancel(cancel(k))\,x \\
& \quad \{ \text{ By Proposition 4.2.2 twice } \} \\
= \; & cancel(cancel(k\,x))
\end{aligned}$$

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

$$\begin{aligned}
& cancel(k\,\overline{i}) \\
& \quad \{ \text{ By Proposition 4.2.3 } \} \\
= \; & cancel(k)\,\overline{i} \\
& \quad \{ \text{ By the inductive hypothesis } \} \\
= \; & cancel(cancel(k))\,\overline{i} \\
& \quad \{ \text{ By Proposition 4.2.3 twice } \} \\
= \; & cancel(cancel(k\,\overline{i}))
\end{aligned}$$

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

$$\begin{aligned}
& \{\text{ By the inductive hypothesis }\} \\
& \mathit{cancel}(k'\,i) = \mathit{cancel}(\mathit{cancel}(k'\,i)) \\
& \{\text{ By Proposition 4.2.2 thrice }\} \\
\Longrightarrow\; & \mathit{cancel}(k')\,i = \mathit{cancel}(\mathit{cancel}(k'))\,i \\
& \{\text{ By removing } i \text{ both sides }\} \\
\Longrightarrow\; & \mathit{cancel}(k') = \mathit{cancel}(\mathit{cancel}(k')) \\
& \{\text{ By Definition 3.4.4 twice }\} \\
\Longrightarrow\; & \mathit{cancel}(k'\,i\,\overline{i}) = \mathit{cancel}(\mathit{cancel}(k'\,i\,\overline{i}))
\end{aligned}$$

□

Proposition 4.2.5. Applying *cancel* on a string $w\,\overline{i}$ is the same as cancelling out w and then cancelling the result appended with $\overline{i}$: for any $w, w' : \mathit{Trace}_A{}^*$, $\mathit{cancel}(w\,\overline{i}) = \mathit{cancel}(\mathit{cancel}(w)\,\overline{i})$.

Proof. The proof follows by string induction on w.

The base case for $w = \varepsilon$ follows by Definition 3.4.4.

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

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

$$\begin{aligned}
& \mathit{cancel}(k\,x\,\overline{i}) \\
& \{\text{ By Proposition 4.2.4 }\} \\
=\; & \mathit{cancel}(\mathit{cancel}(k\,j\,\overline{i})) \\
& \{\text{ By Proposition 4.2.3 twice }\} \\
=\; & \mathit{cancel}(\mathit{cancel}(k\,j)\,\overline{i})
\end{aligned}$$

Case 2: $x = i$

$$\begin{aligned}
& cancel(k\, i\, \overline{i}) \\
& \quad \{ \text{ By Definition 3.4.4 } \} \\
= \;& cancel(k) \\
& \quad \{ \text{ By Proposition 4.2.4 } \} \\
= \;& cancel(cancel(k)) \\
& \quad \{ \text{ By Definition 3.4.4 } \} \\
= \;& cancel(cancel(k)\, i\, \overline{i}) \\
& \quad \{ \text{ By Proposition 4.2.2 } \} \\
= \;& cancel(cancel(k\, i)\, \overline{i})
\end{aligned}$$

□

Proposition 4.2.6. If $cancel(w) = w'\, i$ then $cancel(w\, \overline{i}) = cancel(w')$.

Proof. The result can be proved as follows:

$$\begin{aligned}
& \quad \{ \text{ By Proposition 4.2.5 } \} \\
\Longrightarrow \;& cancel(w\, \overline{i}) = cancel(cancel(w)\, \overline{i}) \\
& \quad \{ \text{ By substitution } \} \\
\Longrightarrow \;& cancel(w\, \overline{i}) = cancel(w'\, i\, \overline{i}) \\
& \quad \{ \text{ By Definition 3.4.4 } \} \\
\Longrightarrow \;& cancel(w\, \overline{i}) = cancel(w')
\end{aligned}$$

□

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 a stack S containing three compensations, $\overline{c}$, $\overline{b}$, and $\overline{a}$: $S = \overline{c} \fatsemi \overline{b} \fatsemi \overline{a} \fatsemi []$ corresponding to 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 $S' = \overline{d} \fatsemi S$. 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 $cancel(\overline{S'_\#}\,\overline{d})$ since nothing would cancel out in a stack consisting solely of compensations. This is expressed in the following lemma.

Lemma 4.2.1. Generalising these observations, given a configuration $(q,S)^r$ which reaches $(q',S')^r$ on string w, the resulting stack S' can be expressed as a function of the original stack and the generated string such that if $(q,S)^r \overset{w}{\Longrightarrow} (q',S')^r$, then $S'_\# = \overline{cancel(\overline{S_\#} w^x_A)}$[11]. 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 $S = S'$.

$$\begin{aligned}
& \{ \text{ By application of Definition 4.2.16 } \} \\
\Longrightarrow \quad & \overline{S_\#} = \overline{S'_\#} \\
& \{ \text{ By Definition 4.2.16 and by Proposition 3.4.1 } \} \\
\Longrightarrow \quad & \overline{S_\#} = cancel(\overline{S'_\#}) \\
& \{ \text{ By Definition 3.4.2 and } w^x_A = \varepsilon \ \} \\
\Longrightarrow \quad & S_\# = \overline{cancel(\overline{S'_\#} w^x_A)}
\end{aligned}$$

Inductive case: $w = k\,x$ — The proof proceeds by a rule-by-rule analysis starting by the rules which trigger on forward configurations, $(q,S)^{\otimes}$ or $(q,S)^{\otimes}_{cf}$.

[11]Recall Definition 3.4.1 where w^x returns w without the non-externally visible activities

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

{ By application of Suc }

$\Longrightarrow \quad S'' = ((i', \bar{i}), \widehat{A}) \fatsemi S'$

{ By Definition 4.2.16 }

$\Longrightarrow \quad S''_\# = \bar{i}\,S'_\#$

{ By inductive hypothesis }

$\Longrightarrow \quad S''_\# = \bar{i}\,\overline{cancel(\overline{S_\#}\,k^x_A)}$

{ By Definition 3.4.2 }

$\Longrightarrow \quad S''_\# = \overline{cancel(\overline{S_\#}\,k^x_A)\,i}$

{ By Proposition 4.2.2 and Definition 4.2.14 }

$\Longrightarrow \quad S''_\# = \overline{cancel(\overline{S_\#}\,(k\,i)^x_A)}$

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

{ By application of Fail and Definition 4.2.16 }

$\Longrightarrow \quad S''_\# = S'_\#$

{ By inductive hypothesis }

$\Longrightarrow \quad S''_\# = \overline{cancel(\overline{S_\#}\,k^x_A)}$

{ By Definition 4.2.14 }

$\Longrightarrow \quad S''_\# = \overline{cancel(\overline{S_\#}\,(k\,↰)^x_A)}$

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

{ By application of NestSuc and Definition 4.2.16 }

$\Longrightarrow \quad S''_\# = S'_\#$

{ By inductive hypothesis }

$\Longrightarrow \quad S''_\# = \overline{cancel(\overline{S_\#}\,k^x_A)}$

{ By Definition 4.2.14 }

$\Longrightarrow \quad S''_\# = \overline{cancel(\overline{S_\#}\,(k\,\tau)^x_A)}$

Case 4: Rule NestFail, $w = k \text{↰}$

{ By application of NestFail and Definition 4.2.16 }

$\implies S''_{\#} = S'_{\#}$

{ By inductive hypothesis }

$\implies S''_{\#} = \overline{cancel(\overline{S_{\#}}\, k^{x}_{A})}$

{ By Definition 4.2.14 }

$\implies S''_{\#} = \overline{cancel(\overline{S_{\#}}\,(k \text{↰})^{x}_{A})}$

Case 5: Rule Nest, $w = k\,x$

where $x \in \{(i_O)_{A'}, ({}_iO)_{A'} \mid i \in \Sigma \wedge O \in 2^{\Sigma} \wedge A' \neq A\} \cup \{\tau, \text{↰}\}$

(due to rules Suc, Fail, Comp, and Dev which can cause rule Nest to trigger)

{ By application of Nest and Definition 4.2.16 }

$\implies S''_{\#} = S'_{\#}$

{ By inductive hypothesis }

$\implies S''_{\#} = \overline{cancel(\overline{S_{\#}}\, k^{x}_{A})}$

{ By Definition 4.2.14 }

$\implies S''_{\#} = \overline{cancel(\overline{S_{\#}}\,(k\,x)^{x}_{A})}$

Next, the rules which trigger on backward configurations, $(q, S)^{\circledcirc}$ or $(q, S)^{\circledcirc}_{cf}$ are considered.

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

{ By application of Comp }

$\Longrightarrow \quad S' = ((i',\bar{i}),\widehat{A}) \fatsemi S''$

{ By Definition 4.2.16 }

$\Longrightarrow \quad S'_\# = \bar{i}\, S''_\#$

{ By inductive hypothesis }

$\Longrightarrow \quad \overline{cancel(\overline{S_\#}\, k^x_A)} = \bar{i}\, S''_\#$

{ By Definition 3.4.2 twice }

$\Longrightarrow \quad cancel(\overline{S_\#}\, k^x_A) = \overline{S''_\#}\, i$

{ By Proposition 4.2.6 and Definition 4.2.14 }

$\Longrightarrow \quad cancel(\overline{S_\#}\,(k\,\bar{i})^x_A) = cancel(\overline{S''_\#})$

{ By Proposition 3.4.1 }

$\Longrightarrow \quad cancel(\overline{S_\#}\,(k\,\bar{i})^x_A) = \overline{S''_\#}$

{ By Definition 3.4.2 }

$\Longrightarrow \quad \overline{cancel(\overline{S_\#}\,(k\,\bar{i})^x_A)} = S''_\#$

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

{ By application of Dev }

$\Longrightarrow \quad S' = d \fatsemi S''$

{ By Definition 4.2.16 }

$\Longrightarrow \quad S'_\# = S''_\#$

{ By inductive hypothesis }

$\Longrightarrow \quad \overline{cancel(\overline{S_\#}\, k^x_A)} = S''_\#$

{ By Definition 4.2.14 }

$\Longrightarrow \quad \overline{cancel(\overline{S_\#}\,(k\,\tau)^x_A)} = S''_\#$

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

$$\begin{aligned}
& \{\text{ By application of NestComp and Definition 4.2.16 }\}\\
\Longrightarrow\quad & S'_\# = S''_\#\\
& \{\text{ By inductive hypothesis }\}\\
\Longrightarrow\quad & \overline{cancel(\overline{S_\#}\, k^x_A)} = S''_\#\\
& \{\text{ By Definition 4.2.14 }\}\\
\Longrightarrow\quad & \overline{cancel(\overline{S_\#}\, (k\,\tau)^x_A)} = S''_\#
\end{aligned}$$

Case 4: Rule Nest, $w = k\,x$

$$(x \in \{(i_O)_{A'}, ({}_iO)_{A'} \mid i \in \Sigma \wedge O \in 2^\Sigma \wedge A' \neq A\} \cup \{\tau, \P\})$$

$$\begin{aligned}
& \{\text{ By application of Nest and Definition 4.2.16 }\}\\
\Longrightarrow\quad & S''_\# = S'_\#\\
& \{\text{ By inductive hypothesis }\}\\
\Longrightarrow\quad & S''_\# = \overline{cancel(\overline{S_\#}\, k^x_A)}\\
& \{\text{ By Definition 4.2.14 }\}\\
\Longrightarrow\quad & S''_\# = \overline{cancel(\overline{S_\#}\, (k\,x)^x_A)}
\end{aligned}$$

□

Theorem 4.2.1. A compensating automaton is self-cancelling, i.e.

$$\forall w : Trace^* \cdot in(q_0) \overset{w}{\Longrightarrow} cf \wedge \boxtimes(cf) \implies w =_c \varepsilon$$

Proof. By applying Definition 4.2.7 and Lemma 4.2.1 on the premise of the implication, then it follows that $[]_\# = \overline{cancel(\overline{[]_\#}\, w^x_A)}$. Since $[]_\# = \varepsilon$, this gets simplified to $\varepsilon = \overline{cancel(w^x_A)}$. Finally, by applying the compensation function on both sides and by Definition 3.4.2, we get $\varepsilon =_c w$ as required.

□

Note that the theorem above proves self-cancellation of a compensating automaton A based on the definition of *cancel* (Definition 3.4.4) which ignores symbols from nested automata. In the next corollary we use the same approach to ignore events from sibling automata as well as from nested automata to extend the result to vectors of compensating automata.

Corollary 4.2.1. A vector of compensating automata is self-cancelling, *i.e.,*

$$\forall w : Trace^{*} \cdot in(\widehat{A}) \overset{w}{\Longrightarrow} cf \wedge \boxtimes(cf) \implies \forall A \in \widehat{A} \cdot w =_c \varepsilon$$

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 4.2.14 — which drops all activities relating to other automata — and Theorem 4.2.1 which states that each trace of a compensating automaton is self-cancelling. □

Corollary 4.2.2. Basic step and compound step behaviour is self-cancelling.

Proof. Since basic steps and compound steps simply constraint the ordering of rule triggering, the resulting behaviour is a special case of the compensating automata semantic rules which have been proven to give self-cancelling behaviour in Corollary 4.2.1. □

4.2.4 Determinism of Compensating Automata

In the context of runtime verification, determinism of the specification language is crucial as it ensures that given the same input, the monitor always returns the same verdict. In this section we show that compensating automata behave deterministically.

Definition 4.2.17. The semantics of a compensating automaton is said to be confluent if and only if given a vector configuration cf and any two possible steps $cf \xrightarrow{x_1} cf_1$ and $cf \xrightarrow{x_2} cf_2$, where $cf_1 \neq cf_2$, there exists a configuration cf' such that $cf_1 \xrightarrow{x_2} cf'$ and $cf_2 \xrightarrow{x_1} cf'$. □

Lemma 4.2.2. The basic semantics of compensating automata (*i.e.,* the rules given in Figure 4.7) is confluent assuming that the automaton only receives a single system event at a time.

Proof. The proof follows by a rule-by-rule analysis on each case of configuration type.

Case 1: cf is a basic forward configuration, $cf = (q, S)^{\ogreaterthan}$

By rule analysis, only Suc and Fail can trigger (but not both by the single event assumption).

If Suc triggers, by Definition 4.2.8 and the single event assumption, at most one element of δ can trigger Suc and thus $cf_1 = cf_2$.

On the other hand, if Fal triggers there is clearly one way the configuration can progress and hence $cf_1 = cf_2$.

Case 2: cf is a basic backward configuration, $cf = (q, S)^{\olessthan}$

By rule analysis, only Comp and Dev can trigger depending on stack S.

If $S = c \mathbin{;} S'$, by definition of Comp the configuration can progress in one way and thus $cf_1 = cf_2$.

If $S = d \mathbin{;} S'$ and by definition of Dev the configuration can progress in one way and thus $cf_1 = cf_2$.

Case 3: cf is a nested forward configuration, $cf = (q, S)^{\ogreaterthan}_{cf'}$

By rule analysis, rules NestSuc, NestFail and Nest can trigger depending on the terminated/compensated status of the nested vector configuration. By Proposition 4.2.1 and propositional logic, the side conditions are mutually exclusive and thus there is only one way the configuration can progress, leading to $cf_1 = cf_2$.

Case 4: cf is a nested backward configuration, $cf = (q, S)^{\olessthan}_{cf'}$

> By rule analysis, rules NestComp and Nest can trigger depending on the terminated/compensated status of the nested vector configuration. By propositional logic, the side conditions are mutually exclusive and thus there is only one way the configuration can progress, leading to $cf_1 = cf_2$.

Case 5: cf is a vector configuration, $cf = \alpha +\!\!+ [cf_1] +\!\!+ \beta +\!\!+ [cf_2] +\!\!+ \gamma$ where α, β, γ are lists of configurations.

> Since rule Vect allows individual configurations to progress independently, the possibility of $cf \xrightarrow{x_1} cf_1$ and $cf \xrightarrow{x_2} cf_2$, where $cf_1 \neq cf_2$ arises.
>
> Taking any two distinct applications of rule Vect,
> $\alpha +\!\!+ [cf_1] +\!\!+ \beta +\!\!+ [cf_2] +\!\!+ \gamma \xrightarrow{x_1} \alpha +\!\!+ [cf_1'] +\!\!+ \beta +\!\!+ [cf_2] +\!\!+ \gamma$, and
> $\alpha +\!\!+ [cf_1] +\!\!+ \beta +\!\!+ [cf_2] +\!\!+ \gamma \xrightarrow{x_2} \alpha +\!\!+ [cf_1] +\!\!+ \beta +\!\!+ [cf_2'] +\!\!+ \gamma$,
> the resulting configuration vectors do not contain any modifications of cf_2 and cf_1 respectively. Thus in both cases cf' exists:
> $\alpha +\!\!+ [cf_1'] +\!\!+ \beta +\!\!+ [cf_2] +\!\!+ \gamma \xrightarrow{x_2} \alpha +\!\!+ [cf_1'] +\!\!+ \beta +\!\!+ [cf_2'] +\!\!+ \gamma$, and
> $\alpha +\!\!+ [cf_1] +\!\!+ \beta +\!\!+ [cf_2'] +\!\!+ \gamma \xrightarrow{x_1} \alpha +\!\!+ [cf_1'] +\!\!+ \beta +\!\!+ [cf_2'] +\!\!+ \gamma$.

□

Corollary 4.2.3. Assuming that the automaton only receives a single system event at a time, basic steps are confluent.

Proof. Basic steps echo the semantic rules with the only constraint of checking for local communication before progressing. Since no rule removes any local channels from the configuration (but possibly adds), no membership check can ever be violated after the triggering of a rule. Furthermore, set union is associative and commutative so that $(L_1 \cup L_2) \cup L_3 = (L_1 \cup L_3) \cup L_2$. Hence by this observation and Lemma 4.2.2 basic steps are confluent. □

Theorem 4.2.2. Compound steps are deterministic:
$\forall cf, cf' \cdot in(\hat{A}) \overset{e:es}{\rightsquigarrow\!\!*} cf \wedge in(\hat{A}) \overset{e:es}{\rightsquigarrow\!\!*} cf' \implies cf = cf'$

Proof. Each compound step, whether $\overset{\tau}{\rightsquigarrow}$ or $\overset{e}{\rightsquigarrow}$, exhausts all the available basic steps such that before considering the next system event there are no possible continuations of the configuration.

By Corollary 4.2.3 and since compound steps consider a system event at a time, the basic steps constituting a compound step are confluent. Consequently by the contrapositive of confluence, *i.e.,* since there are no continuations at the end of a compound step, the end configurations of any step sequence must be equal.

□

4.2.5 Stability in Compensating Automata

Since the proposed architecture of employing compensating automata is to execute the automata in synchrony with the system, monitors bring about instability to the system, *i.e.,* the system has to wait for control to be returned before continuing further. To ensure that the monitor eventually returns control to the system, we reason about the stability of compensating automata. Intuitively, the behaviour of a compensating automata vector is stable if the behaviour is finite. Since compound steps are defined as a finite sequence of steps, then we define stability of a compensating automata in terms of the existence of a compound step under all possible scenarios: a compound step if a τ-step is performed, a compound step if the system signals compensation, and a compound step if the system signals a normal event.

Definition 4.2.18. A compensating automata vector $\widehat{A}$ is said to be stable if and only if for any configuration $cf_{[L]}$ of $\widehat{A}$, compound steps are defined, *i.e.,* there exists $cf_{1[L_1]}$, $cf_{2[L_2]}$, $cf_{3[L_3]}$ such that, $cf_{[L]} \xhookrightarrow{\tau\tau\ldots\tau}\!\bullet cf_{1[L_1]}$, $cf_{[L]} \xhookrightarrow{⁋⁋\ldots⁋}\!\bullet cf_{2[L_2]}$, and $cf_{[L]} \xhookrightarrow{e_{A_1} e_{A_2} \ldots e_{A_n}}\!\bullet cf_{3[L_3]}$ $(\forall i, j : 1..n \cdot i \neq j \implies A_i \neq A_j)$. □

Proposition 4.2.7. Given a vector of compensating automata and a corresponding configuration, there are only a finite number of τ steps (in $cf_{[L]} \xhookrightarrow{\tau\tau\ldots\tau}\!\!\bullet cf'_{[L']}$) which can occur.

The proof follows by induction on the structure of the automaton.

Proof. **Base case: an automaton with no nesting** Let $Q_\#$ represent the non-visited states of an automaton, and $S_\#$ represent the size of the stack of a configuration. Following a transition, the resulting values of two variables are denoted by their primed counterparts.

As an invariant, we show that the following holds over all possible transitions:

$$\begin{aligned} & (Q'_\# = Q_\# - 1 \wedge S'_\# \leq S_\# + 2) \\ \vee \quad & (Q'_\# \leq Q_\# \wedge S'_\# = S_\# - 1) \end{aligned}$$

By rule analysis, there are three rules which can trigger on a configuration of a single automaton with no nesting:

Rule Suc By rule inspection, Suc contributes at most two elements to the stack (one if there is no deviation) and reaches a fresh state, which by Definition 4.2.9 (disallowing any loops over τ's and local channels) cannot have been reached before. This clearly satisfies the first disjunct of the invariant.

Rule Dev By rule inspection, Dev consumes an element from the stack and reaches another state (potentially fresh). This satisfies the second disjunct of the invariant.

Rule Comp By rule inspection, Comp consumes an element from the stack but leaves the state unchanged, satisfying the second disjunct of the invariant.

Due to the fixed size of the automaton and a non infinite stack, the maximum number of steps is finite: given a total number of states Q_m and an initial stack size S_0, the maximal number of transitions, denoted A_m, is $Q_m + 2Q_m + S_0$.

Inductive case 1: an automaton with nesting Further to the previous case, $D_\#$ represents the depth of a nested configuration, and D_m represents the maximum structural depth of the automaton.

As before the three rules Suc, Dev, and Comp can trigger through rule Nest but this time they can transition to a nested configuration of arbitrary depth with maximum D_m. By the inductive hypothesis, we are guaranteed that nested configurations of nested automata can only transition a finite number of times. Furthermore, due to the maximal depth of D_m we are guaranteed that nesting is bounded. Note that if each transition reaches a nested state where in turn each state is nested to the maximal depth and so on, then the number of transitions can be calculated in terms of the geometric series: $\sum_{k=0}^{D_m} D_m A_m ({Q_m}^k)$.

By rule analysis, further to the three rules considered above, there are three rules which can trigger on a nested automaton configuration: NestSuc, NestFail, and NestComp. By rule inspection, the rules do not modify the stack or the state except for rule NestSuc which contributes an element to the stack. This means that at most, NestSuc can contribute Q_m elements if all states are nested and complete successfully.

Due to the fixed size of the automaton, including the bounded depth of nesting, and a non infinite stack, the maximum number of transitions is finite. In terms of the variables being considered, the number of transitions of a nested automaton, denoted N_m is $A_m + Q_m + Q_m \sum_{k=0}^{D_m} D_m A_m ({Q_m}^k)$.

Inductive case 2: a vector of automata A configuration vector can transition over rule Vect, echoing other rules and contributing to the basic steps given in Figure 4.9. Due to the inductive hypothesis, each sub configuration may take a finite number of transitions while the side conditions of the basic steps possibly suppress transitions but cannot contribute further transitions. Given the bounded size of the vector of automata, the number of potential transitions is finite: taking the size of the vector to be $V_{\#}$, and N_m to be the maximum number of transitions of all nested automata in the vector, the maximal number of transitions of a vector of automata is $N_m V_{\#}$.

□

Proposition 4.2.8. Given a configuration $cf_{[L]}$ there exists $cf'_{[L']}$ such that $cf_{[L]} \xhookrightarrow{\Lsh\Lsh\ldots\Lsh}\bullet cf'_{[L']}$.

Proof. By definition of rule Fail any configuration on which the rule triggers turns from forward to backward, implying that the rule cannot trigger more than once on a particular configuration. Since the vector of automata is finite, then there exists a finite number of steps (possibly zero) which can lead to a configuration. □

Proposition 4.2.9. Given a configuration $cf_{[L]}$ there exists $cf'_{[L']}$ such that $cf_{[L]} \xhookrightarrow{e_{A_1} e_{A_2} \ldots e_{A_n}}\bullet cf'_{[L']}$ $(\forall i, j : 1..n \cdot i \neq j \implies A_i \neq A_j)$.

Proof. By the constraint that $\forall i, j : 1..n \cdot i \neq j \implies A_i \neq A_j$ it is ensured that each automaton can only trigger at most once. Since the vector of automata is finite, then there exists a finite number of steps (possibly zero) which can lead to a configuration. □

Theorem 4.2.3. Compensating automata are stable.

Proof. By Definition 4.2.18 and Propositions 4.2.7 – 4.2.9 compensating automata are stable. □

4.3 Programming with Compensating Automata — A Case Study

In the first section of this chapter we motivated the need of a monitor-oriented compensation notation through the e-procurement case study from the literature. Now that we have presented compensating automata, we show how the e-procurement system can be programmed as a vector of five automata shown in Figure 4.12[12]. Figure 4.12(a)[top] listens for events of program R' up to the point where it is confirmed that at least some of the ordered goods are available. Subsequently, Figure 4.12(a)[bottom] 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 Figure 4.12(a)[top] iterates till all the goods are available[13]. One might argue that Figure 4.12(a)[top] is useless since it is not installing any compensations. However, note that all the compensations which follow (in the rest of the automata) 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. If the there is no need to distinguish amongst patterns of system events, then the automaton depicted in Figure 4.12(a)[top] is indeed useless.

Figure 4.12(b) depicts the automaton which compensates for payment. Initially a *PayProc* compensation is installed so that if the procurement fails before payment is affected, any transport costs incurred (hence the local communication guards) can be collected. Note that *PayProc* has programmed compensation so that if the payment for transport cancellation charges fails, an operator is no-

[12]The abbreviations are as follows: calc (calculate), av (available), rec (receive), canc (cancel), req (request), purch (purchase), inv (invoice), proc (process), trans (transport), addr (address), not (notification), ack (acknowledgement), del (delivery), gds (goods).

[13]To provide compensations for all iterations we use parametrised events and dynamically trigger copies of the automata in the spirit of the LARVA framework [40].

tified. If normal payment succeeds then the *PayProc* compensation is replaced by a *Refund* which also takes into consideration the progress of the transport arrangement.

Figure 4.12(c) depicts the automaton which compensates for the 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 Figure 4.12(a) and the *UnreserveGoods* compensation is discarded. Furthermore, Figure 4.12(c) also includes a deviation which enables the procurement system to attempt to correct the address before cancelling the whole procurement — when possible cancelling the whole transaction should be avoided as this is usually the most costly alternative both for the e-procurement system (which effectively loses the proceeds) and for the customer who incurs the cancellation costs.

Finally, Figure 4.12(d) is responsible for installing the *transCancOk* signal and discarding it whenever it is no 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.

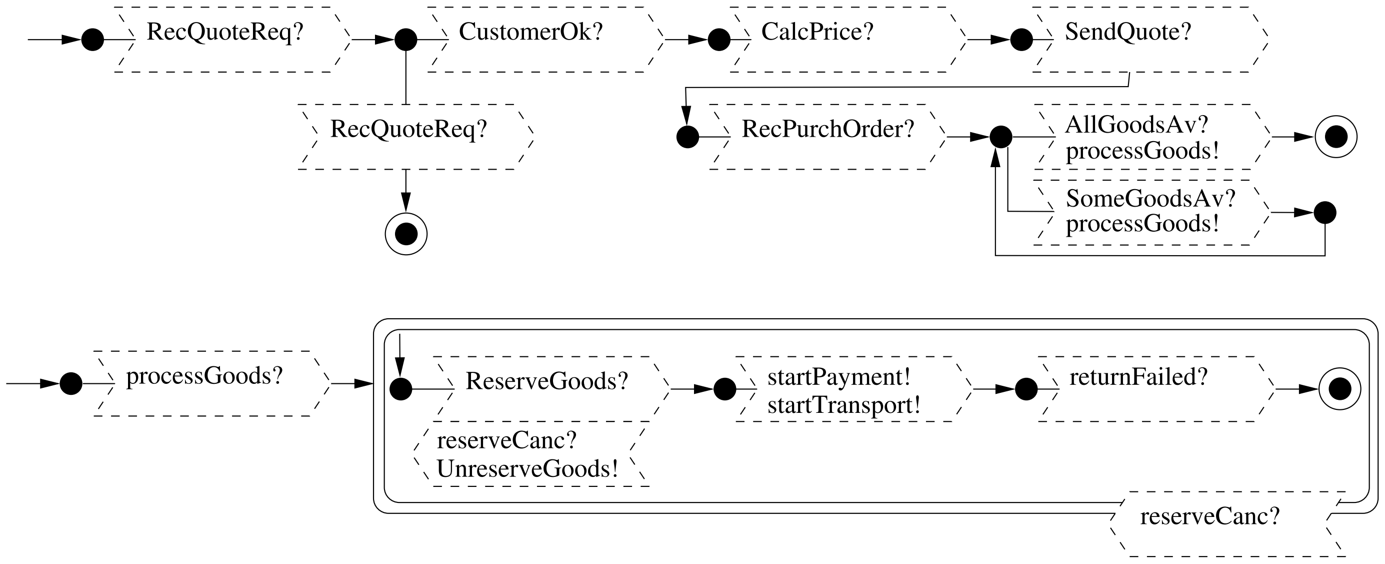
The overall behaviour of programming the compensation manager with these automata and applying it to system S'' (assuming S'' knows the logic corresponding to *UnreserveGoods*, *RecPay*, *Refund*, *etc.*) would satisfy all the features of the e-procurement system as presented in Section 4.1.1.

4.3.1 Other Solutions to the Case Study

We are aware of only one other approach (by Nepal *et al.* [87]) which offers a solution to the e-procurement scenario. Interestingly, the approach is the complete opposite of ours and proposes a model which does not differentiate between normal and exceptional behaviour, and abstracts away from the notion of compensations. They claim that this approach, which is based on a guarded-command language, simplifies the specification of the system. Admittedly, the examples given in [87] are very readable since the model is somewhat similar to a textual specification with many statements of the form: "If *this* happens, then *this* should happen". The disadvantage of this approach is that the "compensation view" is lost, *i.e.,* the relationship between an action and its compensation, the ordering of compensation execution, *etc.*, is not visible from the model. If such a view is not necessary for the programmer, then, indeed, this model should be preferred. However, the compensation view has its advantages: (i) the correspondence between actions and their compensations is useful since one would usually require part of the system state at the time of executing the action to be available during the execution of the compensation; (ii) compensation notations usually clearly encode sequential constraints amongst actions and compensations — information which is useful for ensuring temporal properties. If these advantages are important in a given context, then employing compensating automata might provide the right balance between the compensation view and flexibility.

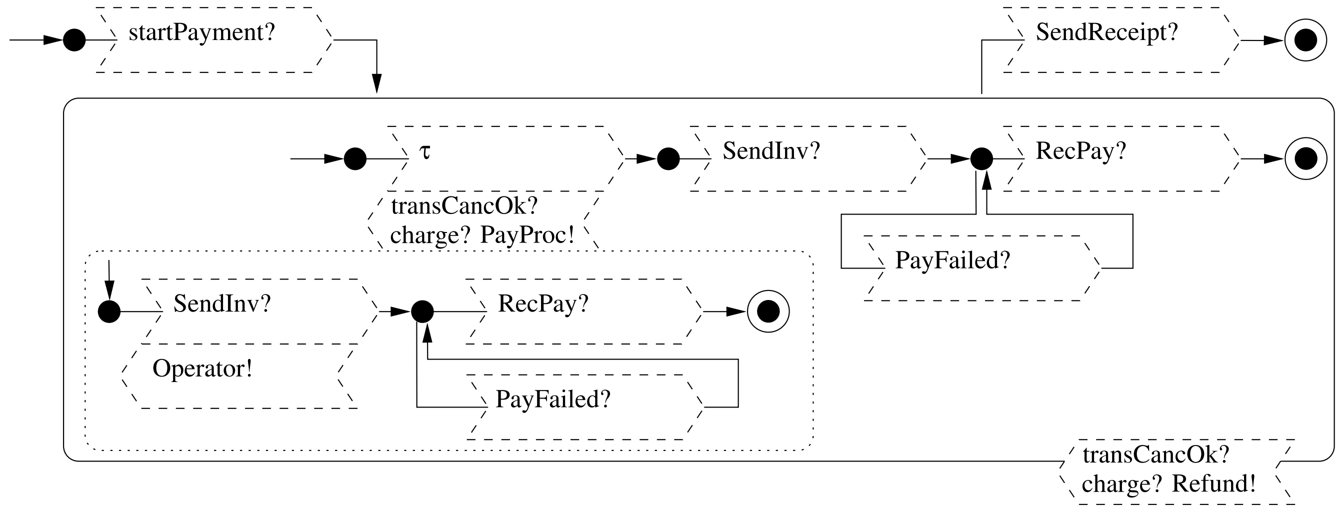
4.4 Related Work

There are numerous formalisations and notations [1, 9, 16, 21, 22, 24, 26, 28, 34, 49, 52, 58, 59, 62, 69, 70, 71, 72, 73, 74, 75, 78, 79, 82, 97] which support compensations including some of which are pictorial (*e.g.,* Petri net-based formalisms [62], communicating hierarchical transaction-based timed automata (CHTTAs) [72], and BMPN [1]). There are three main features distinguishing our work. Firstly, due to the proposed compensation design paradigm which separates compensation concerns from other programming, compensating automata do not have operators which are not directly related to compensations such as alternative forwarding and speculative choice (provided for example in [21, 26]). Secondly, compensating automata provide a set of compensation-dedicated operators which do not require the user to hand-code frequently oc-



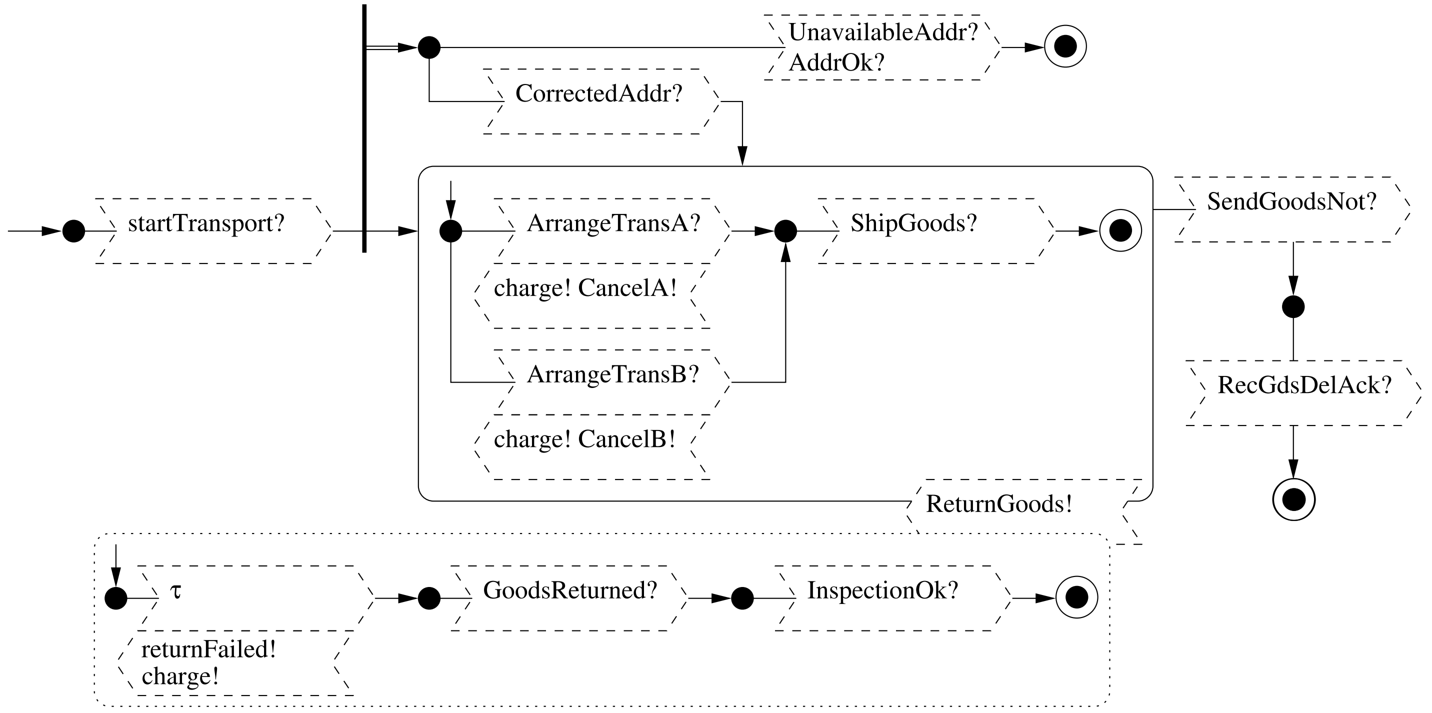
(a)

Figure 4.12: The compensating automata vector for the e-procurement system (cont.)



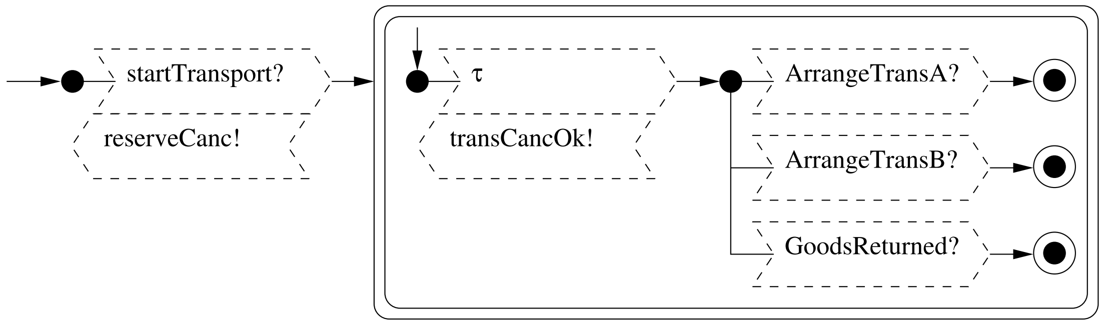
(b)

Figure 4.12: The compensating automata vector for the e-procurement system (cont.)



(c)

Figure 4.12: The compensating automata vector for the e-procurement system (cont.)



(d)

Figure 4.12: The compensating automata vector for the e-procurement system

curring patterns in compensation design (for example compensating automata provide explicit compensation replacement as opposed to for example [24, 59] 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.

A formalism which is arguably similar to ours is that of CHTTAs. However, in CHTTAs compensations are not first-class operators and are instead wired in terms of communication channels. Furthermore, compensations in CHTTAs are programmed in terms of patterns which do not have the unstructured nature of automata.

In the previous chapter, we have presented a bookshop example and shown how this can be encoded in a number of compensation formalisms. To further illustrate how compensating automata relate to the other works, we encode the example in terms of compensating automata and compare it to a number of other of encodings which have been given in Section 3.3. In summary the bookshop example consists of the following important elements:

1. The user first places an order and the compensation for the order placement activity involves two parallel activities: reversing the stock decrement which occurred during the order, and sending an email to the customer with the reason for which the order has been cancelled.

2. A second element is a promotional offer which the customer can freely refuse without affecting the outcome of the order. However, if the offer is refused, a second offer is made. If either of the offers is taken up and the transaction fails to go through, the offer is withdrawn.

3. Following the offer, the bookshop attempts to concurrently pack the order and charge the customer's credit card. If any of these activities fails, the whole order is considered a failure and compensation starts.

4. If both packing and payment are successful, the bookshop concurrently initiates a booking procedure with two couriers; whichever succeeds first is taken up while the other is terminated/cancelled.

5. Compensations are executed in reverse order of their original execution with the concurrent compensation for concurrent execution.

Encoding this arrangement in compensating automata (shown in Figure 4.13) is quite straight forward except for the speculative choice. Programming the speculative choice requires a number of synchronisations which significantly hamper the clarity of the specification. A solution to such synchronisation would be to add syntactic sugar in the form of specialised constructs which hide the channel communication. Another interesting detail is that to easily fit the packing and credit in the sequence, we have used a compensation scope. However, this is not the purpose of the scoping construct. In particular note that the the installed compensation ($Unpack \,||\, Refund$) is the same as the one being discarded. To encode the same logic without using compensation scoping requires a number of additional synchronisation (shown in Figure 4.14) which clutter the specification. Again, a specialised construct through syntactic sugaring would be useful for such patterns.

A significant difference between specifying compensations in compensating automata as opposed to other notations is that compensating automata do not attempt to program the forward execution of the system, but rather simply listen to the events as the system proceeds. This separation of concerns has implications on the specifications since irrelevant system events or irrelevant event orderings can be ignored. For example in the specifications given in Figure 4.14, the specified compensations would match even if courier booking occurred prior to packing or payment. While this might not always be an acceptable compensation program — *e.g.,* if different orderings commend different compensations — it would avoid making the specifications unnecessarily specific. If one would want the compensation program to match only if the expected forward ordering of events is adhered to, then further synchronisation would be required as depicted in Figure 4.15.

In what follows we compare the example encoding in compensating automata to the encodings presented in Section 3.3:

cCSP What is striking in cCSP encoding is its succinctness — a few lines long as opposed to a significant compensating automaton. This is mostly due to the number of specialised operators which provide a direct solution to the standard patterns in the example. The downside of cCSP is its rigidity in programming compensations. For instance there is no way the compensations of sequential operations can be executed in parallel. Thus cCSP is highly compact for typical compensation scenarios but very limited as soon as the problem becomes particular such as the e-procurement scenario.

StAC$_i$ StAC$_i$ is particularly interesting as a notation since like compensating automata it gives scope for a lot of flexibility for handling out of the norm scenarios. The disadvantage of programming with StAC$_i$ is that it feels more like stack programming rather than compensation programming due to the direct stack manipulation it allows. This issue is addressed in compensating automata by providing a number of high level constructs which alleviate the user from programming low level operations.

SOCK Since SOCK is aimed at modelling choreographies, it is considerably different from the rest of the notations being compared. Due to this, the encoding in SOCK highlights the interaction across the parties involved — something which is not visible from the orchestration point of view. The downside of this is that the visibility of communication comes at the cost of obscuring the flow view due to the extra wiring across processes. On a positive note, SOCK, like StAC$_i$, enables a high degree of flexibility in compensation programming including fully customised compensation

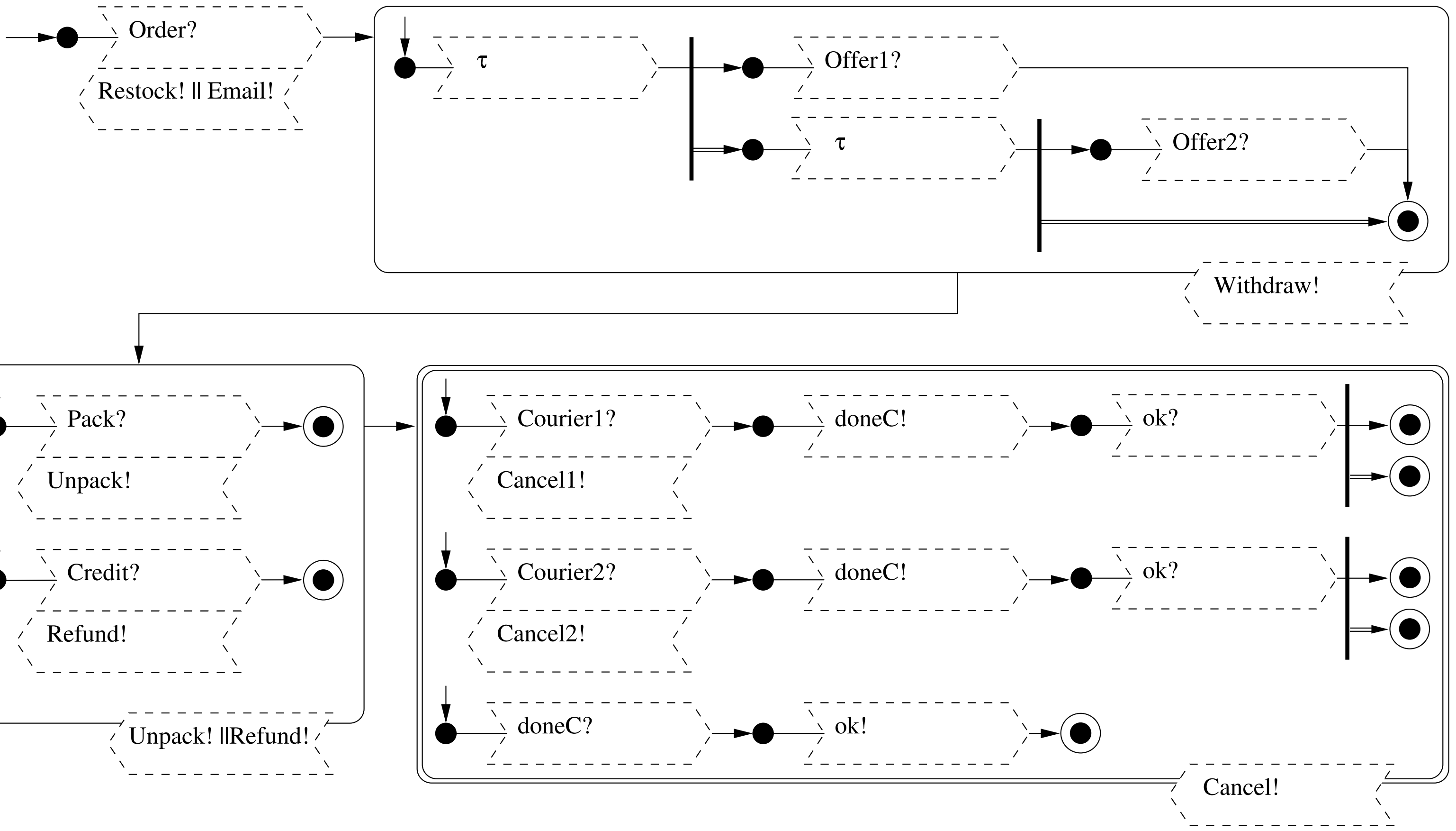


Figure 4.13: The compensating automata vector for the comparison example with synchronised compensations

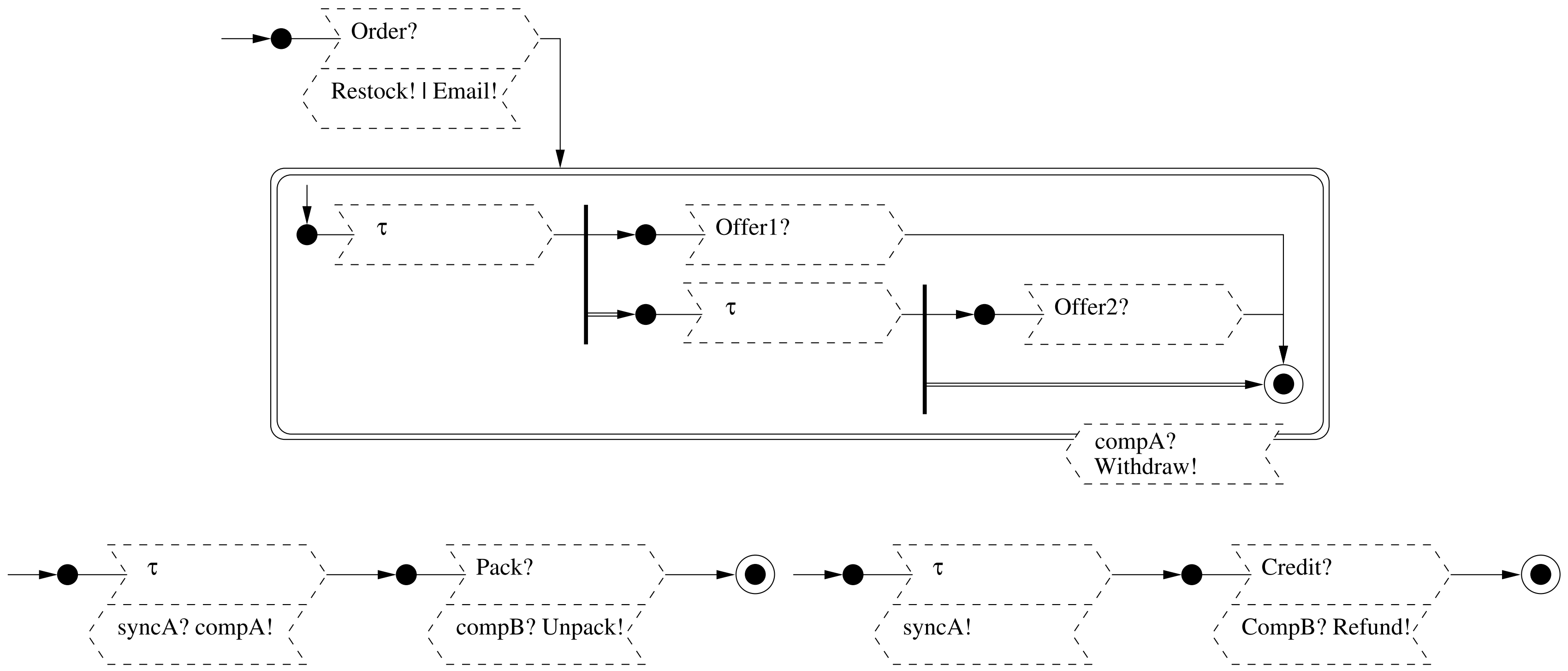


Figure 4.14: The compensating automata vector for the comparison example with synchronised compensations (cont.)

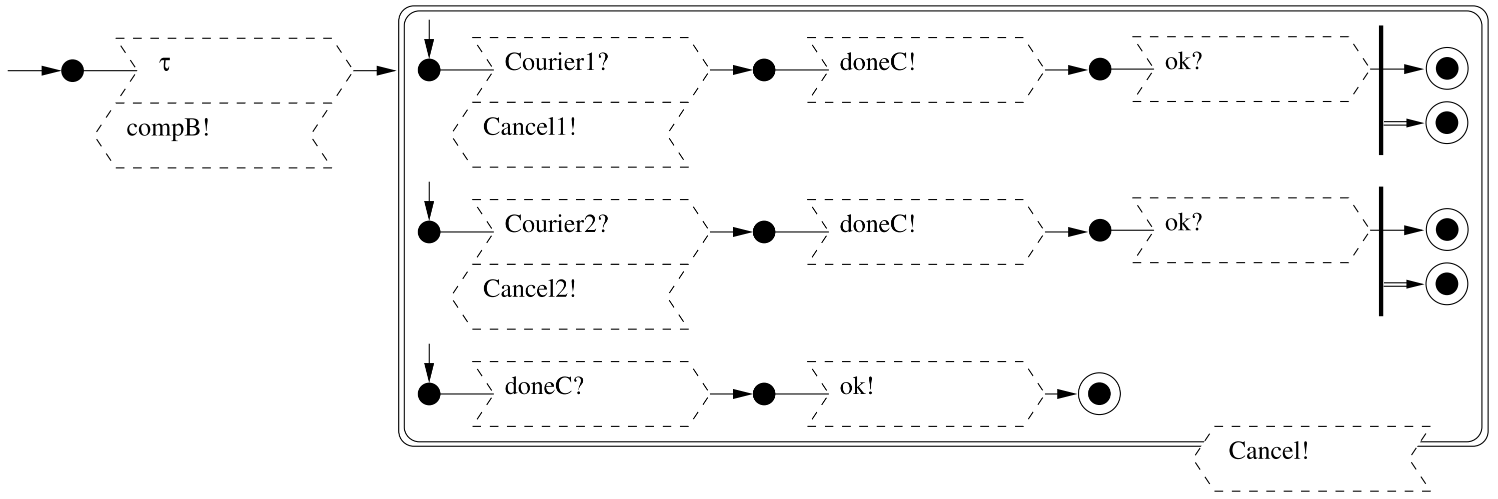


Figure 4.14: The compensating automata vector for the comparison example with synchronised compensations

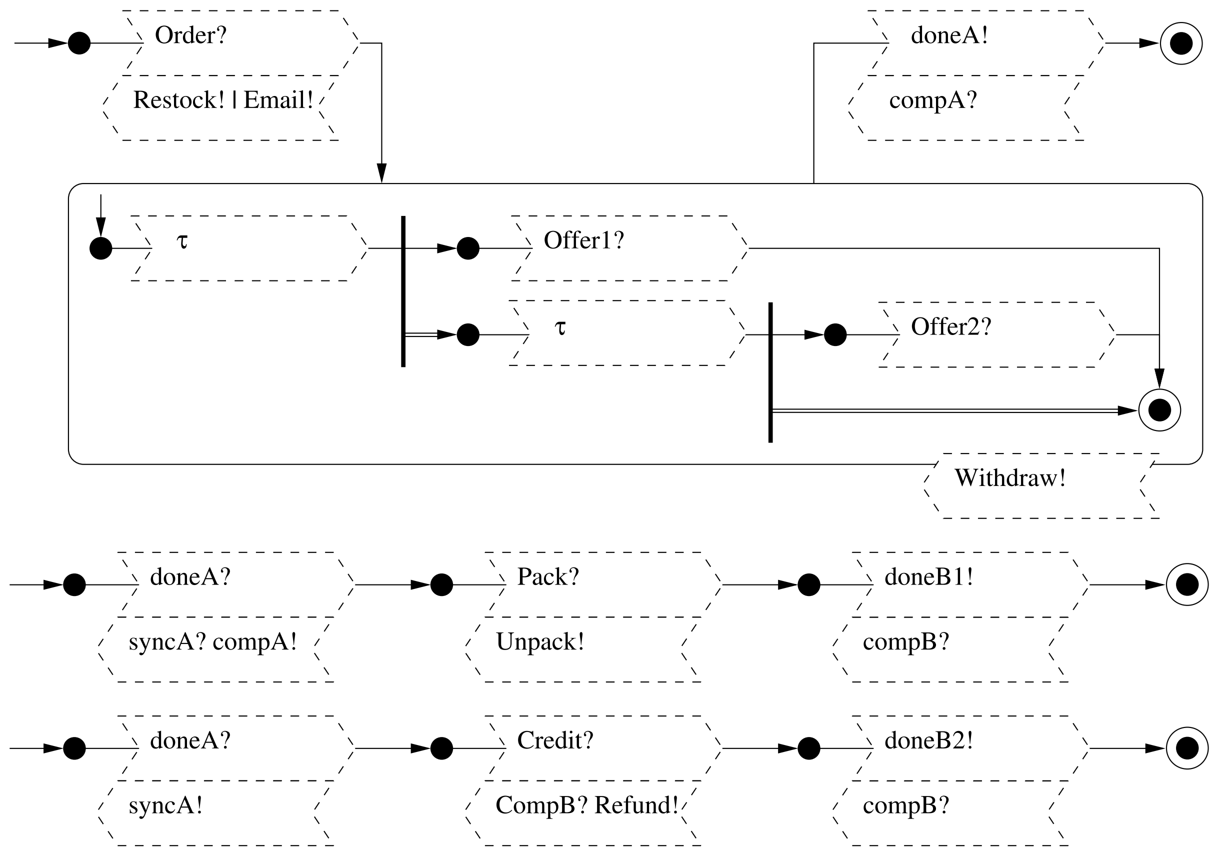


Figure 4.15: The compensating automata vector for the comparison example with full synchronisation (cont.)

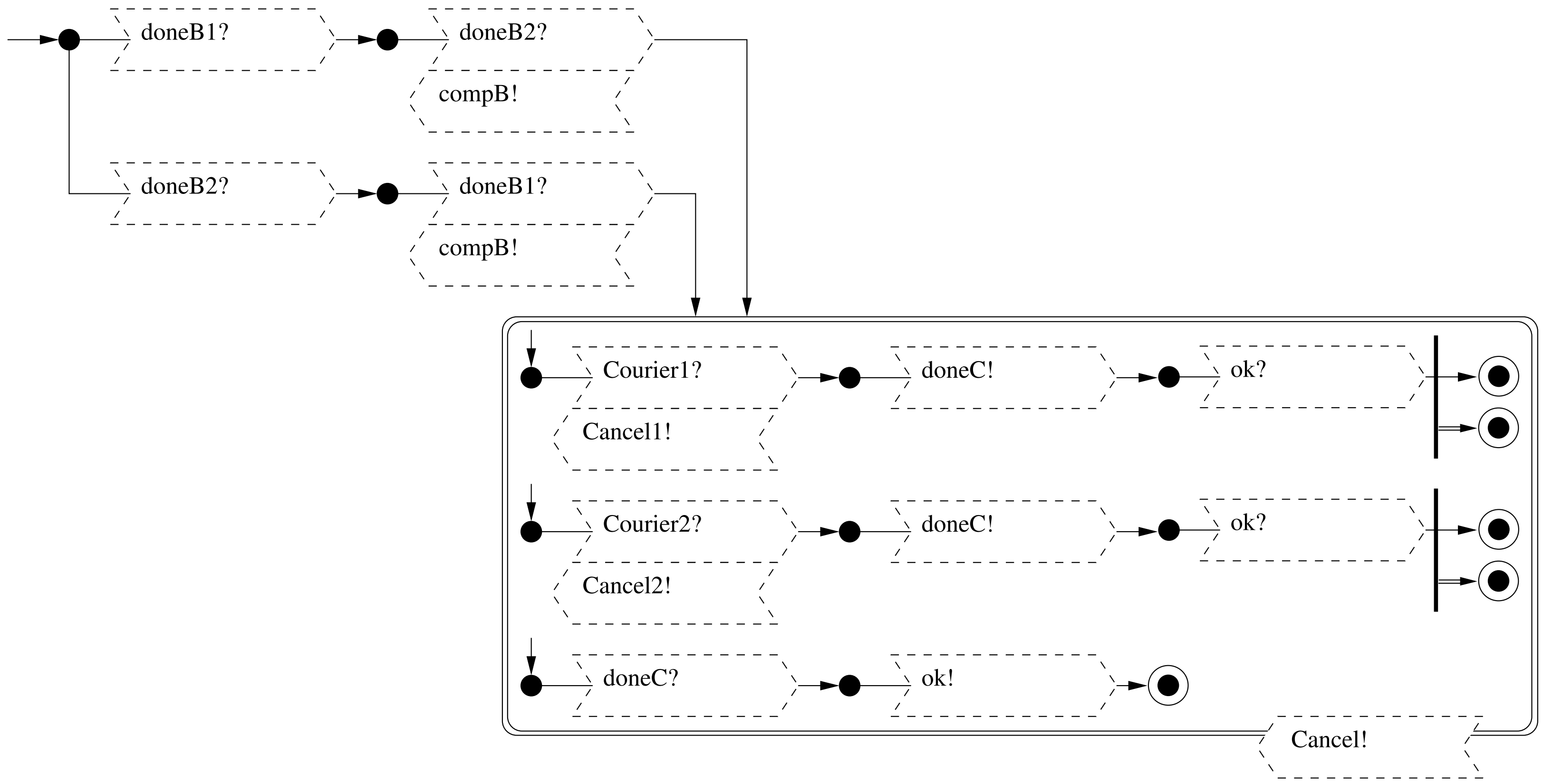


Figure 4.15: The compensating automata vector for the comparison example with full synchronisation

handling. Although this enables one to program highly irregular patterns, we consider SOCK to be an extreme in this regard and offers no "default" means of compensation which can be used in standard cases such as the example above. The consequence is that the programmer is burdened with the extra work and the resulting process definitions are not straightforward to understand.

CHTTAs At face value the example encoded in CHTTAs is very simple and straightforward. However, the problem with CHTTAs is that one cannot model all the aspects of the example and central elements such as the speculative choice had to be completely left out. Although CHTTAs are automata-based, they are actually programmed in a logic-like fashion through LRT patterns meaning that the advantages attributed to automata are lost. On the other hand, if one had to attempt programming directly through CHTTAs, the relationship between actions and their compensations is not at all clear as explained in the review of CHTTAs given in Section 3.3.

Distilling the above discussion and including other general observations, we note the following advantages and disadvantages:

+ The main appeal of compensating automata is that they enable the user to program *only* compensations. Indeed, while the example encoding refers to system events, it does not affect these events, but simply listens out for them.

+ Another significant advantage linked to the previous is that compensation programming is free from any structure restrictions from the system program structure. On the other hand, in the other notations, compensation scoping is generally dictated by transaction scoping while in the case of compensating automata this is managed by a dedicated construct.

- + Most notations either support a fully sequential compensation strategy, a fully parallel one, or a fixed compromise of the two. In the case of compensating automata it is up to the user to decide the level of synchronisation required across compensations, restricting the ordering only when required.
- + A high degree of flexibility is also provided through the unstructured nature synonymous with automata, enabling the programmer to encode choice and repetition in a natural way.
- + Compensating automata are diagrammatic, meaning that some aspects of the specification such as sequentiality would be more immediately visible to the programmer.
- – The flip side of the advantage of greater flexibility is that the user has to manually program the synchronisations across compensations to enforce an ordering.
- – While a number of notations (*e.g.*, [25, 80]) have formally defined translations to and from an industrial standard (usually BPEL [9]) this is not the case (yet) for compensating automata.
- – Another downside of compensating automata is that no model checking techniques have been devised for them (as yet). Note that model checking compensating automata is non-trivial since their state space is potentially infinite due to the unbounded size of the stack.

4.5 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 chapter, we have presented an alternative approach to compensation programming through a monitoring-oriented approach. In summary we have proposed: (*i*) a novel compensation programming 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 and semantics of these automata; (*iv*) proof that compensating automata are self-cancelling, deterministic and stable when used for monitoring; and (*v*) programming compensations for an e-procurement system — showing compensating automata to be useful for handling non-trivial compensation logic.

A limitation of the approach presented in this chapter 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 [24]). In the next chapter we aim to lift this limitation by using monitoring techniques to decide which compensation should be run in the given runtime context.

5. Monitor-Oriented Compensation Programming

In the previous chapter we have proposed a notation for programming two aspects of compensations: *what* — what actions need to be compensated, and *how* — how the actions are to be compensated. Indeed, the vast literature of compensation programming [38] focuses on these two aspects. With this approach we have successfully shown how a system can be alleviated from being cluttered with compensation logic. However, with our proposal the system is still "compensation-aware" since it is the system which has to signal the monitor to execute compensations. Indeed, compensation execution does not occur in a vacuum and compensation programming must also answer the question of *when* to execute compensations.

While many approaches in the literature assume that compensation execution is to be triggered upon a failure [16, 21, 22, 24, 26, 69, 70, 72, 74, 79, 97] or through the exception handler as in the case of BPEL [9], compensation execution might not be directly linked to system failure. For example if a user is detected to be fraudulent, his or her actions might need to be compensated even if the system has not failed. Similarly, in case of an order cancellation, the system has not failed but actions might need to be compensated. Another case in point is the automation of web services adaptation [8, 11, 30, 95, 96] which might trigger compensations in response to deteriorating quality of service (not necessarily failure).

Furthermore, the *when* question is further compounded if an action has more than one possible way of being compensated. For example if a purchase order is being compensated for because of a failure in the payment gateway, the compensation might be different from the case where the user has deliberately cancelled the purchase — the cancellation charges would need to be bore by different parties accordingly. Thus, a question which goes hand in hand with *when* is *which* compensation strategy is to be executed. To the best of our knowledge only StAC$_i$ [24] explicitly supports the specification of multiple compensation strategies. However, in StAC$_i$ all four aspects of *what*, *how*, *when*, and *which* are programmed together and moreover, the programmer also programs the system's (non-compensation) logic within StAC$_i$. On the other hand, in the previous chapter we have proposed a monitor-oriented language which separates compensation programming concerns from other concerns. The same separation approach may be useful for programming the questions of *when* and *which*. Thus contrary to StAC$_i$, we split the programming of a system with compensations into three: (*i*) system programming; (*ii*) the *what* and *how* of compensations; and (*iii*) the *when* and *which* of compensations. For example, consider the case of a financial transaction system where users are carrying out transactions while being monitored for fraud. All users' actions may require to be compensated but the timing and kind of compensation depends amongst other things on whether the user is deemed to be fraudulent or not at the time of compensating. Note that the decision of marking a user as non/fraudulent is a complex one and orthogonal to the management of compensations.

In this chapter, we propose a complete compensation management architecture (Section 5.1) which enables a user to program the compensations separately from the rest of the system and furthermore the *what* and *how* of compensations are programmed separately from the *when* and *which* by exploiting monitoring techniques. Subsequently, in the rest of the chapter we show how our approach has been applied to a realistic case study (Section 5.2), and how it relates to other works in the area (Section 5.3).

5.1 Monitor-Oriented Compensation Programming

In this section we present an architecture which provides a comprehensive compensation programming architecture — alleviating a system from compensation programming up to the point where the system need not be aware of compensations. To this end, the architecture is entirely based on monitoring — hence the term monitor-oriented compensation programming (MOCP) — and is composed of two parts: (i) one which deals directly with compensation aspects such as compensation ordering and discarding, addressing the questions of *what* and *how* to compensate, and (ii) another which deals with triggering compensations at particular points in time, addressing *when* and *which* compensation strategy to execute. Note that the former is involved directly with compensation programming while the latter simply triggers compensation strategies at particular points in time. The first component of such an architecture can be programmed using compensating automata which have been specifically devised for such programming. On the other hand, encoding the *when* and *which* logic of compensations can be done by any specification which can be used for runtime verification and enables the triggering of reparatory actions. For this job we choose DATEs [40] because they are somewhat natural to integrate with compensating automata — they are also automata-based and support channel communication. The two components are integrated as depicted in Figure 5.1. Note that the proposed architecture can also be instantiated using any other suitable specification languages.

The proposed architecture allows the monitoring components to interact with the system through three connections: one which enables the system to

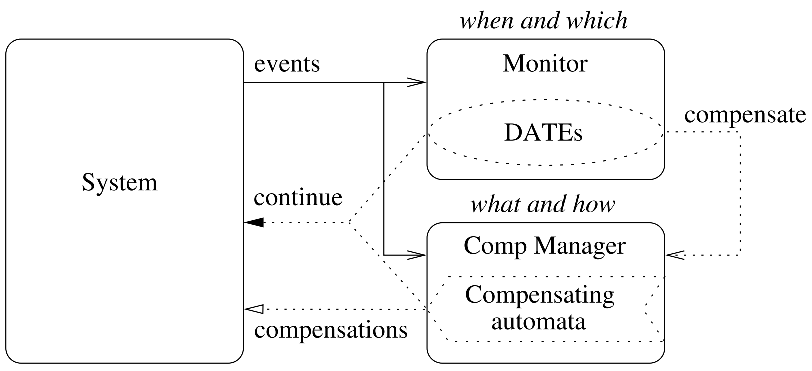


Figure 5.1: The monitor-oriented compensation programming architecture

communicate events, another enabling the monitors to signal the system to continue, and a third one on which the compensation management component instructs the system regarding what compensations to execute. Furthermore, the architecture is event-driven and each event the system emits triggers a series of steps as follows:

1. The system emits an event and waits for the continue signal from the compensation framework.
2. The monitor receives the system event and processes it, emitting a *compensate* signal or a *continue* signal.
3. The compensation manager also receives the system event and processes it.
4. Upon completion, the compensation manager checks whether a *compensate* signal has been received from the monitor.
5. If a *compensate* signal has been received:
   (a) The compensation manager starts emitting compensations to the system.
   (b) While the system is executing compensation instructions, the monitor and compensation manager are still potentially receiving and processing system events.

(c) When the compensation manager has sent all compensation instructions to the system (either the stack is exhausted or a deviation is encountered), it checks once more for the *compensate* signal and if present repeats from Point 5a. Otherwise, execution continues as below.

6. The compensation manager issues a *continue* signal.
7. Once the system detects an active *continue* signal from both the monitor and the compensation manager, the system continues normally.

While the architecture seems to show a single *compensate* line (Figure 5.1), in fact this line is parametrised to enable the monitor to choose whichever compensation strategy is applicable. Similarly, although we show a single monitor, in effect there would typically be several monitors and several compensation managers as shown in Figure 5.2. This arrangement allows a high degree of modularity as we aim to highlight in the next section through a case study.

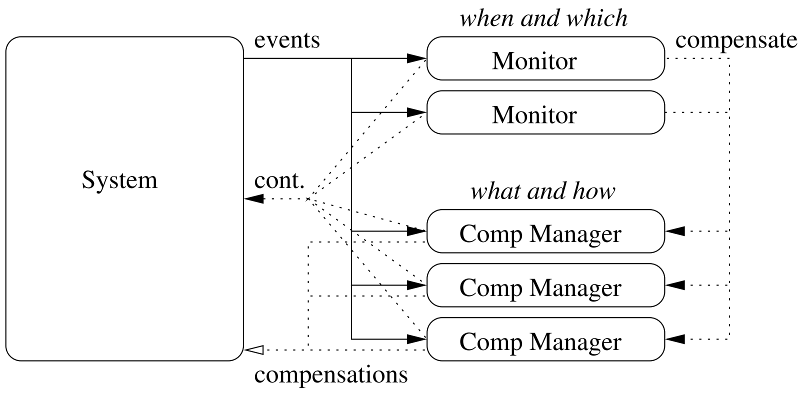


Figure 5.2: The parametrised monitor-oriented compensation programming architecture

5.2 Case Study

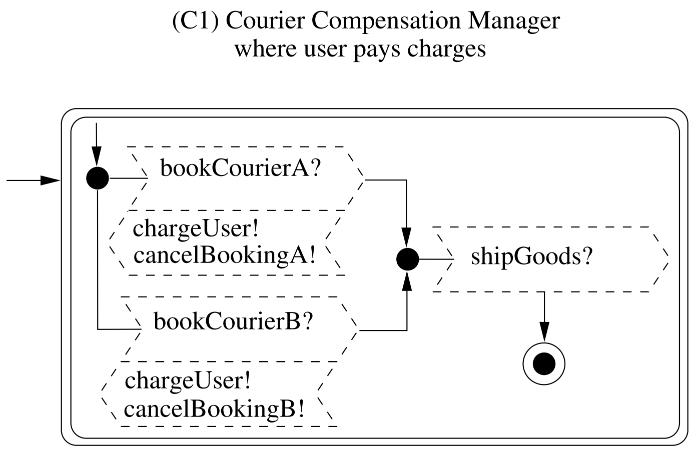
The use of monitor-oriented compensation programming is particularly useful when the decision to activate a compensation strategy is not straightforward.

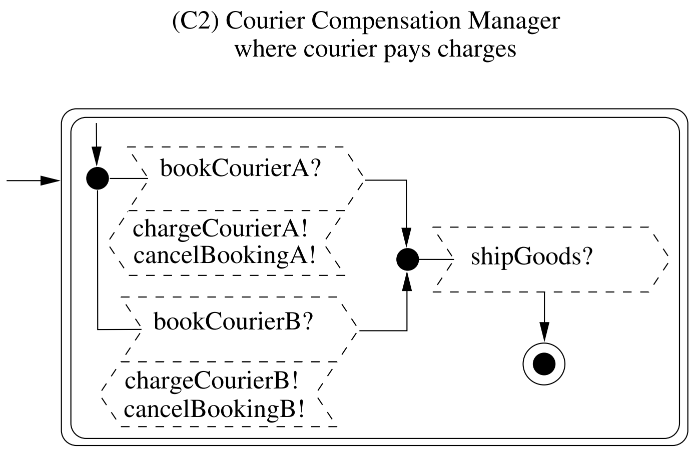
One example where this is the case is a hypothetical e-procurement system[1] which handles payments and shipments of goods. The e-procurement system allows users to create virtual credit cards, load money onto the virtual credit cards from their personal bank accounts, and transfer money across virtual credit cards. Once the user uses money in his or her virtual credit cards to affect a purchase from a third party, the next step would involve the e-procurement attempting to concurrently pay the third party and book a courier on behalf of a user.

A number of possible failures might occur causing the transaction to fail. Cancelling a courier or reversing bank transactions usually incurs a charge and under certain circumstances it might not be possible to cancel such operations (*e.g.,* when the shipment has already left). Furthermore, assuming the compensation is possible, there are at least three parties who might incur the charge: the user, the third party involved (*e.g.,* the courier or the bank), or the e-procurement system. This means that for one courier booking transaction, there are at least three possible compensations (corresponding to *C1*, *C2*, and *C3* in Figure 5.3). Note that compensations are discarded with no replacement once the shipment of goods occurs, *i.e.,* the booking cannot be cancelled. Similarly, for the banking transactions there are at least three possible compensations plus a forth one which keeps track of the credit cards used so that these can be blocked in case of suspicious user behaviour (corresponding to *B1*, *B2*, *B3*, and *B4* in Figure 5.3). Note that in this case once payment succeeds, we assume that the money loads and transfers are not to be undone so that the user only gets the money back after investigation.

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

[1] Inspired from the e-procurement system presented in Section 4.1.1 and the Entropay system presented in Section 6.6.





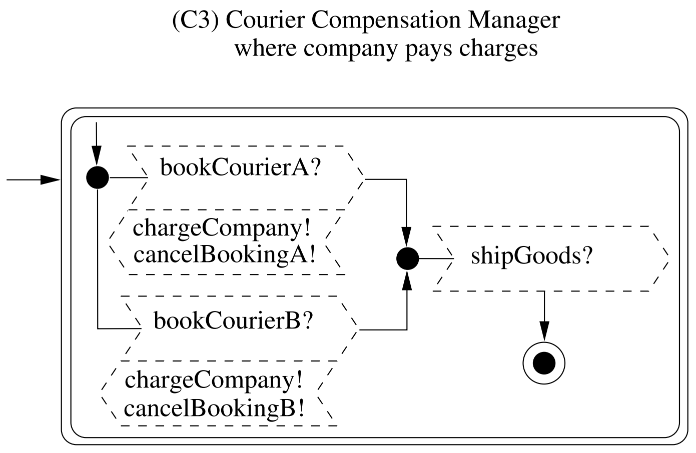
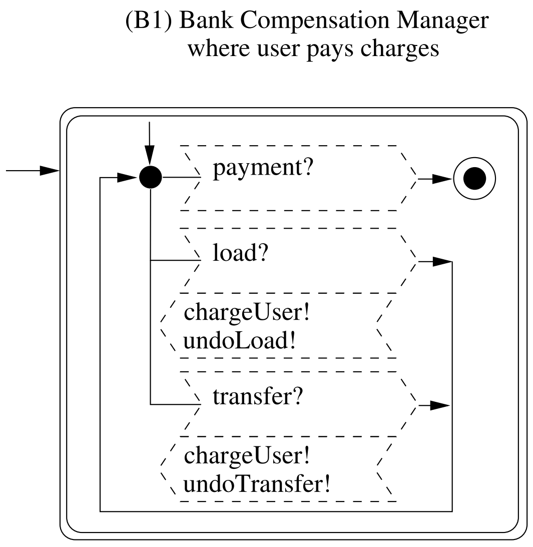


Figure 5.3: The compensating automata vector for the e-procurement system (cont.)



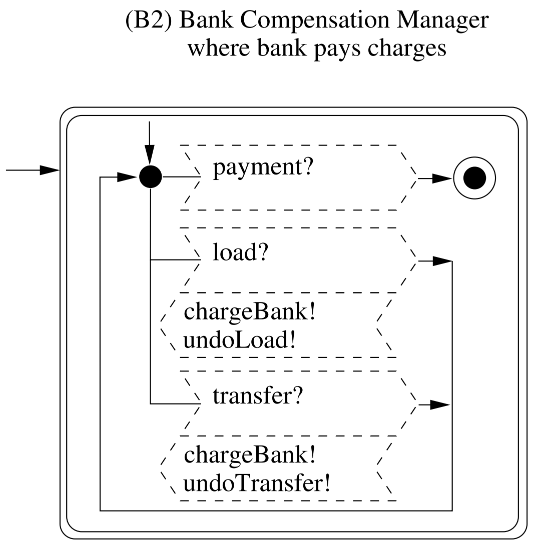
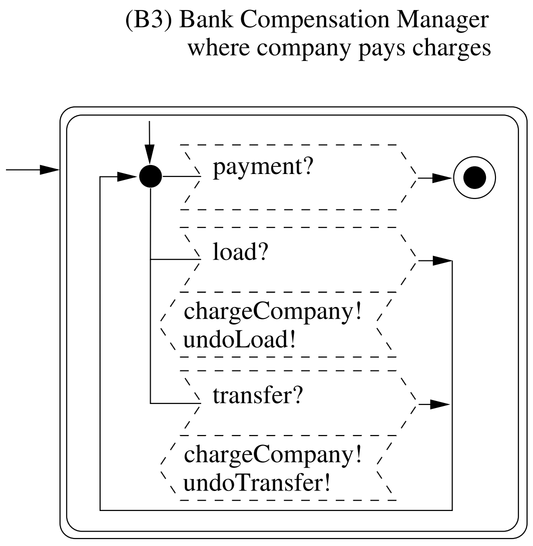


Figure 5.3: Managing compensations for an e-procurement system (cont.)



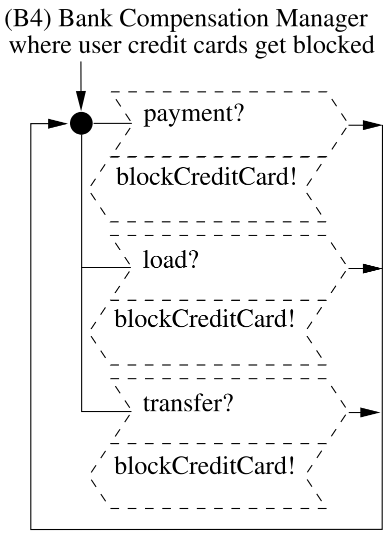
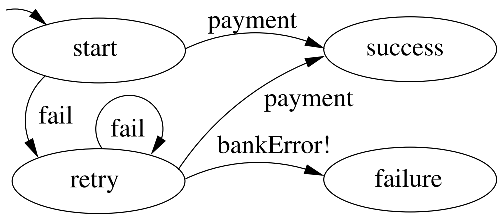


Figure 5.3: Managing compensations for an e-procurement system

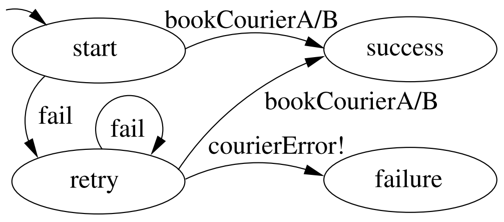
Classification of errors The e-procurement system policy broadly considers three kinds of errors: a bank error, a courier error, or a user cancellation. In the case of a user cancellation it is always the user who should pay for cancellation charges. However, in case of a banking error, the cancellation charges for the banking transactions are incurred by the bank while the cancellation charges for the courier transaction are incurred by the user or the e-procurement system depending on the type of user. A similar approach is taken in case of a courier error. Failure detection is handled by the two simplistic monitors depicted in Figure 5.4(a) and (b).

Classification of users Users are classified as whitelisted, greylisted, or blacklisted. Blacklisted users are suspicious users who cannot be trusted. For this reason these users are not automatically given back their money in case of a failure. Instead, their credit cards are blocked till after human investigation. On the other hand, whitelisted users are trusted customers for whom certain allowances are made such as paying cancellation charges on their behalf. Greylisted users are users who are neither blacklisted nor whitelisted. Each user starts off as greylisted and at a particular point in time a designated monitor (partly shown in Figure 5.4(c)) might classify the user as blacklisted or whitelisted.

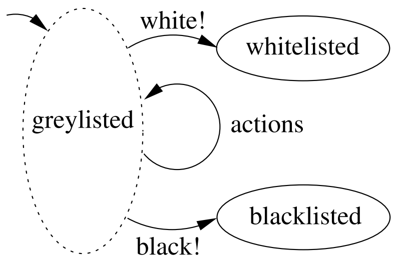
Note that such an e-procurement strategy for handling cancellation includes nine different compensation strategies depending on three user types and three kinds of failure. Clearly, deciding the compensation strategy for such a scenario is not straightforward. By separating the different aspects of compensation programming, the decision can be taken using monitors as depicted in Figure 5.4(d) where compensation strategies (shown in square brackets) can be composed in parallel or sequentially depending on the case. For example in case a blacklisted user cancels the transaction, the payment of the relevant charges should strictly occur before the credit cards are blocked.



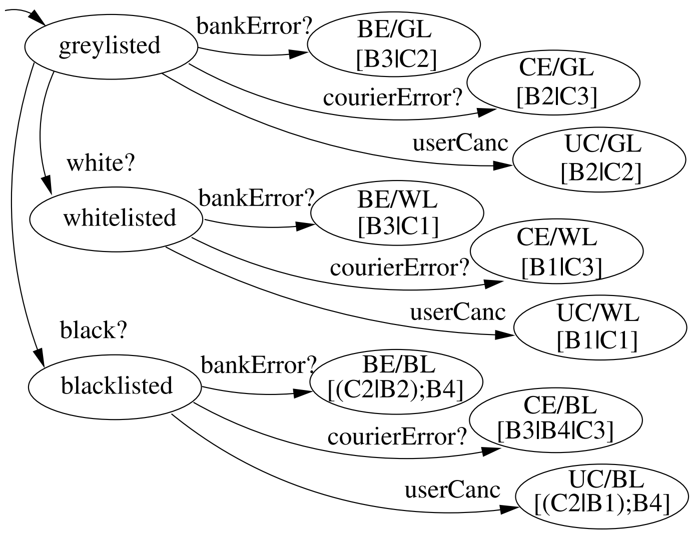
(a) Monitoring for bank errors.



(b) Monitoring for courier errors.



(c) Monitor for whitelisting and blacklisting users.



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

Figure 5.4: Monitoring for triggering compensations for an e-procurement system

5.3 Related Work

Monitor-oriented programming (MOP) [84] has been proposed as a programming paradigm advocating separation of concerns through monitoring. Somewhat analogous to aspect-oriented programming in principle, it differs in that matching occurs through the satisfaction of a formal logic formula rather than over source code pattern matching. Inspired by MOP, we propose monitor-oriented compensation programming (MOCP) where monitors are used for a specific kind of programming, *i.e.,* compensation programming. Compensation programming can be split in the programming of four elements: (i) *what* — what system actions to compensate for; (ii) *how* — how to compensate for the designated system actions; (iii) *when* — when to start compensating; and (iv) *which* — which compensation strategy is chosen if there are more than one way of compensating an action. A fundamental difference between MOP and MOCP is that while MOP matches a pattern to decide *when* to execute a particular logic, in our case MOCP is concerned not only with the *when* but also with the *how* since compensations are programmed on-the-fly while monitoring. MOCP achieves this added expressivity by combining two automata specifications: compensating automata providing the *what* and *how* aspects and DATEs providing the *when* and *which* elements.

In the areas of autonomous adaptation and self-healing a lot of related work [8, 11, 30, 42, 50, 60, 85, 95, 96, 100] has been done in the context of the service-oriented architecture. Of these approaches only [8, 11, 30, 95, 96, 100] support the possibility of executing compensations. These works (with the exception of AO4BPEL [30]) provide a policy language which is able to separate exception handling (and compensation) concerns from the normal business logic but the policy language itself does not go into the details of compensation programming. In other words, the policy language enables a user to specify *what* and *when* to compensate but they do not provide explicit support of specifying *how* to compensate. These approaches are referred to as 'policy languages' in Table 5.1. On the other hand, AO4BPEL provides an aspect-oriented framework which enables the programming of crosscutting concerns such as compensation programming. However, the language does not provide explicit support for programming compensations and is thus left out of Table 5.1.

Most theoretical frameworks which support the specification of compensations [1, 16, 21, 22, 34, 49, 52, 58, 59, 62, 69, 70, 71, 72, 73, 74, 75, 78, 79, 82, 97] (referred to as 'theoretical frameworks' in Table 5.1), enable the user to specify *what* and *how* to compensate but the compensation is activated automatically upon an external failure/signal. On the contrary, other approaches including BPEL and cCSP [9, 26, 28] go further to also enable the user to programmatically invoke the compensation activation, *i.e.,* deciding *when* to execute the installed compensations. For example in the case of BPEL, compensation is optionally invoked from the exception handler enabling full user control of whether or not to execute compensations.

StAC$_i$ [24] gives full control to the user and several compensation strategies can be programmed concurrently (each with an individual compensation stack) and subsequently the user is allowed to program *when* to run *which* compensation stack. This means that StAC$_i$ supports all four aspects of compensation programming which are supported by MOCP. However, MOCP differs from StAC$_i$ in two fundamental ways: (i) MOCP clearly separates compensation specification from compensation activation concerns: with compensating automata able to specify *what* and *how* while DATEs can be used for specifying *when* and *which*. We believe that this separation makes compensation programming more manageable. (ii) Furthermore, MOCP is a monitor-based approach and thus separates compensation concerns from the other concerns. On the other hand, StAC$_i$ requires the programmer to not only program all four aspects of compensation programming but also program the rest of the system concerns including exception handling. Table 5.1 summarises the capabilities of the reviewed compensation programming approaches.

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

Table 5.1: Compensation specification approaches vs. expressivity

5.4 Conclusion

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

In the future we aim to implement our approach and apply it to the case study discussed in this chapter where a number of different compensation strategies would need to be maintained concurrently and executed under different contexts. A possible optimisation is to discard compensation strategies which will surely not be used. Taking the example of the case study, as soon as a user is classified as whitelisted, compensation strategies *B2*, *B4*, and *C2* can be discarded, saving the memory and the time to maintain them.

Part III

Compensations for Runtime Verification

This part gives an account of how compensations can be useful in the context of monitoring, providing the foundational theory, an architecture embodying it, and its application to a real-life industrial case study.

6. Compensation-Based System-to-Monitor Synchronisation

A major concern for the adoption of runtime verification is the inherent overhead that this introduces to the system, with numerous works of research attempting to alleviate this problem particularly through optimisations [10, 17, 19, 48]. While these have been generally successful in reducing the overheads, the problem still hinders adoption by industry as has been confirmed by our experience with the financial transactions industry. The issue is further compounded by the fact that, with possibly few exceptions, existing industrial systems have not been designed with runtime verification in mind and thus no allowance is made for the associated overheads.

One extreme option to address the industry's concern is to adopt asynchronous runtime verification which is virtually non-intrusive given that all system events are usually logged in a database anyway. Subsequently, monitoring can take place on a separate address space without consuming precious system resources. Pure asynchronous monitoring can be depicted as in Figure 6.1, consisting of a system recording events (represented by a circle as in a cassette deck) in a database and subsequently, the monitor plays back (represented by a triangle) the events to check correctness. Note that the system head can move forward without waiting for the monitor head to keep up.

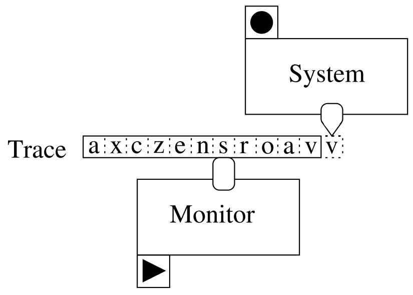


Figure 6.1: Asynchronous monitoring

The main disadvantage of asynchronous monitoring, is that by the time the monitor detects a violation, the system would have probably already progressed, making it hard for the monitor to take effective reparatory actions. This limitation can be alleviated, however, if one can synchronise the system to the monitor. Since the monitor cannot consume further events upon detecting a violation, synchronisation can be achieved by "reversing" the system state till the point where the violation occurred. For this reason we propose to use compensations as a means of synchronisation since these are effectively logical reverses of system actions.

In this context, we present an architecture which enables an asynchronous monitor to achieve synchrony through compensations, thus allowing the monitor to take corrective actions. This approach is particularly suitable for systems such as online transaction systems which usually have inbuilt compensation frameworks which can be exploited for monitor synchronisation. Interestingly, in such systems, fine-grained compensations, in particular circumstances, have to be discarded and replaced by coarser-grained ones. For example, if a fraud has been detected when it is too late to reverse the related purchases, it might be enough to block the offending user's account. Using a box to represent stopping the system, a backward triangle to represent compensations, and double backward triangle to represent coarse-grained compensations, Figure 6.2 depicts the proposed monitoring architecture.

Due to the runtime nature of runtime verification, the choice between syn-

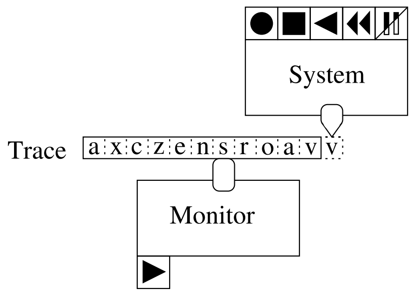


Figure 6.2: The proposed architecture

chronous and asynchronous monitoring might vary depending on the runtime context at hand. For example one might want to generally use synchronous monitoring but switch to asynchrony when the system load is high. Similarly, but more fine-grained, one might prefer untrusted users to be monitored synchronously while trusted users are monitored asynchronously without experiencing any service deterioration due to monitoring. To this end, it is desirable that the system can synchronise and desynchronise on-the-fly by "pausing" to wait for the monitor to keep up and "unpausing" to continue executing, shown in Figure 6.2 as the rightmost button with a diagonal signifying the two actions.

In order to be able to reason about compensation-aware monitoring, and its correctness relative to regular monitoring strategies, we start by characterising synchronous and asynchronous monitoring. In the synchronous version, it is assumed that the system and monitor perform a handshake to synchronise upon each event. In contrast, in the asynchronous approach, the events the system produces are stored in a buffer, and consumed independently by the monitor, which may thus lag behind the system. We then define a compensation-aware monitoring strategy, which monitors asynchronously, but makes sure to undo any system behaviour which has taken place after the event which led to failure.

6.1 Synchronous and Asynchronous Monitoring

To enable reasoning about system behaviour and compensations, we will be talking about finite strings of events. Given an alphabet Σ, we will write Σ^* to represent the set of all finite strings over Σ, with ε denoting the empty string. We will use variables a, b to range over Σ, and v, w to range over Σ^*. We will also assume action τ indicating internal system behaviour, which will be ignored when investigating the externally visible behaviour with $\Gamma = \{\tau\}$ (*cf.* Definition 3.4.1). We will write Σ_τ to refer to the alphabet consisting of $\Sigma \cup \{\tau\}$ and since we will not distinguish amongst strings with τ we overload w to also range over Σ_τ^*.

We will assume a labelled transition system semantics over alphabet Σ for both system and monitor. Given a class of system states S, we will assume the semantics $\rightarrow_{\text{sys}} \subseteq S \times \Sigma \times S$, and similarly a relation $\rightarrow_{\text{mon}}$ over the set of monitor states M. We also assume a distinct system state, $\odot \in S$, identifying a stopped system, and a monitor state, $\otimes \in M$, denoting a monitor which has detected failure. Both $\odot$ and $\otimes$ are assumed to have no outgoing transitions.

Using standard notation, we will write $\sigma \xrightarrow{a}_{\text{sys}} \sigma'$ (resp. $m \xrightarrow{a}_{\text{mon}} m'$) as shorthand for $(\sigma, a, \sigma') \in \rightarrow_{\text{sys}}$ (resp. $(m, a, m') \in \rightarrow_{\text{mon}}$). We write $\overset{w}{\Longrightarrow}_{sys}$ and $\overset{w}{\Longrightarrow}_{mon}$ to denote the reflexive transitive closure of $\rightarrow_{\text{sys}}$ and $\rightarrow_{\text{mon}}$ respectively.

Definition 6.1.1. The transition system semantics of the synchronous composition of a system and monitor is defined over $S \times M$ using the rules given in Figure 6.3. The rule Sync defines how the system and monitor can take a step together, while SyncErr handles the case when the monitor discovers an anomaly. A state (σ, m) is said to be (i) *suspended* if $\sigma = \odot$; (ii) *faulty* if $m = \otimes$; and (iii) *sane* if it is not suspended unless faulty ($\sigma = \odot \implies m = \otimes$).

The set of traces generated through the synchronous composition of system σ and monitor m, written $\text{traces}_{\|}(\sigma, m)$ is defined as follows:

$$\text{traces}_{\|}(\sigma, m) = \{w \mid \exists(\sigma', m') \cdot (\sigma, m) \overset{w}{\Longrightarrow}_{\|} (\sigma', m')\}$$

□

Synchronous Monitoring

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

Asynchronous Monitoring

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

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

Figure 6.3: Semantics of synchronous and asynchronous monitoring

Example 6.1.1. Consider a system σ over alphabet $\{a, b\}$ and a monitor m which consumes an alternation of a and b events starting with a *i.e., abab...* but breaks upon receiving any other pattern. The synchronous composition of the system and monitor takes a step if and only if both the system and the monitor can take a step on the given input. Therefore, if the system performs event a, the system and the monitor can both perform a step: $(\sigma, m) \xrightarrow{a}_{||} (\sigma', m')$. If system σ performs b instead, the system would break: $(\sigma, m) \xrightarrow{b}_{||} (\odot, \otimes)$.

Proposition 6.1.1. A sequence of actions is accepted by the synchronous composition of a system and a monitor, if and only if it is accepted by both the monitor and the system acting independently. Provided that $m' \neq \otimes$, then $(\sigma, m) \overset{w}{\Longrightarrow}_{||} (\sigma', m')$, if and only if $\sigma \overset{w}{\Longrightarrow}_{\text{sys}} \sigma'$ and $m \overset{w}{\Longrightarrow}_{\text{mon}} m'$.

Proof. The proof follows by string induction on w and by application of rule Sync. □

In contrast to synchronous monitoring, asynchronous monitoring enables the system and the monitor to take steps independently of each other. The state of asynchronous monitoring also includes an intermediate buffer between the system and the monitor so as not to lose events emitted by the system which are not yet consumed by the monitor.

Definition 6.1.2. The asynchronous composition of a system and a monitor, is defined over $S \times (\Sigma_\tau)^* \times M$, in terms of the three rules given in Figure 6.3. Rule Async$_S$ allows progress of the system adding the events to the intermediate buffer, while rule Async$_M$ allows the monitor to consume events from the buffer. Finally rule AsyncErr suspends the system once the monitor detects an anomaly. Suspended, faulty and sane states are defined as in the case of synchronous monitoring by ignoring the buffer.

The set of traces accepted by the asynchronous composition of system σ and monitor m, written $\mathsf{traces}_{|||}(\sigma, m)$ is defined as follows:

$$\mathsf{traces}_{|||}(\sigma, m) = \{w \mid \exists(\sigma', w', m') \cdot (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_{|||} (\sigma', w', m')\}$$

$\square$

Example 6.1.2. Taking the same example as before, upon each step of the system an event is added to the buffer; *e.g.*, if the system starts with an event b, the resulting configuration would have b in the buffer with no change to m: $(\sigma, \varepsilon, m) \xrightarrow{b}_{|||} (\sigma', b, m)$. Subsequently, the system may either continue further, or the monitor can consume the event from the buffer and fail:
$(\sigma', b, m) \xrightarrow{\tau}_{|||} (\sigma', \varepsilon, \otimes)$. At this stage the system can still progress further until it is stopped by the rule AsyncErr:

$$(\sigma, \varepsilon, m) \xrightarrow{b}_{|||} (\sigma', b, m) \xrightarrow{\tau}_{|||} (\sigma', \varepsilon, \otimes) \xrightarrow{b}_{C} (\sigma'', b, \otimes) \xrightarrow{\tau}_{|||} (\odot, b, \otimes)$$

6.2 Compensation-Aware Monitoring

When monitoring is used to execute steering (reparation) code upon an anomaly detection, such code would typically be assumed to be executed on the system at the point of violation. In the case of synchronous monitoring the violation time and the detection time are always equal. However, as depicted in Figure 6.4, this is not necessarily the case for asynchronous monitoring — the system can proceed beyond an anomaly before the monitor detects the problem and stops the system. Unfortunately, reparations intended for being executed at violation-time might not necessarily be applicable at detection time *e.g.,* due to the error, some files which the reparation attempts to fix have been deleted. Thus, we enrich asynchronous monitoring with compensation handling so as to 'undo' actions which the system has performed after an error is detected. Through this technique, the detection-time system state is virtually reverted to the violation-time system state — implying that the steering code which was meant for execution at the time of the violation would still be applicable to the time of detection.

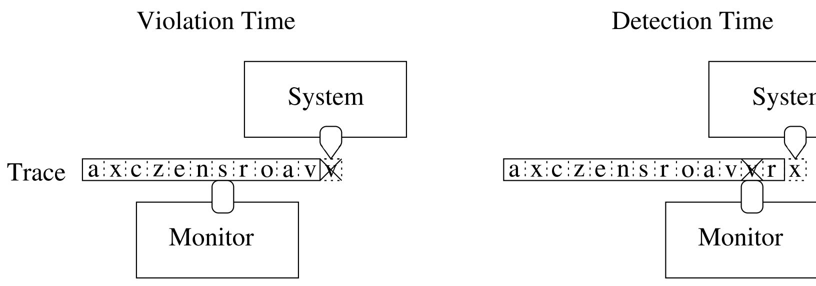


Figure 6.4: Violation time vs. detection time

Compensation-Aware Monitoring

$$\textsc{Comp}\ \frac{}{(\odot, wa, \otimes) \xrightarrow{\overline{a}}_C (\odot, w, \otimes)}$$

Figure 6.5: Semantics of compensation-aware monitoring

Definition 6.2.1. Compensation-aware monitoring semantics $\rightarrow_C$ are identical to asynchronous monitoring rules, but include an additional rule, Comp, which performs a compensating action for each action still lying in the buffer once the monitor detects an anomaly (Figure 6.5).

The set of traces generated through the compensation-aware composition of system σ and monitor m, written $\text{traces}_C(\sigma, m)$, is defined as follows:

$$\text{traces}_C(\sigma, m) = \{w \mid \exists(\sigma', m') \cdot (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', \varepsilon, m')\}$$

Sane, suspended and faulty states are defined as in synchronous and asynchronous monitoring. □

Example 6.2.1. Consider another instance of the system and monitor from the previous example where the monitor has detected a violation and the system has stopped but two actions are still in the buffer:

$$(\sigma,\varepsilon,m)\xrightarrow{b}_C(\sigma',b,m)\xrightarrow{b}_C(\sigma'',bb,m)\xrightarrow{\tau}_C(\sigma'',b,\otimes)\xrightarrow{a}_C(\sigma''',ba,\otimes)\xrightarrow{\tau}_C(\odot,ba,\otimes)$$

In order to synchronise the system with the monitor, compensation actions are executed for the actions remaining in the buffer in reverse order:

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

Proposition 6.2.1. States reachable (under synchronous, asynchronous and compensation-aware monitoring) from a sane state are themselves sane. Similarly, for suspended and faulty states.

Proof. The proof follows by string induction and rule-by-rule analysis. □

Strings accepted by compensation-aware monitoring can be shown to follow a regular pattern. Note that the pattern varies depending on whether the system has been suspended or not. Thus, the following lemma considers two patterns and shows that the monitoring outcome follows each patterns depending on the state of the monitor.

Lemma 6.2.1. For an unsuspended state (σ,ε,m), if $(\sigma,\varepsilon,m)\overset{w}{\Longrightarrow}_C(\odot,v,\otimes)$, then there exist some $w_1,w_2 \in \Sigma^*$ such that the following three properties hold: (i) $w =_x w_1w_2\overline{w}_2$; (ii) $m\overset{w_1}{\Longrightarrow}_{\text{mon}}\otimes$; (iii) $\exists\sigma''\cdot\sigma\overset{w_1vw_2}{\Longrightarrow}_{\text{sys}}\sigma''$.

Similarly, for an unsuspended state (σ,ε,m), if $(\sigma,\varepsilon,m)\overset{w}{\Longrightarrow}_C(\sigma',v,m')$ (with $\sigma'\neq\odot$), then there exists $w_1\in\Sigma^*$ such that the following three properties hold: (i) $w =_x w_1$; (ii) $m\overset{w_1}{\Longrightarrow}_{\text{mon}}m'$; (iii) $\sigma\overset{w_1v}{\Longrightarrow}_{\text{sys}}\sigma'$.

Proof. The proof of the lemma is by induction on string w.

For the base case, with $w = \varepsilon$, we consider the two possible cases separately:

- Given that $(\sigma, \varepsilon, m) \overset{\varepsilon}{\Rightarrow}_C (\odot, v, \otimes)$, it follows immediately that $\sigma = \odot$, $v = \varepsilon$ and $m = \otimes$, and all three statements follow immediately.
- Alternatively, if $(\sigma, \varepsilon, m) \overset{\varepsilon}{\Rightarrow}_C (\sigma', v, m')$, it follows immediately that $\sigma = \sigma'$, $v = \varepsilon$ and $m = m'$. By taking $w_1 = \varepsilon$, all three statements follow immediately.

Assuming the property holds for a string w, we proceed to prove that it holds for a string wa.

By analysis of the transition rules, there are four possible ways in which the final transition can be produced:

(a) Using the rule AsyncErr: $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', v, \otimes) \xrightarrow{\tau}_C (\odot, v, \otimes)$.

(b) Using the rule CompB: $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\odot, va, \otimes) \xrightarrow{\overline{a}}_C (\odot, v, \otimes)$.

(c) Using the rule Async$_S$: $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma'', v, m') \xrightarrow{a}_C (\sigma', va, m')$.

(d) Using the rule Async$_M$: $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', av, m'') \xrightarrow{\tau}_C (\sigma', v, m')$.

The proofs of the four possibilities proceed similarly:

Case (a):

$$(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', v, \otimes) \xrightarrow{\tau}_C (\odot, v, \otimes)$$

By the inductive hypothesis, it follows that there exists w_1' such that:
(i) $w =_x w_1' v$; (ii) $m \overset{w_1'}{\Longrightarrow}_{\text{mon}} \otimes$; (iii) $\sigma \overset{w_1' v}{\Longrightarrow}_{\text{sys}} \sigma'$.

We require to prove that there exist w_1 and w_2 such that:
(i) $w\tau =_x w_1 v w_2 \overline{w}_2'$; (ii) $m \overset{w_1}{\Longrightarrow}_{\text{mon}} \otimes$; (iii) $\exists \sigma'' \cdot \sigma \overset{w_1 v w_2}{\Longrightarrow}_{\text{sys}} \sigma''$.

Taking $w_1 = w_1'$ and $w_2 = \varepsilon$, statement (i) can be proved as follows:

$$
\begin{aligned}
& w\tau \\
& \quad \{ \text{ By statement } (i) \text{ of the inductive hypothesis and } =_x \} \\
=_x \;& w_1'v \\
& \quad \{ \text{ By definition of composition of strings and cancellation } \} \\
= \;& w_1'v\varepsilon\overline{\varepsilon} \\
& \quad \{ \text{ By choice of } w_1 \text{ and } w_2 \} \\
= \;& w_1vw_2\overline{w}_2
\end{aligned}
$$

Statement (ii) follows immediately from statements (ii) of the inductive hypothesis and the fact that $w_1 = w_1'$. Similarly, from statement (iii) of the inductive hypothesis, $\sigma \overset{w_1'v}{\Longrightarrow}_{\text{sys}} \sigma'$, it follows by definition of w_1 and w_2, that $\sigma \overset{w_1vw_2}{\Longrightarrow}_{\text{sys}} \sigma''$.

Case (b):

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

By the inductive hypothesis, it follows that there exist w_1' and w_2' such that: (i) $w =_x w_1'vaw_2'\overline{w}_2'$; (ii) $m \overset{w_1'}{\Longrightarrow}_{\text{mon}} \otimes$; (iii) $\exists \sigma'' \cdot \sigma \overset{w_1'vaw_2'}{\Longrightarrow}_{\text{sys}} \sigma''$.

We require to prove that there exist w_1 and w_2 such that: (i) $w\overline{a} =_x w_1vaw_2\overline{w}_2$; (ii) $m \overset{w_1}{\Longrightarrow}_{\text{mon}} \otimes$; (iii) $\exists \sigma'' \cdot \sigma \overset{w_1vw_2}{\Longrightarrow}_{\text{sys}} \sigma''$.

Taking $w_1 = w_1'$ and $w_2 = aw_2'$, statement (i) can be proved as follows:

$$\begin{array}{ll} & w\overline{a} \\ & \{ \text{By statement } (i) \text{ of the inductive hypothesis} \} \\ =_x & w_1' v a w_2' \overline{w}_2' \overline{a} \\ & \{ \text{By definition of compensation of strings} \} \\ = & w_1' v a w_2' \overline{a w_2'} \\ & \{ \text{By choice of } w_1 \text{ and } w_2 \} \\ = & w_1 v w_2 \overline{w}_2 \end{array}$$

Statement (ii) follows immediately from statements (ii) of the inductive hypothesis and the fact that $w_1 = w_1'$. Similarly, from statement (iii) of the inductive hypothesis, $\sigma \overset{w_1' v a w_2'}{\Longrightarrow}_{\text{sys}} \sigma'$, it follows by definition of w_1 and w_2, that $\sigma \overset{w_1 v w_2}{\Longrightarrow}_{\text{sys}} \sigma'$.

Case (c):

$$(\sigma, \varepsilon, m) \overset{w}{\Rightarrow}_C (\sigma'', v, m') \xrightarrow{\tau}_C (\sigma', va, m')$$

By the inductive hypothesis, it follows that there exist w_1' such that:
(i) $w =_x w_1' v$; (ii) $m \overset{w_1'}{\Longrightarrow}_{\text{mon}} m'$; (iii) $\sigma \overset{w_1' v}{\Longrightarrow}_{\text{sys}} \sigma''$.

We require to prove that there exist w_1 such that:
(i) $wa =_x w_1 va$; (ii) $m \overset{w_1}{\Longrightarrow}_{\text{mon}} m'$; (iii) $\sigma \overset{w_1 va}{\Longrightarrow}_{\text{sys}} \sigma'$.

Taking $w_1 = w_1'$, statement (i) can be proved as follows:

$$\begin{array}{ll} & wa \\ & \{ \text{By statement } (i) \text{ of the inductive hypothesis} \} \\ =_x & w_1' v a \\ & \{ \text{By choice of } w_1 \} \\ = & w_1 v a \end{array}$$

Statement (ii) follows immediately from statements (ii) of the inductive hypothesis and the fact that $w_1 = w_1'$. Similarly, from statement (iii) of the inductive hypothesis, $\sigma \overset{w_1' v}{\Longrightarrow}_{\text{sys}} \sigma''$, it follows by definition of w_1 and the application of rule $\textsc{Async}_S$, that $\sigma \overset{w_1 va}{\Longrightarrow}_{\text{sys}} \sigma'$.

Case (d):

$$(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', av, m'') \xrightarrow{a}_C (\sigma', v, m')$$

By the inductive hypothesis, it follows that there exists w_1' such that:
(i) $w =_x w_1' av$; (ii) $m \overset{w_1'}{\Longrightarrow}_{\text{mon}} m''$; (iii) $\sigma \overset{w_1' av}{\Longrightarrow}_{\text{sys}} \sigma'$.

We require to prove that there exists w_1 such that:
(i) $w\tau =_x w_1 v$; (ii) $m \overset{w_1}{\Longrightarrow}_{\text{mon}} m'$; (iii) $\sigma \overset{w_1 v}{\Longrightarrow}_{\text{sys}} \sigma'$.

Taking $w_1 = w_1' a$, statement (i) can be proved as follows:

$$\begin{array}{ll} & w\tau \\ & \{\text{ By statement } (i) \text{ of the inductive hypothesis }\} \\ =_x & w_1' av \\ & \{\text{ By choice of } w_1 \text{ }\} \\ = & w_1 v \end{array}$$

Statement (ii) follows from statement (ii) of the inductive hypothesis, the application of rule $\textsc{Async}_M$, and the fact that $w_1 = w_1' a$.

Statement (iii) follows immediately from statement (iii) of the inductive hypothesis, $\sigma \overset{w_1'}{\Longrightarrow}_{\text{sys}} \sigma'$, and the fact that $w_1 = w_1' a$.

□

We can now prove that under perfect compensations, synchronous monitoring is equivalent to compensation-aware monitoring. Informally, given that compensations cancel out with their corresponding actions, the allowed system behaviour under synchronous monitoring is equivalent to that under asynchronous monitoring with compensation execution. This result ensures that compensation triggering as defined in the semantics is sane *i.e.,* the triggered compensations precisely cancel out the system actions carried out beyond the violation — no more and no less.

Theorem 6.2.1. Given a sane system and monitor pair (σ, m), the set of traces produced by synchronous monitoring is cancellation-equivalent to the set of traces produced through compensation-aware monitoring:
$\mathsf{traces}_{\|}(\sigma, m) =_c \mathsf{traces}_C(\sigma, m)$.

Proof. To prove that $\mathsf{traces}_{\|}(\sigma, m) \subseteq_c \mathsf{traces}_C(\sigma, m)$, we note that every synchronous transition $(\sigma', m')\xrightarrow{a}_{\|}(\sigma'', m'')$, can be emulated in two or three steps by the compensation-aware transitions (three are required when the monitor fails), $(\sigma', v, m')\overset{a\tau^*}{\Longrightarrow}_C(\sigma'', v, m'')$, leaving the buffer intact:

Case $(m'' \neq \otimes)$: $(\sigma', m')\xrightarrow{a}_{\|}(\sigma'', m'')$ can be emulated by $(\sigma', \varepsilon, m')\xrightarrow{a}_C(\sigma'', a, m')$, followed by $(\sigma'', a, m')\xrightarrow{\tau}_C(\sigma'', \varepsilon, m'')$.

Otherwise: $(\sigma', m')\xrightarrow{a}_{\|}(\sigma'', \otimes)$ can be emulated by $(\sigma', \varepsilon, m')\xrightarrow{a}_C(\sigma'', a, m')$, followed by $(\sigma'', a, m')\xrightarrow{\tau}_C(\sigma'', \varepsilon, \otimes)$, followed by $(\sigma'', \varepsilon, \otimes)\xrightarrow{\tau}_C(\odot, \varepsilon, \otimes)$.

Using this fact, and induction on string w, one can show that if $(\sigma, m)\overset{w}{\Longrightarrow}_{\|}(\sigma', m')$, then $(\sigma, \varepsilon, m)\overset{v}{\Rightarrow}_C(\sigma', \varepsilon, m')$, with $w =_x v$. Hence, $\mathsf{traces}_{\|}(\sigma, m) \subseteq_c \mathsf{traces}_C(\sigma, m)$.
Proving it in the opposite direction ($\mathsf{traces}_C(\sigma, m) \subseteq_c \mathsf{traces}_{\|}(\sigma, m)$) is more intricate.
By definition, if $w \in \mathsf{traces}_C(\sigma, m)$, then $(\sigma, \varepsilon, m)\overset{w}{\Longrightarrow}_C(\sigma', \varepsilon, m')$. We separately consider the two cases of (i) $\sigma' = \odot$, and (ii) $\sigma' \neq \odot$.

- When the final state is suspended ($\sigma' = \odot$):

$$\begin{array}{rl}
& (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\odot, \varepsilon, m') \\
& \{ \text{ By sanity of initial state and Proposition 6.2.1 } \} \\
\Longrightarrow & (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\odot, \varepsilon, \otimes) \\
& \{ \text{ By Lemma 6.2.1 } \} \\
\Longrightarrow & \exists w_1, w_2 \cdot w =_x w_1 w_2 \overline{w}_2 \wedge m \overset{w_1}{\Longrightarrow}_{\text{mon}} \otimes \wedge \exists \sigma'' \cdot \sigma \overset{w_1 w_2}{\Longrightarrow}_{\text{sys}} \sigma'' \\
& \{ \text{ By Proposition 6.1.1 } \} \\
\Longrightarrow & \exists w_1, w_2 \cdot w =_x w_1 w_2 \overline{w}_2 \wedge \exists \sigma''' \cdot (\sigma, m) \overset{w_1}{\Longrightarrow}_{\|} (\sigma''', \otimes) \\
& \{ \text{ By definition of traces}_{\|} \} \\
\Longrightarrow & \exists w_1, w_2 \cdot w =_x w_1 w_2 \overline{w}_2 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m) \\
& \{ \text{ By Proposition 3.4.2 } \} \\
\Longrightarrow & \exists w_1 \cdot w =_c w_1 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m)
\end{array}$$

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

$$\begin{array}{rl}
& (\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', \varepsilon, m') \\
& \{ \text{ By Lemma 6.2.1 } \} \\
\Longrightarrow & \exists w_1 \cdot w =_x w_1 \wedge m \overset{w_1}{\Longrightarrow}_{\text{mon}} m' \wedge \sigma \overset{w_1}{\Longrightarrow}_{\text{sys}} \sigma' \\
& \{ \text{ By Proposition 6.1.1 } \} \\
\Longrightarrow & \exists w_1 \cdot w =_x w_1 \wedge (\sigma, m) \overset{w_1}{\Longrightarrow}_{\|} (\sigma', m') \\
& \{ \text{ By Definition of traces}_{\|} \} \\
\Longrightarrow & \exists w_1 \cdot w =_x w_1 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m) \\
& \{ \text{ By the alphabet of synchronous monitoring } \} \\
\Longrightarrow & \exists w_1 \cdot w =_c w_1 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma, m)
\end{array}$$

Hence, in both cases it follows that:

$$w \in \mathsf{traces}_C(\sigma, m) \implies \exists w_1 \cdot w =_c w_1 \wedge w_1 \in \mathsf{traces}_{||}(\sigma, m)$$

From which we can conclude that:

$$\mathsf{traces}_C(\sigma, m) \subseteq_c \mathsf{traces}_{||}(\sigma, m)$$

□

6.3 Compensation Scopes

The compensations we have used in the previous section undo all extra actions taken *after* an error has occurred. To avoid additional complexity arising from compensation programming, we assumed that each action has a unique compensation, independent of its context. Unfortunately, this approach is simplistic and in real-life situations an action may have different compensations depending on what occurred before or after the action *e.g.,* if an account has been closed after a transfer, then its compensation should not attempt to transfer back the funds. Enhancing the architecture to handle such scenarios is not straightforward. However, one scenario frequently occurring in compensations is that actions which form part of a transaction become locked and impossible to compensate for once the transaction is closed. For example, an online order may be split into a series of transfers of funds between accounts involving the buyer, the seller, the courier company, and possibly different banks. Failure during the transaction should lead to the previous actions to be undone. However, once the full order is processed, none of the subparts of the transaction should be undone. To address this issue, we develop an extension of compensation-aware monitoring to handle compensation-scoping.

To handle compensation scopes, we will allow the system to perform two special actions: $\llparenthesis$ to open a scope, and $\rrparenthesis$ to close the most recently opened scope. These two symbols will be considered as part of the alphabet Σ and we will assume that the system will always produce proper scope markers — at no point will it have produced more $\rrbracket$ than $\llbracket$. To handle scope opening and closing, the *cancel*$(\cdot)$ function definition is extended to drop all $\llbracket$, $\rrbracket$ symbols.

Definition 6.3.1. We say that a string over such an alphabet (including scope markers) is well-scoped if every prefix has no more close scope markers than open scope ones. A string w is said to be balanced, written *balanced*(w), if it contains an equal number of open and close scope markers, and all prefixes are well-scoped. □

Example 6.3.1. To illustrate the use of scopes with compensation-aware monitoring, we look at different system traces with errors captured on prefixes by the monitor, indicating the expected behaviour of the recovery mechanism upon error discovery:

1. If the system performed $ab\llbracket cd\rrbracket e$ (with each single letter indicating an action) by the time the monitor discovered a problem after executing action a, the compensation mechanism must compensate for $b\llbracket cd\rrbracket e$. However, the scope $\llbracket cd\rrbracket$ cannot be undone, meaning that we will compensate by performing $\overline{e}\overline{b}$.

2. If the system has, however, performed $ab\llbracket cd$ by the time the monitor discovered a problem after executing action a, the compensation mechanism will compensate for $b\llbracket cd$ by performing $\overline{d}\overline{c}\overline{b}$.

3. To look at the use of subscopes, if the system's behaviour at the point in time when the monitor discovers an error is $ab\llbracket cd\llbracket ef\rrbracket g$, the behaviour within the subscope $\llbracket ef\rrbracket$ will not be compensated for since the context is closed. However, the outer scope, which is not yet closed, will allow for compensation of actions a, b, c, d and g, depending on the point of the trace where the error is discovered.

4. Now consider a prefix trace $ab\llbracket cd\rrbracket ef\llbracket g$ of the system's behaviour the moment an error is discovered by the monitor after consuming $ab\llbracket c$. The

actions left in the buffer which have to be compensated for are $d \rrparenthesis e f \llparenthesis g$. Since the second scope has not been closed, we will compensate for efg by performing $\overline{g}\overline{f}\overline{e}$. Should action d be compensated for? If d is compensated, we may run into problems since the scope closure indicates that resources may no longer be available. The compensation for d, should thus not be triggered, since it appears within a closed scope.

Definition 6.3.2. Given a trace of actions t, we define $\mathit{strip}(t)$ to be the same trace but removing away all actions occurring within closed scopes and any remaining open scope markers. We define $\mathit{strip}(w)$ to be the shortest string for which there are no further reductions of the form:

$$\begin{array}{ll} \mathit{strip}(w_1 \llparenthesis w \rrparenthesis w_2) = \mathit{strip}(w_1 w_2) & \text{where } \mathit{balanced}(\text{w}) \\ \mathit{strip}(w_1 \llparenthesis w_2) = \mathit{strip}(w_1 w_2) & \text{where } \rrparenthesis \text{ does not appear in } w_2 \\ \mathit{strip}(w_1 \rrparenthesis w_2) = \mathit{strip}(w_2) & \text{where } \llparenthesis \text{ does not appear in } w_1 \end{array}$$

Strings w and w' are said to be *scope-cancellation-equivalent*, written $w =_{\text{sc}} w'$, if they reduce via compensation cancellation and scope stripping to the same string: $\mathit{strip}(w) =_c \mathit{strip}(w')$. As before, we define what it means for a set of strings to be *included in set W' up-to-scope-cancellation*, written $W \subseteq_{\text{sc}} W'$, and set *equality up-to-scope-cancellation*, written $W =_{\text{sc}} W'$. □

Scope stripping is well-defined and cancels with compensation:

Proposition 6.3.1. Scope stripping *strip* is a well-defined function over the domain of well-scoped strings. Furthermore, $w\, \overline{\mathit{strip}(w)} =_{\text{sc}} \varepsilon$.

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

To handle scopes in compensations, we adopt the addition of three rules (given in Figure 6.6) to the compensation-aware monitoring semantics. In the first two cases, whenever a scope closure $\rrparenthesis$ is found in the buffer, the whole scope is removed before proceeding (considering separately whether or not the scope was opened before the error was discovered). In the third case, whenever a scope open symbol $\llparenthesis$ is found, it is simply discarded, since it represents a scope which was opened but not closed by the time the error was identified.

$$\textsc{CloseScope}_M \; \frac{}{(\odot, w\rrparenthesis, \otimes) \xrightarrow{\tau}_{SC} (\odot, w', \otimes)} \; w\rrparenthesis = w' \llparenthesis w'' \rrparenthesis \text{ WITH } \mathit{balanced}(w'')$$

$$\textsc{CloseScope} \; \frac{}{(\odot, w\rrparenthesis, \otimes) \xrightarrow{\tau}_{SC} (\odot, \varepsilon, \otimes)} \; \llparenthesis \text{ DOES NOT APPEAR IN } w$$

$$\textsc{OpenScope} \; \frac{}{(\odot, w\llparenthesis, \otimes) \xrightarrow{\tau}_{SC} (\odot, w, \otimes)}$$

Figure 6.6: Semantics of scoped monitoring

The state sanity-preservation result of Proposition 6.2.1, also holds for scope compensation-aware monitoring.

Proposition 6.3.2. States reachable through scope compensation-aware monitoring from a sane state are themselves sane. Similarly, for suspended and faulty states.

To prove that the modified system is still correct, we need a stronger version of Lemma 6.2.1, which caters for complete contexts which will be discarded when compensations are triggered:

Lemma 6.3.1. For an unsuspended state (σ, ε, m), if $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_{SC} (\odot, v, \otimes)$, then there exist some $w_1, w_2 \in \Sigma^*$ such that the following three properties hold:
(i) $w =_x w_1 v w_2 \overline{strip(w_2)}$ with $\mathit{balanced}(()w_2)$; (ii) $m \overset{w_1}{\Longrightarrow}_{\text{mon}} \otimes$;
(iii) $\exists \sigma'', v' \cdot \sigma \overset{w_1 v' w_2}{\Longrightarrow}_{\text{sys}} \sigma'' \wedge v = strip^+(v')$ ($strip^+$ signifies one or more applications of $strip$).

Similarly, for an unsuspended state (σ, ε, m), if $(\sigma, \varepsilon, m) \overset{w}{\Longrightarrow}_C (\sigma', v, m')$ (with $\sigma' \neq \odot$), then there exists $w_1 \in \Sigma^*$ such that the following three properties hold: (i) $w =_x w_1 v$; (ii) $m \overset{w_1}{\Longrightarrow}_{\text{mon}} m'$; (iii) $\sigma \overset{w_1 v}{\Longrightarrow}_{\text{sys}} \sigma'$.

Proof. The proof of Lemma 6.3.1 is almost identical to that of Lemma 6.2.1, but taking into account the rules CloseScope$_M$, CloseScope, and OpenScope.

Case rule CloseScope$_M$: $(\sigma,\varepsilon,m)\overset{w}{\Longrightarrow}_C(\odot,v'\rrbracket,\otimes)\xrightarrow{\tau}_C(\odot,v,\otimes)$

where $v'\rrbracket = v\llbracket v''\rrbracket$ with $balanced(v'')$

By the inductive hypothesis, it follows that there exist w'_1 and w'_2 such that:
$(i)\ \exists v_1\cdot v'\rrbracket = strip^+(v_1)$. $(ii)\ w =_x w'_1v_1\rrbracket w'_2\overline{strip(w'_2)}$;
$(iii)\ m\overset{w'_1}{\Longrightarrow}_{\text{mon}}\otimes$; $(iv)\ \exists\sigma''\cdot\sigma\overset{w'_1v_1w'_2}{\Longrightarrow}_{\text{sys}}\sigma''$.

We require to prove that there exist w_1 and w_2 such that:
$(i)\ \exists v_2\cdot v = strip^+(v_2)$. $(ii)\ w\overline{a} =_x w_1v_2w_2\overline{strip(w_2)}$;
$(iii)\ m\overset{w_1}{\Longrightarrow}_{\text{mon}}\otimes$; $(iv)\ \exists\sigma''\cdot\sigma\overset{w_1v_2w_2}{\Longrightarrow}_{\text{sys}}\sigma''$.

Taking $v_2 = v_1$, statement (i) can be proved as follows:

$$\begin{aligned}
&\{\text{ By rule CloseScope}_M\text{ and first line in the definition of } strip\ \}\\
v =\ & strip^+(v'\rrbracket)\\
&\{\text{ By inductive hypothesis }\}\\
=\ & strip^+(strip^+(v_1))\\
&\{\text{ By definition of } strip^+ \text{ and choice of } v_2\ \}\\
=\ & strip^+(v_2)
\end{aligned}$$

Taking $w_1 = w'_1$ and $w_2 = w'_2$, statement (ii) can be proved as follows:

$$\begin{aligned}
& w\tau\\
&\{\text{ By statement } (ii) \text{ of the inductive hypothesis }\}\\
=_x\ & w'_1v_1w'_2\overline{strip(w'_2)}\\
&\{\text{ By choice of } w_1, v_2, \text{ and } w_2\ \}\\
=\ & w_1v_2w_2\overline{strip(w_2)}
\end{aligned}$$

Statement (iii) follows immediately from statements (iii) of the inductive hypothesis and the fact that $w_1 = w'_1$. Statement (iv) follows from statements (iv) of the inductive hypothesis and choice of w_1, w_2, and v_2.

Case rule CloseScope: Same as previous case but using the second line in the definition of *strip*.

Case rule OpenScope: Same as first case but using the third line in the definition of *strip*.

□

This allows us to prove the stronger theorem stating the correctness of scoped compensation-aware monitoring, *i.e.,* assuming scoped actions do not need to be compensated and can be ignored, all non-scoped actions are correctly compensated:

Theorem 6.3.1. Synchronous and compensation-aware monitoring with scopes behave in an equivalent manner, $\mathsf{traces}_{\|}(\sigma,m) =_{\mathrm{sc}} \mathsf{traces}_{\mathrm{SC}}(\sigma,m)$. Given a sane system and monitor pair (σ,m):

(i) A trace accepted by synchronous monitoring is also accepted by compensation-aware monitoring with scopes: $\mathsf{traces}_{\|}(\sigma,m) \subseteq_{\mathrm{sc}} \mathsf{traces}_{\mathrm{SC}}(\sigma,m)$.

(ii) A trace accepted by compensation-aware monitoring with scopes can be split into two parts, the first of which is accepted by synchronous monitoring, and the second of which is scope cancellation-equivalent to doing nothing:

$$w \in \mathsf{traces}_{\mathrm{SC}}(\sigma,m) \implies \exists w_1, w_2 \cdot w = w_1 w_2 \wedge w_1 \in \mathsf{traces}_{\|}(\sigma,m) \wedge w_2 =_{\mathrm{sc}} \varepsilon$$

Proof. The correctness of the first part (i) follows from the fact that every synchronous transition can be emulated by two or three scoped compensation-aware rules (as shown in Theorem 6.2.1). This ensures forward language inclusion.

As in the proof of Theorem 6.2.1, the proof of (ii) for $(\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{\mathrm{SC}} (\sigma,\varepsilon,m')$ takes into consideration two cases: (a) $\sigma' = \odot$; and (b) $\sigma' \neq \odot$. Case (b) is identical to the proof of the equivalent case in Theorem 6.2.1. Case (a) can be proved as follows:

$$
\begin{array}{ll}
 & (\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{\mathrm{SC}} (\odot,\varepsilon,m') \\
 & \{ \text{ By sanity of initial state and Proposition 6.3.2 } \} \\
\Longrightarrow & (\sigma,\varepsilon,m) \overset{w}{\Longrightarrow}_{\mathrm{SC}} (\odot,\varepsilon,\otimes) \\
 & \{ \text{ By Lemma 6.3.1 } \} \\
\Longrightarrow & \exists w_1, w_1' \cdot w =_x w_1 w_1' \overline{strip(w_1')} \wedge m \overset{w_1}{\Longrightarrow}_{\mathrm{mon}} \otimes \wedge \exists \sigma'' \cdot \sigma \overset{w_1 w_1'}{\Longrightarrow}_{\mathrm{sys}} \sigma'' \\
 & \{ \text{ By Proposition 6.1.1 } \} \\
\Longrightarrow & \exists w_1, w_1' \cdot w =_x w_1 w_1' \overline{strip(w_1')} \wedge \exists \sigma''' \cdot (\sigma,m) \overset{w_1}{\Longrightarrow}_{||} (\sigma''',\otimes) \\
 & \{ \text{ By definition of } \mathsf{traces}_{||} \} \\
\Longrightarrow & \exists w_1, w_1' \cdot w =_x w_1 w_1' \overline{strip(w_1')} \wedge w_1 \in \mathsf{traces}_{||}(\sigma,m) \\
 & \{ \text{ Adding variable } w_2 = w_1' \overline{strip(w_1)} \} \\
\Longrightarrow & \exists w_1, w_1', w_2 \cdot w =_x w_1 w_2 \wedge w_2 = w_1' \overline{strip(w_1')} \wedge w_1 \in \mathsf{traces}_{||}(\sigma,m) \\
 & \{ \text{ By Proposition 6.3.1 } \} \\
\Longrightarrow & \exists w_1, w_2 \cdot w =_x w_1 w_2 \wedge w_2 =_{\mathrm{sc}} \varepsilon \wedge w_1 \in \mathsf{traces}_{||}(\sigma,m)
\end{array}
$$

This completes the proof. □

Using this result, we can show that scoping still keeps monitoring correct up to ignoring scope content and compensations. The semantics given to scope compensation monitoring gather the scopes in the buffer and only discards them while emptying the buffer. The advantage of this approach, is that the decision of how to handle scopes is left until the compensation triggering phase. Alternatively, one could have discarded actions in the buffer as soon as the system closes a scope, which is less flexible, but may result in smaller buffers being used.

6.4 Desynchronisation and Resynchronisation

Despite compensation-awareness, in some systems it may be desirable to run monitoring synchronously with the system for operations considered risky, only to desynchronise the system from the monitor again once control leaves the risky operation. Intuitively similar to pausing and unpausing the system (as alluded in the introduction of the chapter through the cassette deck metaphor), synchronisation and desynchronisation enable the system-monitor relationship to be more finely controlled. In this section, we investigate a monitoring strategy which can run both synchronously or asynchronously in a non-deterministic manner. Any heuristic used to decide when to switch between modes corresponds to a refinement of this approach.

Adaptive Monitoring

$$\textsc{ReSync}\ \frac{}{(\sigma,\varepsilon,m)\xrightarrow{\tau}_A(\sigma,m)} \qquad \textsc{DeSync}\ \frac{}{(\sigma,m)\xrightarrow{\tau}_A(\sigma,\varepsilon,m)}$$

Figure 6.7: Semantics of adaptive monitoring

Definition 6.4.1. The adaptive monitoring of a system, is defined in terms of the synchronous and asynchronous monitoring rules and two additional ones (given in Figure 6.7). Rule ReSync allows the system to synchronise once the buffer is empty, while rule DeSync allows the monitor to be released asynchronously. By also including the compensation rule Comp, we obtain adaptive compensation-aware monitoring ($\rightarrow_{AC}$) and by further including scoping rules (Figure 6.6), we obtain adaptive compensation-aware scoped monitoring ($\rightarrow_{ASC}$).

The set of traces generated through the adaptive composition of system σ and monitor m, written $\text{traces}_A(\sigma,m)$, is defined as follows:

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

The traces for compensation-aware adaptive composition, denoted

$\mathsf{traces}_{\mathrm{AC}}(\sigma, m)$, can be similarly defined. □

Since adaptive monitoring can be considered as a special case of asynchronous monitoring (having a zero-sized buffer at particular times), then we can show that asynchronous monitoring is equivalent to adaptive monitoring in the traces which can be generated.

Theorem 6.4.1. Asynchronous and adaptive monitoring are observationally indistinguishable: $\mathsf{traces}_A(\sigma, m) =_x \mathsf{traces}_{|||}(\sigma, m)$.

Proof. Proving that $\mathsf{traces}_{|||}(\sigma, m) \subseteq_x \mathsf{traces}_A(\sigma, m)$ is trivial since all the rules which can be used to generate traces in $\mathsf{traces}_{|||}(\sigma, m)$ are also available for traces in $\mathsf{traces}_A(\sigma, m)$.

Proving that $\mathsf{traces}_A(\sigma, m) \subseteq_x \mathsf{traces}_{|||}(\sigma, m)$ is also easy and can be done by showing that both ReSync and DeSync do not affect traces. In fact both rules either introduce or consume an empty buffer while adding a τ to the trace — all actions which clearly leave no effect on traces.

□

Similar to the reasoning behind the previous theorem, compensation-aware monitoring exhibits the same event sequence whether it is adaptive or not.

Theorem 6.4.2. Compensation-aware adaptive monitoring is also indistinguishable from compensation-aware monitoring up to traces:
$\mathsf{traces}_{\mathrm{AC}}(\sigma, m) =_x \mathsf{traces}_{C}(\sigma, m)$.

Proof. The proof is similar to that of the previous theorem. □

An immediate corollary of these results is that compensation-aware adaptive monitoring is cancellation-equivalent to synchronous monitoring.

Corollary 6.4.1. Compensation-aware adaptive monitoring is indistinguishable from synchronous monitoring up to traces: $\mathsf{traces}_{\mathrm{AC}}(\sigma, m) =_x \mathsf{traces}_{||}(\sigma, m)$.

Proof. The proof follows immediately from Theorem 6.4.2 and Theorem 6.2.1. □

While we do not give the counterpart theorems about scoped monitoring, these can be similarly defined as above concluding that adaptive compensation-aware scoped monitoring is equal up-to-scope-cancellation to synchronous monitoring: $\mathsf{traces}_{\mathrm{ASC}}(\sigma, m) =_{\mathrm{sc}} \mathsf{traces}_{||}(\sigma, m)$.

It is important to note that the results hold about trace equivalence, *i.e.,* observational equivalence. In the case of adaptive monitoring, we are increasing the set of diverging configurations since every state can diverge through repeatedly desynchronising and resynchronising. One would be required to enforce fairness constraints on desynchronising and resynchronising rules to ensure achieving progress in the monitored systems.

6.5 A Compensation-Aware Monitoring Architecture

Based on the theory given in the previous sections, we propose an asynchronous compensation-aware monitoring architecture and implementation, cLarva, with a controlled synchronous element. The system exposes two interfaces to the monitor: (i) an interface for the monitor to communicate the fact that a problem has been detected and the system should stop; and (ii) an interface for the monitor to indicate which actions should be compensated for. Note that these correspond precisely to rules AsyncErr and Comp respectively. Furthermore, the actual time of stopping and how the indicated actions are compensated for are decisions left up to the system.

Figure 6.8 shows the four components of cLarva and the communication links between them. The monitor receives system events through the events player from the log, while the system can continue unhindered. If the monitor detects a fault, it communicates with the system so that the latter stops. Depending on the actions the system carried out since the actual occurrence of the fault, the monitor indicates the actions to be compensated for.

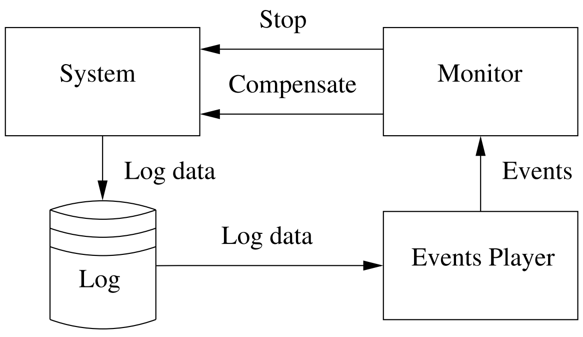


Figure 6.8: The asynchronous architecture with compensations cLarva

To support switching between synchronous and asynchronous monitoring, a *synchronisation manager* component is added as shown in Figure 6.10. All connectors in the diagram are synchronous with the system not proceeding after relaying an event until it receives control from the manager. Figure 6.9 shows the logic of the synchronisation manager which supports two modes of operation encoded in the *if* statement: (i) the first clause deals with the synchronous mode and returns control to the system only after the monitor has processed the event; (ii) the asynchronous mode is programmed in two parallel parts: one which returns control immediately back to the system upon receiving an event (storing it in the buffer), and the other which enables the monitor to consume events from the buffer.

A challenging aspect of the architecture is the stopping of the system when the monitor detects a violation. In real-life scenarios it is usually undesirable to stop a whole system if an error is found as this is considered highly intrusive. However, in many cases it is not difficult to delineate components of the system to ensure that only the relevant parts of the system are stopped. For example, when a transaction is carried out without necessary rights, only it should be stopped and compensated for. Similarly, if a user has managed to illegally login

```
c = PROCEED                 ; set default control to proceed
while (c != STOP)
  if (monitoring_mode == SYNC)
    e = in_event()          ; read event from system
    c = out_event(e)        ; forward to monitor and get its resulting state
    out_control(c)          ; relay control to system
  else
    par                     ; parallel execution
      e1 = in_event()       ; read from system
      addToBuffer(e1)       ; store in buffer
      out_control(c)        ; return control to system
    with
      e2 = readFromBuffer() ; read from buffer
      c = out_event(e2)     ; forward to monitor and get its resulting state
end
```

Figure 6.9: The pseudo code representing the monitoring manager

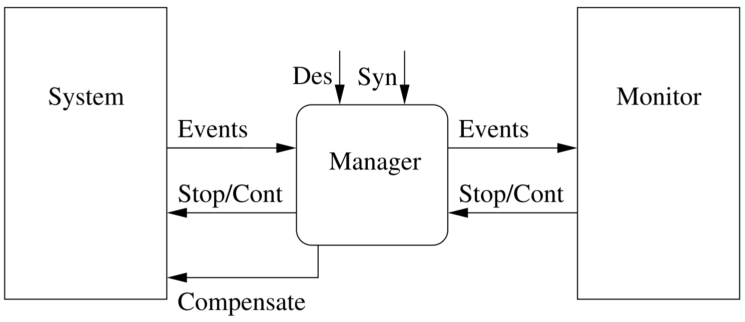


Figure 6.10: The asynchronous architecture with synchronisation and desynchronisation controls

and start a session, then only that user's operations during that session should be stopped and compensated for.

This approach of system decomposition into relatively independent parts can be extended further to simultaneously allow synchronous and asynchronous monitoring. This is further discussed in the next subsections.

6.5.1 Automating the Decision of Monitor Mode Switching

Synchronisation guarantees immediate identification and possible reparation of problems, making it desirable for parts of the system where higher dependability is required. For instance, if a particular transaction is considered high-risk, it would be desirable to synchronise monitoring during the transaction, only to desynchronise once again when a less risky part of the system is reached. Having an architecture which allows switching between synchronous and asynchronous modes of monitoring requires a mechanism to appropriately select the active mode. Although the switching between synchronous and asynchronous monitoring can be done manually, it is much more useful to have an automatic mechanism which handles this feature.

The first issue is how to assess risk associated with particular states and actions, thus ensuring that high risk actions are always monitored synchronously. There are two main ways in which this can be achieved: statically or dynamically. For example, transactions can be statically classified according to the risk they involve, *e.g.,* a transfer between a user's own accounts might be considered as safe but spending a large sum of money might not. On the other hand, one can dynamically keep track of the activities of each user and use pattern matching and statistics to deduce the risks associated with individual users. If a transaction has an associated high risk factor and/or is being carried out by a user with high associated risk, then one might decide that during this action the monitor should switch to synchronous mode.

Another important design issue is *where* to decide de/synchronisation: either within the system itself, or as part of the monitor. Leaving the decision up to the system has the advantage that the system would always be in control of the monitoring mode and the decision can be taken synchronously. On the other hand, this would add an overhead to the system; something which the whole architecture is meant to avoid.

In our case study we opt for a dynamic, monitor-side, asynchronous de/synchronisation decision where the heuristics are themselves implemented as monitors. This strategy avoids any duplication between monitors and heuristics while also avoiding the introduction of additional overheads to the system. Although this might lead to a de/synchronisation decision to be taken late, the problem is minimised by the scheduling strategy discussed in Section 6.5.2. Therefore, updating the architectural view of the system would involve adding a heuristics component (to the architecture shown in Figure 6.10) which is in charge of executing heuristics and signalling the manager to switch between synchronous and asynchronous monitoring. Such a component requires the following connections: (*i*) a connection to the incoming system events — supplying the required information for executing heuristics; and (*ii*) connections to the de/synchronisation signals entering the synchronisation manager. The updated architecture with these modifications is shown in Figure 6.11.

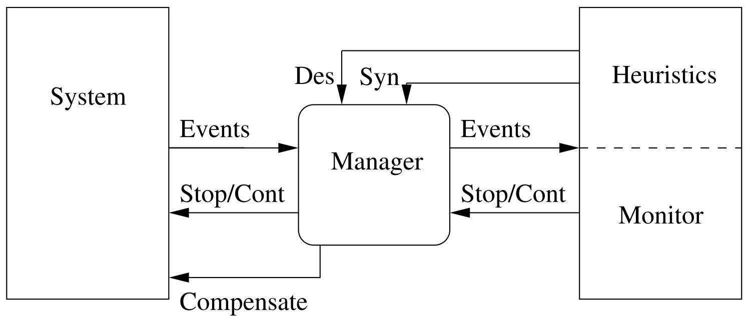


Figure 6.11: The monitoring architecture with heuristics

In practice, this approach requires that the monitoring system can handle parametrised monitoring of transactions/users. Furthermore, it requires a way of decomposing the system into independent components which would simultaneously allow synchronous and asynchronous monitoring. For this reason the monitor-side (including the manager) typically consists of multiple parametrised monitors, each with its own buffer, synchronisation manager and heuristics component. This is expanded further in the following subsection.

6.5.2 Monitor Demultiplexing and Scheduling

Monitors in cLarva are dynamically instantiated for each monitored object. Thus, a system's monitor is in fact composed of many sub-monitors. For example if we are monitoring a number of properties regarding a number of transactions for each logged-in user, then a monitor would be created for each property, for each transaction, for each user. For this reason, although at a high level we have shown the architecture as having one manager, in actual fact it has a buffer for each sub-monitor as illustrated in Figure 6.12.

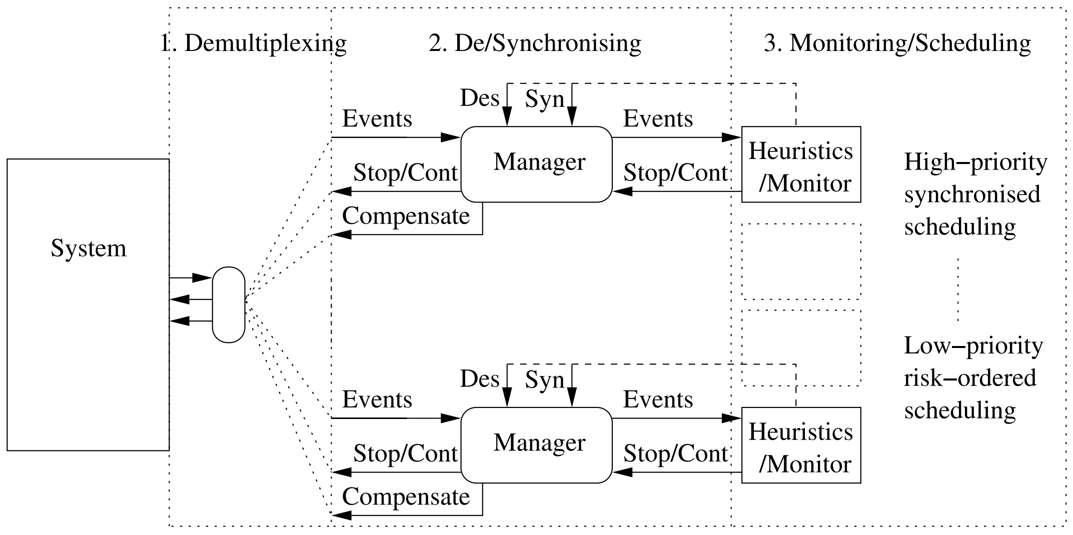


Figure 6.12: The monitoring architecture with heuristics, demultiplexing, and scheduling

To coordinate the execution of the sub-monitors, the following steps are carried out every time a system event is received:

1. The event is replicated to all the relevant sub-monitor buffers. For example, a transaction event would be copied to all buffers pertaining to sub-monitors of that particular transaction.

2. Subsequently, if the sub-monitor is in asynchronous mode, control is immediately passed back to the system. Otherwise the manager first forwards the event to the heuristics and monitoring components, and waits for their response before allowing the system to proceed further.

3. Given the potentially substantial number of sub-monitors, the choice of a scheduling strategy among sub-monitors might be crucial to detect problems as early as possible. A sensible scheduling strategy would be to asso-

ciate a scheduling priority according to the corresponding risk; the higher the risk, the higher the scheduling priority. Naturally, this is over and above the priority that synchronous monitors should have over asynchronous monitors; to ensure minimal disruption to the system — asynchronous monitors should only be allowed to run when no synchronous monitors are running.

After having presented the salient aspects of the monitoring architecture, in what follows, we give an account of how this has been applied to an industrial case study.

6.6 Industrial Case Study

We have applied cLarva on Entropay, an online prepaid payment service offered by Ixaris Systems Ltd[1]. Entropay users deposit funds through funding instruments (such as their own personal credit card or through a bank transfer mechanism) and spend such funds through spending instruments (such as a virtual VISA card or a Plastic Mastercard). The service is used worldwide and thousands of transactions are processed on a daily basis.

The case study implementation closely follows the architecture described in the previous sections including property specifications, compensations for synchronisation, and heuristics for deciding between synchrony and asynchrony. The following subsections focus on these aspects respectively, explaining how each was applied to the case study. Subsequently, the final subsection (Section 6.6.4) concludes by giving information about the monitoring performance and a number of monitoring traces which represent the monitoring behaviour attained.

[1] `http://www.ixaris.com`

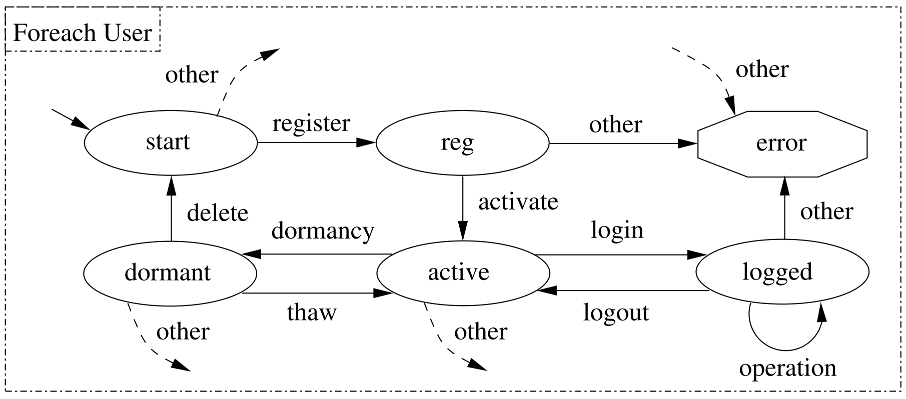


Figure 6.13: The lifecycle property

6.6.1 Specifying Properties for Monitoring Entropay

When monitoring Entropay, we chose to focus on high-level business process properties mainly because these sort of properties typically span over several modules and are therefore difficult to test for. In what follows, we give a classification of properties which were monitored successfully and how compensations were managed in case of a violation detection.

Life cycle A lot of properties in Entropay depend on which phase of the life-cycle an entity is in. Figure 6.13 is an illustration of the user life-cycle, starting with registration and activation, allowing the user to login and logout (possibly carrying out a series of operations in between), and finally, the possibility of putting a user account in and out of dormancy — a state where the user cannot perform financial transactions due to a long period of inactivity.

Implicitly, such a property checks that for a user to perform a particular operation and reach a particular state, the user must be in an appropriate state. Note that the property is monitored for each user.

Another more complex example focusing on a part of the life cycle involves the dormancy feature of Entropay which manages inactive user accounts. If a user account has been inactive for six months, then for security reasons

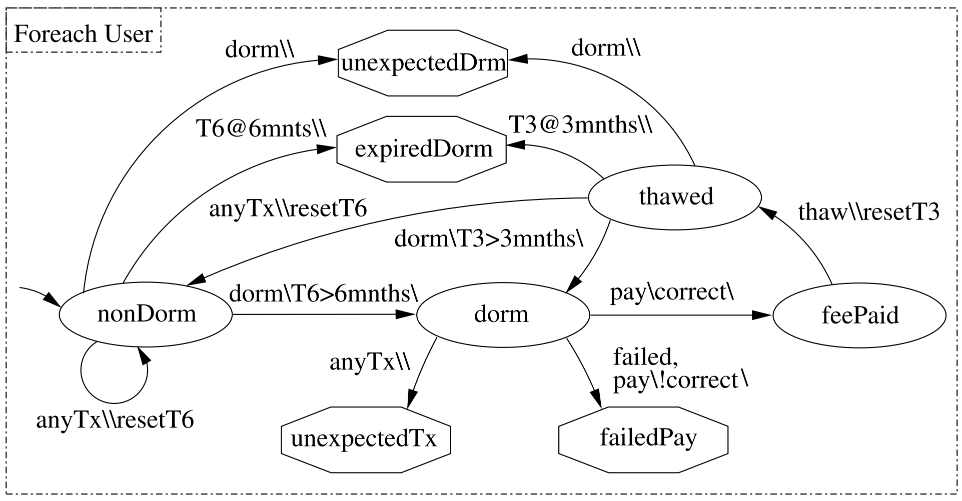


Figure 6.14: The dormancy property

the account is put to *dormant, i.e.,* a state where the account cannot be used unless a permission is requested by the legitimate user. If such a permission is requested and an administration fee is paid, then the account is thawed but put to dormant again if the account still remains unused for another three months. The property is depicted in Figure 6.14. Once again this property is monitored for each user.

Real-time Several properties in Entropay such as the dormancy property just described, have a real-time element: *something* should happen after six months, and then after three months, etc. To model this property in terms of DATEs we have used timers which trigger after a particular period of time elapses since they have been reset (represented in Figure 6.14 as *T6@6mnths*, and *T3@3mnths* respectively) to ensure that a violation of the property is immediately detected[2]. On the other hand, timer checks are conducted to ensure that the relevant actions do not occur earlier than expected. For example if the dormancy process occurs, it must be ensured that enough time has elapsed since the last timer reset (represented by the conditions *T6>6mnths* and *T3>3mnths*).

[2]In practice the timers are set to trigger slightly later to provide some leeway.

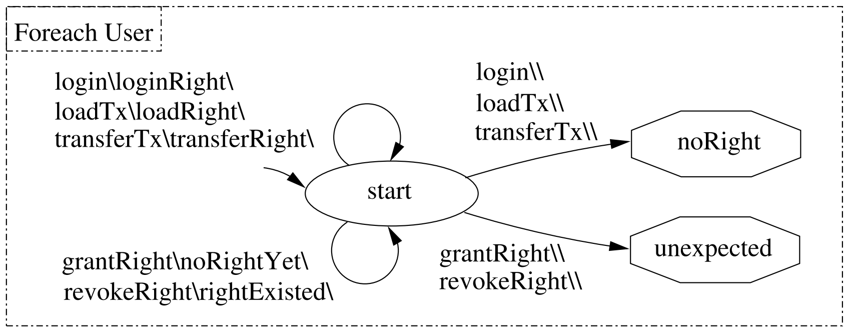


Figure 6.15: The property monitoring rights

Rights User rights are a very important aspect of Entropay's security system. A number of transactions require the user to have the appropriate rights before they are permitted. These properties are monitored by keeping track of rights granting and withdrawal so that the monitor effectively maintains a database of the rights each user holds at any particular time. Consequently, the monitor also tracks each right-requiring action and ensures that the user involved has the corresponding rights. For example Figure 6.15 depicts the rights-checking property for a user login, loading of money onto virtual credit cards, and money transfers. Note that the property has two bad states: one which is reached if the user attempts a transaction without having the right, and another which is reached if a right is granted to a user already having the right or a right is revoked from a user not having the right. The same concept is applied to other operations requiring the user to have particular rights.

Amounts There are various limits (for security reasons) on the frequency of certain transactions and the total amount of money which these transactions constitute. The general idea is depicted in Figure 6.16 where a user performing a *load* operation is checked to ensure that the allowed amount of money which can be spent within a particular time window is adhered to. Note that the administrators of the system may update the applicable limits and different limits apply depending on the user status which may also change during the user life cycle.

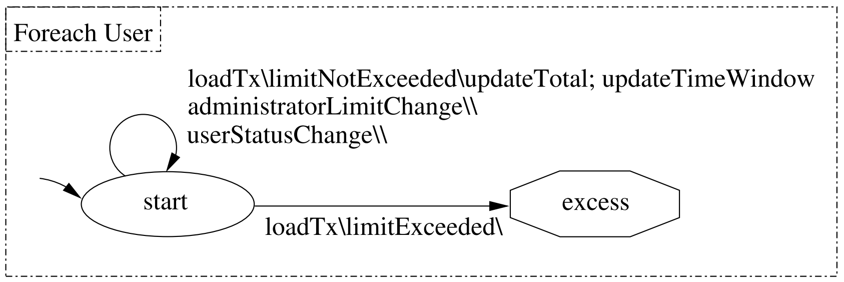

Foreach User
loadTx\limitNotExceeded\updateTotal; updateTimeWindow
administratorLimitChange\\
userStatusChange\\
start
excess
loadTx\limitExceeded\

Figure 6.16: The property monitoring transaction amounts

Although the Entropay properties are relatively light-weight, due to security and performance considerations, it is not desirable to run the monitor synchronously when there is no clear evidence of a potential violation. Users which pose little or no risk to the system and which have been using the system for a number of years should not suffer any service deterioration. The conciliatory approach of cLarva would guarantee added security with the cost of logging under normal execution while incurring overhead only when there is convincing evidence that something is wrong.

6.6.2 Specifying Compensations for the Case Study

In order to support asynchronous monitoring, system actions which significantly modify the state of the system have to be associate to a compensating action which logically undoes the action. The first two columns of Table 6.1 show a number of actions which feature in the Entropay case study together with their corresponding compensations. However, by the time the monitor detects a problem the compensations of the buffered events may no longer be applicable. For example, while the compensation of a login action is a logout, if the user had already performed a logout by the time the monitor attempts to compensate for the login, then it no longer makes sense to perform the logout. Similarly, some actions such as a purchase cannot be cancelled beyond a certain time bound after their occurrence. For simplicity we assume that the time

Action	**Compensation**	**Scope ends**
activate account	deactivate account	deactivate account
login	logout	logout
create virtual credit card	disable virtual credit card	disable/delete virtual credit card
load money onto virtual credit card	withdraw money from virtual credit card	1 hour since load, or money withdrawn
transfer money among virtual credit cards	reverse transfer among virtual credit cards	1 hour since transfer, or money removed from card
purchase with virtual credit card	cancel purchase and return money to virtual credit card	1 hour since purchase, or purchase cancelled

Table 6.1: Actions and their compensations

limit always occurs one hour after the completion of the activity[3]. This aspect is shown in the third column of Table 6.1.

6.6.3 Implementing Heuristics as DATEs

In practice, one expects there to be a substantial overlap between monitors and heuristics. For example to monitor for fraudulent behaviour, one would usually try to measure how much risk is associated with a particular pattern of activities. Such a measure can lend itself useful to decide for or against synchronous monitoring. In this subsection we will show how DATEs can be used for this purpose.

The following features of DATEs are particularly useful for implementing heuristics: (i) it is easy to integrate such heuristics with cLarva which is DATE-based; (ii) DATEs are compositional — different heuristics can be implemented as separate DATEs and then connected together to form a single DATE through

[3]This approach to scoping might seem at odds with the theory presented earlier where scopes cannot intersect (except by inclusion). However, in practice, cLarva would have a monitor for each transaction and thus the scopes would never intersect locally.

channel communication; (*iii*) LarvaStat [37] builds upon Larva, augmenting DATEs with support for statistical properties[4]; and (*iv*) it is easy to keep track of multiple objects at a time through the dynamic mechanism which replicates monitors — one for each object being monitored.

Using these features and considering our case study we opt to implement the following heuristics for each user: (*i*) the monitor uses statistics to calculate the risk factor depending on the series of activities which the user performs; and (*ii*) if the risk factor exceeds a particular threshold, then the monitor is forced to synchronise before the end of a scope (based on Table 6.1).

Thus, we will split the implementation of the heuristics into the following parts: (*i*) a DATE which keeps track of whether a user is currently monitored synchronously or asynchronously; (*ii*) a DATE which keeps track of the risk factor of a user; and (*iii*) a DATE which upon detecting a close-scope event decides whether a user should be monitored synchronously or asynchronously and communicates the decision to (*i*). In what follows, we give the definition of these DATEs:

1. Figure 6.17 shows the main DATE which listens on two channels: *sync*, signifying that the monitor should be synchronised, and *async* to signal that the monitor need no longer remain synchronised.

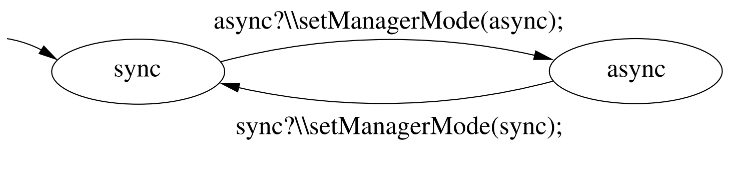


Figure 6.17: The DATE which listens to other DATEs for messages to synchronise or desynchronise the monitor from the system

[4]One of the case studies carried out with LarvaStat involved an intrusion detection system on top of an ftp server, assessing each user, and assigning him or her a risk factor. The risk factor was calculated using two main techniques: (*i*) a Markov chain analysing the user's command sequence, with each ftp command being related to a risk factor, and marking the user as suspicious if the command sequence exceeds a threshold; and (*ii*) the use of statistical moments for the characterisation of abnormal user behaviour, monitoring each user's download and upload behaviour patterns, and assuming a statistically predictable pattern.

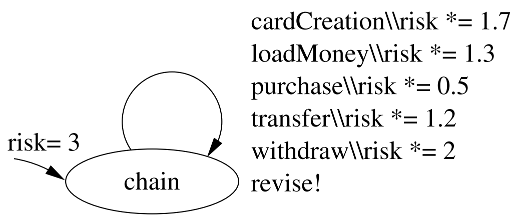


Figure 6.18: The DATE which tracks the user risk factor and notifies of a risk change over channel *revise*

As soon as the DATE receives either of these messages from other monitors, it relays the change to the synchronisation manager. Note that since we would like to apply the heuristics on a per-user basis, the DATE has to be parametrised for each user. The limitation of this approach is that properties which span over multiple users have to be very carefully devised as the monitor states of different users might reflect different synchronisation levels. For this reason communication across DATEs should only occur through channel communication and not through global variables.

2. Another DATE can be useful to measure the perceived user risk factor: for example users who use the money for a purchase are not considered as risky, while users who load money a number of consecutive times, perform several transfers and withdraw the money are considered highly suspicious. This logic is encoded as a Markov chain shown in Figure 6.18.

3. Figure 6.19 illustrates the DATE which would force a monitor to synchronise (by sending a signal to the main DATE (Figure 6.17)) in case a closing scope action is detected and the risk factor of the corresponding user is higher than the threshold. On the other hand, when the risk goes below the threshold, the monitor is desynchronised from the system. Note that new users are considered risky and thus their risk factor is initialised to a higher value than the threshold. Such users are only considered safe after carrying out a pattern of risk-free transactions.

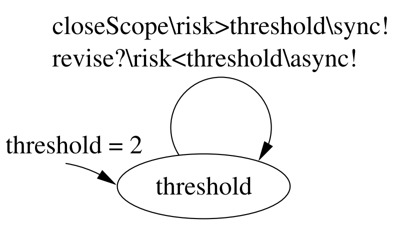

closeScope\risk>threshold\sync!
revise?\risk<threshold\async!
threshold = 2
threshold

Figure 6.19: The DATE which decides whether to synchronise or desynchronise

6.6.4 Monitoring Results

The case study was successfully executed on a sanitized[5] database of 300,000 users with around a million virtual cards. A number of issues have been detected through the monitoring system: (*i*) not all system activities were recorded consistently; (*ii*) some system state was found to be inconsistent, *e.g.,* certain cards which were marked as inactive were still found to be active; (*iii*) in some exceptional cases, system limits did not tally with the overall balance of the transactions monitored.

To demonstrate the results of our case study, in this subsection we give three anonymised[6] system traces in which problems were discovered.

In the following traces, we use ⧁ to indicate an action which is later compensated with an action tagged with ⧀ (actions and their corresponding compensations have been specified in Table 6.1). Also recall that the scope of compensations expires within an hour.

The monitoring heuristics used for generating these traces initialise each monitor with a risk factor of 3 and assumes the threshold 2 when deciding between synchrony and asynchrony (as specified in Figure 6.18).

No rights issue After encountering traces such as the one presented below, the monitor reported that some actions were carried out without the necessary rights. For example a user requires a special right to be allowed to login, to create a virtual credit card and also to load money onto the credit card.

[5]User information was obfuscated for the purpose of this study.
[6]Due to privacy considerations the data in certain fields cannot be exposed.

Figure 6.20 shows excerpts from the system log merged with monitor actions where a user performed actions without the required rights. In this case the monitor was run asynchronously with a high priority scheduling. Once the monitor detected the violation (late), the transactions which occurred after the illegal login were compensated (see last four entries of Figure 6.20).

Timestamp	User	Transaction	$
13:00:35 5-5-2010	user1	register account	n/a
a monitor is dynamically created for user1 with default risk factor 3			
13:05:41 5-5-2010	user1	⧁activate account	n/a
13:05:45 5-5-2010	user1	⧁log in	n/a
user1 logged in without having the right			
13:07:10 5-5-2010	user1	⧁create virtual credit card	n/a
risk factor for user1 increases to 5.1			
13:12:06 5-5-2010	user1	⧁load money onto virtual credit card	100
risk factor for user1 increases to 6.63			
(asynchronous) monitor detects rights violation			
monitor initiates compensation			
13:12:52 5-5-2010	user1	⧀withdraw money from virtual credit card	100
13:12:05 5-5-2010	user1	⧀disable virtual credit card	n/a
13:12:10 5-5-2010	user1	⧀log out	n/a
13:12:13 5-5-2010	user1	⧀deactivate account	n/a

Figure 6.20: Detection by asynchronous monitoring, compensating actions after detection

Late dormancy of user accounts According to the specification, after six months of user monetary inactivity, *i.e.,* no transactions involving money are carried out, the user account should be frozen. Nonetheless, traces such as the one shown in Figure 6.21 were discovered. In this case, there are two compensating activities which have been suggested by the monitor upon the time of detection (third and second trace entries from below). The last activity is the reparation which is carried out after the synchronisation (of the system and the monitor) is complete. Note that although the risk factor for this user was relatively high (ensuring favourable scheduling), this could not lead to synchronisation since no scope closes were encountered.

Timestamp	User	Transaction	$
15:00:38 8-6-2010	user2	account registration	n/a

a monitor is dynamically created for user2 with default risk factor 3

15:15:31 8-6-2010	user2	activate account	n/a
15:15:33 8-6-2010	user2	grant login, card creation rights	n/a
15:15:45 8-6-2010	user2	log in	n/a
15:35:45 8-6-2010	user2	log out	n/a
18:12:14 5-9-2010	user2	log in	n/a
18:42:55 5-9-2010	user2	log out	n/a

by 15:15:31 8-12-2010 *the account of user2 should have been dormant*

17:52:21 18-12-2010	user2	⧁log in	n/a
17:55:50 18-12-2010	user2	⧁create virtual credit card	n/a

risk factor increases to 5.1
(asynchronous) monitor detects non dormant account
monitor initiates compensation

18:00:12 18-12-2010	user2	⧀disable virtual credit card	n/a
18:00:15 18-12-2010	user2	⧀log out	n/a

monitor performs additional corrective action

18:00:20 18-12-2010	user2	change account status to dormant	n/a

Figure 6.21: Detection by asynchronous monitoring, compensating actions after detection, and executing a corrective action after compensating

Excessive money loading to credit cards The system's business logic imposes limits on the amount of money which can be loaded onto a virtual credit card each day, each week and each month. However, two traces similar to the one shown in Figure 6.22 were discovered where the limit for *user3* for a day was $2000. In this case, the risk associated with *user3* not only exceeded the threshold but the heuristic monitor also encountered the closing scope of the money load, and the monitor was thus synchronised. Note that for this reason the system was immediately stopped when attempting to allow the user to load money which exceeded the limit. Since the detection time occurred with the time of violation, no compensations are executed in this case.

Timestamp	User	Transaction	$
risk factor for user3 is 1.6			
`11:05:15 7-10-2010`	`user3`	`log in`	`n/a`
`11:12:16 7-10-2010`	`user3`	`load money onto virtual credit card`	`1000`
risk factor for user3 increases to 2.08			
`11:25:44 7-10-2010`	`user3`	`log out`	`n/a`
monitor synchronises at 12:13:20 (a close scope occurred at `12:12:16`*)*			
`13:15:35 7-10-2010`	`user3`	`log in`	`n/a`
user3 attempts to load $1500 onto virtual credit card			
risk factor for user3 increases to 2.704			
monitor detects violation and stops the activity			

Figure 6.22: Monitor synchronises based on heuristics and stops a violating action from taking place

6.6.5 Discussion

When applied to a suitable case study, adaptive monitor synchronisation is relatively cheap. For example in the case of Entropay, since compensations are part and parcel of the system architecture, then virtually no extra implementation is required. Unfortunately, given the limited time frame for which Entropay was available to carry out the case study, most of the time has been spent on developing the architecture. This means that the implementation of the cLarva architecture is still rudimentary and we have not performed enough experiments to reliably answers questions such as how effective the heuristics were, or what is the best scheduling policy to adopt across monitors. What we can say with conviction is that the non-intrusive monitoring approach we advocate has been welcomed by the industrial partner and it has been effective for detecting significant violations. Furthermore, the theory presented earlier in this chapter provided the assurance that assuming correct compensations, synchronising and desynchronising monitors at runtime through scoped compensation exhibits the same behaviour as synchronous monitoring.

6.7 Related Work

As regards synchronous and asynchronous monitoring, in principle, any algorithm used for synchronous monitoring can be used for asynchronous monitoring as long as all the information available at runtime is still available asynchronously to the monitor through some form of buffer. The inverse, however, is not always true because some monitoring algorithms (such as [91]) require that the complete trace is available at the time of checking so that it can be accessed in reverse. In our case, the monitoring architecture has to support runtime desynchronisation and resynchronisation and thus it was not a question of simply adapting the algorithm but more a question of providing the architectural support. There are numerous algorithms and tools [6, 13, 29, 44, 45, 51, 61, 84, 91, 93] which support asynchronous monitoring — sometimes also known as trace checking or offline monitoring. A number of these tools and algorithms [6, 13, 29, 91] support only asynchrony unlike our approach which supports both synchronous and asynchronous approaches. While a number of other approaches [44, 45, 51, 61, 84, 93] support both synchronous and asynchronous monitoring, no monitoring approach of which we are aware is able to switch between synchronous and asynchronous monitoring during a single execution.

Although the idea of using compensations as a means of synchronisation might be new in the area of runtime verification, this is not the case in other areas, such as distributed games [43, 65, 81]. The problem of distributed games is to minimise the effects on the playing experience due to network latencies. Two general approaches taken are pessimistic and optimistic synchronisation mechanisms. The former waits for all parties to be ready before anyone can progress while the latter allows each party to progress and resolve any conflicts later through rollbacks. The problem which we have addressed in this work is a variant of the distributed game problem with two players: the system and the monitor. In a similar fashion to game synchronisation algorithms, the system compensates to revert to a state which is consistent with the monitor.

From a higher level point of view, the work presented in this chapter aims to control the relationship between the system and the monitor as a means of striking a balance vis-á-vis the monitoring overheads. A number of works [10, 14, 17, 19, 48, 86] have been carried out to optimise the monitoring overhead. The main distinction among these works is that a number of them [10, 14, 48] allow the user to set an upperbound on the resources to be dedicated to monitoring with the possibility of missing violations, while in the other cases [17, 19, 86] it is the tool which decides the resources required but guarantee timely violation detection. Still, none of these approaches enable the switching between the guarantees, which is effectively what we do through switching between synchronous and asynchronous monitoring. The work which somewhat resembles our approach is that of Bartocci *et al.* [14] which supports the use of *criticality levels* to decide which monitoring instances should be given priority. However, this still does not guarantee the timely violation detection for such instances.

6.8 Conclusion

Notwithstanding the rigorous testing which critical systems undergo, problems still arise particularly due to the unpredictable environment under which such systems operate. This scenario motivates the need for monitoring such systems during normal use where the occurrence of an error might imply serious repercussions. However, the problem with monitoring is that it adds overheads to the system which might already be under pressure during peak hours of usage. This motivates the use of asynchronous monitoring which minimises the overhead to logging system events.

When the monitor has detected a problem, then the only option to synchronise the system and the monitor is by reversing the system state. Using compensations, compensation-aware monitoring can be used to restore the system state in case an error is detected late. To handle real-life scenarios where compensations expire we have designed compensation-aware monitoring to handle scopes such that when a compensation terminates, it is no longer compensable. Moreover, pausing and unpausing the system can provide added flexibility to enable switching between synchronous and asynchronous monitoring at runtime. Referred to as adaptive monitoring, this approach can incorporate heuristics which are able to automatically steer the monitoring system from synchrony to asynchrony and vice-versa. As a heuristic example we have demonstrated the use of perceived user risk for switching to monitor synchrony upon scope closure detection. The theoretical framework has been applied to an industrial case study where each user is monitored individually for the perceived risk and the user's monitors are synchronised or desynchronised accordingly.

Although adaptive monitoring offers a highly flexible monitoring architecture, it is based on a limited model of compensations where compensations are specified on a per-action basis. In the absence of a language for specifying compensation synchronisation, keeping the model simple has been preferred with the purpose of avoiding error-prone complexities. In Chapter 8 we lift this limitation by allowing the system-monitor synchronisation to be specified through compensating automata. Furthermore, in this chapter we have assumed system-monitor synchronisation to happen either through compensation or through pausing the system to wait for the monitor; both operations which affect the system. In the next chapter, we investigate a way through which it is the monitor which synchronises to the system instead of the other way round, giving more flexibility to the architecture.

7. Monitor-to-System Synchronisation

Whenever new or modified stateful properties are to be monitored on a system, monitoring has to start processing the system trace from its beginning (which might be years' worth of logs) — ignoring any part of the trace might yield wrong monitoring results since the monitor state would not have correctly evolved over the whole trace. Having a monitoring architecture where the monitor can take corrective measures is of little use if the monitor is processing events which are too old. Even using compensations as proposed in the previous chapter might not be applicable since compensations typically expire. To tackle this issue, we propose *fast-forward monitoring* which provides a means of going through the trace (or an abstraction of it) to quickly infer the state (or a somewhat similar state) the monitor would have reached if the trace was monitored normally. By ignoring the intermediate monitoring states, fast-forward monitoring promises to be significantly faster than normal monitoring.

In this chapter we define a generic theory of monitor fast-forwarding and then we instantiate the approach on the monitoring tool Larva.

7.1 A Theory of Monitor Fast-Forwarding

The idea behind monitor fast-forwarding is to allow the monitor to abstract away parts of the trace but still reach the same configuration that would have been reached had the monitor progressed normally. At its simplest level, this would just involve a function to abstract traces into shorter ones. However, in practice, this is usually not sufficient, and one would require abstracting also the trace-processing transition system. Given a monitoring system, its fast-forward version is another monitoring system, with three additional components: (i) a mapping from the original monitoring system to the fast one; (ii) a trace abstraction which transforms an actual trace into a shorter or a more efficiently processable one; and (iii) a mapping from the fast monitoring system to the original one.[1]

Definition 7.1.1. Given a monitoring transition system $M = \langle \mathcal{C}, c_0, \rightarrow_M \rangle$ with bad states $\mathcal{C}_B$, the transition system $\langle \mathcal{A}, a_0, \rightarrow_A \rangle$ with total domain translation functions $\blacktriangleright\!\blacktriangleright \in \mathcal{C} \rightarrow \mathcal{A}$, $\blacktriangleleft\!\blacktriangleleft \in \mathcal{A} \rightarrow \mathcal{C}$, and trace abstraction function $\alpha \in \Sigma^* \rightarrow \Sigma^*$, is said to be a fast-forward version of M. $\square$

Typically, for actual fast-forwarding, the length of an abstracted trace is shorter than the original trace[2]: $\mathit{length}(\alpha(w)) \leq \mathit{length}(w)$. The trace abstraction function is assumed to map the empty string to itself: $\alpha(\varepsilon) = \varepsilon$ (which follows directly if the abstraction always shortens traces). Apart from modifying the trace for faster processing, fast-forward monitoring may also sacrifice monitoring precision (generating false positives/negatives) in favour of efficiency.

Definition 7.1.2. A transition system A is an *exact* fast-forward version of a monitoring transition system M, with functions $\blacktriangleright\!\blacktriangleright$, $\blacktriangleleft\!\blacktriangleleft$ and α if whenever $c \overset{w}{\Longrightarrow}_M c'$:

[1]Note that the following definitions are based on Definition 2.1.8 stated on page 13.

[2]We do not enforce this in the definition to allow for abstractions to an extended alphabet which although lengthens the trace, would still be processed faster.

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies c' =_M \blacktriangleleft\!\blacktriangleleft(a')$$

It is said to be an *over-approximated* fast-forward version if whenever $c \overset{w}{\Longrightarrow}_M c'$:

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies c' \sqsubseteq_M \blacktriangleleft\!\blacktriangleleft(a')$$

It is an *under-approximation* if $c \overset{w}{\Longrightarrow}_M c'$ implies:

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies \blacktriangleleft\!\blacktriangleleft(a') \sqsubseteq_M c'$$

We will write $c \overset{w}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} c'$ if there exist a and a' such that (i) $a = \blacktriangleright\!\blacktriangleright(c)$; (ii) $c' = \blacktriangleleft\!\blacktriangleleft(a')$; and (iii) $a \overset{\alpha(w)}{\Longrightarrow}_A a'$. □

Proposition 7.1.1. Going to an exact fast-forwarded system and back does not change the observable behaviour of the resulting monitor: $\blacktriangleleft\!\blacktriangleleft \circ \blacktriangleright\!\blacktriangleright \subseteq =_M$.

Proof. The proof follows directly from the definition of exact fast-forwarded systems and the fact that $\alpha(\varepsilon) = \varepsilon$. □

Given a monitoring transition system M and an exact fast-forwarded version M_A, then monitoring a trace partially in M and partially in the fast-forwarded version M_A is equivalent to monitoring it completely with the original monitor M.

Theorem 7.1.1. Given a monitoring transition system $M = \langle \mathcal{C}, c_0, \rightarrow_M \rangle$ with bad states $\mathcal{C}_B$, and an exact fast-forward transition system $M_A = \langle \mathcal{A}, a_0, \rightarrow_A \rangle$ with functions $\blacktriangleright\!\blacktriangleright$, $\blacktriangleleft\!\blacktriangleleft$, and α, given $w = w_1 w_2 \ldots w_n$, then $w \in \mathcal{B}(c_0)$ if and only if there exists $c_n \in \mathcal{C}_B$ and states $c_i \in \mathcal{C}$ such that:

$$c_0 \overset{w_1}{\Longrightarrow}_M c_1 \overset{w_2}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} c_2 \overset{w_3}{\Longrightarrow}_M c_3 \overset{w_4}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} \ldots \overset{w_{n-1}}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} c_{n-1} \overset{w_n}{\Longrightarrow}_M c_n$$

If the fast-forward system is an over-approximation, then the above is a forward implication. Similarly, for an under-approximation, only the backward implication is guaranteed to hold.

Proof. This result follows by induction on the number of parts string w is split into. □

Interestingly, while it is generally undesirable to use over- or under-approximations, in certain scenarios one might be ready to compromise having false negatives in the case of an over-approximation and false positives in the case of an under-approximation[3]. While in this work we focus on using monitor fast-forwarding to initialise monitors quickly — *fast monitor bootstrapping* — in the following list we suggest a number of applications of fast-forwarding monitoring, highlighting where it is preferable to use exact or approximate fast-forward monitoring:

Fast monitor bootstrapping Whenever monitors have to be instantiated on a system with a long recorded history, running the standard monitor on the long traces may take prohibitively long. An alternative is to process the traces using an exact fast-forwarded version of the monitor. Approximate fast-forwarding can also be useful if, either we are assured that there are no errors to be caught on the stored history (in which case we can use an under-approximation) or if we prefer to process the history quickly ensuring that any errors are caught (in which case, an over-approximation would be applicable).

Burst monitoring In systems where resources are committed only at particular points in time, it can be beneficial to accumulate and process the system trace only at these moments in time. For instance, in a transaction processing system where all database modifications are committed at the end of a transaction one may, for example, collect the full trace of a transaction and process it using a fast-forwarded monitor. If an exact fast-forward may still be too expensive to check and performance is an issue, one may choose to apply over-approximations for transactions by blacklisted users and under-approximations for whitelisted ones.

[3]By definition, over-approximation may detect bad behaviour which doesn't actually exist since the abstraction might consider more behaviours to be bad than the concrete counterpart. The converse is true for under-approximation.

Synchronous/asynchronous monitoring In the case of asynchronous monitoring, fast-forward monitoring can be used whenever the monitor is lagging too much behind the system. Moreover, in monitoring systems such as the case study presented in the previous chapter where asynchronous monitoring can be synchronised at runtime, fast-forward monitoring can be used for a quick synchronisation.

To facilitate the use of monitor fast-forwarding to applications such as the ones listed above, in the next section we instantiate the theory to the runtime verification architecture Larva.

7.2 Instantiating Fast-Forwarding to Larva

In Larva we use monitor fast-forwarding to start monitors from a particular point in a system's history, *i.e.,* fast monitor bootstrapping. This is crucial for industrial systems so that when new properties are introduced or existing ones modified, the monitor comes up to scratch with the system as soon as possible — monitoring years' of data would waste monitoring time which could be used to start monitoring more recent (and thus more relevant) data.

Instantiating the theory of fast-forwarding to monitor bootstrapping in Larva would include deciding the two translation functions $\blacktriangleright\blacktriangleright$ and $\blacktriangleleft\blacktriangleleft$, the trace abstraction function α, and the fast-forward transition relation $\rightarrow_A$. The following list expands each of these aspects, generalising the approach taken in Example 7.2.1:

- In the case of Larva it is assumed that the translation function $\blacktriangleright\blacktriangleright$ obtains the list of objects which should be monitored during fast-forwarding while the reverse translation function $\blacktriangleleft\blacktriangleleft$ drops the additional information.
- The trace which originally includes all the system events, is collapsed to

the trace elements which are required for deciding the state of each monitor.

- The fast-forward monitor which handles fast bootstrapping, first obtains $\overline{M}$ (representing a vector of monitors[4]) by instantiating a monitor for each entity indicated by ►► and subsequently allows each entity to perform one step through the fast-forward transition relation $\rightarrow_A$ — updating its monitor state based on the abstracted trace. Such a configuration step should include three aspects corresponding to the configuration components $\overline{q}$, ct, and θ respectively:

  1. The state of the respective monitor (an element of $\overline{q}$) is updated *e.g.,* monitors of users whose account has been put into dormant state in the recent past (according to the system event log) should be in the state *dorm*.
  2. The clocks of each monitor (*i.e., ct*) are set *e.g.,* when the last financial transaction took place so that it can be ensured that inactive users are actually put into the dormant state.
  3. The values of the variables of each monitor (*i.e.,* θ) are set *e.g.,* counting the number of transactions which the user has carried out so that monitors can check that the allowed quota has not been exceeded.

Example 7.2.1. As an example of monitor fast-forwarding, we use a Larva monitor from the case study which we presented in the previous chapter. To aid the reader, we again give a short description of the property.

To ensure the security of inactive accounts, users who are inactive for more than six months are suspended, *i.e.,* put in *dormancy mode,* and an administration fee is charged. If a user asks for his account to be reactivated, then the request is granted but the account is switched to dormant once more if the user

[4]The definitions have been given in Chapter 2, Section 2.1.2.

still remains inactive for another three months. This property is monitored for each system user and thus a replica of each monitor is instantiated for each user. Excerpts of the Larva script which specifies the dormancy property is given in Figure 7.1[top] while the depiction of the DATE automaton is given in Figure 7.1[bottom][5].

```
Foreach (String user) {
   Variables {
      Clock T6, T3; }
   Events {
      expired6 = {T6 @ 183 days}
      anyTx = {event(currency, amount, type) where type = "generic"}
      ... }
   Property dormancy {
      States {
         Bad { expiredDorm, unexpectedDorm, failedPay, unexpectedTx }
         Normal { dorm, thawed, feepaid }
         Starting { nonDorm } }
      Transitions {
         nondorm -> nondorm [anyTx\\resetT6]
         nondorm -> dorm [dorm\expectedT]
         ... } } }
```

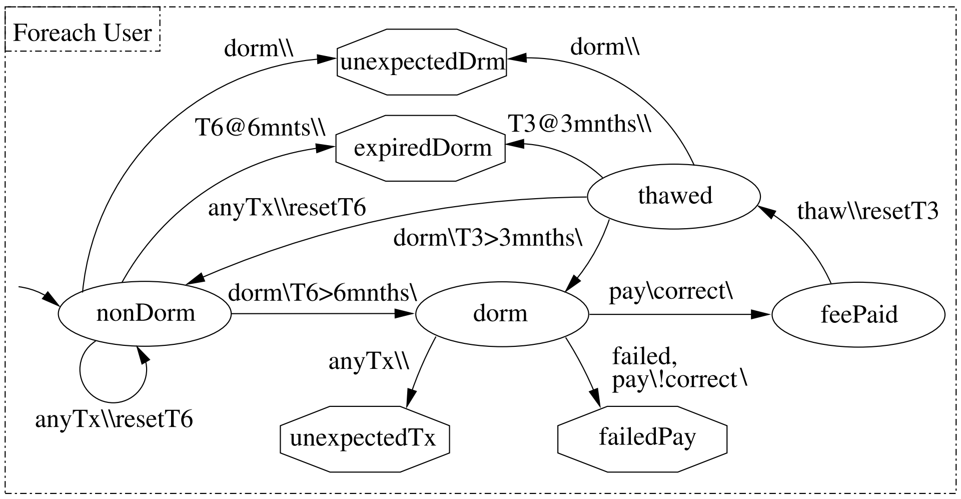


Figure 7.1: The dormancy property expressed as a Larva script [top] and as a DATE [bottom]

Referring to the dormancy example, we illustrate how the theory of fast-forwarding can be instantiated for initialising dormancy monitors for the existing users of a system. Fast-forward initialisation, or *fast monitor bootstrapping*,

[5]For brevity we use *dorm* for *dormancy*, *Tx* for *transaction*, and *T* for *Timing* or *Timer*.

entails a choice of a cut-off point during the system's life-cycle (as shown in Figure 7.2) which marks up to which point the monitor is expected to fast-forward. In the rest of the example this point is referred to as the *point of initialisation*.

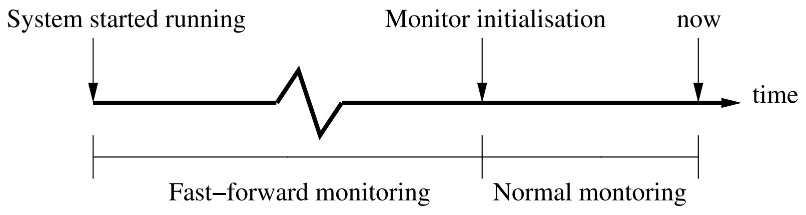


Figure 7.2: The typical timeline of fast monitor bootstrapping

For the sake of this example we will focus on deciding whether a user is in the *nonDorm* state, or the *dorm* state and set the clocks accordingly. To facilitate this decision, we need two pieces of information: (i) a list of system users, and (ii) for each user, the timestamp for the latest transaction or the latest switch to dormancy; whichever is most recent (this information would enable us to deduce the active automaton state and the timers' values).

Thus, assuming a monitor $\langle \mathcal{C}, c_0, \rightarrow_c \rangle$ and a system trace s, the corresponding translation function ►► transforms c_0 into a configuration of type $\mathcal{C} \times \Psi$ (using information from the system state) where Ψ is the list of users currently active in the system. The reverse translation function ◄◄ simply drops Ψ from the resulting configuration. The trace abstraction function α drops all events except for the most recent activity of each user (with respect to the point of initialisation). As for the fast-forward transition system, $\rightarrow_A$, for each user there are conceptually two options: either his or her last activity was a normal transaction, in which case the user is in the *nonDorm* state and the stopwatch should be set to trigger six months from the latest transaction, or the last activity turned the user into dormant and hence the user is in the *dorm* state.

Over and above instantiating the theory of monitor-fast-forwarding to Larva, appropriate constructs have to be provided as part of the specification script to aid the user program the necessary abstractions. This is further discussed in the next section.

7.3 Adapting Larva Scripts

To enable users to easily program fast monitor bootstrapping, we have augmented the Larva script structure. Recall that Larva provides the *foreach* construct which enables monitors to be replicated for distinct objects of a particular type. Furthermore, *foreach* components can be nested (*e.g.,* for monitoring each credit card of each user) and the outermost *foreach* components are enclosed in a *global* component which can be used to monitor properties which are not replicated, *i.e.,* not related to particular objects. To initialise a *global* component what is required is to give a value to variables, clocks and update the state of any global property automata. To this end, we introduce a novel construct to the Larva specification: *initializeIf*. The code in the *initializeIf* component triggers on a particular condition (indicating that the monitor is in fast-forward mode) enabling the user to specify a Java method which returns a hashmap with variable/clock/automata names as keys and the corresponding intended values as the hashmap values. No setting needs to be done if the variables/clocks/states have not progressed from their default initialisation. Note that the *initializeIf* component corresponds to the fast-forward transition relation $\rightarrow_A$ and also the trace abstraction function α since it extracts a minimal set of events to perform the initialisation.

Yet it is not enough to be able to initialise a monitor for the global context — the Larva script should also allow the script writer to specify a means of deducing the number of users in the system for whom a monitor should be replicated and initialised. For this reason, each *foreach* may contain an *initially* component (apart from an *initializeIf* component) which can specify a method returning an array with all the objects for which a monitor should be created[6]. In our ex-

[6]Larva supports *foreach* components for tuples of objects. Thus, the initially method actually returns an array of arrays where each array supplies an element of the tuple.

ample the *initially* method returns an array of user id's. Note that the *initially* component corresponds to Ψ as presented in Example 7.2.1, contributing to $\blacktriangleright\!\blacktriangleright$. Furthermore, in the case of Larva the reverse translation function $\blacktriangleleft\!\blacktriangleleft$ is implicit and is done conceptually by dropping Ψ.

The approach described in this section has been successfully applied to the live data of an industrial case study with promising results as elaborated in the next section.

7.4 Applying Monitor Fast-Forwarding to Entropay

Since Entropay had been up and running for more than a year at the time of applying monitoring and it was envisaged that monitors would have to be modified or added regularly, fast monitor bootstrapping was crucial to make monitoring feasible.

Using the constructs introduced in Section 7.3, Figure 7.3 shows how using two SQL queries (marked with *(1)* and *(2)* in the code excerpt) we deduce when the last successful transaction occurred for a particular user and whether the user has been recently (since the last successful transaction) put into dormant state. Using this information we set (marked with *(3)*) the corresponding clocks to trigger when the user should be put to dormant in the future. Moreover, if the user is currently dormant, then the corresponding *dormancy* automaton (shown in Figure 7.3) is to be in state *dorm*. Note that we assume that the current system state does not contain errors and consequently our fast-forwarding is an under-approximation. Finally, a third SQL query (marked with *(4)*) is used to obtain the list of active users for each of whom a monitor has to be instantiated.

The approach described in this section has been successfully applied to the live data of an industrial case study with promising results as elaborated in the final section of the chapter.

```
Foreach (String user) {
  Initializeif (init) {
    static HashMap<String, Object> initializeifUser(String user) {
      HashMap<String, Object> list = new HashMap<String, Object>();
    //obtain last successful user transaction
(1)   rs = st.executeQuery(
           SELECT timestamp FROM transaction_table
           WHERE id=@user AND timestamp < @initializationTime
           ORDER BY timestamp DESC);
      latestTrans = rs.getLong("timestamp");

    //check if user is currently dormant
(2)   rs = st.executeQuery(
           SELECT timestamp FROM log_table WHERE id=@user
           AND event="USER_DORMANT" AND timestamp < @initializationTime
           ORDER BY timestamp DESC);
      latestDorm = rs.getLong("timestamp");

(3)   if (latestDorm > latestTrans) {
        //i.e. user is currently dormant
        //therefore put automaton into "dorm" state
      } else {
        //i.e. user is not dormant
        //set clock to expire 6 months after last transaction occurred
      }
...
     return list; } }

//code given in the previous example goes here
  Variables {...} Events {...} Property ...
//code given in the previous example ends here

  Initially {
    static ArrayList initiallyUsers() {
      ...
(4)   rs = s.executeQuery(SELECT id FROM users_table;);
      while (rs.next())
        list.add(rs.getString("id"));
      return list;
  } } }
```

Figure 7.3: The dormancy example augmented with fast bootstrapping code

7.4.1 Performance Results

The monitor was deployed on data representing activities starting from 23rd December 2008. Data before this date was considered to be too old and would waste monitoring time which could more beneficially be used to monitor recent data pertaining to users which are more probably still active at the time of monitoring. To quickly bootstrap the monitors up to 23rd December 2008, we used the fast-forward technique on 58 weeks of data starting from 8th November 2007.

Using a Dual Core AMD Opteron Processor at 1.81GHz running Windows XP x64 with 2Gb RAM, the monitors successfully fast-forwarded through 58 weeks in 35 hours. Subsequently, the monitors were run on the available data (at the time when this case study was carried out) dating till 8th September 2009 (including 37 weeks of data) consisting of millions of transactions. This process took 552 hours (approximately 23 days) equating to less than 15 hours of processing per one week's data. Proportionately, monitoring the 58 weeks of data would have roughly taken the monitor 36 days to come at par with the live system as opposed to the day and a half with fast-forwarded initialisation. This time saving is crucial when one would need to receive immediate feedback upon deploying new monitors, particularly if remedy actions can be taken based on monitoring results.

7.4.2 Discussion

The downside of the current instantiation of monitor fast-forwarding is that the user has to program the fast-forwarding abstractions manually. From our experience, coming up with fast-forwarding monitors is more challenging than devising normal monitors. The reason is that normal monitoring is usually more similar to the typical specifications accompanying industrial systems, while the logic needed for fast-forward monitoring can only be obtained by having an intimate knowledge of the system at hand. For example, referring back to the dormancy example, the industrial specifications are written in the following imperative style: (i) *The cycle starts when a registered user is inactive for six months, at that point the user account must be put to dormant.* (ii) *Whilst dormant, the user may not perform any transactions but may ask to be reactivated.* (iii) *If the user* *has been reactivated but does not carry out a financial transaction for another three months, the user account is deactivated again.* This logic can almost be directly translated into normal monitors with states and transitions. On the other hand, programming under-approximating (*i.e.,* assuming the system worked correctly before monitoring started) fast-forward monitoring would requires declarative knowledge such as: (i) *If the user has performed financial transactions since the last time he or she has been dormant, then the user must be active.* (ii) *On the other hand, if no transactions have been carried out since, then the user is still dormant.* (iii) *Yet another possibility is that if the user has carried out (only) non-financial transactions, then the user has been thawed but not fully activated yet.* Although statements such as the latter can usually be inferred from the former, they are not typically written in technical specifications and engineers are more accustomed to the imperative style of specifications.

One use of more generic fast-forwarding in our case study would be to enable the monitor to keep up with the system in case asynchronous monitoring is consistently slower than the system. However, as yet, we have never encountered monitors which are not able to keep up with the system. In our case study results it is noteworthy that once monitors come at par with the system, it is not a problem for the monitors to keep up (with slightly more than two hours of processing for a day's events). Still, if one had to adapt the above given code for fast-forwarding asynchronous monitoring, this can be done by simply adding a condition in the SQL statements to ignore entries before a particular date (the date where the slow asynchronous monitors have reached).

7.5 Related Work

To the best of our knowledge the idea of fast-forward monitoring is novel. A notion which relates to fast-forward monitoring is counterexample shrinking [76, 99] from the area of testing. For example QuickCheck [64], a model-based testing tool for Erlang, attempts to find a shorter trace when a bug is found. This simplifies the developers' task of debugging since it is easier to understand what happened in a simpler trace. Note that counterexample shrinking is a special case of our fast-forwarding theory: (i) the translation functions are the identity functions, (ii) the trace abstraction function returns a shorter or simpler trace, and (iii) the same identical oracle that is used during testing is used during shrinking. In this case exact fast-forwarding ensures that the same bug that was exhibited during the original trace is also exhibited when monitoring the simpler one.

7.6 Conclusion

The main problem with asynchronous monitoring is that the monitor may fall indefinitely behind the system, meaning that it might be too late to take any corrective measures by the time the monitor detects a violation. For this reason, it is desirable to have ways in which the relationship between the system and the monitor can be controlled. From the monitor point of view, we have presented monitor fast-forwarding — a means by which the monitor can keep up with the system. The theory of fast-forwarding has been instantiated for the monitoring tool Larva by enhancing its script with two new components which enable users to specify fast monitor bootstrapping. Furthermore, we have shown the usefulness of this approach for an industrial case study where monitors have to process millions of records before starting to process relevant events.

8. Synchronisation Programming with Compensating Automata

A significant limitation of our introduction of compensations to synchronise monitoring (namely the architecture shown in Figure 6.8) is the assumption that each individual action has an individual compensation. In many real-life scenarios, there may be several ways in which an action may be compensated depending on the context; *e.g.,* a payment may be refunded free of charge if a system error occurs and for a fee in case of a customer cancellation. Furthermore, frequently individual compensations are discarded at some point and replaced by coarser-grained compensations, *i.e.,* having one compensation for several actions. For example it is common that once a transaction is closed and shipment leaves, the transaction can only be compensated by another transaction which ships back the goods.

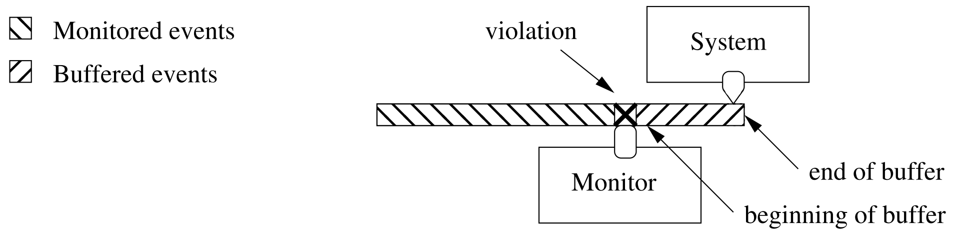


Figure 8.1: Monitored and buffered events

Another limitation of the architecture is that it is very inflexible with respect to the actions it compensates for — it automatically compensates for all the actions which are in the buffer (depicted in Figure 8.1 as the list of events which the system has produced but the monitor has not yet consumed) at the time of error detection. There are at least two reasons why one might want to modify this: (*i*) Not all actions in the buffer may need to be compensated (*e.g.,* only the most risky operations are compensated). (*ii*) Some actions which have already been monitored — *i.e.,* not in the buffer at detection-time — might need to be compensated: *e.g.,* in the case of a fraud this is usually detected late even when monitoring synchronously, in which case one may desire to undo certain actions which have been already monitored.

While these limitations have been partially lifted through compensation scoping, the cLarva architecture presented in Chapter 6 can still be considered inflexible and useful only in particular scenarios. To lift these limitations in this chapter we explain how compensating automata (presented in Chapter 4) can be integrated with cLarva and show how the modified architecture can be applicable to a realistic case study.

8.1 Proposed Architecture

The cLarva architecture is able to signal the system to compensate for a sequence of actions, *i.e.,* the actions remaining in the buffer at detection time after removing actions within closed scopes. To allow more flexibility as regards the choice of compensations, we introduce an additional *compensation management* component within the monitoring architecture as shown in Figure 8.2. By listening to the system's events, the compensation manager dynamically manages what compensations are to be executed if (and when) the monitor detects a problem. When the monitor identifies a problem, it signals the compensation manager with a compensation strategy which is to be carried out in the spirit of monitor-oriented compensation programming. Such a strategy would include a choice of various compensating automata whose accumulated compensations are to be executed (sequentially or concurrently).

An important consideration in the managing of compensations is usually the point at which the violation had occurred and the point at which the monitor has detected a problem (depicted in Figure 8.1). These two points are indicated by the dedicated signals: *beginning-of-buffer* (*BoB*) and the *end-of-buffer* (*EoB*) respectively. Using these signals the compensation manager and the monitor can take into consideration the fact that the events they will receive following the *BoB* have occurred after a violation, while the receiving *EoB* means that all the events have been received.

Incorporating these modifications into the architecture presented in Chapter 6, we add the compensation manager component which receives all the events as received by the monitor, and the two extra signal *BoB* and *EoB* as shown in Figure 8.2.

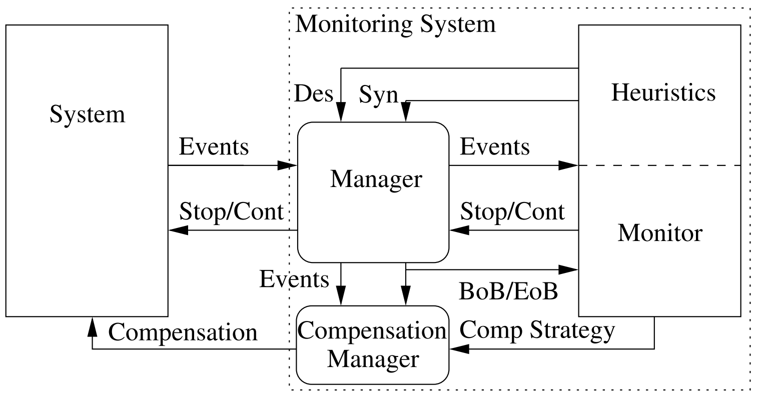


Figure 8.2: The cLarva architecture with the compensation manager

Note that unlike the previous architecture where the monitor simply communicated the buffered activities to be compensated, the new architecture delegates the compensation synchronisation to the compensation manager. This is a significant difference since it is no longer the system which has to decide how to compensate for the actions in the monitor buffer but rather a separate module, keeping the system uncluttered.

To facilitate the programming of the compensation manager, a compensation programming language relieves the user from having to manually program compensations. To this end, we propose compensating automata which have been devised specifically for programming compensations. An advantage of using compensating automata is that they can be seamlessly integrated with the monitors as proposed in Chapter 5, using monitors to decide *when* and *which* compensation strategy to be carried out (hence the *compensation strategy* signal from the monitor to the compensation manager in Figure 8.2).

In what follows, we give two examples to show how this architecture can be useful for synchronising the system with a monitor if a problem is detected late.

8.2 Case Study

To illustrate how one could use compensating automata to synchronise the system to the monitor, we use a case study inspired from the Entropay industrial case study (presented in Chapter 6). For simplicity we assume that after a user logs in, he or she may perform a sequence of operations made of repeating any of the following three: (i) loading money from an external source onto a virtual credit card, (ii) transferring money across virtual credit cards, and (iii) purchasing through a virtual credit card. The sole aim of this case study is to show how compensating automata can be used to synchronise the system to the monitor and thus we abstract away from the monitoring logic which detects the violation itself. Rather, we assume that some monitor signals that a failure has occurred and that some other monitor is responsible of listening to failures and triggering compensations (examples of the latter are shown in Figure 8.3[top] and Figure 8.4[top]). The following points outline two examples of system failure where synchronising the system is done through compensating automata:

Excessive loads When the monitor detects that the limit of money that can be loaded has been exceeded, the relevant subsystem is stopped and a

BoB signal is issued (*i.e.,* a violation has been detected and the system has stopped), followed by the events in the buffer (possibly including further loads which have been performed after the limit had been exceeded). Figure 8.3[bottom][1] shows a compensating automaton which accumulates compensations after it receives the *BoB* signal, *i.e.,* compensations are only accumulated for loads which are still recorded in the buffer (and had not been processed by the monitor)[2]. Once all the events in the buffer have been processed, the *EoB* signal is issued and the monitor depicted in Figure 8.3[top] triggers *[EXCESS]* to execute the accumulated compensations. Note that in this case, the fixed approach adopted in Chapter 6 would not have been appropriate since not all the buffered events required compensating (only the load events). The programmed compensating automata depicted in Figure 8.3[bottom] automatically ignores any events which do not match with the pattern.

Fraud detection The scenario is considerably different when compensating for the actions of a user who have been detected to be a fraud. In such a case, one would not only want to compensate till the point where the user was detected to be fraudulent since the user has probably been fraudulent all along[3]. Although the compensating automaton which compensates for a load action by a fraudulent user (given in Figure 8.4[bottom]) appears to be similar to the one used for compensating excessive loads, there are two significant differences:

- The *fraud* compensating automaton does not wait for the *BoB* signal

[1]We use the following abbreviations: *dec* (decrease), *inc* (increase), *crd* (card), *avl* (available), *act* (actual), *bal* (balance), *int* (internal), *ext* (external), *not* (notify), *opr* (operator), *rec* (receive), *settl* (settlement), *src* (source), and *dst* (destination).

[2]For simplicity we assume that the money which has been loaded in excess has not yet been used for transfers and/or purchases and hence we compensate for load operations only. In an actual implementation one would need to quantify the excess and also compensate for any operations which have only been possible due to the excess.

[3]For the monitor to detect a pattern of fraudulent behaviour, the pattern would have already started earlier.

before starting to configure compensations. This means that the compensating automaton will compensate for *all* the loads of a user; not only those which remain in the buffer.

- Another difference is that completed transactions are not undone but instead the credit cards involved are frozen. The rationale of this choice is that fraud detection is usually investigated further by a human operator. Thus, to avoid unnecessary inconvenience to innocent users who might be detected as fraudulent, only incomplete transactions are compensated — immediately stopping any impending high-risk activity.
- While excessive loading of money is an issue concerning money loads, a fraud permeates all the user's operations (including transfers and purchases). Thus, although we have focused only on money loads for the sake of this example, an extended version of the fraud detection case study is given in Appendix A.

Note once more that the architecture of Chapter 6 is not suitable for synchronisation in this example since events which have been removed from the buffer are also compensated.

8.3 Conclusion

Whilst synchronising a system with its monitor through compensations is beneficial to enable asynchronous monitoring to give feedback to the system, a simplistic compensation architecture such as that presented in the previous chapter would not always suffice. To cater for compensation complexities, the user would have to manually introduce additional compensation logic at the system side. By externalising such compensation logic into a separate module and programming it through compensating automata we have provided full support for complex compensation programming. Furthermore, using monitor-oriented compensation programming we have seamlessly integrated the compensation manager with the monitors within the existing architecture. This enables the explicit programming of *when* and *which* synchronisation strategy to be used depending on the violation detected. Finally, we have applied the architecture to two examples based on an industrial case study to show where synchronisation programming is crucial to enable automatic recovery under asynchronous monitoring.

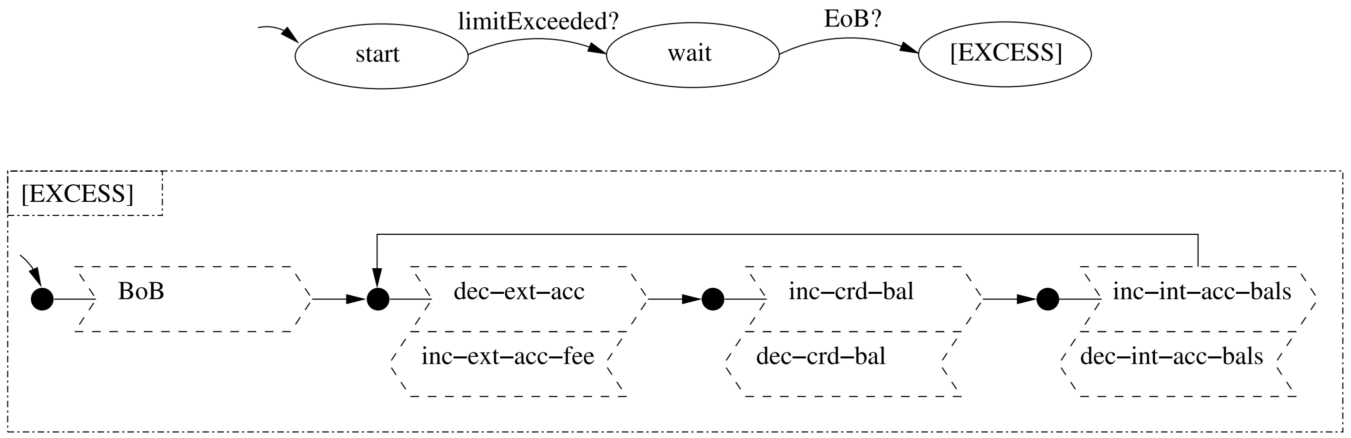


Figure 8.3: A monitor which detects load limit excess and the *EoB* signal [top], a compensating automaton which manages compensations when excessive loads occur [bottom]

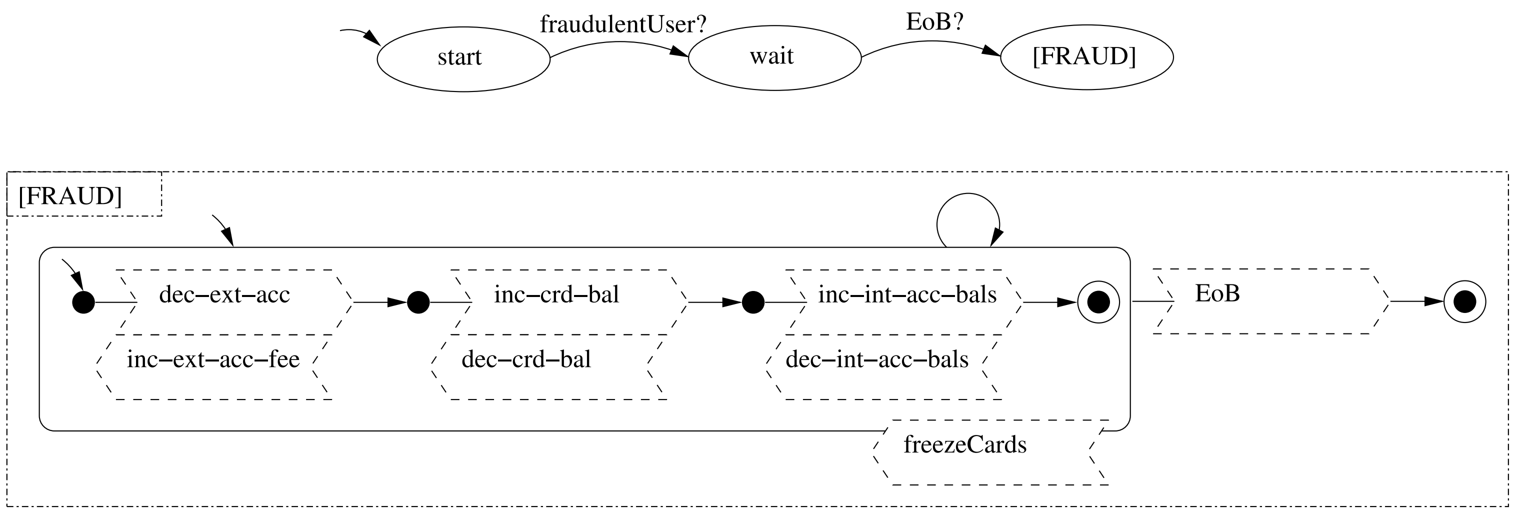


Figure 8.4: A monitor which detects user fraud and the *EoB* signal [top], a compensating automaton which manages compensations when a fraudulent user has performed a load [bottom]

Part IV

Conclusions

This part summarises the previous two parts and discusses directions for future work, ending with some final remarks.

9. Conclusion

Starting out with two seemingly unconnected topics, this study has shown compensations and runtime verification to be significantly related. On the one hand, runtime verification has helped alleviate the challenge of programming compensations through separating compensation programming concerns from others. This not only eases the programming of such concerns due to the separation but also introduces added flexibility by decoupling the structure of compensations from the structure of the system they are compensating for. To support this approach, we have proposed compensating automata, a compensation-dedicated automata-based notation which provides specialised constructs for programming compensations. Through a non-trivial case study from the literature we have shown compensating automata to be highly expressive.

Although compensating automata free the system from being aware of the compensation logic, the latter still has to be aware of when to execute compensations. To this end, we have shown how compensating automata can be integrated into a monitoring architecture resulting in a framework where compensation programming can be entirely delegated to an external monitoring module, leaving the system uncluttered.

On the other hand, compensations can be also useful for providing solutions for significant issues in the context of monitoring, notably the problem of runtime overheads induced through monitoring. While to date asynchronous monitoring has been synonymous with non-steering monitoring, enabling the system and the monitor to synchronise upon error detection lifts this limitation. The possibility of such a synchronisation significantly relaxes the coupling between a system and its monitor, relieving the system from intrusiveness. Naturally, this might introduce a significantly longer delay between the occurrence and detection of a violation — something which might not always be acceptable. For this reason our framework allows the use of monitor heuristics which automatically decide when to synchronise and desynchronise the monitor and the system, enabling fine-grained control over the tradeoff between monitor intrusiveness and error detection latency. This approach has been applied to an industrial case study with encouraging results and further scope to refine the architecture. The case study made us aware of the need of fast-forward monitoring, a technique which has proved crucial for initialising stateful monitors when the system has been running for a number of years. Moreover, we have also become aware of the limitations of our framework, particularly in the system-monitor synchronisation protocol which always follows a rigid pattern of compensation execution. To this extent, we have described a way of enhancing the framework, enabling the user to program the synchronisation protocol through compensating automata.

9.1 Future Work

There are several directions in which the work presented here can be taken in the future:

Language analysis Although we have formally specified compensating automata, we have not yet carried out an in depth analysis from a language perspective. From initial considerations, compensating automata seem to be a special case of Aho's nested stack automata [3]: nested stack automata allow reading of any stack elements while compensating automata only allow access to the top most stack element at particular points of execution (during compensation execution). This would imply that the language is an indexed grammar [2] and thus that it shares a lot of properties with context-free grammars vis-à-vis language closure properties. In the future, we aim to verify or refute these claims by performing a thorough study of the language properties of compensating automata.

Syntactic sugar The compensation example given in Section 4.4 shows that there are recurring patterns of synchronisation across parallel automata which the programmer has to hand code through local channel communication. In the future we aim to provided such patterns as special constructs which compile to basic compensating automata syntax. Of particular interest are two constructs which have been discussed in the example: (i) one which synchronises the start of a number of parallel automata; and (ii) another which synchronises the end of concurrent compensation execution before continuing with compensating earlier actions.

Model checking Since compensating automata execute compensation actions in the context of a system event trace, it might be useful to model check the execution orderings which might be triggered, *e.g.,* ensuring that two actions are always executed in a particular order. This would guarantee added assurance that a compensating automata specification actually encodes the expected behaviour. Model checking compensating automata is non-trivial since the state space of compensating automata is potentially infinite due to the unbounded size of the stack.

Real-time Real-time aspects have not been incorporated within compensating automata. This might be done for example by supporting timed events, enabling compensations to be installed, discarded and/or replaced upon a timeout. Given that real-life scenarios involving business transactions usually entail significant timing constraints, this would be an interesting addition.

Industrial application Whilst we have successfully applied our theory and techniques on industrial and industry-inspired case studies, there is still a lot of scope for further experimentation. In particular, monitor-oriented compensation programming through compensating automata has not been tried out on industrial-scale case studies. Indeed such applications to real-life scenarios would be vital for transforming the current prototype toolset into a mature and reliable one.

9.2 Concluding Thoughts

A major insight obtained through combining runtime verification with compensations is that whilst it has been argued that compensations are "not enough" for real-life scenarios [57], through this work it has been shown that it is not the compensation mechanism which is the problem but the tightly system-coupled way compensations are usually programmed. Through the adoption of monitoring-oriented programming we have shown that compensations can still be useful in very complex scenarios.

This work has hopefully shed more light on the usefulness of the runtime verification approach to programming real-life software which is becoming increasingly complex. Unfortunately, convincing industrial case studies are still lacking in the area of runtime verification. One of the major contributing factors probably being the intrusiveness of synchronous monitoring. To this extent we have attempted to show how intrusiveness can be reduced to a minimum by synchronising and desynchronising monitors at runtime — making it more practical for industrial adoption. We hope that in the future these insights lead to more uptake of runtime verification particularly in the security-intensive areas where compensations are frequently used.

A. System-Monitor Synchronisation Extended Case Study

Further to the case study given in Section 8.2 concerning the programming of system-monitor synchronisation through compensating automata, we give an extended version of the fraud example. The main difference is that the compensating automaton provided in this section manages all the financial operations in a user's life cycle as opposed to managing only the *load* operations.

For simplicity we assume that a user life-cycle is a repetition of a choice of three actions: loading money from a bank account onto a virtual credit card, transferring money from one virtual credit card to another, and using a virtual credit card to affect a purchase. If a user is detected to be fraudulent, all actions (before and after being detected as fraudulent) are compensated. A significant difference between the programmed compensations for fraud and those for excessive money loading is that in the case of fraud, completed transactions are not undone but instead the credit cards involved are frozen. The rationale of this choice is that fraud detection is usually investigated further by a human operator. Thus to avoid unnecessary inconvenience to innocent users who might be detected as fraudulent, only incomplete transactions are compensated. Figure A.1[1] shows the compensating automata involved in programming compen-

[1]We use the following abbreviations: *dec* (decrease), *inc* (increase), *crd* (card), *avl* (available),

sations for the life-cycle of a user who has been detected as fraudulent. For readability, parts of the main compensating automaton (shown last) are shown before and marked with curly brackets, namely {*LOAD*}, {*TRANS*}, and {*PURCH*}. Managing compensations for these three operations mainly involves monitoring balance changes and recording compensations accordingly.

act (actual), *bal* (balance), *int* (internal), *ext* (external), *not* (notify), *opr* (operator), *rec* (receive), *settl* (settlement), *src* (source), and *dst* (destination).

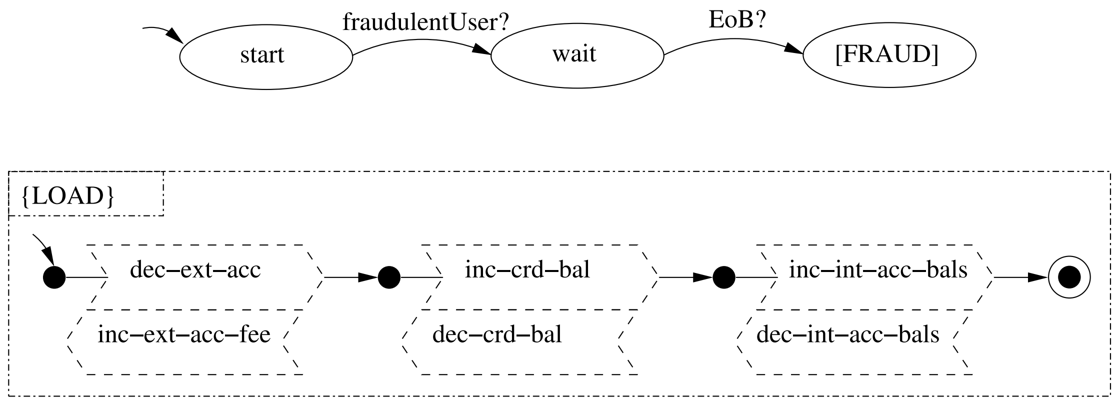


Figure A.1: A user life-cycle compensating automaton which manages compensations for a user life-cycle (cont.)

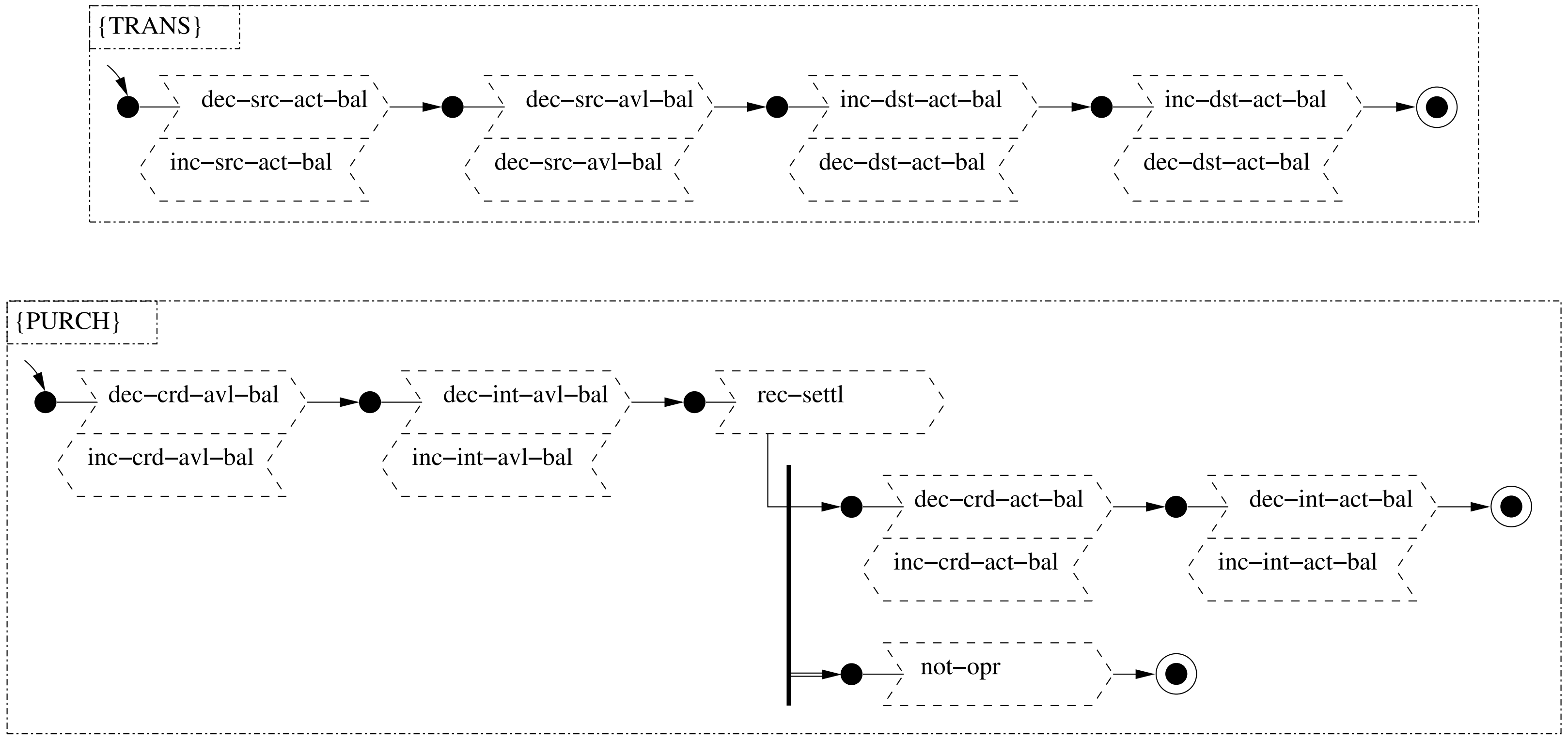


Figure A.1: A user life-cycle compensating automaton which manages compensations for a user life-cycle (cont.)

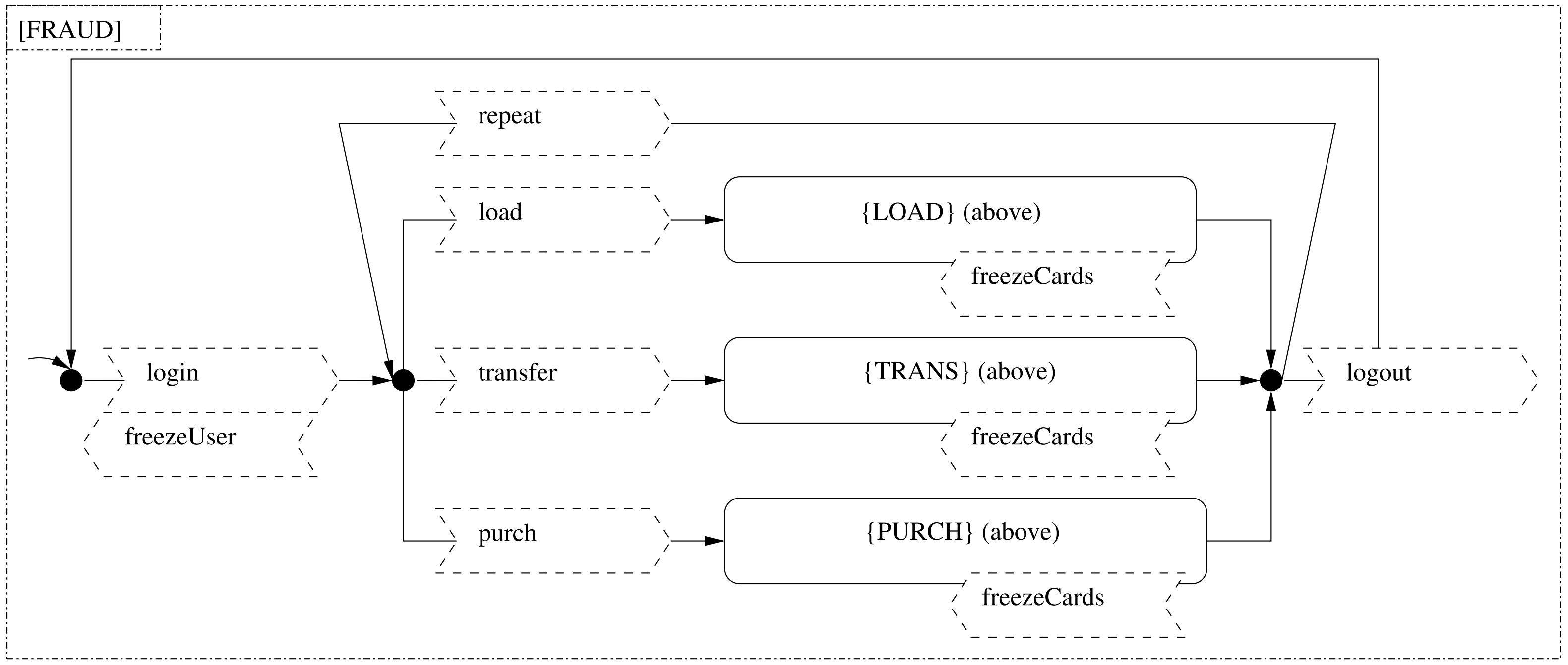


Figure A.1: A user life-cycle compensating automaton which manages compensations for a user life-cycle

B. Publications

The work presented in this thesis includes the following published and unpublished papers. The contribution of the respective authors is explained below.

Published Papers

1. Monitor-Oriented Compensation Programming, Christian Colombo and Gordon J. Pace, to appear in Graph Transformation and Visual Modeling Techniques. 2013.

   Pace reviewed the paper several times.

2. Fast-Forward Runtime Monitoring — An Industrial Case Study, Christian Colombo and Gordon J. Pace, in Runtime Verification, volume 7687 of Lecture Notes in Computer Science, pages 214–228, Springer, 2012.

   Pace reviewed the paper and rewrote Section 3.1.

3. Recovery within Long Running Transactions, Christian Colombo and Gordon Pace, to appear in ACM Computing Surveys, 2012.

   Pace performed several reviews of the paper and rewrote parts of Section 5.

4. Safer Asynchronous Runtime Monitoring Using Compensations, Christian Colombo, Gordon Pace and Patrick Abela, volume 41 of Formal Methods in System Design, pages 269–294, 2012.

Pace reviewed the paper several times, rewrote parts of Section 3 and wrote Section 4. Abela reviewed the parts related to Ixaris Ltd.

5. Compensation-Aware Runtime Monitoring, Christian Colombo, Gordon J. Pace and Patrick Abela, in Runtime Verification (RV), volume 6418 of Lecture Notes in Computer Science, pages 214–228, Springer, 2010.

   Pace reviewed the paper several times and rewrote parts of Section 3. Abela reviewed the parts related to Ixaris Ltd.

Papers Submitted for Publication

1. Fully Monitor-Oriented Compensation Programming, Christian Colombo and Gordon J. Pace, Department of Computer Science, University of Malta. 2012.

   Pace reviewed the paper.

Technical Reports

1. Separating Compensation Concerns and Programming them with Compensating Automata, Christian Colombo and Gordon J. Pace, Department of Computer Science, University of Malta. Technical Report CS2012-01, 2012.

   Pace reviewed the paper several times.

2. Programming Compensations for System-Monitor Synchronisation, Christian Colombo and Gordon Pace, internal report 03-WICT-2012, University of Malta, 2012.

   Pace reviewed the paper.

3. A Compensating Transaction Example in Twelve Notations, Christian Colombo and Gordon J. Pace, Department of Computer Science, University of Malta. Technical Report CS2011-01, 2011.

   Pace reviewed the paper several times.

4. An Architecture Supporting Compensation-Aware Monitoring, Christian Colombo, Gordon J. Pace and Patrick Abela, internal report 01-WICT-2010, Malta.

   Pace reviewed the paper several times.

5. Offline Runtime Verification with Real-Time Properties: A Case Study, Christian Colombo, Gordon J. Pace and Patrick Abela, internal report 01-WICT-2009, Malta.

   Pace reviewed the paper several times.

References

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

[2] A. V. Aho. Indexed grammars — an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, 1968.

[3] A. V. Aho. Nested stack automata. *Journal of the ACM*, 16:383–406, July 1969.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *International Colloquium on Automata, Languages and Programming (ICAL)*, pages 169–178. Springer, 1999.

[6] J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, 2003.

[7] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services v1.1, 2003. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf (Last accessed: 2010-02-17).

[8] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani. PAWS: A framework for executing adaptive web-service processes. *IEEE Software*, 24:39–46, 2007.

[9] 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).

[10] M. Arnold, M. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. *SIGPLAN Notes*, 43:143–162, 2008.

[11] L. Baresi and S. Guinea. Self-supervising BPEL processes. *IEEE Transactions on Software Engineering*, 37(2):247–263, 2011.

[12] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *First International Conference, RV 2010*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.

[13] H. Barringer, A. Groce, K. Havelund, and M. Smith. An entry point for formal methods: Specification and analysis of event logs. In *Formal Methods in Aerospace (FMA)*, volume 6 of *Electronic Proceedings in Theoretical Computer Science*, pages 16–21, 2009.

[14] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster. Adaptive runtime verification. In *Runtime Verification (RV)*, Lecture Notes in Computer Science, 2012. to appear.

[15] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64, 2011.

[16] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[17] E. Bodden, F. Chen, and G. Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *Aspect-oriented software development (AOSD)*, pages 3–14. ACM, 2009.

[18] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with tracematches. In *Runtime Verification (RV)*, pages 22–37. Springer, 2007.

[19] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 525–549. Springer, 2007.

[20] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Principles of Programming Languages (POPL)*, pages 209–220. ACM, 2005.

[21] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Principles of Programming Languages (POPL)*, pages 209–220. ACM, 2005.

[22] R. Bruni, H. C. Melgratti, and U. Montanari. Nested commits for mobile calculi: Extending join. In *IFIP International Conference on Theoretical Computer Science*, pages 563–576. Kluwer, 2004.

[23] M. J. Butler and C. Ferreira. A process compensation language. In *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2000.

[24] M. J. Butler and C. 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.

[25] M. J. Butler, C. Ferreira, and M. Y. Ng. Precise modelling of compensating business transactions and its application to BPEL. *Journal of Universal Computer Science*, 11(5):712–743, 2005.

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

[27] M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2004.

[28] M. J. Butler and S. Ripon. Executable semantics for compensating CSP. In *European Performance Engineering Workshop (EPEW) / Web Services and Formal Methods (WS-FM)*, Lecture Notes in Computer Science, pages 243–256. Springer, 2005.

[29] F. Chang and J. Ren. Validating system properties exhibited in execution traces. In *Automated Software Engineering (ASE)*, pages 517–520. ACM, 2007.

[30] A. Charfi and M. Mezini. AO4BPEL: An aspect-oriented extension to BPEL. *World Wide Web*, 10(3):309–344, 2007.

[31] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, 2003.

[32] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *International Conference on Tools and Algorithms for the construction and analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 2005.

[33] F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. *SIGPLAN Notes*, 42(10):569–588, 2007.

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

[35] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.

[36] S. Colin and L. Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[37] C. Colombo, A. Gauci, and G. J. Pace. Larvastat: Monitoring of statistical properties. In *Runtime Verification (RV)*, volume 6418 of *Lecture Notes in Computer Science*, pages 480–484. Springer, 2010.

[38] C. Colombo and G. Pace. Recovery within long running transactions. *ACM Computing Surveys*, 45(3), 2013. To appear.

[39] C. Colombo and G. J. Pace. A compensating transaction example in twelve notations. Technical report, Department of Computer Science, University of Malta, 2011. Technical Report CS2011-01.

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

[41] C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE, 2009.

[42] M. Colombo, E. Di Nitto, and M. Mauri. Scene: a service composition execution environment supporting dynamic changes disciplined through rules. In *International conference on Service-Oriented Computing (ICSOC)*, pages 191–202. Springer, 2006.

[43] E. Cronin, A. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1):67–73, 2004.

[44] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. In *International Workshop on Dynamic Analysis (WODA)*, pages 1–7. ACM, 2005.

[45] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *International Symposium on Temporal Representation and Reasoning (TIME)*, pages 166–174. IEEE, 2005.

[46] C. T. Davies, Jr. Recovery semantics for a DB/DC system. In *ACM annual conference*, pages 136–141. ACM, 1973.

[47] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[48] M. B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Automated Software Engineering (ASE)*, pages 228–237. IEEE, 2008.

[49] C. Eisentraut and D. 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.

[50] A. Erradi, P. Maheshwari, and V. Tosic. Ws-policy based monitoring of composite web services. In *European Conference on Web Services (ECOWS)*, pages 99–108. IEEE, 2007.

[51] S. A. Ezust and G. V. Bochmann. An automatic trace analysis tool generator for estelle specifications. In *Applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 175–184. ACM, 1995.

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

[53] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems in the bip framework with formally-proved sound and complete instrumentation. *SOftware and SYstem Modeling (SOSYM)*. to appear.

[54] H. Garcia-Molina and K. Salem. Sagas. In *Management of Data (SIGMOD)*, pages 249–259. ACM, 1987.

[55] A. Gauci, A. Francalanza, and G. J. Pace. Distributed system contract monitoring. *Journal of Logic and Algebraic Programming (JLAP)*, 2013. to appear.

[56] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases (VLDB)*, pages 144–154. IEEE, 1981.

[57] P. Greenfield, A. Fekete, J. Jang, and D. Kuo. Compensation is not enough. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 232–239. IEEE, 2003.

[58] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response service invocations. In *Application of Concurrency to System Design (ACSD)*, pages 190–198. IEEE, 2008.

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

[60] S. Guinea, G. Kecskemeti, A. Marconi, and B. Wetzstein. Multi-layered monitoring and adaptation. In *International Conference on Service-Oriented Computing (ICSOC)*, volume 7084 of *Lecture Notes in Computer Science*, pages 359–373. Springer, 2011.

[61] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.

[62] Y. He, L. Zhao, Z. Wu, and F. Li. Formal modeling of transaction behavior in WS-BPEL. In *Computer Science and Software Engineering (CSSE)*, pages 490–494. IEEE, 2008.

[63] T. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[64] J. Hughes. Quickcheck testing for fun and profit. In *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007.

[65] D. Jefferson. Virtual time. In *International Conference on Parallel Processing (ICPP)*, pages 384–394. IEEE, 1983.

[66] G. Kiczales. Aspect-oriented programming. In *Software Engineering (ICSE)*, page 313. ACM, 2005.

[67] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[68] I. Lanese, C. Vaz, and C. Ferreira. On the expressive power of primitives for compensation handling. In *Programming Languages and Systems (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 366–386. Springer, 2010.

[69] I. Lanese and G. Zavattaro. Programming sagas in SOCK. In *Software Engineering and Formal Methods (SEFM)*, pages 189–198. IEEE, 2009.

[70] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

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

[72] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. Design and verification of long-running transactions in a timed framework. *Science of Computer Programming*, 73:76–94, 2008.

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

[74] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2008. http://rap.dsi.unifi.it/cows.

[75] A. Lapadula, R. Pugliese, and F. 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.

[76] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Automated Software Engineering (ASE)*, pages 417–420. ACM, 2007.

[77] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[78] J. Li, H. Zhu, and J. He. Algebraic semantics for compensable transactions. In *Theoretical Aspects of Computing (ICTAC)*, volume 4711 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2007.

[79] J. Li, H. Zhu, G. Pu, and J. He. Looking into compensable transactions. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0:154–166, 2007.

[80] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.

[81] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.

[82] M. Mazzara and S. Govoni. A case study of web services orchestration. In *Coordination Models and Languages (COORDINATION)*, volume 3454 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[83] P. M. Melliar-Smith and B. Randell. Software reliability: The role of programmed exception handling. *ACM Software Engineering Notes*, 2:95–100, 1977.

[84] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 14:249–289, 2012.

[85] O. Moser, F. Rosenberg, and S. Dustdar. Event driven monitoring for service composition infrastructures. In *International Conference on Web Information Systems Engineering (WISE)*, pages 38–51. Springer, 2010.

[86] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Runtime Verification (RV)*, volume 7186 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2012.

[87] S. Nepal, A. Fekete, P. Greenfield, J. Jang, D. Kuo, and T. 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.

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

[89] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57. IEEE, 1977.

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

[91] G. Roşu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, RIACS, 2001.

[92] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering (ASE)*, 12(2):151–197, 2005.

[93] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering (ASE)*, 12(2):151–197, 2005.

[94] U. Sammapun, A. Easwaran, I. Lee, and O. Sokolsky. Simulation of simultaneous events in regular expressions for run-time verification. *Electronic Notes in Theoretical Computer Science*, 113:123–143, 2005.

[95] M. Schäfer, P. Dolog, and W. Nejdl. Engineering compensations in web service environment. In *International Conference on Web Engineering (ICWE)*, pages 32–46. Springer, 2007.

[96] N. A. Tahamtan and W.-D. team. WS-DIAMOND: Web services - DIAgnosability, MONitoring and Diagnosis. In *E. di Nitto, A. Sassen, P.Traverso and A. Zwegers (Eds), At your service, Chapter 9, MIT Press*, page Chapter 9. MIT Press, May 2007.

[97] C. Vaz, C. Ferreira, and A. Ravara. Dynamic recovering of long running transactions. *Trustworthy Global Computing*, 5474:201–215, 2009.

[98] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10:167–195, 1978.

[99] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[100] L. Zeng, H. Lei, J.-J. Jeng, J.-Y. Chung, and B. Benatallah. Policy-driven exception-management for composite web services. In *Conference on E-Commerce (CEC)*, pages 355–363. IEEE, 2005.