IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Formal Specification, Verification and Analysis of Long-running Transactions

PhD Program in Computer Science and Engineering

XXIV Cycle

By

Anne Kersten Kauer

2013

The dissertation of Anne Kersten Kauer is approved.

Program Coordinator: Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Supervisor: Roberto Bruni, University of Pisa

Supervisor: Carla Ferreira, Universidade Nova de Lisboa

Tutor: Maria Grazia Buscemi, Alberto Lluch Lafuente, IMT Institute for Advanced Studies, Lucca

The dissertation of Anne Kersten Kauer has been reviewed by:

Marco Carbone, IT University of Copenhagen

Hernán Melgratti, University of Buenos Aires

Irek Ulidowski, University of Leicester

IMT Institute for Advanced Studies, Lucca

2013

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor Roberto Bruni for his constant support, guidance and encouragement throughout the last four years. Discussions with him have always been a great supply for new ideas and problem solutions. I would also like to thank Carla Ferreira for giving me the opportunity to come to Lisbon and for exploring with me the world of dynamic logic. Thanks to both of them, as well as to Ivan Lanese and Giorgio Spagnolo for their help and work in the papers we published together. Thanks to my tutors Alberto Lluch Lafuente and Marzia Buscemi, and the other professors at IMT as well as to Hugo Torres Vieira in Lisbon for their support. Moreover I want to thank the reviewers for their good suggestions and comments.

I am grateful to my friends and colleagues at IMT and in Dresden for coffee, drinks, discussions, music, games and just being there. Thanks to my family for their support, and last, but never least to Bernhard and Bruno, without you I would not have made it.

Vita

2011 **ERASMUS period**

Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Departamento de Informática

Since 2009 **PhD student in Computer Science and Engineering**

IMT Institute for Advanced Studies Lucca, Italy

Thesis: Formal Analysis, Specification and Verification of Long-Running Transactions

Supervisor: Roberto Bruni (University of Pisa)

Courses attended on Formal Methods, Semistructured Databases, Global Computing, Web Algorithmics, Information Theory, Software Engineering and Advanced Networking Architectures

2002 to 2008 **Diploma (MSc) in Computer Science**

Technical University of Dresden, Germany

Thesis: A monad-based approach to the semantics of Concurrent Haskell

2000 to 2002 **Abitur (A-Levels)**

Gymnasium Dresden Plauen

Publications

1. Roberto Bruni, Anne Kersten, Ivan Lanese, and Giorgio Spagnolo, A New Strategy for Distributed Compensations with Interruption in Long-Running Transactions, In *WADT*, pages 42 – 60, 2010
2. Roberto Bruni, Carla Ferreira, and Anne Kersten Kauer, First-order dynamic logic for compensable processes. In *COORDINATION*, pages 104 – 121, 2012
3. Roberto Bruni and Anne Kersten Kauer. LTS Semantics for Compensation-based Processes. In *TGC*, LNCS, pages 112 – 128, 2012

Presentations

1. A. Kersten, "On the Semantics of Distributed Compensations with Interruption," at *20th International Workshop on Algebraic Development Techniques*, 2010.
2. A. Kersten, "On the Semantics of Distributed Compensations with Interruption," at *Universidade Nova de Lisboa*, Lisbon, Portugal, 2011.

Abstract

Compensation-based long-running transactions (LRTs) are the main error recovery mechanism in business process modelling languages. Correctly implementing LRTs is difficult, especially in a concurrent setting. To ease this task we are developing a full-fledged formal approach to the description, design, analysis and verification of long-running transactions.

The existing calculi Sagas and compensating CSP rely on different compensation policies regarding concurrent processes. Unfortunately they either require synchronization before compensating or they include unrealistic traces. We therefore propose a new policy that improves existing ones using realistic distributed compensations.

In this thesis we formalize the behaviour of the new policy using three semantics: i) a denotational semantics to compare it with previous policies, ii) an operational semantic based on an encoding into Petri nets as a foundation for richer semantic domains that record causal dependencies between events, iii) an easily extendable small-step SOS semantics, that facilitates model checking.

We prove the correspondence between the different semantics showing their observational equivalence. Moreover we develop a tool for each of the semantics in Maude to improve and validate our theory.

Finally, we introduce a logical framework to model and analyse LRTs based on dynamic logic. We use it to derive sufficient conditions under which a compensable program is guaranteed to restore a correct state after a fault.

Chapter 1

Introduction

1.1 Motivation

In the early days of the Internet, web sites used to be static and their implementation rather simple. With the huge increase of users as well as more and more additional requirements to web applications, new concepts and techniques for scalability as well as partitioning had to be found. While previously a few hundred people had access to web pages, now several billion people use the internet every day. Moreover emerging paradigms like cloud computing or Web 2.0 with large online communities and user generated content call for a more dynamic and interactive approach.

One of the most prominent solutions, that has been put in place, are web services, an instance of *service-oriented computing*. Service-oriented computing comprises different techniques and concepts to create modular software solutions. A service provides a defined functionality and is publicly available, as well as its semantic description. Services are designed and implemented independently and can be combined to compose larger applications.

Services are loosely coupled, *i.e.*, there is no strong connection between components. Moreover in such an open environment like the Internet, communication may be unreliable. This may lead to application

Atomicity A transaction's changes to the state are atomic: either all happen or none happen.

Consistency A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

Durability Once a transaction completes successfully (commits), its changes to the state survive failures.

Figure 1: ACID properties for transactions taken from [GR92]

faults that are hard to debug. Therefore we need different fault handling mechanisms to guarantee a correct execution of complex workflows. One such mechanism are long-running transactions (henceforth abbreviated with LRTs) that, as the name says, are transactions that take a long amount of time. While classic database transactions use locking and rollback for error recovery, such mechanisms are not feasible for LRTs due to the fact that the usage of common resources is long lasting. For these reasons it is difficult to guarantee the usual ACID properties (see Figure 1) for this kind of transactions. The concept of compensation has been introduced to provide a flexible solution to this problem. The idea is that LRTs are divided into smaller subtransactions, each associated with a compensation. The compensation is installed once the subtransaction commits and used to undo the action in case of a later abort of the LRT. Compensations are executed in the reverse order of installation. Note that, in many cases they cannot recover the original state, *e.g.* undoing a message send operation is not possible, but it can be compensated by sending, *e.g.*, a new message. Thus reaching a state similar to the starting one is usually sufficient. Compensation-based LRTs are today's de facto standard for business process modelling languages.

We are interested in developing a *full-fledged formal approach to the de-*

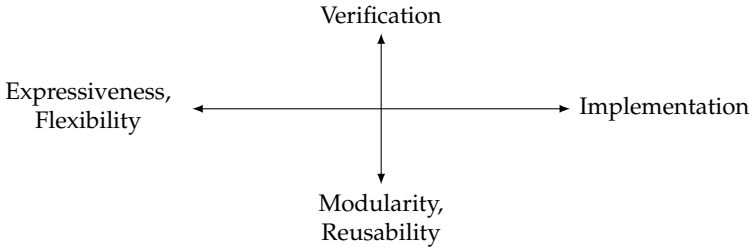


Figure 2: Requirements for the right level of abstraction

scription, design, analysis and verification of long-running transactions. There exists a variety of approaches trying to model LRTs with compensations, however (so far) there is no universally accepted approach to represent LRT core primitives. To contribute towards a widely agreed theory for LRTs is the general aim of this thesis.

1.2 Objectives

There exists a variety of approaches modelling LRTs. There are calculi based on different languages, some have an implementation, others offer a type system, or define a notion of equivalence or correctness. However, as we want to point out again, none of them is still widely agreed to be the best option to model LRTs.

The general aim of this thesis is to define a *suitable process algebraic abstraction for LRTs* that possibly *improves or extends existing proposals*. But what do we mean with improving? Some requirements are given in Figure 2. As we can see there are four aspects:

Expressiveness More expressiveness allows us to represent more complex problems, but can also introduce *e.g.* nondeterminism or infinite behaviour. We consider computational and representational expressiveness. Computational expressiveness regards how much a calculus can express from a mathematical point of view, while representational expressiveness considers how natural the calculus can model common problems. A calculus with a high com-

putational expressiveness can encode different kinds of features, but then it is difficult to reason about such features. Moreover a large representational distance may lead to the necessity to encode a property for verification. But then the source of a problem may actually be the encoding, instead of a problem in the system. On the other hand a model where the analysis can be conducted efficiently can be not expressive enough.

A mutual encoding can demonstrate that two calculi are equally expressive or that one is strictly more expressive than the other by giving a counter example. However such an encoding has to be semantics preserving. Thus one can transfer or rule out properties of other calculi.

Modularity A higher level of modularity allows us to easily extend the calculus or exchange small parts. That way we can reuse parts of the calculus. Moreover by partitioning the calculus the formal analysis can be more efficient by considering smaller parts. Following this approach we may define a calculus that is correct by construction.

Implementation An implementation of the calculus provides an environment to assess the validity of a theory. It can be used to check the correctness of the system and improve the theory, especially if problems become more complex leading to a combinatorial explosion, that cannot be handled without tool support. It provides a platform to verify properties, find counter examples and implement case studies.

Verification There exist different verification methods, among them logics, equivalences or types. But in each case the aim is to show that the calculus can adhere to some rules, fulfil a few guarantees or satisfy a certain property or correctness criteria.

While it may be easy to achieve one or two properties, the difficulty is to keep a balance between all four different ones. More expressiveness usually leads to more complexity which makes verification more

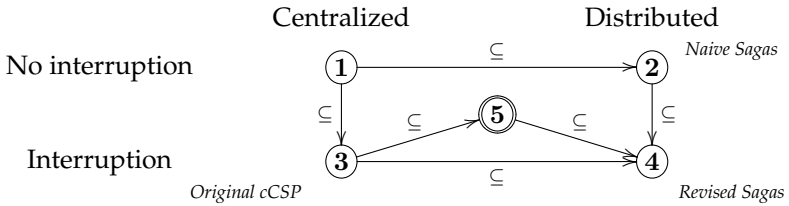


Figure 3: Compensation policies (arrows stand for trace inclusion)

difficult. On the other hand well defined verification methods may complicate an implementation or reduce the level of modularity. Another aspect is to provide an abstraction that can be used by the stakeholders. For example LRTs are largely used in business processes, where workflow like notations are preferred to process algebraic ones.

1.3 Results

In this section we summarize the results accomplished towards our general aim of developing a formal approach towards the description, design, analysis and verification of long-running transactions. Our research focuses on the semantics of two workflow based calculi, namely Sagas [BMM05] and compensating CSP [BHF04] (cCSP). They model LRTs on a level high enough to be independent of specific interaction mechanisms. Thus they are closer to both the Business Process Modelling Notation (BPMN) and the Business Process Execution Language (BPEL), the standard for implementing web services.

The core fragment of Sagas has been sufficient to characterise different compensation policies for parallel processes. A thorough analysis is presented in [BBF+05] by comparing the Sagas calculus with cCSP along two axes of classification: i) interruption of siblings in case of an abort (interruption versus no interruption) and ii) whether compensations are started at the same time or siblings can start their compensation on their own (centralised versus distributed). Combing the two dimensions led

to four different policies as displayed in Figure 3.

As it turns out, none of the four originally defined semantics is entirely satisfactory. Policies one to three are too restrictive: they miss the possibility to stop a sibling branch and to activate compensations as soon as possible, because typically activities and compensations have a cost. Policies missing interruption (cases one and two) can have sibling branches that finish their execution anyway, even though they will have to compensate. In case three, branches might have to wait until they are allowed to continue together with their siblings. The second and fourth policies on the other hand are unrealistic: they include an oracle mechanism where a branch may start its compensation even though the error has not occurred yet. An optimal, realistic semantics should be more "permissive" (*i.e.*, allowing more traces) than policy three but less than four.

The first part of this thesis is concerned with the conceptual modelling of transactions, where some general abstract properties can be proven that are independent from the actual nature of the activities involved in the workflow. In this first part of the thesis, we introduce a new policy for parallel Sagas with interruption. The new policy is "optimal" in the following sense. It guarantees that distributed compensations may only be started after an error actually occurred, but compensations can start as soon as possible.

Furthermore we present a denotational and two operational semantics modelling the new policy. The denotational semantics extends the existing trace semantics of cCSP. This allows us to formally compare our policy to the previously defined ones.

The first operational semantics is based on an encoding into Petri nets. This provides a graphical presentation based on a well-known model of concurrency, thus paving the way to the straightforward derivation of richer semantic domains than traces. Such domains allow to record causal dependencies between events. This has practical consequences in the tracking of the faults that triggered a compensation. Thus it is more informative than the denotational semantics, because it accounts for the branching of processes arising from the propagation of

interrupts.

The second operational semantics is a small-step semantics based on labelled transition systems. It is easier to parse than the Petri net model due to the sophisticated mechanism needed for handling interrupts. While in the denotational and the Petri net semantics we focused on other aspects the small-step semantics can be extended with more complex operators for processes including choice or iteration adding more expressiveness.

We were able to prove the correspondence between the different semantics showing their observational equivalence. Moreover we developed a tool for each of the semantics to improve and validate our theory.

The second and last part of the thesis is concerned with an actual programming instance of the previously studied framework. While we focused in the first chapters mostly on modularity and implementation of our calculus this second part of the thesis is concerned with verification. We introduce a logical framework based on dynamic logic to show properties regarding programs including compensations and LRTs. Such compensable programs cannot only be composed in sequence and in parallel, but we allow as well choice and iteration. We replace the abstract activities used in the rest of the thesis with concrete assignments over shared memory variables. The main result establishes some sufficient conditions under which a compensable program is guaranteed to always restore a correct state after a fault.

1.4 Structure

In Chapter 2 we present related work regarding calculi for LRTs. We give an in-depth description of the workflow based calculi Sagas and cCSP. Starting with a shared syntax for both we present their two semantics.

In Chapter 3 we first summarize the comparison of Sagas versus cCSP from Bruni *et al.* [BBF+05]. From this relation we are able to deduce a *new policy* for handling compensations in concurrent programs that *improves existing ones* by allowing the activation of compensations autonomously, but only after an actual error occured. We define its denotational se-

mantics and show its formal relation to the previously defined policies. Moreover we present tool support for the denotational semantics.

Chapter 4 introduces an operational semantics for the newly defined policy based on an encoding into Petri nets. After giving some background on Petri nets we present the encoding of Sagas processes and show its *correspondence to the denotational semantics*. We prove that the encoding satisfies some high-level properties and extend it to represent other policies. The last section shows an implementation of the encoding.

In Chapter 5 we present a small-step operational semantics based on labelled transition systems. We show its *observational equivalence* to the previously defined semantics using a weak bisimulation. The chapter includes as well possible presentations of other policies, syntactic extensions and tool support.

Chapter 6 presents a *dynamic logic for long-running transactions*. We first recall some background on dynamic logic and then introduce our own logic adding concurrency and compensations. We define serializability for programs, *i.e.*, a concurrent execution of a program is equivalent to some serial execution in the sense that the first and the last state are equivalent. Using serializability we are able to prove properties for the logic, among them a sufficient condition under which a compensable program is *guaranteed to have a correct compensation*. The chapter describes tool support for the logic as well as how interruption can be added.

Running Example In the following chapters we will use a running example to give a better understanding of how LRTs and compensations work in the presented models. Figure 4 shows a simple graphical presentation of the respective workflow modelling the booking of trip. The outer box refers to the transaction scope. Each of the smaller boxes corresponds to a subtransaction where the forward action is above the dashed line and its compensation below. The control flow is represented by connecting arrows with the direction giving the order of the forward execution.

In this example a trip is booked to visit an event like a concert or a football match in another city. First the transaction reserves the ticket for

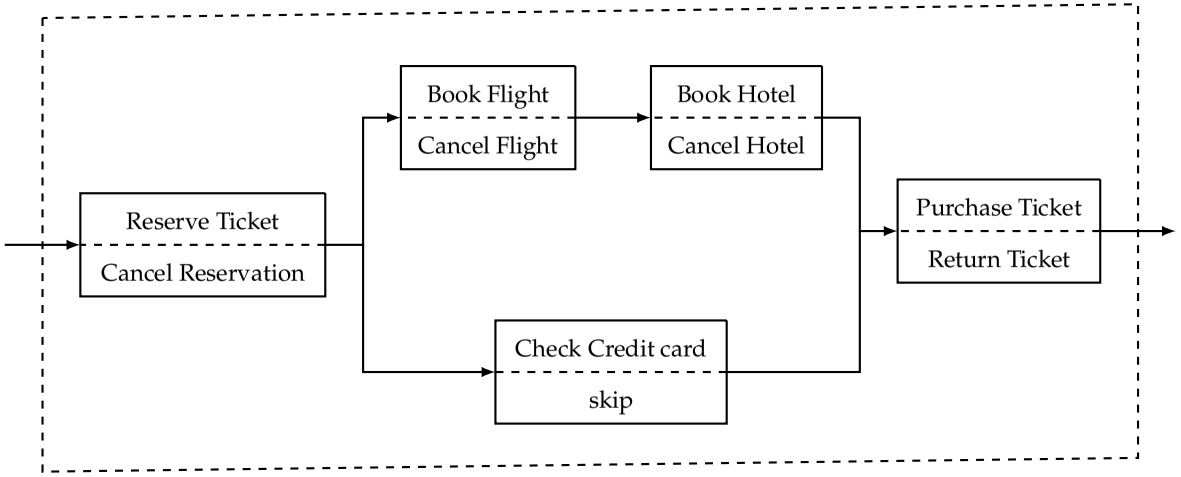


Figure 4: Example of a workflow for a LRT

the event. Then there is a parallel composition where in one branch a flight and a hotel are booked and in the other branch the credit card is checked. After the concurrent execution the ticket is purchased. Each action has a compensating activity. For the ticket purchase the ticket will be returned. For both the booking of the hotel and the flight the respective booking is cancelled. Equally for the ticket reservation the reservation is cancelled. For the credit card check the compensation is empty.

1.5 Origin

Though parts of this thesis are original to this work some have already been published by the author. We summarize here the contents of published articles:

- [BKLS10] presents the deduction of the new policy, its denotational semantics as well as its encoding into Petri nets and shows the correspondence of the two semantics.
- [BK12] presents the small-step semantics with the proof of correspondence. It shows representations of other policies as well as possible extensions.
- [BFK12] introduces the dynamic logic for compensable programs and shows the conditions under which a compensable program has a correct compensation.

Chapter 2

Background

In this chapter we describe previous work in the formal modelling of long-running transactions (henceforth LRTs). We give a brief overview of the literature and then focus on the process calculi Sagas [BMM05] and compensating CSP (cCSP) [BHF04].

We define a syntax derived from both Sagas and cCSP, that we also use in the following chapters. Next we present the different semantics, the big-step operational one for Sagas and the denotational semantics for cCSP.

The last section presents the language Maude with its underlying theory called rewriting logic. We will use Maude to implement the models presented in this thesis.

2.1 Existing calculi modelling long-running transactions

For a formal specification and analysis we need a model of concurrency. A natural choice is to use process calculi as their theory is well studied, including semantics, behavioural equivalences, type systems and logics to verify properties. Another advantage is the algebraic structure of processes, that favours compositional specifications and their illustration to non-experts. Indeed most of the existing formal approaches to LRTs use

process calculi. A few of them extend the π-calculus [Mil99], while others are inspired by process calculi like CSP [Hoa85] or CCS [Mil89]. We give here a short summary of existing approaches to model LRTs using process calculi and how they relate together.

LRTs were first mentioned in the context of database theory. In [GMS87] the authors introduce the problem of transactions, that take a long amount of time. They call such transactions sagas. A saga runs over a long period of time, therefore it cannot lock its database accesses as it usually would imply blocking other transactions for too long. The classical locking and rollback mechanisms for error recovery are not suitable. Instead a saga consists of several atomic and isolated subtransactions. Each of these subtransactions is paired with a compensation which can be executed in case of a later fault. The compensation will in a way undo (*i.e.*, compensate) the corresponding forward action, leading the system back to a consistent state.

The choice to transfer these ideas to service-oriented computing seems natural. Services are combined in a transactional manner to ensure the consistency of the system. However, the heterogeneity as well as the inherent loose coupling of services cannot guarantee the classical ACID properties of transactions (Atomicity, Consistency, Isolation, Durability). In this setting the use of LRTs and compensations for error recovery is a simple and straightforward solution.

The first attempts to combine the concepts of atomicity of actions and process calculi were [Gla90] and [GMM88, GMM90] that extend CCS with atomic actions.

There are two main approaches for modelling LRTs in service-oriented computing. The first one abstracts away from the actual implemention of services and focuses more on the control flow between components. One of the first languages to take this approach is StAC [BF00, CGV$^+$02, BF04]. Its novel building block is a compensation pair consisting of two processes. As the computation progresses compensations are piled in a stack such that in case of an error they can be executed in reverse order. Note that in a parallel composition each branch has its own stack. The language provides special primitives for the commit and abort of a trans-

action. If the transaction commits the compensations are discarded. If it aborts compensations are activated. Its operational semantics is given in a richer intermediate language called StAC$_i$. However the syntax included a too large set of primitives and its semantics became soon very complex, having to deal with spurious aspects.

Two refinements of StAC were developed in parallel. One called compensating CSP (cCSP) [BHF04] extends the standard CSP calculus [Hoa85] with a transaction scope and compensations. While StAC has a rich syntax and a rather intricate operational semantics, cCSP comes with a simpler syntax and a denotational semantics. Moreover it adds a formal correctness criteria using a cancellation semantics. The other one, the process calculus Sagas [BMM05] is also inspired by StAC, moreover it transfers the ideas of a saga in databases from [GMS87] to service-oriented computing. It provides a big-step operational semantics. In this chapter we will give a detailed description of both Sagas and cCSP as a starting point for the work presented in this thesis.

There are several articles extending Sagas or cCSP. In [LZ09] an encoding of Sagas in SOCK [GLG$^+$06] is given together with a new semantics that is closer to the usage of compensations in practice. In [Lan10] dynamic compensations are added to the basic Sagas calculus. A small-step semantics for cCSP is introduced in [BR05], while [RB09] implements this semantics in PVS. In [CLW12] an extension of cCSP including recursion is presented, this is handled in a failure-divergence semantics. The paper [FM07] defines a calculus very similar to Sagas and cCSP with set consistency as a notion of correctness. As we explain later, Sagas has also been applied to the formalization of web-service standards [ES08].

A different approach compared to these workflow based calculi are languages modelling the communication between components. Such approaches are often based on a notion of name mobility and hence they extend the π-calculus [Mil99]. The first ones to extend it with transactions were Berger and Honda [BH00]. In order to model the two-phase commit protocol they introduce primitives for message loss, timers, process failure and persistance. The calculus webπ [LZ05] extends the asynchronous π-calculus with timed transaction and static compensations. It

does not support nesting of transactions. A fragment of webπ called webπ_∞ [ML06] discards the constriction regarding time.

Another language focusing on the interaction between services is Committed Join (cJoin) [BMM04b, BMM04a, BMM11] that extends the Join calculus [FG96]. Compensations are staticly defined, the nesting of transactions is allowed. Furthermore different transactions may be joined in order to communicate between different partners in a negotiation.

The calculus committed ccpi [BM07a] combines the ideas of cJoin with ccpi [BM07b]. The latter adds primitives from concurrent constraint programming to the π-calculus in order to model service level agreements.

In [VFR08] the authors present the calculus dcπ. It introduces dynamic recovery upon input. Each input operation is associated with a compensating process. If the input is executed (i.e. the process received an output on the respective channel) the new compensation is installed in parallel to the existing one. Transactions have a scope with a special action. If the action is executed (within the transaction or from the outside) the transaction is aborted and the gathered compensations are activated. Compensations may also be predefined inside a transaction before any execution. The paper also defines a type system for dcπ.

Though dcπ relaxes the static installation of compensations by putting them in parallel, in [LVF10] the idea of dynamic recovery is fully accomplished. Compensations are updated with the help of λ-calculus like functions. Encodings for both static and parallel in dynamic recovery are given. This shows that dynamic recovery is strictly more powerful. The authors also define should-testing equivalence and weak bisimulation. Another similar calculus is introduced in [VF09] together with a correctness criteria. In [LZ13] the termination of a simple extension of the π-calculus regarding static, parallel or dynamic compensation is analysed. A similar line of research was already taken in [BZ09] where the termination of an extension of CCS regarding failure operators and replication or recursion is analysed. Not that there are a few works on exceptions for session types where models similar to the ones for compensations are proposed (*e.g.* [CHY08, Car09]).

The calculus TransCCS [dVKH10a, dVKH10b] extends CCS with transactions. It provides instead of compensations for each single action an overall compensation for a transaction. It is thus able to model automatic error recovery.

Furthermore there are several calculi that try to formalize the Web Services Business Process Execution Language (WS-BPEL). WS-BPEL is nowadays the standard for specifying interactions between web services. It was developed from IBM's WSFL (Web Services Flow Language) and Microsoft's XLANG by putting their concepts together. The full specification of WS-BPEL can be found under [BPE13]. Among the most known BPEL engines there are ActiveBPEL [Act13], Apache ODE [Apa13] and the Oracle BPEL Process Manager [Ora13].

In [PZQ+06] the authors introduce BPEL0, a subset of WS-BPEL including LRTs and compensations. It provides a notion of equivalence based on bisimulation. The article [LPT08] presents B*lite*, it allows partner links, process termination, message correlation and LRTs with compensation handlers. Using their specification they do experiments with three existing WS-BPEL engines showing that they differ among each other as well as from the official WS-BPEL specification. The BPEL$_{fct}$ calculus [ES08] formalizes fault, compensation and termination handling in WS-BPEL. An encoding of Sagas into BPEL$_{fct}$ is also presented.

Moreover there are so-called service oriented calculi, that try to model service-oriented computing directly. They provide low-level primitives to handle faults of various kinds, including compensations. We already mentioned SOCK [GLG+06, GLMZ09]. The language Jolie [Jol13] is an implementation based on it. COWS [LPT07] introduces a kill primitive, that terminates transactions of a given name, and the protection block, protecting processes from being terminated from the outside. The Conversation Calculus [VCS08] does not provide compensation handling directly, instead it uses a try-catch operator to handle errors. In [CFV08] it is shown that this is enough to model cCSP.

An overview of different calculi under the aspect of error and compensation handling is given in [FLR+11]. A good survey regarding LRTs and related topics is [CP13]. A table summarizing the different approaches

	Underlying Language	Compensation Definition	Nesting	Highlights	References
StAC		static	yes		[BF00]
cCSP	CSP	static	yes	cancellation semantics	[BHF04]
Sagas		static	yes		[BMM05]
Dynamic Sagas		dynamic	yes		[Lan10]
coreT		static	no	set consistency	[FM07]
webπ	asynch. π	static	no	includes time	[LZ05]
webπ_∞	asynch. π	static	no		[ML06]
dcπ	asynch. π	parallel	yes		[VFR08]
Generalized dcπ	synch. π	dynamic	yes		[LVF10]
cCalc	asynch. π	dynamic	yes	correctness mapping	[VF12]
cJoin	Join	static	yes	transaction merge	[BMM11]
Committed ccpi	synch. π	static	yes		[BM07a]
TransCCS	CCS	static	yes		[dVKH10a]
Conversation Calculus		-	yes	no compensation	[VCS08]
BPEL0	WS-BPEL	static	yes		[PZQ+06]
B*lite*	WS-BPEL	static	yes		[LPT08]
BPEL$_{fct}$	WS-BPEL	static	yes	encoding of Sagas	[ES08]
SOCK		dynamic	yes		[GLMZ09]
COWS		static	yes	protection block	[LPT07]

Figure 5: Comparison of the presented calculi

mentioned in this section is given in Figure 5.

Another approach worth mentioning regards instead of compensations the actual reversibility of processes. It was first modelled for CCS called RCCS in [DK04, DK05] and later extended in [DKS07]. [PU07] presents a more general approach. The authors include reversibility in languages with SOS-semantics (Structural Operational Semantics) and use CCS as an example. More recently the idea has been applied to the higher order π-calculus as $\rho\pi$ in [LMS10, LMSS11, LMS12]. In the above presented calculi this can be realized with a perfect undo as compensation.

In the line of reversibility we also consider [ABDZ07] that investigates software transactional memory. It adds classic transactions to CCS that are executed in an optimistic fashion. In case of a concurrent update on the shared memory the transaction is rolled back to its initial state.

2.2 Syntax

This section presents the core syntax that we use in the following chapters. It combines the main operators of Sagas and cCSP.

Definition 1. *The set of all parallel sagas is defined by the grammar:*

$$
\begin{array}{lllll}
(\text{ACT}) & A, B & ::= & a \mid \mathit{skip} \mid \mathit{throw} \\
(\text{STEP}) & X & ::= & A \div B \\
(\text{PROCESS}) & P, Q & ::= & X \mid P; Q \mid P|Q \\
(\text{SAGA}) & S, T & ::= & A \mid S; T \mid S|T \mid \{\!|P|\!\}
\end{array}
$$

Atomic actions A include generic activities $a \in \mathcal{A}$, where $\mathcal{A}$ is an infinite set of activities, the vacuous activity skip and the always faulty activity throw. In a compensation pair $A \div B$, the activity B compensates A. We write skipp for $\mathit{skip} \div \mathit{skip}$ and throww for $\mathit{throw} \div \mathit{skip}$. A (compensable) process is a compensation pair or the sequential or parallel composition of processes. We use $\{\!|P|\!\}$ to enclose a compensable process within a transaction. A transaction or saga can be composed sequentially and in parallel.

We sometimes refer to sequential Sagas when we consider this syntax without parallel composition. Moreover in the following chapters we include additional elements like choice or iteration in advanced stages.

Example 1. *We use the syntax of Definition 1 to represent the transaction of Figure 4. Abbreviating actions using initials, the saga in the example can be written as:*

$$\{\![\mathsf{rT} \div \mathsf{cR}\ ;\ ((\mathsf{bF} \div \mathsf{cF}\ ;\ \mathsf{bH} \div \mathsf{cH}) \mid \mathsf{cC} \div \mathit{skip})\ ;\ \mathsf{pT} \div \mathsf{retT}]\!\}$$

2.3 Semantics for Sequential Sagas

The following sections present the operational semantics of Sagas taken from [BMM05]. We start with the sequential part of Sagas.

A saga consists of (several) atomic actions. Each of these actions is associated with a compensation. If an action fails, then the previously installed compensations are activated. Note that if a subtransaction fails, nothing is executed, thus no compensation is necessary. Likewise a LRT is executed in an all-or-nothing fashion, *i.e.*, either everything is executed leading to success or nothing is done and a failure is returned. However in this setting also compensations may fail, which leads to an inconsistent state where the transaction was only partially executed. We say that the saga terminates abnormally or less formally it crashes. The system needs the information of success, failure or crash for any continuation or concurrent execution. Likewise the semantics has to keep track of this status, to start compensations, stop the execution or inform parallel branches. Therefore we introduce some additional notation. We define a set $\mathcal{R} = \{\boxdot, \boxtimes, \circledast\}$ for the three possible result states of a compensable program. The symbol $\boxdot$ stands for a commit or successful execution, $\boxtimes$ for abort and $\circledast$ for a crash, $\square$ ranges over $\mathcal{R}$.

Moreover single actions may either succeed or fail, an information we provide with a context Γ.

Definition 2 (Context)**.** *We call a function $\Gamma : \mathcal{A} \to \{\boxdot, \boxtimes\}$ a context that assigns to each activity its result, where $\boxdot$ stands for success and $\boxtimes$ for failure.*

Note that a single action (or subtransaction as we recall from [GMS87]) cannot terminate abnormally. Either it will be executed completely with success or nothing is done and the system returns a failure. Without loss of generality we assume that each activity is named differently. Thus a context has the form $\Gamma = a_1 \mapsto \Box_1, \ldots, a_n \mapsto \Box_n$, where $a_i \neq a_j$ for all $i \neq j$ and $'$,$'$ stands for the disjoint union.

We use the following equivalences as a notational convention to reduce the number of rules:

$$A \div \mathit{skip} \equiv A \qquad \text{(Null compensation)}$$
$$\mathit{skip}; P \equiv P; \mathit{skip} \equiv P \qquad \text{(Null process)}$$
$$(P; Q); R \equiv P; (Q; R) \qquad \text{(Associativity of Sequential Composition)}$$

We say that the activity skip as a compensation is equivalent to having no compensation. Thus basic activities can be very easily integrated in compensable programs. Moreover the action skip can be written as the left and right identity for sequential composition. The last equation defines the associativity of the sequential composition.

We define the semantics of sequential Sagas modulo structural congruence as the relation $\Gamma \vdash S \xrightarrow{\alpha} \Box$ given by the inference rules in Figure 6. The symbol Γ denotes a context, S a saga and $\Box$ its result after complete execution under the context Γ. The label α denotes a process without compensations. It represents the order of execution when executing S under Γ.

Furthermore we define an auxiliary relation $\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle$ that describes the semantics of a compensable process P under a certain context Γ. The labels β and β' denote the installed compensations before and after the execution of P, i.e. they are both processes that do not contain compensations themselves. In the following we describe the different rules of Figure 6.

Rule (SKIP) describes the simple case of the vacuous activity skip. It always succeeds without changing the compensations.

For a compensation pair $A \div B$ we consider three cases. The first differentiation depends on the execution of A according to the context Γ. In the first case, rule (S-ACT), activity A commits and installs the compensation B in front of the existing compensation process β. Thus in case of a

(SKIP)

$$\Gamma \vdash \langle \mathit{skip}, \beta \rangle \xrightarrow{\mathit{skip}} \langle \boxdot, \beta \rangle$$

(S-ACT)

$$A \mapsto \boxdot, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{A} \langle \boxdot, B; \beta \rangle$$

(S-CMP)

$$\frac{\Gamma \vdash \langle \beta, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \boxdot, \mathit{skip} \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, \mathit{skip} \rangle}$$

(F-CMP)

$$\frac{\Gamma \vdash \langle \beta, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \boxtimes, \mathit{skip} \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \circledast, \mathit{skip} \rangle}$$

(S-STEP)

$$\frac{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \boxdot, \beta'' \rangle \quad \Gamma \vdash \langle Q, \beta'' \rangle \xrightarrow{\alpha'} \langle \square, \beta' \rangle}{\Gamma \vdash \langle P; Q, \beta \rangle \xrightarrow{\alpha;\alpha'} \langle \square, \beta' \rangle}$$

(A-STEP)

$$\frac{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \square, \mathit{skip} \rangle}{\Gamma \vdash \langle P; Q, \beta \rangle \xrightarrow{\alpha} \langle \square, \mathit{skip} \rangle} \qquad \square \in \{\boxtimes, \circledast\}$$

Figure 6: Semantics of sequential Sagas.

(SAGA)

$$\frac{\Gamma \vdash \langle P, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \square_1, \beta \rangle}{\Gamma \vdash \{\!|P|\!\} \xrightarrow{\alpha} \square_2} \qquad \square_2 = \begin{cases} \boxtimes & \text{if } \square_1 = \circledast \\ \boxdot & \text{if } \square_1 = \boxdot \vee \square_1 = \boxtimes \end{cases}$$

(SAGA-S-ACT)

$$A \mapsto \boxdot, \Gamma \vdash A \xrightarrow{A} \boxdot$$

(SAGA-A-ACT)

$$A \mapsto \boxtimes, \Gamma \vdash A \xrightarrow{\mathit{skip}} \boxtimes$$

(SAGA-S-STEP)

$$\frac{\Gamma \vdash S \xrightarrow{\alpha} \boxdot \quad \Gamma \vdash T \xrightarrow{\alpha'} \square}{\Gamma \vdash S; T \xrightarrow{\alpha;\alpha'} \square}$$

(SAGA-A-STEP)

$$\frac{\Gamma \vdash S \xrightarrow{\alpha} \boxtimes}{\Gamma \vdash S; T \xrightarrow{\alpha} \boxtimes}$$

Figure 7: Rules for sagas

later abort the last added compensation is always executed first. In other words, compensations are executed backwards, in the reverse order of installation.

In the other two cases the activity A fails according to Γ. Remember as A is not executed the compensating activity B is discarded. However as there was a failure the previously installed compensation β must be executed. We differentiate the two cases where either β is successful or β fails. In the first case (S-CMP) the whole process aborts, while in the latter case (F-CMP) the process terminates abnormally. Note that by definition compensations do not contain compensations themselves. As a result in the final state there is only the empty compensation $skip$ left.

The next two rules describe the execution of the sequential composition $P; Q$ depending on whether P succeeds or fails. In the first case (S-STEP) P is executed with the previously installed compensation β and commits with a changed compensation β'' (adding P's compensation). The changed compensation β'' is passed on to the continuation Q. The result of Q's execution with the compensation β' is also the final result of the execution of the sequential process. The observable control flow is the sequential composition of the control flow α of P and α' of Q. Note that if Q fails it executes its own compensation as well as P's and the previously installed β.

The second rule (A-STEP) stands for the case where P fails in a sequential composition $P; Q$. In this case the compensation β will be activated (see (S-CMP) and (F-CMP)). The continuation Q is discarded.

The rule (SAGA) states that a compensable process within a saga is executed starting with the trivial compensation $skip$. The final result of the execution of the compensable process is returned ignoring the compensation β. If the saga aborted and successfully compensated the result $\boxtimes$ is turned into a commit $\boxdot$. In (SAGA-S-STEP) and (SAGA-A-STEP) sequential composition of processes is defined as expected.

This concludes the sequential part of the semantics.

2.4 Semantics for Parallel Sagas

In this section the semantics is extended in order to handle parallel composition. We say as a notational convention that parallel composition is commutative and associative with *skip* being the identity. We present first a naive semantics for Sagas which is then modified to include interrupts.

2.4.1 Naive Semantics for Sagas

The naive semantics extends the semantics for sequential Sagas with the inference rules given in Figure 8. The new set of rules regards the possible execution of a parallel composition $P|Q$. In every rule each branch is executed starting with an empty compensation. The differences arise with the possible result states for each branch.

In the first rule, (S-PAR), both branches commit. The observable control flow, the label $\alpha|\alpha'$, denotes all possible interleavings of α and α'. The resulting compensations of the two branches are put in parallel in front of the existing compensation β. Thus in case of a future abort the compensations for P and Q will be executed in parallel and when both finish successfully β will be activated.

The remaining rules describe what happens if at least one branch aborts.

In the rule (F-PAR-NAIVE-1) both branches abort and are successfully compensated. This activates the compensation β and depending on its result the whole process either aborts or crashes. The observable control flow is the interleaving of the runs for P and Q (including compensations) followed by β's execution.

For the rules (F-PAR-NAIVE-2) and (F-PAR-NAIVE-3) one branch crashes, *i.e.*, after aborting its compensation fails. In rule (F-PAR-NAIVE-2) the other branch commits and its compensation will be activated. In the other rule (F-PAR-NAIVE-3) the second branch aborts as well. Note that in any case the whole process will reach an inconsistent state, *i.e.*, a $\boxtimes$, and thus the previously installed compensation β will never be activated.

(S-PAR)

$$\frac{\Gamma \vdash \langle P, skip\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle Q, skip\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta''\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, (\beta'|\beta''); \beta\rangle}$$

(F-PAR-NAIVE-1)

$$\frac{\Gamma \vdash \langle P, skip\rangle \xrightarrow{\alpha} \langle \boxtimes, skip\rangle \quad \Gamma \vdash \langle Q, skip\rangle \xrightarrow{\alpha'} \langle \boxtimes, skip\rangle \quad \Gamma \vdash \langle \beta, skip\rangle \xrightarrow{\alpha''} \langle \square_1, skip\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha|\alpha');\alpha''} \langle \square_2, skip\rangle}$$

$$\square_2 = \begin{cases} \boxtimes & \text{if } \square_1 = \boxdot \\ \circledast & \text{otherwise} \end{cases}$$

(F-PAR-NAIVE-2)

$$\frac{\Gamma \vdash \langle P, skip\rangle \xrightarrow{\alpha} \langle \circledast, skip\rangle \quad \Gamma \vdash \langle Q, skip\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle \beta', skip\rangle \xrightarrow{\alpha''} \langle \square, skip\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{\alpha|(\alpha';\alpha'')} \langle \circledast, skip\rangle}$$

(F-PAR-NAIVE-3)

$$\frac{\Gamma \vdash \langle P, skip\rangle \xrightarrow{\alpha} \langle \circledast, skip\rangle \quad \Gamma \vdash \langle Q, skip\rangle \xrightarrow{\alpha'} \langle \square, skip\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha|\alpha')} \langle \circledast, skip\rangle} \quad \square \in \{\boxtimes, \circledast\}$$

(F-PAR-NAIVE-4A)

$$\frac{\Gamma \vdash \langle P, skip\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle Q, skip\rangle \xrightarrow{\alpha'} \langle \boxtimes, skip\rangle \quad \Gamma \vdash \langle \beta', skip\rangle \xrightarrow{\alpha''} \langle \boxtimes, skip\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha;\alpha'')|\alpha'} \langle \circledast, skip\rangle}$$

(F-PAR-NAIVE-4B)

$$\frac{\Gamma \vdash \langle P, skip\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle Q, skip\rangle \xrightarrow{\alpha'} \langle \boxtimes, skip\rangle \quad \Gamma \vdash \langle \beta', skip\rangle \xrightarrow{\alpha''} \langle \boxdot, skip\rangle \quad \Gamma \vdash \langle \beta, skip\rangle \xrightarrow{\alpha'''} \langle \square_1, skip\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{((\alpha;\alpha'')|\alpha');\alpha'''} \langle \square_2, skip\rangle}$$

$$\square_2 = \begin{cases} \boxtimes & \text{if } \square_1 = \boxdot \\ \circledast & \text{otherwise} \end{cases}$$

Figure 8: Naive semantics of parallel Sagas.

$$\text{(SAGA-PAR)}\quad \frac{\Gamma \vdash S \xrightarrow{\alpha_1} \square_1 \quad \Gamma \vdash T \xrightarrow{\alpha_2} \square_2}{\Gamma \vdash S|T \xrightarrow{\alpha_1|\alpha_2} \square}$$

$$\square = \begin{cases} \boxdot & \text{if } \square_1 = \boxdot \wedge \square_2 = \boxdot \\ \boxtimes & \text{otherwise} \end{cases}$$

Figure 9: Semantics of parallel sagas

In the last two rules, (F-PAR-NAIVE-4A) and (F-PAR-NAIVE-4B), one branch fails and successfully compensates. The other one commits and due to the failure its compensation will be activated. In rule (F-PAR-NAIVE-4A) this compensation fails leading the system to an inconsistent state. In rule (F-PAR-NAIVE-4B) the compensation is successfully executed and the previously installed compensation is activated.

The handling of the parallel composition of sagas is displayed in Figure 9 with (SAGA-PAR). Both sagas execute indepently. The resulting state is a $\boxdot$ if both branches committed, otherwise it is $\boxtimes$.

Example 2. *Consider the transaction of Example 1 again. Figure 10 shows a derivation of this saga using the naive semantics. We assume that the booking of the hotel fails while the other actions succeed. Thus the context* Γ *maps the action* bH *to* $\boxtimes$ *and the others to* $\boxdot$*. We omit the context* Γ *in the derivation of Figure 10, moreover the label of the final observed workflow is* $\mathsf{rT}; (\mathsf{bF}; \mathsf{cF}|\mathsf{cC}; \mathit{skip}); \mathsf{cR}$*, i.e., we first reserve the ticket, then the flight is booked and cancelled again and the credit card is checked (with no real compensation), finally the reservation for the ticket is cancelled again. Note that in the naive semantics the displayed proof tree is the only possible derivation under* Γ*, i.e., the execution is deterministic.*

2.4.2 Revised Semantics for Sagas

We present the revised semantics for Sagas as introduced in [BMM05]. It extends the naive semantics with the possibility to interrupt sibling branches. In the naive semantics branches of a parallel composition are always fully executed, even if a failure is already inevitable. However each action (including compensations) may involve a cost that we would

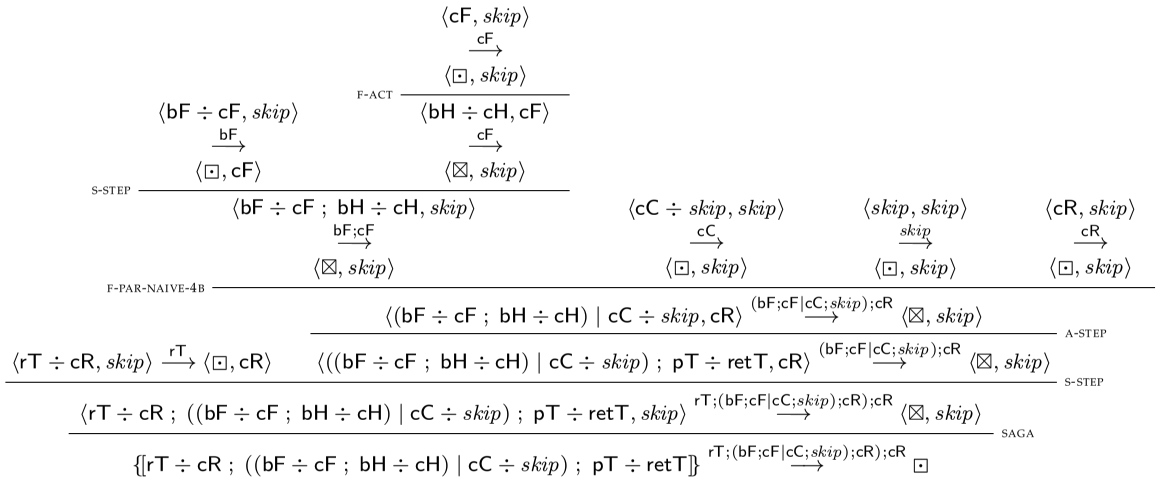


Figure 10: Derivation for the naive semantics

like to avoid if possible. With an abort the naive semantics not only executes superfluous actions, it also has to compensate these actions. In the revised semantics failing branches are allowed to interrupt siblings. This reduces the arising cost of evitable actions.

To distinguish a state where a branch was interrupted and thus forced to abort from a state with a normal failure we enrich the result set $\mathcal{R}$. We add two new symbols $\overline{\mathcal{R}} = \mathcal{R} \cup \{\overline{\boxtimes}, \overline{\circledast}\}$ with $\overline{\boxtimes}$ for a forced abort with a successful compensation and $\overline{\circledast}$ for a forced abort with a failing compensation. Thus we can deal with the partial execution of a process. Note that the result of a saga still may only be in $\mathcal{R}$, that is a whole saga cannot be "forced to abort".

We introduce a binary operator $\wedge$ on $\overline{\mathcal{R}}$ to simplify the rules for the different cases of handling parallel composition. Let $\wedge$ be associative and commutative. Its composition is defined in the following table:

$\wedge$	$\boxdot$	$\boxtimes$	$\circledast$	$\overline{\boxtimes}$	$\overline{\circledast}$
$\boxdot$	$\boxdot$	–	–	–	–
$\boxtimes$	–	$\boxtimes$	$\circledast$	$\boxtimes$	$\circledast$
$\circledast$	–	$\circledast$	$\circledast$	$\circledast$	$\circledast$
$\overline{\boxtimes}$	–	$\boxtimes$	$\circledast$	$\overline{\boxtimes}$	$\overline{\circledast}$
$\overline{\circledast}$	–	$\circledast$	$\circledast$	$\overline{\circledast}$	$\overline{\circledast}$

A parallel composition can only commit if both branches commit as can be seen in the first entry. Note that there is no other possibility to combine $\boxdot$ as with an evident failure the branch will be interrupted. In the rest of the table a crash always subsumes a normal abort while normal failure or crash subsumes a forced one.

The revised semantics reuses the inference rules for sequential Sagas given in Figure 6. The rule (A-STEP) is changed allowing for $\square$ also the interrupted alternatives $\overline{\boxtimes}$ or $\overline{\circledast}$. The new rules for the revised semantics are shown in Figure 11.

The main novelty is introduced with the rule (FORCED-ABORT). It gives the possibility to force the execution of the compensation before the forward process is finished. If the compensation succeeds the result of the overall process is a forced abort $\overline{\boxtimes}$, if the compensation fails the result is a forced crash $\overline{\circledast}$.

(FORCED-ABT)

$$\frac{\Gamma \vdash \langle \beta, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \Box_1, \mathit{skip} \rangle}{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box_2, \mathit{skip} \rangle} \quad \Box_2 = \begin{cases} \overline{\boxtimes} & \text{if } \Box_1 = \boxdot \\ \overline{\circledast} & \text{otherwise} \end{cases}$$

(S-PAR)

$$\frac{\Gamma \vdash \langle P, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \boxdot, \beta' \rangle \quad \Gamma \vdash \langle Q, \mathit{skip} \rangle \xrightarrow{\alpha'} \langle \boxdot, \beta'' \rangle}{\Gamma \vdash \langle P | Q, \beta \rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, (\beta' | \beta''); \beta \rangle}$$

(F-PAR)

$$\frac{\Gamma \vdash \langle P, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \Box_1, \mathit{skip} \rangle \quad \Gamma \vdash \langle Q, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \Box_2, \mathit{skip} \rangle}{\Gamma \vdash \langle P | Q, \beta \rangle \xrightarrow{\alpha|\alpha'} \langle \Box_1 \wedge \Box_2, \mathit{skip} \rangle}$$

$$\Box_1 \in \{\circledast, \overline{\circledast}\} \wedge \Box_2 \in \{\boxtimes, \circledast, \overline{\boxtimes}, \overline{\circledast}\}$$

(C-PAR)

$$\frac{\Gamma \vdash \langle P, \mathit{skip} \rangle \xrightarrow{\alpha} \langle \Box_1, \mathit{skip} \rangle \quad \Gamma \vdash \langle Q, \mathit{skip} \rangle \xrightarrow{\alpha'} \langle \Box_2, \mathit{skip} \rangle \quad \Gamma \vdash \langle \beta, \mathit{skip} \rangle \xrightarrow{\gamma} \langle \Box_3, \mathit{skip} \rangle}{\Gamma \vdash \langle P | Q, \beta \rangle \xrightarrow{(\alpha|\alpha');\gamma} \langle \Box_1 \wedge \Box_2 \wedge \Box_4, \mathit{skip} \rangle}$$

$$\Box_1, \Box_2 \in \{\boxtimes, \overline{\boxtimes}\} \text{ and } \Box_4 = \begin{cases} \overline{\boxtimes} & \text{if } \Box_3 = \boxdot \\ \overline{\circledast} & \text{otherwise} \end{cases}$$

Figure 11: Revised semantics of parallel Sagas

The other rules handle the concurrent part of Sagas. Rule (S-PAR) is the same as in the naive semantics, *i.e.*, it stands for the case where both branches commit.

The other two rules consider the failure of the parallel composition, either naturally or by force. Rule (F-PAR) corresponds to the rules (F-PAR-NAIVE-2) and (F-PAR-NAIVE-3). One branch aborts or is forced to abort and its compensation fails as well, *i.e.*, its result is either $\circledast$ or $\overline{\circledast}$. Remember that if a branch fails the other one cannot commit, thus the other branch has to abort as well, if necessary by force. As at least one branch terminated abnormally the previously installed compensation β will not be executed. Note that the result of this process is built using the operator $\wedge$.

The last rule (C-PAR) handles the case where both branches abort (either forced or not) and successfully compensate (comparable to rule (F-PAR-NAIVE-1)). As each branch was successfully compensated the previously installed compensation β is activated. The overall result is built from the results of both branches as well as a forced abort or crash for the result of β using the operator $\wedge$. Note that if both branches were forced to abort, the overall result will be a forced abort as well, but if one failed naturrally the overall result will be normal abort. If the previously installed compensation fails this will turn into a crash.

Example 3. *In Figure 12 we present a possible derivation in the revised semantics that is not possible in the naive one. From Example 2 we reuse the process and context Γ, which we omit in the proof tree. Note that we can interrupt the credit card check now, thus the final observed flow is* $\mathsf{rT};(\mathsf{bF};\mathsf{cF}|\mathit{skip});\mathsf{cR}$. *Note that there is also a derivation similar to the naive semantics resulting in the same observed flow.*

2.5 Compensating CSP and a Denotational Semantics

This section presents the denotational semantics of cCSP. The interpretation was first given in [BHF04]. We will reuse the syntax from Definition 1.

We start with some notation. A saga or process is interpreted by a set of traces, compensable programs by sets of pairs of traces, where the first element is the forward trace and the second element the backward or compensation trace. Each trace is a string $s\langle\omega\rangle$ consisting of a sequence of actions $s \in \mathcal{A}^*$ called the observable flow. We say that ω is the final event, defined such that $\omega \in \Omega = \{\checkmark, !, ?\}$. The symbols $\checkmark, !$ and $?$ stand for success, failure and yield to an interrupt. Note that a transaction cannot be interrupted, thus the final symbol for a saga is either a commit $\checkmark$ or a failure !. Slightly abusing the notation, we let $p, q, ...$ range over traces and also observable flows, $pp, qq, ...$ range over pairs of traces, ϵ denotes the empty observable flow.

Figure 13 presents the definitions for trace composition. There are

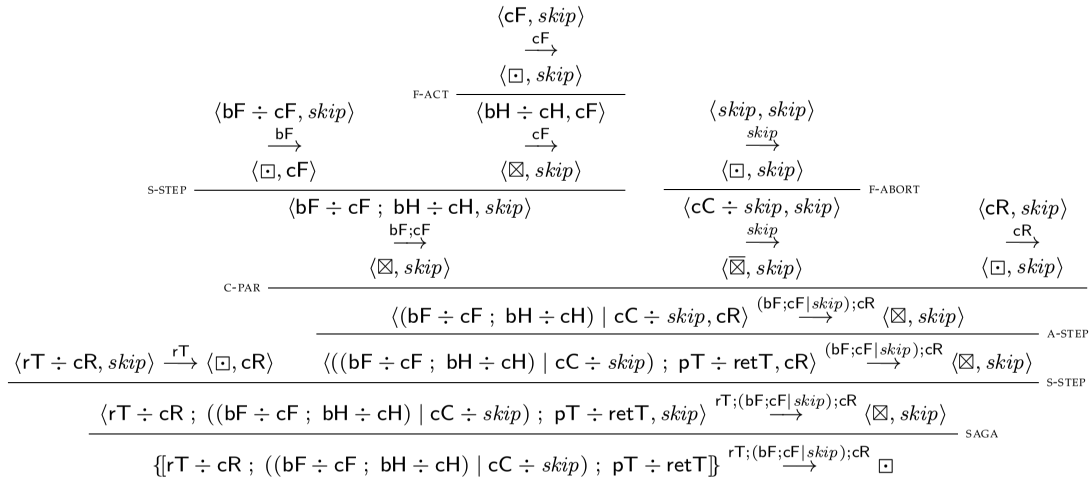


Figure 12: Derivation for the revised semantics

COMPOSITION OF STANDARD TRACES

Sequential $\begin{cases} p\langle\checkmark\rangle; q \triangleq pq \\ p\langle\omega\rangle; q \triangleq p\langle\omega\rangle \text{ when } \omega \neq \checkmark \end{cases}$

Parallel $p\langle\omega\rangle||q\langle\omega'\rangle \triangleq \{r\langle\omega\&\omega'\rangle \mid r \in (p|||q)\},$

where

ω	!	!	!	?	?	✓
ω'	!	?	✓	?	✓	✓
$\omega\&\omega'$	!	!	!	?	?	✓

and $\begin{cases} p|||\epsilon \triangleq \epsilon|||p \triangleq \{p\} \\ Ap|||Bq \triangleq \{Ar \mid r \in (p|||Bq)\} \cup \{Br \mid r \in (Ap|||q)\} \end{cases}$

COMPOSITION OF COMPENSABLE TRACES

Sequential $\begin{cases} (p\langle\checkmark\rangle, p'); (q, q') \triangleq (pq, q'; p') \\ (p\langle\omega\rangle, p'); (q, q') \triangleq (p\langle\omega\rangle, p') \text{ when } \omega \neq \checkmark \end{cases}$

Parallel $(p, p')||(q, q') \triangleq \{(r, r') \mid r \in (p||q) \wedge r' \in (p'||q')\}$

Figure 13: Trace composition in the denotational semantics

two parts, one for the composition of standard traces, the other part for the composition of compensable traces, *i.e.* pairs of traces. Both can be combined in sequence and parallel. For standard traces the sequential composition continues if the first part succeeds and otherwise stops. Parallel composition returns the set given by interleaving the two observable flows followed by a combination of the two final symbols using the operator $\&$.

For compensable traces the definition of sequential composition is given first. There is a case differentiation depending on the final symbol of the forward trace. If the forward flow of the first trace commits, the continuation is appended and the compensation is installed in reverse order. If the forward flow does not commit the continuation is discarded. The parallel composition returns the set of all pairs such that the first element is in the parallel composition of the two forward traces and the second element in the parallel composition of the two backward traces.

TRACES OF SAGAS

$$a \triangleq \{a\langle\checkmark\rangle\} \quad \mathit{skip} \triangleq \{\langle\checkmark\rangle\} \quad \mathit{throw} \triangleq \{\langle!\rangle\}$$
$$S;T \triangleq \{p;q \mid p \in S \wedge q \in T\}$$
$$S|T \triangleq \{r \mid r \in (p||q) \wedge p \in S \wedge q \in T\}$$
$$\{[P]\} \triangleq \{p\langle\checkmark\rangle \mid (p\langle\checkmark\rangle, q) \in P\} \cup \{pq \mid (p\langle!\rangle, q) \in P\}$$

TRACES OF COMPENSABLE PROCESSES

$$\begin{array}{lcl} A \div B & \triangleq & \{(A\langle\checkmark\rangle, B\langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\ \mathit{skipp} & \triangleq & \{(\langle\checkmark\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\ \mathit{throww} & \triangleq & \{(\langle!\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\ P;Q & \triangleq & \{pp;qq \mid pp \in P \wedge qq \in Q\} \\ P|Q & \triangleq & \{rr \mid rr \in (pp||qq) \wedge pp \in P \wedge qq \in Q\} \end{array}$$

Figure 14: Denotational semantics of cCSP

In Figure 14 the denotational semantics of cCSP is given. The first part defines the traces for sagas. Starting with traces for basic actions each action $a \in \mathcal{A}$ is interpreted by the trace containing the action a followed by the commit symbol $\checkmark$. Note that we assume that every action is successful. There is only one failing action throw. Its interpretation is defined next, as well as the vacuous and always successful activity skip. The sequential and parallel composition of sagas is defined using the sequential and parallel composition of traces. The last item, the transaction scope, is the most interesting. It takes the interpretation of a compensable program, *i.e.*, a pair of traces. If the forward trace is successful the compensation is discarded. In case of a failure the forward trace is followed by the compensation. Note that any yielding traces are ignored.

The part for compensable programs starts with the interpretation of compensation pairs. Either the forward activity succeeds and the compensation is installed or it yields before any execution. The interpretations of skipp and throww are unwrapped accordingly. Finally sequential and parallel composition are defined similarly to their saga counterparts depending on the definition of traces of compensable programs.

Example 4. *We consider the transaction from Example 1 where we replace the booking of the hotel with* throww*. To build the set of traces for this transaction*

we present the traces of its subterms. Consider the left branch of the parallel composition including throww first. Its trace set is

$$\mathsf{bF} \div \mathsf{cF}\ ;\ \mathit{throww} \triangleq \{(\langle ?\rangle, \langle \checkmark \rangle), (\mathsf{bF}\ \langle ?\rangle, \mathsf{cF}\ \langle \checkmark \rangle), (\mathsf{bF}\ \langle !\rangle, \mathsf{cF}\ \langle \checkmark \rangle)\}$$

From this set and the interpretation of the compensation pair $\mathsf{cC} \div \mathit{skip}$ *we build the traces for the parallel composition:*

$$\begin{array}{l}(\mathsf{bF} \div \mathsf{cF}\ ;\ \mathit{throww}) \mid \mathsf{cC} \div \mathit{skip} \triangleq \{\\ \quad (\langle ?\rangle, \langle \checkmark \rangle), (\mathsf{bF}\ \langle ?\rangle, \mathsf{cF}\ \langle \checkmark \rangle), (\mathsf{bF}\ \langle !\rangle, \mathsf{cF}\ \langle \checkmark \rangle)\\ \quad (\mathsf{bF}\ \mathsf{cC}\ \langle ?\rangle, \mathit{skip}\ \mathsf{cF}\ \langle \checkmark \rangle), \quad (\mathsf{bF}\ \mathsf{cC}\ \langle ?\rangle, \mathsf{cF}\ \mathit{skip}\ \langle \checkmark \rangle),\\ \quad (\mathsf{cC}\ \mathsf{bF}\ \langle ?\rangle, \mathit{skip}\ \mathsf{cF}\ \langle \checkmark \rangle), \quad (\mathsf{cC}\ \mathsf{bF}\ \langle ?\rangle, \mathsf{cF}\ \mathit{skip}\ \langle \checkmark \rangle),\\ \quad (\mathsf{bF}\ \mathsf{cC}\ \langle !\rangle, \mathit{skip}\ \mathsf{cF}\ \langle \checkmark \rangle), \quad (\mathsf{bF}\ \mathsf{cC}\ \langle !\rangle, \mathsf{cF}\ \mathit{skip}\ \langle \checkmark \rangle),\\ \quad (\mathsf{cC}\ \mathsf{bF}\ \langle !\rangle, \mathit{skip}\ \mathsf{cF}\ \langle \checkmark \rangle), \quad (\mathsf{cC}\ \mathsf{bF}\ \langle !\rangle, \mathsf{cF}\ \mathit{skip}\ \langle \checkmark \rangle)\}\end{array}$$

We see that a compensable process is interpreted by several yielding traces. These will be ignored in the final result set. It is defined as follows:

$$\begin{array}{l}\{\![\mathsf{rT} \div \mathsf{cR}\ ;\ ((\mathsf{bF} \div \mathsf{cF}\ ;\ \mathit{throww}) \mid \mathsf{cC} \div \mathit{skip})\ ;\ \mathsf{pT} \div \mathsf{retT}]\!\}\\ \quad \triangleq \{\quad \mathsf{rT}\ \mathsf{bF}\ \mathsf{cC} \quad \mathit{skip}\ \mathsf{cF}\ \mathsf{cR}\ \langle \checkmark \rangle\\ \qquad\quad \mathsf{rT}\ \mathsf{bF}\ \mathsf{cC} \quad \mathsf{cF}\ \mathit{skip}\ \mathsf{cR}\ \langle \checkmark \rangle\\ \qquad\quad \mathsf{rT}\ \mathsf{cC}\ \mathsf{bF} \quad \mathit{skip}\ \mathsf{cF}\ \mathsf{cR}\ \langle \checkmark \rangle\\ \qquad\quad \mathsf{rT}\ \mathsf{cC}\ \mathsf{bF} \quad \mathsf{cF}\ \mathit{skip}\ \mathsf{cR}\ \langle \checkmark \rangle\\ \qquad\quad \mathsf{rT}\ \mathsf{bF} \qquad \mathsf{cF}\ \mathsf{cR}\ \langle \checkmark \rangle \quad \}\end{array}$$

The result set includes four traces regarding the possible interleaving of forward and backward flow seperately where each branch is fully executed. In the last trace the credit card check is interrupted, i.e., it is not executed.

2.6 Maude

In the following chapters we will present supporting tools for each model we describe. We use these implementations to experiment with the newly proposed calculi and to check the correctness of the system and improve the theory. In this section we present the programming language Maude [CDE+07, Mau13a] that we use for this purpose.

Maude is based on rewriting logic that was first introduced by Meseguer in [Mes92]. It is a logic of concurrent change, that deals naturally with

highly nondeterministic concurrent computations. It can be used as a semantic framework representing distributed systems, but also as a logical framework to represent a logic.

Rewriting logic has a static part that represents the states of a system or the formulas of a logic. For this an underlying equational logic on terms is used, in the case of Maude this is membership equational logic [BM03, BM06]. It supports subsorting and equational axioms with conditions. In this functional part of rewriting logic an algebraic specification is defined. Such an algebra $\Omega = (\Sigma, E)$ is given by a set of sorts and an equational signature Σ consisting of the syntax of terms and axioms E to simplify more complex terms.

On top of this algebraic specification conditional rewrite rules are defined. This is the dynamic part of rewriting logic that describes transitions between states or inference rules for a logic.

We note that implementing a model in Maude has several advantages. Its ability to capture both static as well as dynamic behaviour makes it a very well suited tool to implement a process calculus. Moreover we say that rewriting logic has an "ϵ representational distance" [Mes12]. That means that the represented system and its representation in rewriting logic are isomorphic, *i.e.*, there is a direct representation as a rewrite theory.

Figure 15 presents the Maude specification for the algebra of natural numbers (example taken from the Maude manual [Mau13b]). The underlying sort in this algebra called `Nat` is introduced using the keyword `sort`. A Maude specification can be many-sorted. Using the keyword `subsort` we can impose an order on the different sorts. In the example the sort of nonzero natural numbers `NzNat` is a subsort of `Nat`.

After the definition of sorts we specify the operations of the signature using the keyword `op`. Note that it is possible to use mixfix notation and not only prefix notation where we use underscores as placeholders. In square brackets we can specify attributes for operations. In the example the constant `0` and the operator `s` are defined as constructors using `ctor`. The derived operation + is associative and commutative using the attributes `assoc` and `comm`. Moreover for + we specify equations us-

```
fmod NAT is
  sorts Nat NzNat .
  subsort NzNat < Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [assoc comm] .

  vars N M : Nat .

  eq 0 + N = N .
  eq s(N) + M = s (N + M) .
endfm

Maude> reduce in NAT : s(0) + s(s(0)) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s(s(s(0)))
```

Figure 15: Functional Maude module for the natural numbers

ing the keyword `eq`. We can also define conditional equations with `ceq`, memberships with `mb` and conditional memberships with `cmb`.

Using the Maude interpreter we can reduce well-defined terms to reach a normal form as shown in Figure 15 where we compute $1 + 2$.

An implementation is structured in modules. We distinguish between functional modules with sort and operator definitions, equations and memberships, and rewrite system modules, that include also rewrite rules. There are several predefined modules, for example for booleans, natural numbers or integers, as well as parametric modules like lists or sets. These can be imported using the keyword `inc`.

An example for a Maude specification using rewrite rules can be found in Figure 16 (example taken from the Maude manual [Mau13b]). It defines a Petri Net for a vending machine selling cakes for a dollar and apples for three quarters. You can only buy something with a dollar, but the machine changes four quarters to a dollar. Rules are specified with the keyword `rl` and `crl` for a conditional rule. The module defines

```
mod PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  var M : Marking .
  op __ : Marking Marking -> Marking [assoc comm] .
  ops $ q a c : -> Place .

  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a  q .
  rl [change] : q  q  q  q => $ .
endm

Maude> rewrite in PETRI-NET : $ $ q q q q q .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result Marking: q q a c c

Maude> search in PETRI-NET : $ $ q q q q q =>! a M .

Solution 1 (state 13)
states: 16  rewrites: 24 in 0ms cpu (0ms real)
         (~ rewrites/second)
M --> q q c c

Solution 2 (state 14)
states: 16  rewrites: 24 in 0ms cpu (0ms real)
         (~ rewrites/second)
M --> q q q a c

Solution 3 (state 17)
states: 19  rewrites: 27 in 0ms cpu (0ms real)
         (~ rewrites/second)
M --> a a c

Solution 4 (state 18)
states: 19  rewrites: 27 in 0ms cpu (0ms real)
         (~ rewrites/second)
M --> q a a a

No more solutions.
states: 19  rewrites: 27 in 0ms cpu (0ms real)
         (~ rewrites/second)
```

Figure 16: Maude module for a Petri Net

three rules, one for buying a cake, one for buying an apple and one for changing four quarters. The figure shows as well how we can rewrite terms and search the state space using the Maude interpreter. With the command `rewrite` Maude returns the first final (non-reduceable and non-rewritable) result. With `search` we can get several results matching a certain pattern.

There is a large number of applications for Maude. Our main interest lies in its ability to represent semantics of process calculi and programming languages. In the functional part a denotational semantics (inductively defined on the syntax) can be translated straightforwardly into equational modules. Moreover [VMO06] defines standard ways to encode SOS specifications in rewrite theories using Maude. The authors demonstrate their approach implementing languages like CCS or LOTOS. A broad overview regarding semantics represented in rewrite theories and Maude can be found in [MR11] including a formal model of C [ER12]. Thus Maude provides an easily manageable and sufficient platform to implement process calculi.

Maude has been used as a logical framework for instance to model rewriting logic itself in [CDE+99] or to represent higher order logic [NSM01]. Other recent applications of Maude consider for example real-time systems [ÖM02, ÖBM10, BMÖ12], probabilistic systems [KSMA03] or cloud computing [WEMM12]. An extensive analysis of rewriting logic, Maude, and applications can be found in [Mes12].

Various tools were developed for Maude, for example an inductive theorem prover [Hen08], a debugger [RVMOC12], but also a built-in LTL model checker [EMS03]. It can also be used to specify case studies connecting our ideas to practice. There are built-in search facilities, *i.e.*, it is possible to explore the reachable configurations starting from an initial one searching for states that satisfy (user-definable) logic predicates. Moreover new strategies can be defined to explore the state space based on the current application.

We use Maude's functional part to implement the denotational semantics in Section 3.4. In Section 4.6 we implement the encoding of Sagas processes into Petri nets in Maude and use a rewrite theory to compute

the execution inside an encoded Petri net. Using the ideas of [VMO06] in Section 5.6 we describe the implementation of the small-step semantics for Sagas. While in those sections we use Maude as a semantic framework, in Section 6.4 we use it as a logical one to represent inferences in the dynamic logic for long-running transactions.

Chapter 3

Developing a new Policy

In the previous chapter we presented the semantics of the process calculi Sagas [BMM05] and compensating CSP [BHF04]. In this chapter we continue with some more background presenting their comparison [BBF+05]. It shows that while their sequential part is largely equivalent the handling of compensations in a concurrent setting differs in the two languages. This leads to four different policies. Starting from these differences we deduce a new policy to describe the semantics of long-running transactions (LRTs). This new policy is in a sense optimal that branches may compensate independently but only once an error actually occurred. We present the denotational semantics of this new policy and show its relation to the previously defined policies. The last section introduces a prototypical engine based on the rewriting logic Maude [CDE+07].

The first section summarizes [BBF+05]. The remaining content of this chapter was developed in collaboration with Giorgio Spagnolo and first published in [Spa10]. Compared to [Spa10] we introduce here additionally the policy of Section 3.2.1 along with its formal relation shown in Theorems 3 and 4. Moreover Section 3.4 describes tool support that was originally presented in [Spa10] and is shown here in an improved version. The content was later extended in [BKLS10].

3.1 Sagas versus cCSP

In the previous chapter we presented the two process calculi Sagas and cCSP. While we can use the same syntax the two calculi provide two different semantics. The interpretation of Sagas is a big-step operational semantics while cCSP uses traces in a denotational semantics. Both languages model LRTs in a similar manner, the obvious question is how similar they are. In this section we compare the two calculi pointing out similarities and differences. The section follows closely [BBF+05], a joint work by the authors of both Sagas and cCSP.

To compare the two semantics we compare the set of traces from the denotational semantics with the labels for the operational semantics. We first focus on the sequential part. The main problem for this part is that while in cCSP actions are always successful and only *throw* fails, Sagas uses a context to determine the success or failure of an action. To solve this problem the authors define an encoding to map the set of traces from cCSP to the labels in the operational semantics and vice versa. From Sagas to cCSP failing actions are mapped to *throw* while successful actions remain the same. We can show that for every transaction $\{\![P]\!\} \xrightarrow{\alpha} \Box$ the label α in the operational semantics is equivalent to a trace of the encoded transaction. From passing from cCSP to Sagas a special context is created that maps every action to success and every *throw* to failure. Also in this case it can be shown that for any trace of a transaction there is an equivalent label in the operational semantics using the context from the encoding.

Next we consider concurrency. Here we distinguish how compensations are activated in case of a failure along two dimensions. Consider Sagas first that has already two different interpretations. In the naive semantics each branch in a parallel composition is fully executed and in case of an error afterwards compensated. The revised semantics on the other hand allows interruption of parallel branches reducing the amount of executed actions. Interruption will be our first differentiation.

Now consider cCSP. It includes interruption as well. However while for Sagas each branch can start its compensation on its own, in cCSP

there is a synchronisation point. That way, each branch executes forward activities and once every branch finishes, compensations are activated. We say that compensations in Sagas are distributed while in cCSP they are centralized.

Regarding these two aspects we can state four different policies of how to use compensations in a concurrent setting:

1. No interruption and centralized compensation.

   Siblings cannot be interrupted and once each concurrent process is finished, compensations are activated.

2. No interruption and distributed compensation.

   Siblings cannot be interrupted, each single concurrent process may start its compensation on its own.

3. Centralized interruption.

   Concurrent branches may be interrupted, compensations are only activated once each branch finished its execution.

4. Distributed interruption.

   Concurrent branches may be interrupted and compensations may start independently.

Policy #3 corresponds to the semantics of cCSP while the #2 and #4 correspond to the naive and revised semantics of Sagas.

Due to the different policies the encoding from the sequential part cannot simply be lifted to include concurrency. In [BBF+05] the authors present for each policy a version of the operational and denotational semantics. In each case they extend the encoding to show the equivalence of the two semantics. Moreover they prove a relation between the four different policies using the denotational semantics. We will present here the four policies using the denotational semantics. Remember that cCSP originally modelled policy #3. The Figures 17 and 18 show the original semantics. We use subscripts to indicate the definition specific to a policy. We are going to introduce the other policies by difference.

COMPOSITION OF STANDARD TRACES

Sequential $\begin{cases} p\langle\checkmark\rangle; q \triangleq pq \\ p\langle\omega\rangle; q \triangleq p\langle\omega\rangle \text{ when } \omega \neq \checkmark \end{cases}$

Parallel $p\langle\omega\rangle|||q\langle\omega'\rangle \triangleq \{r\langle\omega\&\omega'\rangle \mid r \in (p|||q)\}$,

where

ω	!	!	!	?	?	✓
ω'	!	?	✓	?	✓	✓
$\omega\&\omega'$	!	!	!	?	?	✓

and $\begin{cases} p|||\epsilon \triangleq \epsilon|||p \triangleq \{p\} \\ Ap|||Bq \triangleq \{Ar \mid r \in (p|||Bq)\} \cup \{Br \mid r \in (Ap|||q)\} \end{cases}$

COMPOSITION OF COMPENSABLE TRACES

Sequential $\begin{cases} (p\langle\checkmark\rangle, p'); (q, q') \triangleq (pq, q'; p') \\ (p\langle\omega\rangle, p'); (q, q') \triangleq (p\langle\omega\rangle, p') \text{ when } \omega \neq \checkmark \end{cases}$

Parallel $(p, p')||(q, q') \triangleq_{1,3} \{(r, r') \mid r \in (p||q) \wedge r' \in (p'||q')\}$

Figure 17: Composition of traces in the denotational semantics in policy #3

TRACES OF SAGAS

$a \triangleq \{a\langle\checkmark\rangle\} \quad skip \triangleq \{\langle\checkmark\rangle\} \quad throw \triangleq \{\langle!\rangle\}$

$S; T \triangleq \{p; q \mid p \in S \wedge q \in T\}$

$S|T \triangleq \{r \mid r \in (p||q) \wedge p \in S \wedge q \in T\}$

$\{\![P]\!\} \triangleq \{p\langle\checkmark\rangle \mid (p\langle\checkmark\rangle, q) \in P\} \cup \{pq \mid (p\langle!\rangle, q) \in P\}$

TRACES OF COMPENSABLE PROCESSES

$A \div B \triangleq_{3,4} \{(A\langle\checkmark\rangle, B\langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\}$

$skipp \triangleq \{(\langle\checkmark\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\}$

$throww \triangleq \{(\langle!\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\}$

$P; Q \triangleq \{pp; qq \mid pp \in P \wedge qq \in Q\}$

$P|Q \triangleq \{rr \mid rr \in (pp||qq) \wedge pp \in P \wedge qq \in Q\}$

Figure 18: Denotational semantics of policy #3

Compared to cCSP and its semantics the first policy removes interruption. We redefine the traces for compensation pairs

$$A \div B \triangleq_{1,2} \{(A\langle\checkmark\rangle, B\langle\checkmark\rangle)\}$$

removing the yielding trace. Similarly for *skipp* and *throww* the yielding trace is removed.

Policy #2 does not allow interruption as well, thus it also uses the redefined traces for compensation pairs from the previous case. Moreover compensations are distributed in parallel compositions. We have to change the definition for the interleaving of pairs of traces:

$$\begin{array}{lll}
(p\langle\checkmark\rangle, p')||(q\langle\checkmark\rangle, q') & \triangleq_{2,4} & \{(r\langle\checkmark\rangle, r'\langle\checkmark\rangle)|r \in (p|||q) \wedge r'\langle\checkmark\rangle \in (p'||q')\} \\
& & \cup\{(r\langle?\rangle, \langle\checkmark\rangle)|r\langle\checkmark\rangle \in (pp'||qq')\} \\
(p\langle\omega\rangle, p')||(q\langle\omega'\rangle, q') & \triangleq_{2,4} & \{(r\langle\omega\&\omega'\rangle, \langle\checkmark\rangle)|r\langle\checkmark\rangle \in (pp'||qq')\} \\
& & \text{if } \omega\&\omega' \in \{!, ?\}
\end{array}$$

In the successful case a yielding trace is added. Moreover if the resulting final symbol is a fail or a yield the resulting forward trace is the interleaving of the forward and backward traces combined, while the compensation is empty. Due to the additional yielding trace we have to change sequential composition as well such that it does not allow a ? in the first component.

Policy #4 reuses the original definition of compensation pairs including the interrupted trace. Moreover compared to the original semantics the distributed interleaving of compensable traces is used as in policy #2.

Using this representation of the four different policies for the concurrent semantics the authors show a subset relation.

Theorem 1. [BBF+05] *Let P be a concurrent compensable process, and let $\{\![P]\!\}_i$ denote the set of traces of $\{\![P]\!\}$ when considering the policy $i = 1, ..., 4$. Then, the four trace semantics satisfy the following diagram:*

$$\begin{array}{lccc}
 & \{\![P]\!\}_1 & \xrightarrow{\subseteq} & \{\![P]\!\}_2 \quad \textit{Naive Sagas} \\
 & \downarrow\subseteq & & \downarrow\subseteq \\
\textit{Original cCSP} & \{\![P]\!\}_3 & \xrightarrow{\subseteq} & \{\![P]\!\}_4 \quad \textit{Revised Sagas}
\end{array}$$

The proof is done by induction on the structure of compensable P for each inclusion. The following corollary reduces the result to sequential processes.

Corollary 1. *Let P be a sequential compensable process (i.e., it contains no parallel composition operator), then $\{\![P]\!\}_1 = \{\![P]\!\}_4$.*

3.2 Coordinated Compensation

We start this section with an example to clarify the differences between the above policies and eventually point out their shortcomings. Our aim is to deduce a new policy that will overcome these shortcomings. We present the different sets of traces obtained for the saga $\{\![(A \div A'; B \div B')|(C \div C'; \mathit{throww})]\!\}$, that may stand for example for a workflow for booking a trip. Activity A stands for booking a flight, B for booking a hotel, while C is the credit card check. Compensations are the cancelling of the respective bookings and the sending of a failure message by email for the credit card check. A fault is issued after the credit card check. Note that in general we do not observe a failure, but from the definition of the transaction we know that it has to happen sometime after C is executed but before its compensation C'.

For each policy we present the different sets of traces. The example shows that all semantic inclusions in Theorem 1 are strict for the particular process under consideration. We omit the final symbol ✓, denoting success, for the sake of readability.

For case one, centralized compensation without interruption, the resulting set of traces is $S_1 \equiv (AB|||C); (B'A'|||C')$. Roughly, all branches are fully executed forward and only then (indicated by ;) their (interleaved) compensation is started. In this case the failure must happen before any compensation is executed.

For case two, distributed compensation without interruption, the set of traces is $S_2 \equiv ABB'A'|||CC'$, where each branch separately starts its compensation. Thus activity C may be compensated after the failure without waiting for the completion of the first branch, like in the trace $CC'ABB'A'$. This trace is not allowed in the first policy. Moreover, the

interleaving of $ABB'A'$ and CC' includes traces like $ABB'A'CC'$ where compensation B' is observed before the *throww* is issued (*i.e.*, before the execution of C), a property we will discuss further for policy #4.

For case three we have $S_3 \equiv S_1 \cup (A|||C); (A'|||C') \cup CC'$. While the first part corresponds to the set S_1 of policy #1 the rest of the set results from allowing the branch for the activities A and B to be interrupted. In these cases either only activity A or neither A nor B are executed. As in the first policy compensations are activated only when each branch is ready, *i.e.*, all forward activities precede all compensating activities. Thus the failure must happen sometime before any compensation is executed. Note that both S_2 and S_3 include the set S_1 from the first policy. However policy #2 and #3 are not comparable. Both include traces that are not present in the other policy. For example S_2 includes the trace $CC'ABB'A'$ where the compensation C' is executed before the forward activities A and B. On the other hand S_3 includes the trace CC' where the first branch is interrupted.

Policy #4, distributed interruption, is the most liberal. It allows the set of traces $S_4 \equiv CC' \cup AA'|||CC' \cup ABB'A'|||CC'$. Branches can be interrupted and activate their compensation themselves. The set S_4 includes both S_2 and S_3. Note that, like in case two, compensation A' or B' may be executed before C, *i.e.*, before the error occurred like in the trace $AA'CC'$.

We already indicated that none of the four originally defined semantics is entirely satisfactory. A main issue is that typically activities and compensations have a cost. Without interruption (cases one and two) sibling branches finish their forward execution, even though a failure is already imminent and they will have to compensate. Thus they execute futile actions that could have been avoided. In case three, branches might have to wait until they are allowed to continue together with their siblings, costing the system time and thus reducing its performance. We conclude that policies one to three are too restrictive: it is important to have the possibility to stop a sibling branch and to activate compensations as soon as possible reducing any additional cost. That leaves policy #4, but, as policy #2, it is unrealistic: it allows a guessing mech-

$$
\begin{array}{rcl}
(p\langle\checkmark\rangle, p')|||(q\langle\checkmark\rangle, q') & \triangleq_5 & \{(r\langle\checkmark\rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} \\
(p\langle\omega\rangle, p')|||(q\langle\omega'\rangle, q') & \triangleq_5 & itp((p\langle\omega\rangle, p'), (q\langle\omega'\rangle, q')) \cup \\
 & & itp((q\langle\omega'\rangle, q'), (p\langle\omega\rangle, p')) \\
 & & \quad \text{when } \omega, \omega' \neq \checkmark \\
(p\langle\omega\rangle, p')|||(q\langle\omega'\rangle, q') & \triangleq_5 & \emptyset \qquad \text{otherwise} \\
itp((p\langle\omega\rangle, p'), (q\langle\omega'\rangle, q')) & \triangleq_5 & \{((p|||q_1)\langle\omega\rangle, (p'||q_2q')) \mid q = q_1q_2\}
\end{array}
$$

Figure 19: Parallel composition of compensable traces in the denotational semantics for policy #5

TRACES OF COMPENSABLE PROCESSES

$$
\begin{array}{lcl}
A \div B & \triangleq_5 & \{(A\langle\checkmark\rangle, B\langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle), (A\langle?\rangle, B\langle\checkmark\rangle)\} \\
skipp & \triangleq & \{(\langle\checkmark\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\
throww & \triangleq & \{(\langle!\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\
P; Q & \triangleq & \{pp; qq \mid pp \in P \wedge qq \in Q\} \\
P|Q & \triangleq & \{rr \mid rr \in (pp||qq) \wedge pp \in P \wedge qq \in Q\}
\end{array}
$$

Figure 20: Denotational semantics of policy #5 with $pp||qq$ as in Figure 19

anism where a branch may start its compensation by predicting that an error will occur in a sibling. A realistic semantics should be more "permissive" (i.e., allowing more traces) than policy three but less than four. In this new semantics, the traces of the above example would be $S \equiv S_3 \cup (CC'AA') \cup (AB|||CC'); B'A'$. Note that $S_3 \subset S \subset S_4$.

We call this new fifth policy *coordinated compensation*. It is "optimal", in the sense that it guarantees that distributed compensations may only be started after an error actually occurred, but compensations can start as soon as possible.

The new policy differs from the original cCSP (policy #3) by slightly changing the semantics of compensation pairs, to allow a successfully completed activity to yield, and the semantics of parallel composition, to allow distributed compensation without any guessing mechanism.

The new definition of the interleaving of compensable traces is dis-

played in Figure 19. The definition for successful traces is equivalent to the previous definition. For failing or interrupted traces we use a special function itp. Let $pp = (p\langle\omega\rangle, p')$ and $qq = (q\langle\omega'\rangle, q')$ with $\omega, \omega' \neq \checkmark$. The function $itp(pp, qq)$ returns the set of all compensable traces obtained by interleaving the activities of p with that of any prefix q_1 of $q = q_1 q_2$ as forward activities, together with the interleaving of p' with the residual $q_2 q'$ of qq (after removing the prefix q_1). Several cases are possible. If $\omega =!$ and $\omega' =?$, then it means that qq will be interrupted by the fault raised from pp, which is ok, because qq will yield after all activities in p have been observed. The resulting forward traces have ! as final event. If $\omega =?$ and $\omega' =!$, then it means that pp is yielding to a sibling different from qq, and therefore pp can legitimately start compensating without waiting for qq to raise the fault. In this case the resulting forward traces have $?$ as final event. If $\omega = \omega' =?$ then it means that pp receives the interrupt before qq. If $\omega = \omega' =!$ then it just means that pp is the first to raise the fault.

In Figure 20 the traces for compensable processes in policy #5 are given. Compared to the original cCSP semantics we include interruption for compensation pairs before and after execution. The rest of the semantics remains the same, though using the new definition of interleaving of parallel composition.

3.2.1 Notification and distributed compensation

Note that also for policy #5 we have a similar version without interrupt that we present in this section. We call this policy *Notification and distributed compensation* (policy #6) to emphasize the fact that siblings are notified about the fault, not really interrupted. Since compensations are distributed and the fault is not observable, it can happen that a notified thread starts compensating even before the sibling that actually aborted. However, contrary to policy #2, a thread cannot guess the presence of faulty siblings, so it is not possible to observe a forward activity of the only faulty thread after a compensation activity of a notified thread. Thus policy #6 defines a sound variant of policy #2 where unrealistic traces are

$$
\begin{array}{lcl}
(p\langle\checkmark\rangle, p')||(q\langle\checkmark\rangle, q') & \triangleq_6 & \{(r\langle\checkmark\rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} \cup \\
& & itp((p\langle?\rangle, p'), (q\langle?\rangle, q')) \cup \\
& & itp((q\langle?\rangle, q'), (p\langle?\rangle, p')) \\
(p\langle\omega\rangle, p')||(q\langle\omega'\rangle, q') & \triangleq_6 & itp((p\langle\omega\rangle, p'), (q\langle\omega'\rangle, q')) \cup \\
& & itp((q\langle\omega'\rangle, q'), (p\langle\omega\rangle, p')) \\
& & \quad \text{when } \omega, \omega' \neq \checkmark \\
(p\langle\checkmark\rangle, p')||(q\langle\omega'\rangle, q') & \triangleq_6 & itp((p\langle?\rangle, p'), (q\langle\omega'\rangle, q')) \cup \\
& & itp((q\langle\omega'\rangle, q'), (p\langle?\rangle, p')) \\
& & \quad \text{when } \omega' \neq \checkmark \\
(p\langle\omega\rangle, p')||(q\langle\checkmark\rangle, q') & \triangleq_6 & itp((p\langle\omega\rangle, p'), (q\langle?\rangle, q')) \cup \\
& & itp((q\langle?\rangle, q'), (p\langle\omega\rangle, p')) \\
& & \quad \text{when } \omega \neq \checkmark
\end{array}
$$

Figure 21: Parallel composition of compensable traces in the denotational semantics for policy #6

discarded. In this way policy #6 relates to policies #1 and #2 as policy #5 relates to policies #3 and #4.

Regarding the running example from the beginning of this section the set of traces allowed in policy #6 is $S_6 \equiv S_1 \cup (AB|||CC'); B'A'$. We can easily see that $S_6 \subset S$. Compared to policies #1 and #2 it includes traces like $CC'ABB'A'$ that are not allowed in policy #1 as compensation C' is executed before the complete forward flow finished. On the other hand it does not include unrealistic traces like $ABB'A'CC'$ where compensation B' is observed before C is executed and the actual error occurred.

As in policies #1 and #2 we remove the yielding traces from compensation pairs as well as from *skipp* and *throww*. Thus the definition of parallel composition of compensable traces has to be adapted. Figure 21 shows the new definition where we add for successful traces also a respective yielding one. This additional trace implies the change of sequential composition as in policy #2 where we do not allow $?$ in the first component.

3.3 Formal Relation

This section presents how the new semantics of policy #5 is related to previous policies. Our first result establishes a formal relation with policies #1–4.

Theorem 2. *Let P be a compensable process, and let $\{\![P]\!\}_i$ denote the denotational semantics of saga $\{\![P]\!\}$ (i.e., its set of traces) according to policy #i. Then we have: $\{\![P]\!\}_3 \subseteq \{\![P]\!\}_5 \subseteq \{\![P]\!\}_4$.*

Proof. The inclusion $\{\![P]\!\}_3 \subseteq \{\![P]\!\}_5$ follows by proving by structural induction on P the following implications for any p, p' and any $\omega \in \{?, !\}$:

$$\begin{array}{lcl} (p\langle\checkmark\rangle, p') \in_3 P & \Rightarrow & (p\langle\checkmark\rangle, p') \in_5 P \ \wedge \ (p\langle ?\rangle, p') \in_5 P \\ (p\langle\omega\rangle, p') \in_3 P & \Rightarrow & (p\langle\omega\rangle, p') \in_5 P \end{array}$$

where $\in_i$ denotes membership according to policy #i. For both implications the base cases regarding compensation pairs, *skipp* and *throww* hold trivially. For a sequential composition $P = P_1; P_2$ the implication is shown by applying the induction hypothesis to both P_1 and P_2. The most interesting case is parallel composition. Consider the first implication and assume that $P = P_1|P_2$. The successful case holds quite trivially. We want to show that if $(p\langle\checkmark\rangle, p') \in_3 P$ then $(p\langle ?\rangle, p') \in_5 P$. By the definition of parallel composition there exist p_1, p_1', p_2, p_2' such that

$$\begin{array}{l} (p_1\langle\checkmark\rangle, p_1') \in_3 P_1 \\ (p_2\langle\checkmark\rangle, p_2') \in_3 P_2 \\ p \in p_1 ||| p_2 \qquad p' \in p_1' ||| p_2' \end{array}$$

By the induction hypothesis we know that $(p_1\langle ?\rangle, p_1') \in_5 P_1$ and $(p_2\langle ?\rangle, p_2') \in_5 P_2$. Using the definition of parallel composition in policy #5 we can prove that $(p\langle ?\rangle, p') \in_5 P$ by taking $(p\langle ?\rangle, p') \in itp((p_1\langle ?\rangle, p_1'), (p_2\langle ?\rangle, p_2'))$ where we set the parameters q_1, q_2 as $q_1 = p_2$ and $q_2 = \epsilon$. The other implication can be shown similarly.

The inclusion $\{\![P]\!\}_5 \subseteq \{\![P]\!\}_4$ follows by proving that for any p, p':

$$\begin{array}{lcl} (p\langle\checkmark\rangle, p') \in_5 P & \Rightarrow & (p\langle\checkmark\rangle, p') \in_4 P \\ (p\langle !\rangle, p') \in_5 P & \Rightarrow & \exists q, q' \ \ (pq\langle !\rangle, q') \in_4 P \text{ with } p' = qq' \\ (p\langle ?\rangle, p') \in_5 P & \Rightarrow & \exists q, q', \omega \ \ (pq\langle\omega\rangle, q') \in_4 P \text{ with } p' = qq' \\ & & \qquad\qquad \text{and } \omega \in \{\checkmark, ?, !\} \end{array}$$

We show this by structural induction on P. For all cases the first implication is trivial from the definition of policies #4 and #5. For the second

implication the base case only regards $throww$ where the implication follows trivially. In the third implication there are several base cases regarding compensation pairs, $throww$ and $skipp$. In each case we let $q = \epsilon$ and $q' = p'$.

For the induction step we first consider sequential composition. For the second implication let $P = P_1; P_2$ with $(p\langle!\rangle, p') \in_5 P$. If P_1 fails the implication follows from the induction hypothesis by taking $q = \epsilon$ and $q' = p'$. If P_1 succeeds, P_2 has to fail. Then by the definition of sequential composition there exist p_1, p_1', p_2, p_2' such that

$$\begin{array}{ll} (p_1\langle\checkmark\rangle, p_1') \in_5 P_1 & \\ (p_2\langle!\rangle, p_2') \in_5 P_2 & \\ p = p_1 p_2 \qquad & p' = p_2' p_1' \end{array}$$

We apply the induction hypothesis to P_1 and P_2 (with $p_2' = r\ r'$):

$$(p_1\langle\checkmark\rangle, p_1') \in_4 P_1 \quad (p_2 r\langle!\rangle, r') \in_4 P_2$$

Using the definition of sequential composition yields

$$(p_1 p_2 r\langle!\rangle, r' p_1') = (p r\langle!\rangle, r' p_1') \in_4 P$$

Together with $p' = p_2' p_1' = r\ r' p_1'$ this proves the implication. The third implication for sequential composition follows the same reasoning.

Next we consider parallel composition. For the second implication this leads to several cases. Let $P = P_1|P_2$ with the trace pair $(p\langle!\rangle, p') \in_5 P$. We consider the case where P_1 fails and P_2 is interrupted. By the definition of parallel composition there exist traces p_1, p_1', p_2, p_2' with $p_2 = qq'$ such that

$$\begin{array}{ll} (p_1\langle!\rangle, p_1') \in_5 P_1 & \\ (p_2\langle?\rangle, p_2') \in_5 P_2 & \\ p \in p_1 ||| q \qquad & p' \in p_1' || q' p_2' \end{array}$$

We apply the induction hypothesis to P_1 and P_2:

$$\begin{array}{l} \exists r, r'.(p_1 r\langle!\rangle, r') \in_4 P_1 \wedge p_1' = r\ r' \\ \exists s, s', \omega.(p_2 s\langle\omega\rangle, s') \in_4 P_2 \wedge p_2' = s\ s' \wedge \omega \in \{?, !\} \end{array}$$

We build the parallel composition as defined for policy #4. We get the set of traces $(t\langle!\rangle, \langle\checkmark\rangle) \in_4 P$ where $t \in p_1\ r\ r' || p_2\ s\ s' = p_1\ p_1' || p_2\ p_2'$. Our aim is to show that $(pp'\langle!\rangle, \langle\checkmark\rangle) \in_4 P$, thus we have to build $t = p\ p' \in p_1 ||| q; p_1' || q' p_2' \subseteq p_1\ p_1' || q\ q'\ p_2' = p_1\ p_1' || p_2\ p_2'$. This proves the implication. The other cases where either only P_2 or both fail can be shown similarly.

The case of parallel composition for the last implication follows the same reasoning. □

The process $\{\![(A \div A'; B \div B')|(C \div C'; throww)]\!\}$ from the previous section witnesses that all semantic inclusions between the different policies can be strict.

Corollary 2. *If P is a sequential process (i.e., it contains no parallel composition operator), then $\{\![P]\!\}_5 = \{\![P]\!\}_4$.*

Proposition 1. *The policies #2 and #5 are not comparable by inclusion.*

Proof. We show that there exists a process P such that neither $\{\![P]\!\}_2 \subseteq \{\![P]\!\}_5$ nor $\{\![P]\!\}_5 \subseteq \{\![P]\!\}_2$ hold. Take $P = \{\![A \div A' \mid (B \div B'; throww)]\!\}$. Then the trace $p = AA'BB'\langle\checkmark\rangle \in \{\![P]\!\}_2$, but $p \notin \{\![P]\!\}_5$, because A' is observed before B (and therefore before the fault occurs). Moreover, the trace $q = BB'\langle\checkmark\rangle \in \{\![P]\!\}_5$, but $q \notin \{\![P]\!\}_2$, because it involves the interruption of process $A \div A'$. This is not allowed in policy #2. □

For policy #6 we can state similar to Theorem 2 the following:

Theorem 3. *Let P be a compensable process, and let $\{\![P]\!\}_i$ denote the denotational semantics of saga $\{\![P]\!\}$ (i.e., its set of traces) according to policy #i. Then we have: $\{\![P]\!\}_1 \subseteq \{\![P]\!\}_6 \subseteq \{\![P]\!\}_2$.*

Proof. The proof is similar to the one for Theorem 2. For the first inclusion $\{\![P]\!\}_1 \subseteq \{\![P]\!\}_6$ we have to prove the following:

$$\begin{array}{lcl} (p\langle\checkmark\rangle, p') \in_1 P & \Rightarrow & (p\langle\checkmark\rangle, p') \in_6 P \\ (p\langle!\rangle, p') \in_1 P & \Rightarrow & (p\langle!\rangle, p') \in_6 P \end{array}$$

Note that policy #1 does not include any yielding traces.

The second inclusion $\{\![P]\!\}_6 \subseteq \{\![P]\!\}_2$ follows by proving that for any p, p':

$$\begin{array}{lcl} (p\langle\checkmark\rangle, p') \in_6 P & \Rightarrow & (p\langle\checkmark\rangle, p') \in_2 P \\ (p\langle!\rangle, p') \in_6 P & \Rightarrow & \exists q, q' \;\; (pq\langle!\rangle, q') \in_2 P \text{ with } p' = qq' \\ (p\langle?\rangle, p') \in_6 P & \Rightarrow & \exists q, q', \omega \;\; (pq\langle\omega\rangle, q') \in_2 P \text{ with } p' = qq' \\ & & \qquad\qquad \text{and } \omega \in \{?, !\} \end{array}$$

□

Theorem 4. *Let P be a compensable process, and let $\{\![P]\!\}_i$ denote the denotational semantics of saga $\{\![P]\!\}$ (i.e., its set of traces) according to policy #i. Then we have: $\{\![P]\!\}_6 \subseteq \{\![P]\!\}_5$.*

Proof. We prove by structural induction the following implications for any p, p' and any $\omega \in \{?, !\}$:

$$\begin{array}{lcl} (p\langle\checkmark\rangle, p') \in_6 P & \Rightarrow & (p\langle\checkmark\rangle, p') \in_5 P \ \wedge \ (p\langle?\rangle, p') \in_5 P \\ (p\langle\omega\rangle, p') \in_6 P & \Rightarrow & (p\langle\omega\rangle, p') \in_5 P \end{array}$$

The proof is similar to showing the first inclusion of Theorem 2. □

3.4 Tool Support

This section presents an implementation of the denotational semantics. It includes the original cCSP as well as the other four policies (along with policy #5). The tool is implemented in Maude (see Section 2.6 for details). It was originally developed in collaboration with Giorgio Spagnolo [Spa10] and is here in a slightly improved version. The complete code can be found at [Sou13].

The implementation is structured in six modules, one for each of the five policies and one including the definitions shared by all of them.

One of the basic sorts is called `trace` used for a single trace. Its constructor is

```
op _<_> : names end -> trace [ctor] .
```

The sort `names` stands for a string of actions and `end` for the possible final symbols. These can either be `ok` for a commit (symbol ✓), `!` for a failure or `?` for an interrupt. For the sake of the presentation we take the set of natural numbers as the set of actions.

The sort `trace` is a subsort of `traceset`, that stands for sets of traces. The combination of traces for sagas as given in Figure 17 is defined using the sort `traceset`. A process or saga is defined using the sort `sagaprocess` that is a supersort of names. It allows sequential and parallel composition. To compute the set of traces for a saga we use an environment. The constant `nilenv` stands for an empty environment

```
Maude> reduce in cCSP3 :
       ((1 ; 2) || (3 ; 4)) @ (3 - !) .
rewrites: 27 in 0ms cpu (0ms real) (~ rewrites/second)
result traceset: 1 2 3 < ! > U 1 3 2 < ! > U
                 3 1 2 < ! >
```

Figure 22: Example of a saga

that maps every action to success. Otherwise the environment maps single names to a respective `end` symbol (including ?) using

```
op _-_ : name end -> env .
```

The environment is applied to a `sagaprocess` using the operator `@`.

An example for a computation is given in Figure 22 where we derive the set of traces for a parallel composition of two sagas. In the example the environment states that the action `3` in the right branch fails. During the computation first the environment is mapped down to single actions. Next the environment is resolved giving a trace with the name of the action and its final symbol determined by the environment. Then the computation goes bottom up again building from the singleton traces the result set. In the example it consists of three failing traces. Note that the aborting action is included in the final trace.

The definition of the tool continues with the extension to compensable processes. The basic sort here is a single trace pair consisting of two traces:

```
op _,_ : trace trace -> tracepair [ctor ] .
```

As for traces we define a sort `tracepairset` for sets of trace pairs. Compensable Processes are defined using the sort `cprocess` that can be compensation pairs or the sequential or parallel composition of compensable processes. The system uses an environment as well to determine the resulting state of single actions. It is applied to `cprocess` using the operator `#`.

```
Maude> reduce in cCSP1 :
       [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 146 in 0ms cpu (5ms real)
(~ rewrites/second)
result traceset:
    1 3 2 4 < ok > U 1 3 4 2 < ok > U
    3 1 2 4 < ok > U 3 1 4 2 < ok >

Maude> reduce in cCSP2 :
       [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 224 in 0ms cpu (0ms real)
(~ rewrites/second)
result traceset:
    1 2 3 4 < ok > U 3 4 1 2 < ok > U
    1 3 2 4 < ok > U 1 3 4 2 < ok > U
    3 1 2 4 < ok > U 3 1 4 2 < ok >

Maude> reduce in cCSP3 :
       [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 396 in 4ms cpu (1ms real)
(111000 rewrites/second)
result traceset:
    3 4 < ok >     U
    1 3 2 4 < ok > U 1 3 4 2 < ok > U
    3 1 2 4 < ok > U 3 1 4 2 < ok >

Maude> reduce in cCSP4 :
       [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 600 in 4ms cpu (1ms real)
(150000 rewrites/second)
result traceset:
    3 4 < ok >     U
    1 2 3 4 < ok > U 3 4 1 2 < ok > U
    1 3 2 4 < ok > U 1 3 4 2 < ok > U
    3 1 2 4 < ok > U 3 1 4 2 < ok >
```

Figure 23: Example of a transaction regarding the first four policies

```
Maude> reduce in cCSP5 :
       [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 1732 in 4ms cpu (4ms real)
(444500 rewrites/second)
result traceset:
    3 4 < ok >     U
                     3 4 1 2 < ok > U
    1 3 2 4 < ok > U 1 3 4 2 < ok > U
    3 1 2 4 < ok > U 3 1 4 2 < ok >

Maude> reduce in cCSP6 :
       [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 724 in 0ms cpu (1ms real)
(~ rewrites/second)
result traceset:
                     3 4 1 2 < ok > U
    1 3 2 4 < ok > U 1 3 4 2 < ok > U
    3 1 2 4 < ok > U 3 1 4 2 < ok >
```

Figure 24: Example of a transaction regarding the new policies #5 and #6

The modules for the different policies define the parallel composition of both traces and trace pairs as well as the definition of compensation pairs.

In Figures 23 and 24 the different results for each of the six policies are given for a simple example transaction. The transaction is a parallel composition where the left branch is a compensation pair and the right branch a compensation pair followed by a *throww*. The environment is empty, thus every basic activity is mapped to success. The example shows the different inclusions from Theorem 1 and 2. In the module `cCSP1` representing the first policy the set of traces is the smallest. Each of these traces first interleaves the forward actions followed by interleaving of the compensating actions. This set is included in each of the other policies and reported in the two bottom lines of each reduction. Policy #2 in the module `cCSP2` adds two traces to the set. These traces allow the branches to run their compensation independently. The third policy on the other hand in module `cCSP3` adds one trace for interruption to the first set. The set for policy #4 as the result of module `cCSP4` includes

```
Maude> reduce in cCSP5 : {3 4,ok} u {3 4 1 2,ok} in
                  [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 1736 in 4ms cpu (4ms real)
(445500 rewrites/second)
result Bool: true

Maude> reduce in cCSP5 :
   {1 2 3 4,ok} in [(1 / 2 || (3 / 4 ; throww)) # nilenv] .
rewrites: 1739 in 8ms cpu (4ms real)
(223125 rewrites/second)
result Bool: false
```

Figure 25: Example showing the use of the tool predicate `in`

both policy #2 and #3. Compared to policy #4 the fifth policy in module `cCSP5` removes the unrealistic trace where the left branch starts compensating before the error occurred. Policy #6 in module `cCSP6` on the other hand removes the unrealistic trace compared to policy #2.

The tool defines some comparing predicates for trace sets. We can show the inclusion of different trace sets as well as the equivalence, the difference and the intersection. Figure 25 shows the use of the inclusion predicate in the fifth policy. Regarding the previous example we show here using the tool that both the distributed and the interrupted trace are included in the result while the unrealistic trace where the left branch compensates before the error occurred is not included.

3.5 Conclusion

In this chapter we formally defined a new policy for handling compensations in a concurrent setting. It improves existing approaches by allowing branches to independently activate compensations without any synchronisation point (unlike policies #1 and #3) and discarding unrealistic traces (policies #2 and #4). We presented its denotational semantics and showed its relation to previous policies. Finally we introduced tool support for all five policies implementing their denotational semantics in Maude.

Chapter 4

A Graphical Presentation

In the previous chapter we deduced a new policy for handling compensations in a concurrent setting and presented its denotational semantics. We called this new policy *coordinated compensation*. It allows interruption of sibling branches and the autonomous activation of compensation as soon as an error occurred. In this chapter we introduce an operational characterization of the coordinated semantics, and we prove a correspondence result between the operational semantics presented here and the denotational semantics of the previous chapter.

We base our operational semantics on a mapping of sagas into Petri nets [Rei85]. On one side, Petri nets are a well-known model of concurrency and allow us to give a simple account of the interactions between the different activities in a saga. On the other side, this mapping allows us to exploit the well-developed theory of Petri nets and the related tools. The choice for Petri nets has been inspired by the interesting work by Acu and Reisig [AR06] aiming to add compensations to a simpler class of nets called workflow nets.

In the remaining part of the chapter we show that the semantics satisfies some expected high-level properties. Furthermore we present encodings of the four other policies presented in Chapter 3 and a tool implementing the encoding. The content of this chapter was first published in [BKLS10].

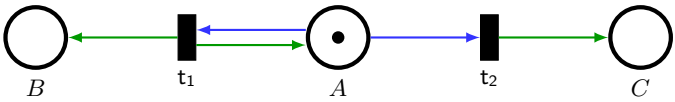


Figure 26: Example of a Petri net

4.1 Background on Petri nets

In this section we give a basic introduction to Petri nets. In a nutshell a Petri net can be seen as a graph where nodes are called places and edges are transitions. Such transitions can have multiple sources and multiple targets. To add dynamics, places can be marked with tokens that using transitions are moved, split , joined, generated or deleted. The token is consumed if it is in the set of incoming places (preset) for an executing transition. Such a transition when executed (or fired), consumes a token from each of its incoming places and produces a token in each outgoing one.

Example 5. *In Figure 26 we present a Petri net with three places and two transitions. Incoming arcs for transitions are blue, outgoing arcs are green. Initially there is a token in A. The first transition $\mathtt{t}_1$ consumes the token in A and produces a token in A again and one in B. Transition $\mathtt{t}_2$ consumes the token in A and produces a token in C. Thus the net produces tokens in B until the second transition fires and produces a token in C. Note that both transitions compete for the same source.*

Formally we define Petri nets as follows:

Definition 3 (Petri net)**.** *A Petri net graph is a triple (P, T, F), where:*

- *P is a finite set of places,*
- *T is a finite set of transitions, disjoint from P and*
- *the flow relation $F \subseteq (P \times T) \cup (T \times P)$ is a set of edges.*

Given a Petri net graph, a marking U for the net is a multiset of places such that $U : P \to \mathbb{N}$. We call Petri net any net N equipped with an initial marking U_N.

The preset of a transition t is the set of its input places: ${}^\bullet t = \{p \mid (p, t) \in F\}$; its postset is the set of its output places: $t^\bullet = \{p \mid (t, p) \in F\}$. Definitions of pre- and postsets of places ${}^\bullet p$ and $p^\bullet$ are analogous. We denote

$$\frac{U : P \to \mathbb{N}}{U : U \to U \in \mathcal{T}(N)}$$

$$\frac{t \in T_N}{t : {}^\bullet t \to t^\bullet \in \mathcal{T}(N)}$$

$$\frac{r : U \to V,\ r' : U' \to V' \in \mathcal{T}(N)}{r + r' : U + U' \to V + V' \in \mathcal{T}(N)}$$

$$\frac{r : U \to V,\ s : V \to W \in \mathcal{T}(N)}{r; s : U \to W \in \mathcal{T}(N)}$$

Figure 27: Inference rules for $\mathcal{T}(N)$.

the empty multiset by 0, multiset union by $+$, multiset difference by $-$, multiset inclusion by $\subseteq$ and write $a \in U$ if $U(a) > 0$.

Definition 4 (Firing)**.** *Given a Petri net graph (P, T, F) and a marking U a transition t is* enabled *in U if ${}^\bullet t \subseteq U$. A transition t enabled in U can* fire *leading to the marking $V = U - {}^\bullet t + t^\bullet$.*

A multiset of transitions can fire concurrently, if U contains enough tokens to cover all their presets. After [MM90], we denote computations over a Petri net as terms of the algebra $\mathcal{T}(N)$ freely generated by the inference rules in Fig. 27 modulo the axioms below (whenever both sides are well-defined):[1]

$$\begin{array}{rl}
\text{monoid:} & (p + q) + r = p + (q + r) \\
& r + r' = r' + r \\
& 0 + r = r \\
\text{category:} & (p; q); r = p; (q; r) \\
& r; V = r = U; r \\
\text{functorial:} & (p; p') + (q; q') = (p + q); (p' + q')
\end{array}$$

Each term $r : U \to V \in \mathcal{T}(N)$ defines a concurrent computation over N, from the marking U to the marking V. We write $N \xrightarrow{*} V$ and say that

[1] For category-minded theorists, the computations form the arrows of a freely generated, strictly symmetric, strict monoidal category whose objects are markings.

$$
\begin{array}{lllll}
\text{(ACT)} & A, B & ::= & a \mid \mathit{skip} \mid \mathit{throw} \\
\text{(STEP)} & X & ::= & A \div B \\
\text{(PROCESS)} & P, Q & ::= & X \mid P; Q \mid P|Q \\
\text{(SAGA)} & S, T & ::= & A \mid S; T \mid S|T \mid \{\![P]\!\}
\end{array}
$$

Figure 28: Syntax for Sagas

V is *reachable* in N, if there exists a computation $r : U_N \to V \in \mathcal{T}(N)$ where U_N is the initial marking of the net N. A Petri net N is *1-safe* if for any place a and for any reachable marking V we have $V(a) \leq 1$. We say that $r : U_N \to V \in \mathcal{T}(N)$ is *maximal* if no transition is enabled in V. We will use this notion in Theorem 5.

4.2 From Sagas to Petri nets

In this section we present an encoding of Sagas into Petri nets. We define the Petri net graph associated to a process by structural induction on the syntax which was defined in Section 2.2 and is shown once more in Figure 28 for the reader's convenience. We distinguish between compensable processes and sagas. A compensable process can be a compensation pair with a forward action and its compensation, or the sequential or parallel composition of compensable processes. The Petri net for a compensable process should model the following behaviour: It should imitate the forward flow as well as the backward flow and in case of a fault it should be able to inform siblings. Moreover if it receives the information of a fault it should activate compensations.

At the high-level view, the Petri net for each compensable process is a black box with six external places to be interfaced with the environment (Fig. 29). Places F_1 and F_2 are used for propagating the forward flow of execution: a token in F_1 starts the execution and a token in F_2 indicates that the execution has ended successfully. Places R_1 and R_2 control the reverse flow: a token in R_1 starts the compensation, a token in R_2 indicates that the compensation has ended successfully. The place I_1 is

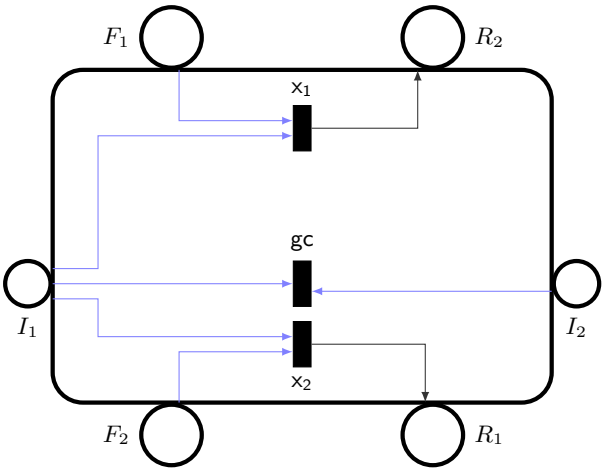


Figure 29: A compensable process P

used to interrupt the process from the outside while a token in I_2 is used to inform the environment that an error has occurred. The figure highlights that three auxiliary transitions will be present in any process: two of them (transitions x1 and x2) handle the catching of the interrupt and reversal of the flow, the other one (transition gc) handles the disposal of the interrupt in case the process already produced a fault. This garbage collection consumes the interrupt tokens that have no further purpose.

Supposedly for compensable processes we expect to have the following kinds of computations:

Successful (forward) computation: from marking F_1 the net reaches marking F_2.

Compensating (backward) computation: from marking R_1 the net reaches marking R_2.

Aborted computation: from marking F_1 the net reaches marking R_2+I_2.

Interrupted computation: from marking F_1+I_1 the net reaches marking R_2.

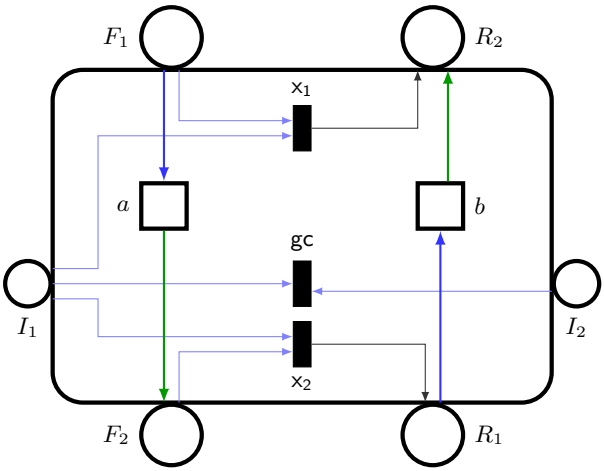


Figure 30: Petri net for a compensation pair $a \div b$

In Theorem 5 we will formalize this behaviour and prove that our model behaves accordingly.

The nets for compensable processes are depicted in Figures 30 to 33. When drawing transitions we use larger boxes for representing sagas activities and thinner black-filled boxes for auxiliary transitions. As in Figure 26 incoming arcs for transitions are blue, outgoing arcs are green. For a compensation pair $a \div b$ (Figure 30) there is a transition called a that consumes a token in F_1 and produces a token in F_2 representing the execution of the forward flow, as well as a transition b that consumes a token in R_1 and produces a token in R_2 corresponding to the reverse flow. The net for *skipp*, not shown here, replaces the transitions a and b by the vacuous silent activity *skip*.

The net for the primitive *throww* is displayed in Figure 31. The transition k models the abort of the transaction. It consumes the token in F_1 for the forward flow and produces a token in R_2 for the continuation of the reverse flow and in I_2 to inform the environment of the abort.

In the net for the sequential composition for a compensable process $P; Q$ (Figure 32) the token in F_3 produced by P for the forward flow is passed on to start the execution of the process Q. Equally the token

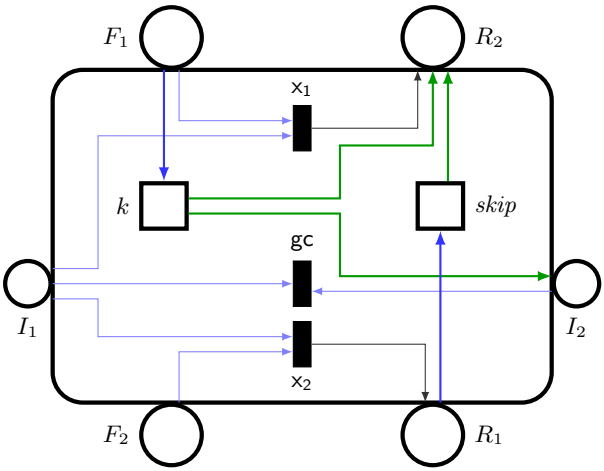


Figure 31: Petri net for *throww*

in R_3 for the reverse flow produced by Q is passed on to P to start its compensation. In the encoding P and Q share the places for I_1 and I_2.

For the parallel composition $P|Q$ (Figure 33) we use two subnets for the two processes, with external places $PF_1, PF_2, \ldots$ and $QF_1, QF_2, \ldots$ respectively. To start the execution of the forward flow there is a fork transition producing tokens in PF_1 and QF_1 as well as an additional token MEX working as a semaphore. Without an error or an interrupt it is collected together with the tokens in PF_2 and QF_2 at the end of the execution in the transition join producing a token in F_2. If something goes wrong the token in MEX is collected to prevent the completion of the forward flow. A fork-and-join mechanism is used also for the reverse flow, though no additional tokens are needed. If during the forward computation an interrupt is received, *i.e.*, there is a token in I_1, it is split using the transition i_{in} into PI_1 and QI_1 which are processed by P and Q. Here we need the semaphore MEX to guarantee that the interrupt is only split during the execution of the parallel composition. If an error occurs inside one of the processes P or Q, the places PI_2 and QI_2 are used to inform the respective sibling and the environment: the transition i_{p1} (respectively i_{p2}) consumes a token from PI_2 (respectively QI_2) and

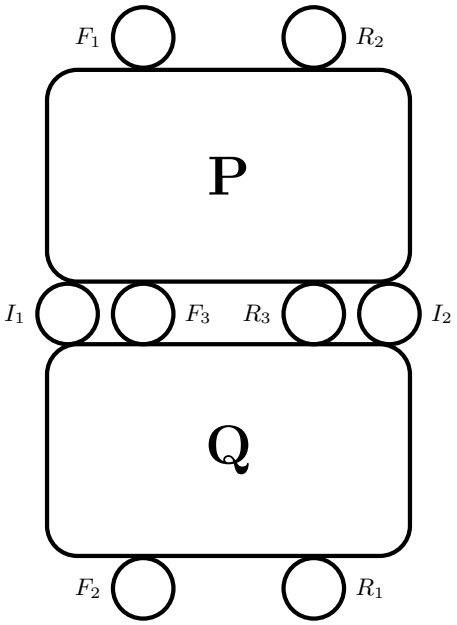


Figure 32: Petri net for a sequential composition $P; Q$

produces a token in I_2 and in QI_1 (respectively in I_2 and in PI_1). The semaphore MEX guarantees in this case that only one branch sends the interrupt to the environment, and that no interrupt is sent if an external interrupt has been already received. As usual we have the possibility to interrupt the process in the beginning or end of the execution with the transitions x_1 and x_2, and the garbage collecting transition.

A standard process or saga can be a transaction, a basic action or the sequential or parallel composition of sagas. The Petri net for a saga has just three places to interact with the environment: F_1 starts its flow, F_2 signals successful termination, and E raises a fault. There is no reverse flow and we do not allow interruption of sagas, thus we can drop the other places. Intuitively for standard processes a computation starting in F_1 will lead either to F_2 or to E.

Figure 34 shows the Petri net for a transaction scope. It embeds the

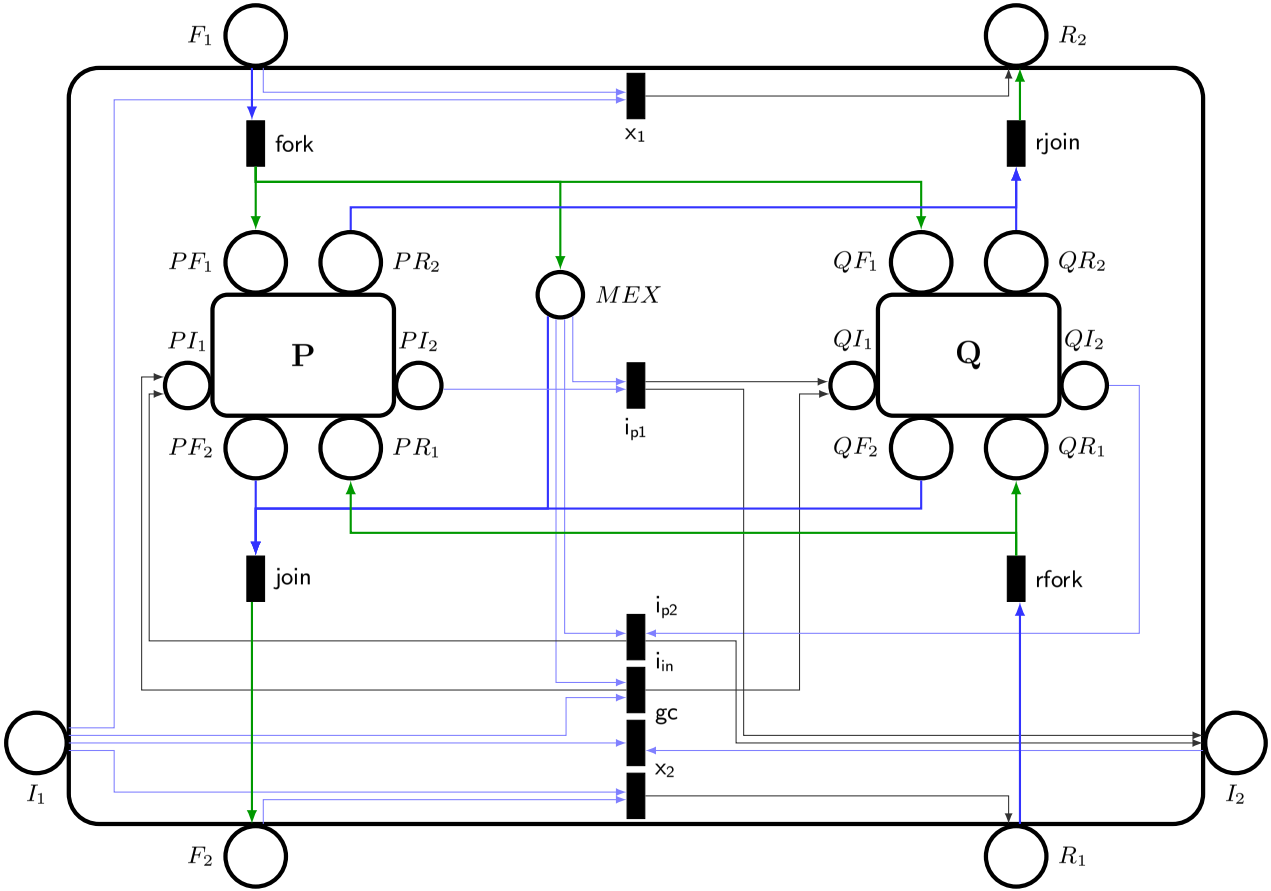


Figure 33: Petri net for a parallel composition $P|Q$

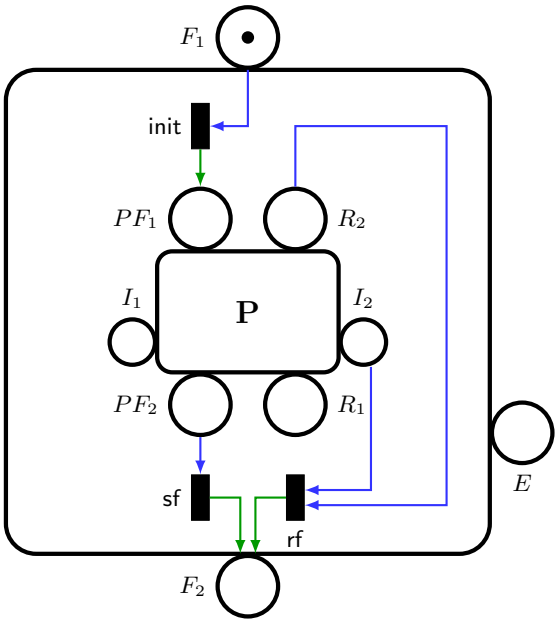
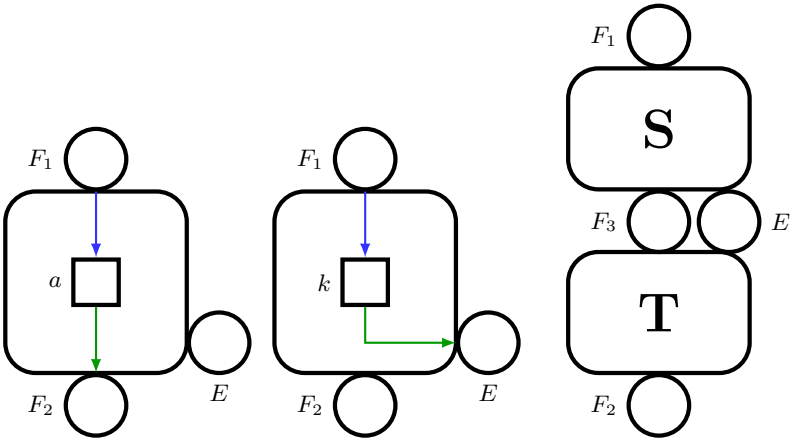


Figure 34: Petri net for a transaction

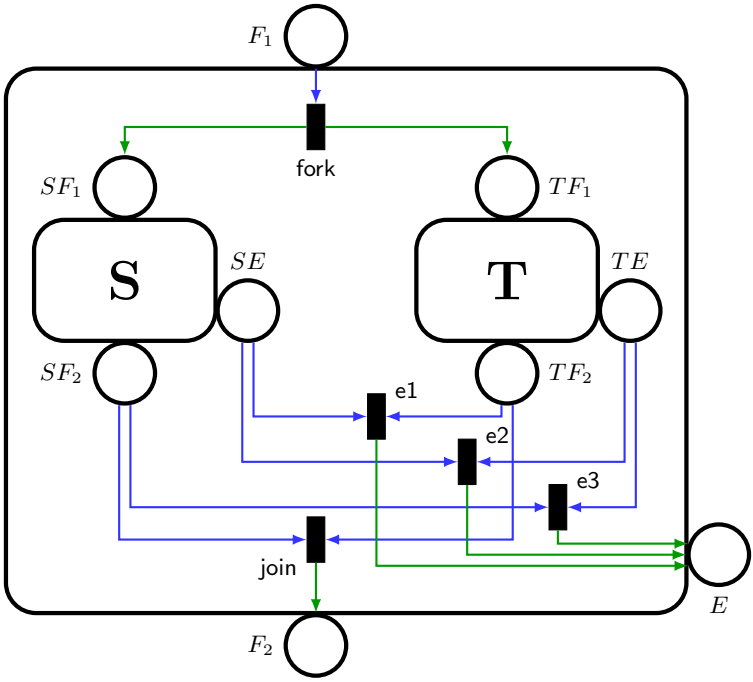
structure of a compensable process with six external places into the structure of a saga with three places. The transaction is activated putting a token in the place PF_1 to start the forward flow of the compensable process. We expect only two possible results: Either the transaction is successful, then the token will end in PF_2 and passed on to F_2 (transition sf). Or the transaction aborts and compensates, then we expect a token in R_2 and I_2. These are as well passed on to F_2 (transition rf), thus F_2 signals that the transaction has reached a consistent state. Place E on the other hand corresponds to an inconsistent state. So far it cannot be generated by a transaction as we do not allow the failure of compensations.

The nets for a basic activity, *throw*, sequential and parallel composition of sagas are displayed in Figure 35.

Example 6. *Figure 36 shows an example of an encoded transaction* $\{\![A \div B; \mathit{throww}]\!\}$ *consisting of a sequential composition of a compensation pair and a* throww*. Given a token in* F_1 *the transition* init *fires first. Then activity* A *is*



(a) A basic activity **(b)** *throw* **(c)** Sequential composition



(d) Parallel composition

Figure 35: Petri nets for sagas

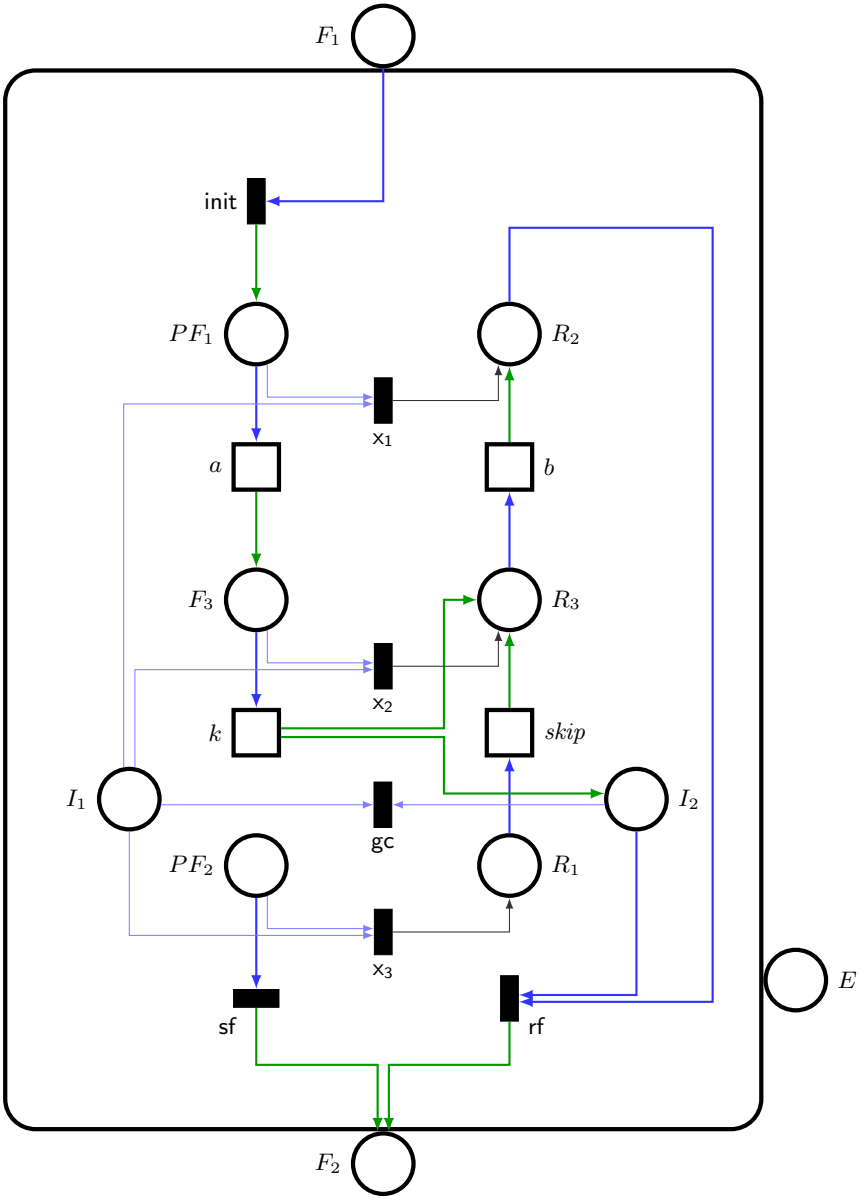


Figure 36: Example Petri net for the encoded saga $\{\![A \div B; throww]\!\}$

executed followed by transition k that activates the compensation and sends an interrupt via I_2 to the environment. After the compensation B the transition rf *fires taking the transaction back to a consistent state.*

We will now show that every (compensable) process satisfies some basic behavioural properties, which match the intuition given on page 61 concerning the meaning of the different places and transitions.

Theorem 5. *Given a compensable process P and the corresponding net N_P with external places F_1, F_2 for the forward flow, R_1, R_2 for the reverse flow and I_1, I_2 for interrupts, we can state the following properties:*

1. *Every maximal execution of the net N_P with initial marking F_1 is either of the form $f_P : F_1 \to F_2$ (successful computation), or of the form $a_P : F_1 \to R_2 + I_2$ (aborted computation);*
2. *Every maximal execution of the net N_P with initial marking R_1 is of the form $r_P : R_1 \to R_2$ (backward computation);*
3. *Every maximal execution of the net N_P with initial marking $F_1 + I_1$ is of the form $i_P : F_1 + I_1 \to R_2$ (interrupted computation).*

Proof. The proof is by structural induction. The theorem holds trivially for compensation pairs as well as for the primitives *skipp* and *throww*. For the sequential composition the proof follows from the induction hypothesis.

Consider a parallel composition $P|Q$ of two compensable processes. For the first case assume a token is in F_1. Using fork we create a marking $PF_1 + QF_1 + MEX$. Depending on the behaviour of the two subnets we consider two cases. If both branches succeed, then by induction hypothesis there are $f_P : PF_1 \to PF_2$ and $f_Q : QF_1 \to QF_2$. The join transition consumes both PF_2 and QF_2 as well as MEX and creates a token in F_2 as desired. Note that a transition like $f_P + f_Q$, due to the distributive law given in [MM90], corresponds to all the possible interleaving of the two transitions.

In the other case we assume that P aborts (the case where Q aborts is symmetric). By induction hypothesis there is $a_P : PF_1 \to PR_2 + PI_2$. The transition $\mathsf{i_{p1}}$ creates a token in QI_1, thus for Q we go to the case of interrupted computation with $i_Q : QF_1 + QI_1 \to QR_2$. The hypothesis follows firing the transition rfork. Note that i_Q may include computations where Q aborts as well. Then QI_1 and QI_2 are garbage collected by transition gc.

The case of backward computation is easy: Starting with a token in R_1 the transition rfork is fired producing PR_1 and QR_1. The induction hypothesis then leads to tokens PR_2 and QR_2 triggering rjoin. This satisfies the hypothesis.

Finally, consider the case of interrupted computation. If we fire the transition x_1 before fork the thesis holds trivially. Equally if after fork both processes run successfully and the join is fired, then x_2 is taken, producing a token in R_1. As described in the previous case we reach R_2. Now assume instead that transition i_{in} is taken during forward computation, producing as marking $PF_1 + QF_1 + PI_1 + QI_1$. We can apply the induction hypothesis to reach a marking $PR_2 + QR_2$. Executing transition rjoin this satisfies the hypothesis. □

Note that we call *interrupted* any computation that consumes the token I_1: It may as well happen that the net autonomously aborts (due to some *throww*), and the token I_1 is consumed by the garbage collection transition.

We can state similar to the theorem above some behavioural properties for sagas:

Corollary 3. *Let S be a saga and N_S its corresponding net with external places F_1, F_2, E and initial marking F_1. Then any maximal execution of N_S leads either to F_2 or to E.*

Proof. The proof is by structural induction on S. It is trivial except for the case of a transaction scope. Here we apply the result of Theorem 5 to show that the Petri net for a compensable process with initial marking F_1 has only two possible final markings. Both lead with either transition sf or rf to marking F_2. □

The next proposition states a different behavioural property:

Proposition 2. *Let P be a compensable process and N_P the corresponding net with external places $F_1, F_2, R_1, R_2, I_1, I_2$. Then:*

(i) N_P with initial marking $F_1 + I_1$ is 1-safe, and

(ii) N_P with initial marking $R_1 + I_1$ is 1-safe.

Moreover, let S be a saga and let N_S be its corresponding net with external places F_1, F_2, E and initial marking F_1. Then, N_S is 1-safe.

Proof. The proof is by structural induction on P and S. Note that the only way to generate multiple tokens in one place is when several branches in a concurrent process abort and multiple interrupts are sent. Here the semaphore MEX guarantees that still only one branch can actually broadcast the interrupt while the others are collected by garbage collection. □

As a consequence of the proposition we can state the following corollary:

Corollary 4. *Let P be a compensable process and N_P the corresponding net with external places $F_1, F_2, R_1, R_2, I_1, I_2$. Then:*

(i) N_P with initial marking F_1 is 1-safe, and

(ii) N_P with initial marking R_1 is 1-safe.

4.3 Correspondence

In this section we prove that the presented operational semantics using Petri nets models the policy *coordinated compensation* introduced in the previous chapter. We therefore show its observational equivalence to the denotational semantics (Figure 37).

In order to show the correspondence we need to introduce some kind of observation for the operational semantics.

Definition 5. *Let P be a compensable process and N_P its corresponding net. For any $f \in \mathcal{T}(N_P)$ we define the set $label(f)$ of action sequences as follows:*

$$\begin{aligned} label(a) &= \{a\} \text{ for any basic activity } a \\ label(k) &= \{k\} \text{ for any throw transition } k \\ label(f_1; f_2) &= label(f_1)label(f_2) \\ label(f_1 + f_2) &= label(f_1)|||label(f_2) \\ label(f) &= \epsilon \text{ otherwise} \end{aligned}$$

where juxtaposition and interleaving are defined element-wise.

It is immediate to check that the function $label$ is well-defined, in the sense that it is invariant with respect to the equivalence axioms on $\mathcal{T}(N)$. To demonstrate the use of this function consider a compensation pair

COMPOSITION OF STANDARD TRACES

$$\textbf{Sequential} \quad \begin{cases} p\langle\checkmark\rangle; q \triangleq pq \\ p\langle\omega\rangle; q \triangleq p\langle\omega\rangle \text{ when } \omega \neq \checkmark \end{cases}$$

$$\textbf{Parallel} \quad p\langle\omega\rangle||q\langle\omega'\rangle \triangleq \{r\langle\omega\&\omega'\rangle \mid r \in (p|||q)\},$$

where

ω	!	!	!	?	?	$\checkmark$
ω'	!	?	$\checkmark$	?	$\checkmark$	$\checkmark$
$\omega\&\omega'$	!	!	!	?	?	$\checkmark$

and

$$\begin{cases} p|||\epsilon \triangleq \epsilon|||p \triangleq \{p\} \\ Ap|||Bq \triangleq \{Ar \mid r \in (p|||Bq)\} \cup \{Br \mid r \in (Ap|||q)\} \end{cases}$$

COMPOSITION OF COMPENSABLE TRACES

$$\textbf{Sequential} \quad \begin{cases} (p\langle\checkmark\rangle, p'); (q, q') \triangleq (pq, q'; p') \\ (p\langle\omega\rangle, p'); (q, q') \triangleq (p\langle\omega\rangle, p') \text{ when } \omega \neq \checkmark \end{cases}$$

$$\begin{array}{rcll} (p\langle\checkmark\rangle, p')||(q\langle\checkmark\rangle, q') & \triangleq_5 & \{(r\langle\checkmark\rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} & \\ (p\langle\omega\rangle, p')||(q\langle\omega'\rangle, q') & \triangleq_5 & itp((p\langle\omega\rangle, p'), (q\langle\omega'\rangle, q')) \cup & \\ & & itp((q\langle\omega'\rangle, q'), (p\langle\omega\rangle, p')) & \\ & & \text{when } \omega, \omega' \neq \checkmark & \\ (p\langle\omega\rangle, p')||(q\langle\omega'\rangle, q') & \triangleq_5 & \emptyset \quad \text{otherwise} & \end{array}$$

Parallel

$$itp((p\langle\omega\rangle, p'), (q\langle\omega'\rangle, q')) \triangleq_5 \{((p|||q_1)\langle\omega\rangle, (p'||q_2q')) \mid q = q_1q_2\}$$

TRACES OF SAGAS

$$a \triangleq \{a\langle\checkmark\rangle\} \quad skip \triangleq \{\langle\checkmark\rangle\} \quad throw \triangleq \{\langle!\rangle\}$$

$$S;T \triangleq \{p; q \mid p \in S \wedge q \in T\}$$

$$S|T \triangleq \{r \mid r \in (p||q) \wedge p \in S \wedge q \in T\}$$

$$\{[P]\} \triangleq \{p\langle\checkmark\rangle \mid (p\langle\checkmark\rangle, q) \in P\} \cup \{pq \mid (p\langle!\rangle, q) \in P\}$$

TRACES OF COMPENSABLE PROCESSES

$$\begin{array}{lcl} A \div B & \triangleq_5 & \{(A\langle\checkmark\rangle, B\langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle), (A\langle?\rangle, B\langle\checkmark\rangle)\} \\ skipp & \triangleq & \{(\langle\checkmark\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\ throww & \triangleq & \{(\langle!\rangle, \langle\checkmark\rangle), (\langle?\rangle, \langle\checkmark\rangle)\} \\ P; Q & \triangleq & \{pp; qq \mid pp \in P \wedge qq \in Q\} \\ P|Q & \triangleq & \{rr \mid rr \in (pp||qq) \wedge pp \in P \wedge qq \in Q\} \end{array}$$

Figure 37: Denotational semantics for Sagas according to policy #5

$A \div B$. With initial marking $F_1 + I_1$ a possible computation in the encoded Petri net is $A \times_2 B$. Applying the $label$ function the result is $A\ B$. Moreover we define a function $filter(f)$ that removes every k from a label f. Using this definition we can now formulate the correspondence theorem.

Theorem 6 (Correspondence)**.** *Let P be a compensable process and N_P the corresponding net with external places F_1, F_2 for the forward flow, R_1, R_2 for the reverse flow and I_1, I_2 for interrupts. The correspondence of denotational and (maximal computations of the) operational semantics is given as follows:*

1. *$(p\langle\checkmark\rangle, q\langle\checkmark\rangle) \in P$ iff there is a computation $f : F_1 \to F_2 \in \mathcal{T}(N_P)$ with $p \in label(f)$ and a computation $r : R_1 \to R_2 \in \mathcal{T}(N_P)$ with $q \in label(r)$.*

2. *$(p\langle!\rangle, q\langle\checkmark\rangle) \in P$ iff there is a computation $a : F_1 \to I_2 + R_2 \in \mathcal{T}(N_P)$ with label $pkq' \in label(a)$ for some q' such that $q = filter(q')$.*

3. *$(p\langle?\rangle, q\langle\checkmark\rangle) \in P$ iff there is a marking U and two computations $f : F_1 \to U, i : U + I_1 \to R_2 \in \mathcal{T}(N_P)$ such that $p \in label(f)$ and $q = filter(q')$ for some $q' \in label(i)$.*

Proof. The proof is by induction on the structure of the process (which corresponds to an induction on the structure of the corresponding net), with a case analysis similar to the one of Theorem 5.

The theorem holds trivially for compensation pairs, $skipp$ and $throww$.

Let us consider sequential composition $P; Q$. We have two kinds of traces: the ones where P succeeds, and the ones where it does not. Let us consider the first case. By induction hypothesis we have a computation $f_P : F_1 \to F_3$ with $label(f_P) = p$. Then we have again a case analysis according to the behaviour of Q. We consider just the case of success, the other being similar. In this case, by the induction hypothesis we have a computation $f_Q : F_3 \to F_2$ with $label(f_Q) = p'$ and then computations $r_Q : R_1 \to R_3$ with $label(r_Q) = q'$ and $r_P : R_3 \to R_2$ with $label(r_P) = q$. Thus the two computations $f_P; f_Q$ and $r_Q; r_P$ satisfy the thesis. The case where P does not succeed is trivial by induction hypothesis.

Let us consider now parallel composition $P|Q$. We have again two possibilities: either both P and Q succeed, or not. Let us consider the first case. Operationally, first transition fork is executed. Then the induction hypothesis is applied to the two subnets for the forward flow. Finally

transition join is executed. The analysis of the backward flow is similar. It is easy to see that labels are the desired ones.

Let us consider the case where at least one branch aborts. First transition fork is executed. We have then a case analysis according to the behaviour of the two subnets. Assume P aborts. By induction hypothesis there is a computation $a_P : PF_1 \to PI_2 + PR_2$. Assume that Q is interrupted. Again by induction hypothesis, there are computations $f_Q : QF_1 \to U$ and $i_Q : U + QI_1 \to QR_2$. The only constraint on the possible interleaving is that i_Q may only start after PI_2 has been produced. This is the behaviour captured by function itp in the parallel composition of compensable traces.

In the case of double abort the two subnets start compensating on their own, thus there is no synchronization constraint to be satisfied (both the sets defining function itp become not empty). The two notifications are garbage collected.

The case of external interrupt is similar. The only difference is that if the interrupt is processed after the two processes have finished their computations successfully, then transition x_2 is used. In the denotational semantics there is no clause corresponding to this, but this produces the same traces of two yielding computations (and we always have a yielding computation for each successful one).

□

The correspondence can be extended to sagas as follows:

Corollary 5. *Let S be a saga and N_S its corresponding net with external places F_1, F_2 and E. Then:*

1. *$p\langle\checkmark\rangle \in_5 S$ iff there is a computation $f : F_1 \to F_2 \in \mathcal{T}(N_S)$ with $p \in label(f)$ and*

2. *$p\langle!\rangle \in_5 S$ iff there is a computation $a : F_1 \to E \in \mathcal{T}(N_S)$ with $p \in label(a)$.*

Proof. All the cases are trivial but the one for a transaction. This case follows from Theorem 6. The only possibilities for the internal compensable process are to succeed, causing the success of the transaction, or to abort and compensate, causing again the success of the transaction. Note that there is no possibility of external interrupt. Instead notifications to the outside are discarded. □

4.4 Logical Properties

In this section we show that a transaction $\{\!|P|\!\}$ satisfies some basic logical properties following their intuitive behaviour. First, following [BMM05], we define the concept of order of the activities in a transaction. To this end we need activities with a unique name. Also, we consider $throww$ as a forward activity and use subscripts to distinguish between multiple occurrences of the same activity, like in $\{\!|throww_1 | A_1 \div B; A_2 \div C; throww_2|\!\}$.

We let $\mathbf{A}(S)$ be the set of activities of a transaction including $throww$s.

Definition 6 (Order of a transaction)**.** *The strict order of a transaction S is the least transitive relation $\prec_S$ such that:*

1. *if $A \div A'$ occurs in S then $A \prec_S A'$;*
2. *if $P; Q$ occurs in S then $A \prec_S B$ for each forward activity A occurring in P and any forward activity B in Q;*
3. *if $A \div A'$ and $B \div B'$ occur in S and $A \prec_S B$ then $B' \prec_S A'$.*

We let $pred_S(A) \triangleq \{B \in \mathbf{A}(S) \mid B \prec_S A\}$ be the set of the predecessors of the activity A w.r.t. the order $\prec_S$. We say a sequence $A_1 A_2 ... A_n$ *respects* the order $\prec_S$ if $A_i \prec_S A_j$ implies $i < j$ for any $1 \leq i < j \leq n$.

Theorem 7 (Completion)**.** *Let $S = \{\!|P|\!\}$ be a transaction. If P contains no $throww$ activities, then it will succeed. In this case there exists a unique maximal computation $f \in \mathcal{T}(N_P)$ with initial marking F_1 and it leads to F_2. Furthermore, $label(f)$ is the set of possible interleavings of all forward activities in $\mathbf{A}(S)$ that respect $\prec_S$.*

Proof. By induction on the structure of the process inside the transaction S. □

Theorem 8 (Successful compensation)**.** *Let $S = \{\!|P|\!\}$ be a transaction. If P contains at least a $throww$ activity, then it will abort and it will be compensated. In this case all the maximal computations in the net N_P with initial marking F_1 end in $R_2 + I_2$. Then, for any such computation $a : F_1 \rightarrow R_2 + I_2 \in \mathcal{T}(N_P)$ we have that each possible label in $filter(label(a))$ satisfies the conditions below:*

1. *activities in the label respect the order $\prec_S$;*

2. *any activity A such that $A \prec_S throww_i$ for all $throww_i$ in $\mathbf{A}(S)$, occurs in the label;*

3. *no forward activity A such that there exists a $throww_i$ in $\mathbf{A}(S)$ with $throww_i \prec_S A$, occurs in the label;*

4. *if activity A' is the compensation of activity A, then A occurs in the label iff A' occurs in the label;*

5. *there exists at least one $throww_i$ such that all activities in $pred_S(throww_i)$ appear in the label and they precede each compensation activity A' appearing in the label.*

Moreover, for any action sequence q satisfying conditions 1–5 above, there exists a maximal computation $a : F_1 \to R_2 + I_2$ such that $q \in filter(label(a))$.

Proof. First we prove by structural induction using Theorem 5 that any maximal computation starting from F_1 ends in $R_2 + I_2$ with no token ever appearing in F_2 (needed for proving property 3, below).

Next, we take any $a : F_1 \to R_2 + I_2$ and $q \in filter(label(a))$ and show that 1–5 hold for q.

Properties 1 and 4 are proved by structural induction on P.

For property 2, since $A \prec_S throww_i$ for all $throww_i$, there must exist P' and Q' such that $P = C[P'; Q']$ for some context $C[\cdot]$, where P' contains A and Q' contains all $throww_i$. Then, we conclude by structural induction on the shape of the context $C[\cdot]$, by applying Theorem 7 to P'.

For property 3, let $throww_i$ be such that $throww_i \prec_S A$, therefore there exist P' and Q' such that $P = C[P'; Q']$ for some context $C[\cdot]$, where P' contains $throww_i$ and Q' contains A. Therefore by applying the first argument of this proof to P' we know that activities in Q' are never enabled.

For property 5 we proceed by contradiction. Suppose that there exists a compensation activity A' such that for any activity $throww_i$ an activity $B_i \in pred_S(throww_i)$ can be found such that A' precedes B_i in q. Without loss of generality, let A' be the leftmost such activity appearing in q. Hence $q = A_1 \cdots A_n A' q'$ for some forward activities $A_1, \ldots, A_n$ and sequence q' that contains all B_i's. But then the firing of $A_1 \cdots A_n$ leads to a marking where no $throww_i$ is enabled and therefore A' cannot be enabled, which is absurd.

For the last part, let q be any action sequence that satisfies conditions 1–5 and let $A_1 \cdots A_n$ be the subsequence of q formed by forward activities. By condition 4, their backward activities $A'_1, \ldots, A'_n$ are the

only other activities that appear in q and we let $A'_{i_1} \cdots A'_{i_n}$ be the corresponding subsequence of q. By conditions 1–3 the sequence of transitions $A_1 \cdots A_n$ can be fired (possibly firing additional fork and join) starting from F_1. By condition 5 there is a $throww_i$ such that all activities in $pred_S(throww_i)$ are in $A_1, \ldots, A_n$ and therefore the transition k associated with $throww_i$ is enabled after the firing of $A_1 \cdots A_m$ for some $m \leq n$, which is a prefix of q. Note that the propagation of interrupts by transitions $\mathsf{i_{in}}$, $\mathsf{i_{p1}}$ and $\mathsf{i_{p2}}$ can be delayed until A_n is fired. Therefore the sequence of transitions $A_1 \cdots A_n A'_{i_1} \cdots A'_{i_n}$ is fireable, which induces a computation $a : F_1 \to R_2 + I_2$. It remains to show that $q \in filter(label(a))$, which can be done by induction on the number of action switches needed to transform $A_1 \cdots A_n A'_{i_1} \cdots A'_{i_n}$ to q exploiting the functorial axiom. Note that in fact one has to swap only some forward actions A_i (for $i > m$) and some backward actions A'_j (possibly with the interrupt propagation transitions that enables A'_j), such that $A_i \not\prec_S A'_j$ (by condition 1). □

Conditions 1 to 4 correspond to the conditions already presented in [BMM05] (translated into our notation) , while condition 5 does not hold in [BMM05]: it characterizes the fact that our semantics allows only realistic traces, where compensations are not started before a fault is actually executed. Since faults are removed from labels, we consider that a fault can be executed when all the observable activities preceding it have been executed, and thus enables the failing action.

Example 7. *Consider the Saga $S = \{[C \div C' \mid (A; throww_1) \mid (B; throww_2)]\}$. Here we have $pred_S(throww_1) = \{A\}$ and $pred_S(throww_2) = \{B\}$. Then the trace $CC'AB$ can not be observed, since C' is preceded by neither A nor B. However, this trace is valid according to the semantics of policy #4.*

4.5 Dealing with other compensation policies

In this section we present how other compensation policies can be modelled using Petri nets. Figure 38 presents an overview of the previously introduced policies that are distinguished along two aspects. The first differentiation regards interruption of siblings while the other aspect concerns the centralized or distributed activation of compensations. We al-

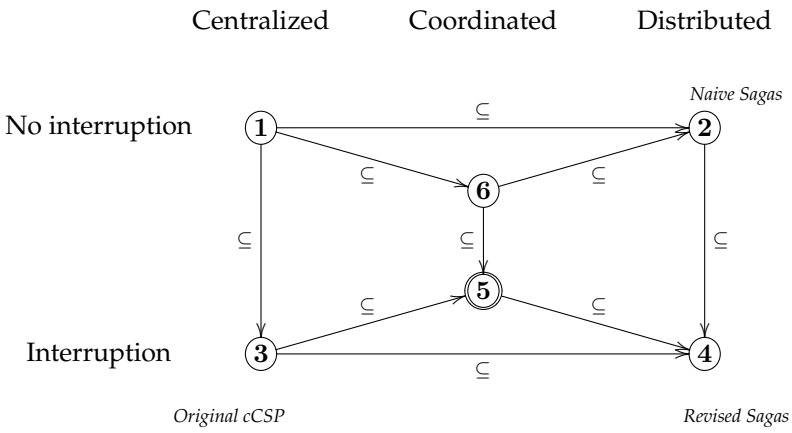


Figure 38: Compensation policies (arrows stand for trace inclusion)

ready presented the encoding of policy #5 and introduce here the other encodings.

As a first item we consider the coordinated compensation without interrupt (policy #6). It can be obtained from policy #5 by removing any transitions $\mathsf{x} : F + I_1 \to R$ used to interrupt a process. We only allow interrupting transitions $\mathsf{x_P} : PF_2 + PI_1 \to PR_1$ and $\mathsf{x_Q} : QF_2 + QI_1 \to QR_1$ for a parallel composition $P|Q$ at the end of each branch. That way if one branch aborts the other can receive the interrupt after finishing its forward flow and start compensating. Moreover while in policy #5 for a sequential composition $P;Q$ the place I_1 is shared, in policy #6 we introduce a new place PI_1 while Q uses I_1. This ensures if P is a parallel composition and we receive an interrupt on I_1 then Q will still be executed.

Section 4.5.1 presents the encoding for policy #3 while in Section 4.5.2 we show the encoding of policy #4. Both sections give a correspondence theorem to the respective denotational semantics. At the end of each section we show how to obtain the encoding for the respective policy without interruption (policies #1 and #2).

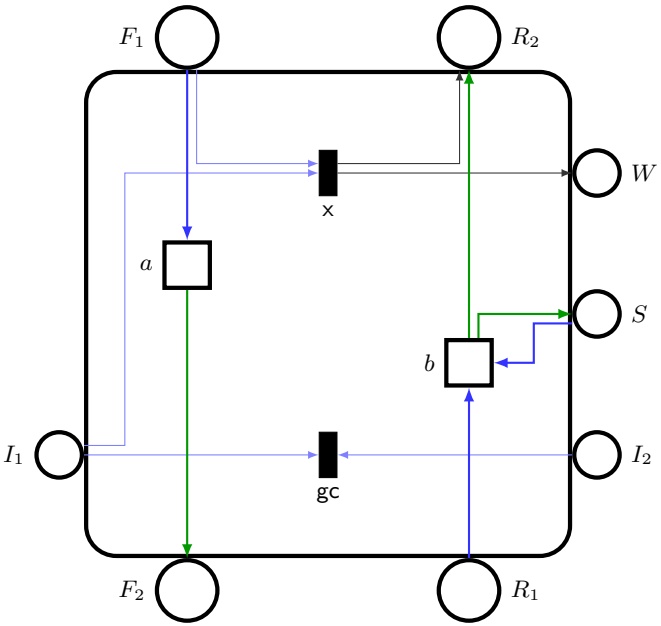


Figure 39: Petri net for a compensation pair according to policy #3

4.5.1 Centralized Compensation

In the centralized case with interruption (policy #3), compensations of concurrent processes are activated after a synchronization point that guarantees each branch has either aborted or was interrupted by another (aborting) branch. The encoding works similar to the one for policy #5, with some additions in order to guarantee that compensations are only started once every branch is informed about the abort. We add the following places to the interface of every compensable process:

- the place W is used for branches that are waiting, they have been interrupted or aborted and wait for the confirmation to start their compensation,

- a place S that will start the execution of the compensation.

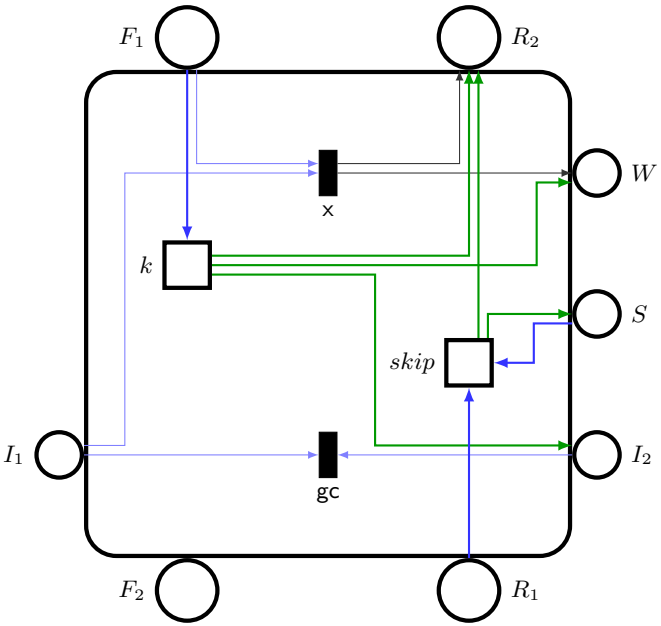


Figure 40: Petri net for *throww* according to policy #3

Additionally to the specific transitions for the encoding of each compensable process we define the following transition that are part of every encoding. These transitions are:

$$\begin{array}{rlcl} \mathsf{x}: & F_1 + I_1 & \to & R_2 + W \\ \mathsf{gc}: & I_1 + I_2 & \to & \emptyset \end{array}$$

The first indicates that the forward flow was interrupted and is now waiting to start compensating, the other one is used for garbage collection. Note that we do not allow interruption at the end (transition x_2 in Figure 29) due to the correspondence with the denotational semantics.

For a compensation pair $A \div B$ (Figure 39) to the transitions given above we add $a : F_1 \to F_2$, executing the activity A, and $b : R_1 + S \to R_2 + S$ for executing compensation B given a token in S. The token in S is put back for any continuation. That way we can guarantee branches compensate only if they have a token in S indicating that each branch

finished its forward flow.

For the primitive *throww* (Figure 40) instead we add the transition $k : F_1 \to R_2 + I_2 + W$ indicating an abort and waiting for possible other siblings to finish their forward flow. For the sequential composition we define new intermediate places for the forward and reverse flow. The places I_1, I_2 are shared as in policy #5, additionally S and W are shared as well.

The encoding for the parallel composition (Figure 41) extends the one from policy #5 with the handling of the waiting mechanism. Apart from the encoding of the two branches with fresh names and the usual transitions for standard processes, the following transitions are added:

$$
\begin{array}{rlcl}
\mathsf{fork}: & F_1 & \to & PF_1 + QF_1 + MEX + H \\
\mathsf{join}: & PF_2 + QF_2 + MEX + H & \to & F_2 \\
\mathsf{rfork}: & R_1 & \to & PR_1 + QR_1 + H \\
\mathsf{rjoin}: & PR_2 + QR_2 + PS + QS & \to & R_2 + S \\
\mathsf{sfork}: & S + H & \to & PS + QS \\
\mathsf{wjoin}: & PW + QW & \to & W \\
\mathsf{x_P}: & PF_2 + PI_1 & \to & PR_1 + PW \\
\mathsf{x_Q}: & QF_2 + QI_1 & \to & QR_1 + QW \\
\mathsf{x_2}: & F_2 + I_1 & \to & R_1 + W \\
\mathsf{i_{in}}: & I_1 + MEX & \to & PI_1 + QI_1 \\
\mathsf{i_{p1}}: & PI_2 + MEX & \to & I_2 + QI_1 \\
\mathsf{i_{p2}}: & QI_2 + MEX & \to & I_2 + PI_1
\end{array}
$$

The first four are used for the fork and join of the forward and the reverse flow. The token MEX is used as before for the handling of interrupts. The place H ensures that we split the token S for compensating only inside the parallel composition, as can be seen in the transition sfork. The next one, wjoin joins the waiting branches. The transitions $\mathsf{x_P}$ and $\mathsf{x_Q}$ allow the interruption of a branch when it has finished, equally $\mathsf{x_2}$ interrupts the computation at the end of the forward flow, while the last three handle the propagation of interrupts similar to policy #5.

For a saga $\{\![P]\!\}$ (Figure 42) with external places F_1, F_2 for the forward flow and E for an error we introduce fresh names for the encoding of P.

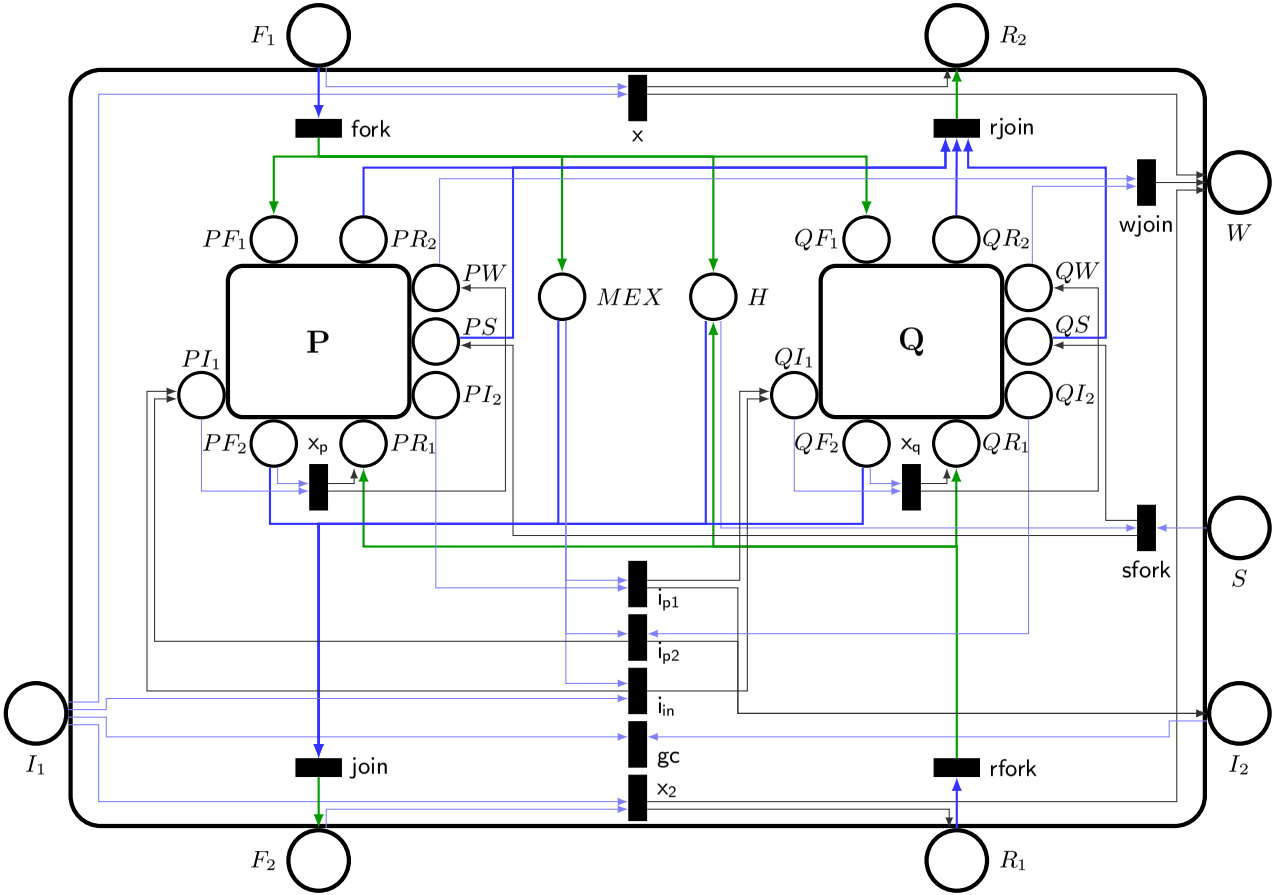


Figure 41: Petri net for a parallel composition in policy #3

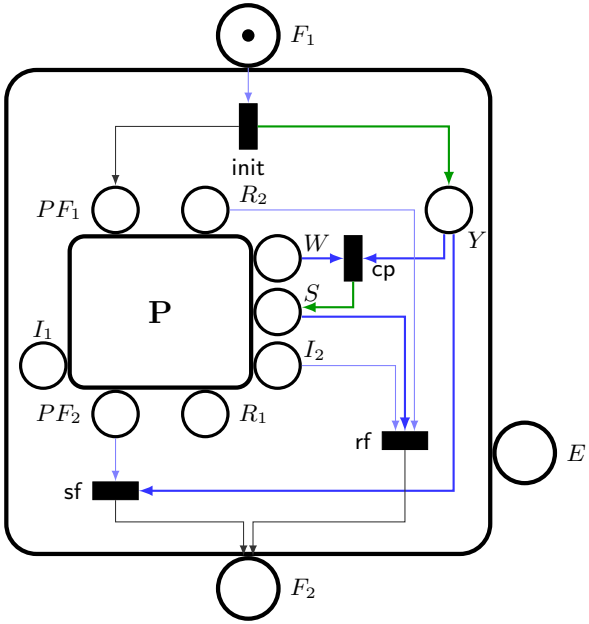


Figure 42: Petri net for a transaction according to policy #3

Then we add the following transitions:

$$\begin{array}{rlcl} \mathsf{init}: & F_1 & \to & PF_1{+}Y \\ \mathsf{cp}: & Y + W & \to & S \\ \mathsf{sf}: & PF_2{+}Y & \to & F_2 \\ \mathsf{rf}: & PR_2 + PI_2{+}S & \to & F_2 \end{array}$$

where the first one starts the transaction and puts a token in the new place Y which indicates that this is the outermost process of the transaction. The second one takes this token in Y and one in W indicating each concurrent computation is waiting and returns a token in S that will activate compensations. The last two transitions are as in policy #5 taking a successful or a successfully compensated computation back to the normal forward flow.

Compared to the denotational semantics for policy #3 (see Section 3.1) we state the following correspondence for compensable processes, where

we use $\in_3$ to indicate traces according to policy #3:

Theorem 9 (Correspondence policy #3)**.** *Let P be a compensable process and N_P the corresponding net according to policy #3 with external places F_1, F_2 for the forward flow, R_1, R_2 for the reverse flow, I_1, I_2 for interrupts, W for waiting and S for activating compensations. The correspondence of denotational and (maximal computations of the) operational semantics is given as follows:*

1. *$(p\langle\checkmark\rangle, q\langle\checkmark\rangle) \in_3 P$ iff there is a computation $f : F_1 \to F_2 \in \mathcal{T}(N_P)$ with $p \in label(f)$ and a computation $r : R_1 + S \to R_2 + S \in \mathcal{T}(N_P)$ with $q \in label(r)$.*

2. *$(p\langle!\rangle, q\langle\checkmark\rangle) \in_3 P$ iff there is a marking U and a computation $a : F_1 \to U + I_2 + W \in \mathcal{T}(N_P)$ with $p' \in label(a)$ for some p' such that $p = filter(p')$ and a computation $r : U + S \to R_2 + S \in \mathcal{T}(N_P)$ with $q \in label(r)$.*

3. *$(p\langle?\rangle, q\langle\checkmark\rangle) \in_3 P$ iff there is a marking U and a computation $i : F_1 + I_1 \to U + W \in \mathcal{T}(N_P)$ with $p \in label(i)$ and $r : U + S \to R_2 + S \in \mathcal{T}(N_P)$ such that $q \in label(r)$.*

Proof. The proof is by structural induction similar to the proof for Theorem 6. Note that we have a clear distinction between forward and backward flow in both the denotational semantics, as the two elements of the pair, and in the operational semantics due to the waiting mechanism with places W and S. □

The first policy, centralized compensation without interrupt, can be easily modelled changing the encoding for policy #3 as follows. First we remove the possibility for compensable processes to be interrupted in the beginning, *i.e.* the transition $\mathsf{x} : F + I_1 \to R$ that is included in the encoding of every process. The only possibility to interrupt a process is at the end of a branch in a parallel composition $P|Q$ where we leave the transitions $\mathsf{x_P}$ and $\mathsf{x_Q}$. That way if either P or Q aborts while the other branch is successful we do not get a deadlock but instead can start compensations. As in policy #6 for a sequential composition $P; Q$ we introduce the new place PI_1 while Q uses I_1.

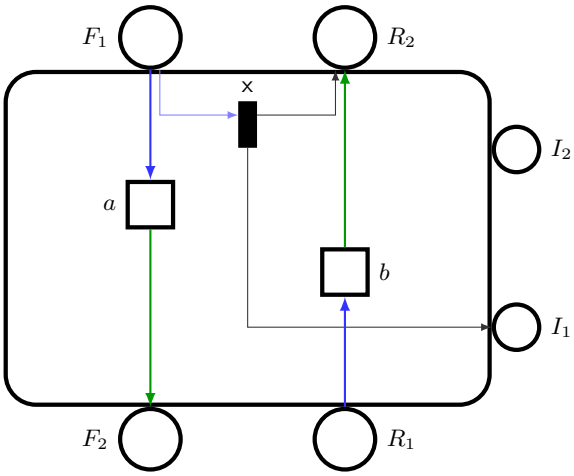


Figure 43: Petri net for a compensation pair according to policy #4

4.5.2 Distributed Compensation

In the distributed case (policy #4) concurrent processes can start their compensations on their own. This includes cases where a branch activates its compensation before an actual error occurred. Nevertheless the encoding has to distinguish those cases where a process aborted and where a process started its compensation assuming a fault.

The encoding of compensable processes uses six places. As for policy #5 there are two places for the forward flow F_1 and F_2, as well as two for the reverse flow R_1 and R_2. We use the place I_2 to keep the information that a process has aborted on its own while in contrast to policy #5 we use the place I_1 in order to know that a process without an explicit abort started executing its compensation.

Possible final configurations for an execution starting with a token in F_1 of compensable processes should be either a token in F_2 for a successful computation, a token in R_2 and I_2 for an aborted computation or a token in R_2 and I_1 for an interrupted computation. Note that inside a saga only the first two are correct computations.

For P a compensable process and N_P the corresponding net with ex-

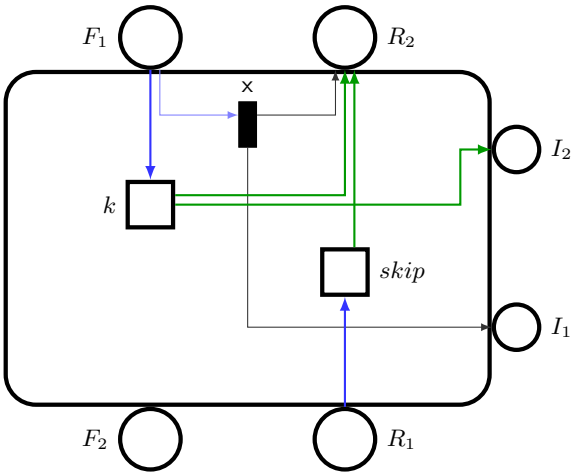


Figure 44: Petri net for *throww* according to policy #4

ternal places F_1, F_2 for the forward flow, R_1, R_2 for the reverse flow and I_1, I_2 for interrupts the net N_P always includes the following transition: $\mathsf{x} : F_1 \to R_2 + I_1$, *i.e.*, in every state of the computation the reverse flow can be started. The encoding for a compensation pair $A \div B$ (Figure 43) adds the respective transitions for A and B, while for *throww* (Figure 44) the transition k is added. The encoding of sequential composition remains as in policy #5 (Figure 32).

For a parallel composition $P|Q$ (Figure 45) both P and Q are encoded with new names. Additionally we need the following transitions:

$$
\begin{array}{rlcl}
\mathsf{fork}: & F_1 & \to & PF_1 + QF_1 \\
\mathsf{join}: & PF_2 + QF_2 & \to & F_2 \\
\mathsf{rfork}: & R_1 & \to & PR_1 + QR_1 \\
\mathsf{rjoin}: & PR_2 + QR_2 & \to & R_2 \\
\mathsf{x_P}: & PF_2 & \to & PR_1 + PI_1 \\
\mathsf{x_Q}: & QF_2 & \to & QR_1 + QI_1 \\
\mathsf{x_2}: & F_2 & \to & R_1 + I_1 \\
\mathsf{ijoin_1}: & PI_1 + QI_1 & \to & I_1 \\
\mathsf{ijoin_2}: & PI_2 + QI_1 & \to & I_2 \\
\mathsf{ijoin_3}: & PI_1 + QI_2 & \to & I_2 \\
\mathsf{ijoin_4}: & PI_2 + QI_2 & \to & I_2
\end{array}
$$

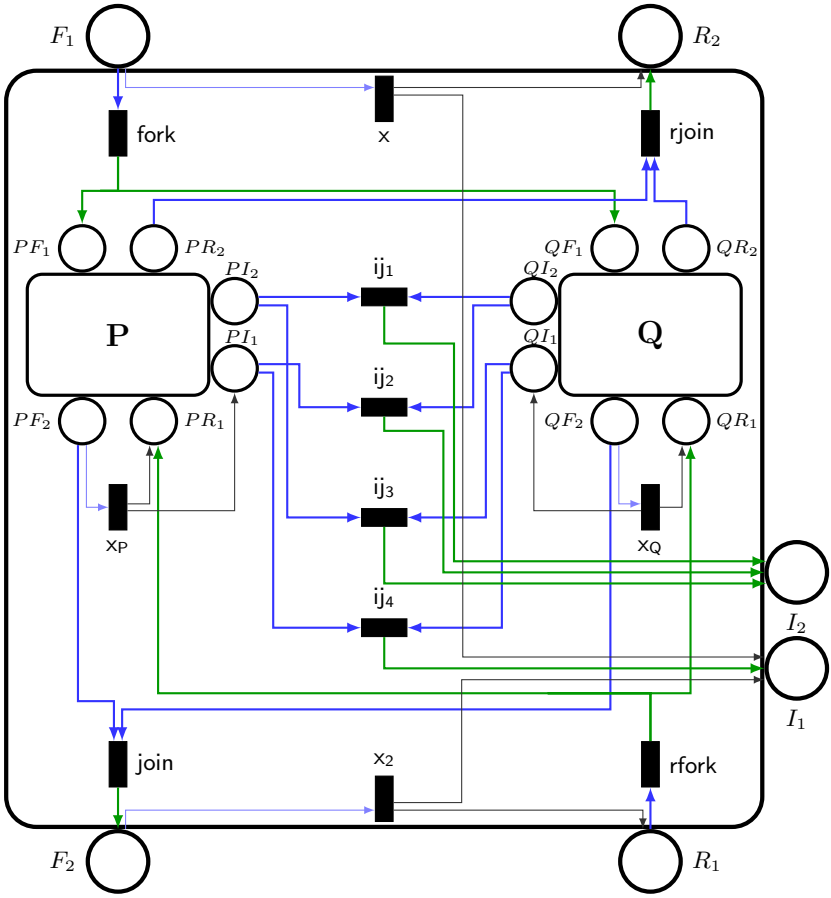


Figure 45: Petri net for a parallel composition $P|Q$ in policy #4

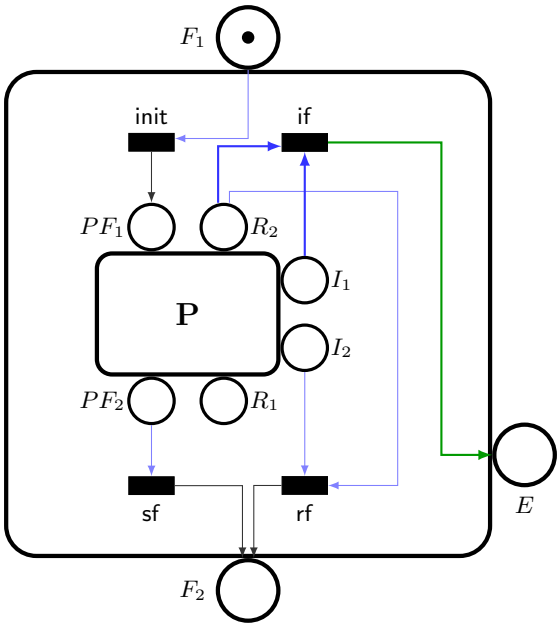


Figure 46: Petri net for a transaction according to policy #4

The first transitions handle the fork and join for the forward and reverse flow. The next ones start the reverse flow independently, while the last four rules are different variants for joining the interrupt depending on where an error occurred.

The encoding of a transaction (Figure 46) is the same as in policy #5 with an additional transition if. It takes a token in R_2 and I_1 each as the final configuration and will lead to an error state.

Compared to the denotational semantics for policy #4 (see Section 3.1) we state the following correspondence for compensable processes:

Theorem 10 (Correspondence policy #4)**.** *Let P be a compensable process and N_P the corresponding net according to policy #4 with external places F_1, F_2 for the forward flow, R_1, R_2 for the reverse flow and I_1, I_2 for interrupts. The correspondence of denotational and (maximal computations of the) operational semantics is given as follows:*

1. $(p\langle\checkmark\rangle, q\langle\checkmark\rangle) \in_4 P$ *iff there is a computation* $f : F_1 \to F_2 \in \mathcal{T}(N_P)$

with $p \in label(f)$ *and a computation* $r : R_1 \to R_2 \in \mathcal{T}(N_P)$ *with* $q \in label(r)$.

2. $(p\langle! \rangle, q\langle \checkmark \rangle) \in_4 P$ *iff there is a computation* $a : F_1 \to I_2 + R_2 \in \mathcal{T}(N_P)$ *with label* $p'q \in label(a)$ *for some* p' *such that* $p = filter(p')$.

3. $(p\langle ? \rangle, q\langle \checkmark \rangle) \in_4 P$ *iff there is a computation* $i : F_1 \to I_1 + R_2 \in \mathcal{T}(N_P)$ *such that* $pq \in label(i)$.

Proof. The proof is by structural induction similar to the proof of Theorem 6. □

To encode policy #2 we have to change the encoding of policy #4 as follows: First we remove the transition $\mathsf{x} : F \to R + I_1$ inherent in every compensable process. We keep the remaining transitions including $\mathsf{x_P}, \mathsf{x_Q}$ and x_2 for parallel composition. This guarantees that a branch in a parallel composition cannot be interrupted during computation. However it may start its compensation independently after finishing successfully. As for policies #1 and #6 we do not share the place I_1 in a sequential composition $P; Q$, but instead introduce place PI_1 for the encoding of P while Q uses I_1.

4.6 Tool Support

In this section we present a tool modelling the encoding of compensable processes and sagas into Petri nets and the execution inside a Petri net. We implemented policy #5 as well as the policies presented in the previous section. The tool is written in Maude (see Section 2.6 for details) and can be found at [Sou13].

The implementation is structured in several modules. The first one called `PETRI` implements Petri nets. We define a sort for places such that `op F : Nat -> Place` and other constructors using instead of `F` the operators `R`, `I1` or `I2`. A `Place` is a subsort of `Marking` that can be combined using the operator `+`. The sort `Transition` is defined by the operator

```
op _ - _ > _ : Marking Names Marking -> Transition .
```

```
rl [fire]  : (M  , ((M - A > M2), S) ,  N)
          => (M2 , ((M - A > M2), S) , (N A)) .
crl [fire] : ((M + M1)  , ((M - A > M2), S) ,  N)
          => ((M2 + M1) , ((M - A > M2), S) , (N A))
          if M1 =/= empty .
```

Figure 47: Rules for firing a transition in a Petri net where `M, M1, M2` are markings, `N` and `A` names and `S` a set of transitions

where a `Marking` (the preset of the transition) is mapped to another `Marking` (the postset of the transition) observing the label `Names`. We then use a configuration consisting of the current marking a set of transitions and the yet observed labels for the rewrite rule to fire transitions (Figure 47).

The module `SAGA` defines the syntax for compensable processes (sort `Cprocess`) and standard processes (sort `Sagaprocess`) as given in Figure 28. For simplicity we use natural numbers to represent actions.

The encoding is defined using the following two operators:

```
op enc : Sagaprocess -> TransitionSet .
op enc : Cprocess -> TransitionSet .
```

These are abbreviations used to initialize the actual encoding operator `encc` that takes as arguments apart from the process also the external places and a counter. The counter is needed to generate new places (as in the encoding of parallel composition) while through the external places the corresponding net can be addressed. The module `SAGA` defines this encoding for standard processes while the encoding of compensable processes differs in each policy and thus is defined in different modules, one for each policy. Exemplary we present here the encoding of policy #5 in module `SAGA5`.

The extended encoding function for compensable processes is defined such that

```
op encc : Cprocess Place Place Place
      Place Place Place Nat -> Encoding .
```

where `Encoding` stands for a pair consisting of a natural number for the counter and the resulting transition set. It is initialized by the equation

```
ceq enc(PP) = T
  if
  F1 := F(0) /\ F2 := F(1) /\
  R1 := R(0) /\ R2 := R(1) /\
  I1 := I1(0) /\ I2 := I2(0) /\
  Ctr := 2 /\
  Ctr2 . T := encc(PP, F1, F2, R1, R2, I1, I2, Ctr) .
```

As can be seen the six places for `encc` are F_1, F_2 for the forward flow, R_1, R_2 for the backward flow and I_1, I_2 for interrupts.

The operator `encc` encodes processes according to the presented Petri nets in Figures 30 to 35. For example, the encoding of a compensation pair is defined such that

```
eq encc(A / B, F1, F2, R1, R2, I1, I2,Ctr) =
      (Ctr . ((F1 - A > F2), (R1 - B > R2),
      ((F1 + I1) - nil > R2), ((F2 + I1) - nil > R1),
      ((I1 + I2) - nil > empty))) .
```

where the first two transitions represent the forward and backward flow observing A or B respectively, and the other transitions are auxiliary transitions x_1 and x_2 for interrupting the forward flow and gc for garbage collection. Note that for auxiliary transitions the label is `nil`, thus the observed flow of an encoded Petri net only contains basic activities.

Figure 48 shows an example for encoding a transaction consisting of a parallel composition of a compensation pair and a *throww*. Note that the operator `enc` always puts the place `F(0)` as the starting place for the forward flow F_1 and `F(1)` as the final place F_2. In Figure 49 we use this fact to compute the possible final configurations of the corresponding Petri net starting with a token in F_1 (abbreviating `enc([1 / 2 || throww])` as constant `T`). We can see there are two possible results depending on the observed flow. In the first solution the observed flow is empty corresponding to the case where the left branch is interrupted before any execution. The other solution observes `1 2` where the left branch is in-

```
Maude> reduce in SAGA5 : enc([1 / 2 || throww]) .
rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
result TransitionSet:
     (F(0) - nil > F(5) + F(7) + MEX(5)),
     (F(3) - nil > F(1)),
     (F(5) - 1 > F(6)),
     (F(7) - nil > R(8) + I2(7)),
     (R(3) - nil > R(5) + R(7)),
     (R(5) - 2 > R(6)),
     ((F(0) + I1(3)) - nil > R(4)),
     ((F(3) + I1(3)) - nil > R(3)),
     ((F(5) + I1(5)) - nil > R(6)),
     ((F(6) + I1(5)) - nil > R(5)),
     ((F(7) + I1(7)) - nil > R(8)),
     ((F(8) + I1(7)) - nil > R(7)),
     ((R(4) + I2(3)) - nil > F(1)),
     ((R(6) + R(8)) - nil > R(4)),
     ((I1(3) + I2(3)) - nil > empty),
     ((I1(3) + MEX(5)) - nil > I1(5) + I1(7)),
     ((I1(5) + I2(5)) - nil > empty),
     ((I1(7) + I2(7)) - nil > empty),
     ((I2(5) + MEX(5)) - nil > I1(7) + I2(3)),
     ((I2(7) + MEX(5)) - nil > I1(5) + I2(3)),
     (F(6) + F(8) + MEX(5)) - nil > F(3)
```

Figure 48: Example encoding in module `SAGA5`

terrupted after execution. In both cases the final marking corresponds to F_2.

Figure 50 shows the corresponding search graph for the example computation. Going through the graph we can see the following transitions:

- From `state 0` to `state 1` transition fork is executed.
- From `state 1` to `state 2` transition 1 is executed.
- From `state 1` to `state 3` transition k is executed.
- From `state 2` to `state 4` transition k is executed.
- From `state 3` to `state 4` transition 1 is executed.

```
Maude> search in SAGA5 :
   F(0),enc([1 / 2 || throww]),nil =>! C:Configuration .

Solution 1 (state 11)
states: 13  rewrites: 39 in 0ms cpu (1ms real)
            (~ rewrites/second)
C:Configuration --> F(1),T,nil

Solution 2 (state 13)
states: 14  rewrites: 41 in 4ms cpu (1ms real)
            (10250 rewrites/second)
C:Configuration --> F(1),T,1 2

No more solutions.
states: 14  rewrites: 41 in 4ms cpu (2ms real)
            (10250 rewrites/second)
```

Figure 49: Computation of a transaction in the tool

- From `state 3` to `state 5` transition ip_2 is executed.
- From `state 4` to `state 6` transition ip_2 is executed.
- From `state 5` to `state 6` transition 1 is executed.
- From `state 5` to `state 7` transition x_{P1} is executed.
- From `state 6` to `state 8` transition x_{P2} is executed.
- From `state 7` to `state 9` transition rjoin is executed.
- From `state 8` to `state 10` transition 2 is executed.
- From `state 9` to `state 11` transition rf is executed.
- From `state 10` to `state 12` transition rjoin is executed.
- From `state 12` to `state 13` transition rf is executed.

As we explained in this example we can search for any maximal computation of an encoded process. Moreover exploiting the search graph we can show any reachable state from a given marking. We can also

```
state 0, Configuration: F(0),T,nil
arc 0 ===> state 1 (rl M,S,... [label fire] .)

state 1, Configuration: (F(5) + F(7) + MEX(5)),T,nil
arc 0 ===> state 2 (crl (M + M1),S,... [label fire] .)
arc 1 ===> state 3 (crl (M + M1),S,... [label fire] .)

state 2, Configuration: (F(6) + F(7) + MEX(5)),T,1
arc 0 ===> state 4 (crl (M + M1),S,... [label fire] .)

state 3, Configuration: (F(5) + R(8) + I2(7) + MEX(5)),T,nil
arc 0 ===> state 4 (crl (M + M1),S,... [label fire] .)
arc 1 ===> state 5 (crl (M + M1),S,... [label fire] .)

state 4, Configuration: (F(6) + R(8) + I2(7) + MEX(5)),T,1
arc 0 ===> state 6 (crl (M + M1),S,... [label fire] .)

state 5, Configuration: (F(5) + R(8) + I1(5) + I2(3)),T,nil
arc 0 ===> state 6 (crl (M + M1),S,... [label fire] .)
arc 1 ===> state 7 (crl (M + M1),S,... [label fire] .)

state 6, Configuration: (F(6) + R(8) + I1(5) + I2(3)),T,1
arc 0 ===> state 8 (crl (M + M1),S,... [label fire] .)

state 7, Configuration: (R(6) + R(8) + I2(3)),T,nil
arc 0 ===> state 9 (crl (M + M1),S,... [label fire] .)

state 8, Configuration: (R(5) + R(8) + I2(3)),T,1
arc 0 ===> state 10 (crl (M + M1),S,... [label fire] .)

state 9, Configuration: (R(4) + I2(3)),T,nil
arc 0 ===> state 11 (rl M,S,... [label fire] .)

state 10, Configuration: (R(6) + R(8) + I2(3)),T,1 2
arc 0 ===> state 12 (crl (M + M1),S,... [label fire] .)

state 11, Configuration: F(1),T,nil

state 12, Configuration: (R(4) + I2(3)),T,1 2
arc 0 ===> state 13 (rl M,S,... [label fire] .)

state 13, Configuration: F(1),T,1 2
```

Figure 50: Search graph for the computation of Figure 49

use the search command to match the result state to a given pattern, *e.g.*, looking for reachable states that observed a particular action.

Using this feature of Maude we used the tool to improve our encoding. For example we were able to search for final states that did not match the behaviour formalized in Theorem 5.

Extending the tool we could use the LTL model checker of Maude to prove properties for a given process. As an example we could prove that a net for a particular encoded process is 1-safe as shown in Proposition 2. Another extension could show the correspondence of the operational and denotational semantics given a particular process.

4.7 Conclusion

In this chapter we presented an operational semantics for Sagas based on an encoding into Petri nets and showed that computations are equivalent to labels in the denotational semantics. Moreover we presented behavioural and logical properties that hold for every encoded process. We showed as well encodings for other compensation policies including a correspondence with the respective denotational semantics and presented a tool implementing the encoding for each presented policy.

Chapter 5

Small-step SOS semantics for Sagas

In this chapter we introduce a small-step semantics for modelling long-running transactions. In the previous chapters starting from Sagas and cCSP we introduced a new policy for handling compensations in concurrent processes and presented first a denotational semantics and then an operational one based on an encoding into Petri nets. The Petri net model is more informative than the trace semantics, because it accounts for the branching of processes arising from the propagation of interrupts. Unfortunately, the sophisticated mechanism needed for handling interrupts introduces many auxiliary places and transitions that make the Petri net model quite intricate to parse and difficult to extend.

Our aim is to provide an operational semantics for the new policy whose main requirements are: i) it must follow the small-step style of operational semantics, so to account for the branching caused by the propagation of interrupts; ii) it must adhere to the Petri net semantics; iii) other policies can be implemented without radical redesign; iv) it must be easy to introduce other features, like choice, iteration, and faulty compensations (crashes).

In this chapter we propose a labelled transition system (LTS) semantics that meets all the above requirements. The main result consists of

the correspondence theorems with the existing semantics. We started by considering the optimal policy #5 and were guided by the correspondence with the Petri net semantics to correct many wrong design choices in our first attempts. The main result is the proof that our LTS semantics matches the Petri nets semantics in Chapter 4 up to weak bisimilarity. This gives a way to read markings as (weak bisimilar) terms of a process algebra that describes the run-time status of the process.

The contents of this chapter was first published in [BK12] and shows here extended proofs as well as a presentation of tool support.

5.1 Labelled transition system for sequential Sagas

In this section we define a small-step LTS semantics for the sequential part of the Sagas calculus. Note that we focus on policy #5 in the next two sections, however for sequential processes the semantics is the same in each policy. We use the syntax defined in Section 2.2, though ignoring parallel composition for now. Later we extend our definitions to concurrency.

To be able to reason on intermediate states in the execution of a process we introduce a runtime syntax. Initially a process is always constructed using the basic syntax, the runtime syntax is only present during execution.

$$
\begin{array}{lllll}
\text{(COMP)} & C & ::= & A \mid C;C \mid \mathbf{nil} & \\
\text{(PROCESS)} & P & ::= & A \div B \mid P;Q \mid P\$C \mid [C] & \\
\text{(SAGA)} & S & ::= & A \mid S;T \mid \mathbf{nil} &
\end{array}
$$

First we add a distinct type for compensations. They can either be basic activities A, the sequential composition of compensations $C;C$ or **nil**. With **nil** we denote completion of a compensation, in the sense that, e.g., the compensation $\mathbf{nil};C$ can never execute activities in C. For compensable processes, $P\$C$ denotes a process P running with the already installed compensation C. Compensations of P will be installed on top of C once P is finished. Notably, each process can be assigned a compensation, i.e., the operational semantics does not rely on a global stack of com-

pensation. When parallel composition will be considered in Section 5.2, this will make the order in which concurrent actions are interleaved be inessential for the order of execution of their compensations, as in the Petri net semantics (Chapter 4). The completion of forward activities is denoted by $[C]$ instead of **nil**, because we need to consider the installed compensation C (informally, $[C]$ can be read as $\mathbf{nil}\$C$). We also add **nil** for marking the completion of a saga.

The small-step semantics is defined by three LTSs, one for each syntax category. Let $\Omega = \{\boxdot, \boxtimes\}$. A *context* Γ is a function $\Gamma : \mathcal{A} \to \Omega$ that maps a basic activity to $\boxdot$ or $\boxtimes$ depending on whether it commits or aborts, with $\Gamma(\mathit{skip}) = \boxdot$ and $\Gamma(\mathit{throw}) = \boxtimes$. We remind that it is tacitly assumed that compensation activities cannot fail. Given the set of compensations $\mathcal{C}$, the set of compensable processes $\mathcal{P}$ and the set of sagas $\mathcal{S}$, we let $\mathcal{S}_C = \mathcal{C}$, $\mathcal{S}_P = \Omega \times \mathcal{P}$, $\mathcal{S}_S = \Omega \times \mathcal{S}$ denote sets of States. The component Ω in a state describes whether the process can still commit (it can still move forward) or must abort (a fault was issued that needs to be compensated). Note that states of the LTS for compensations have clearly no Ω component. Sagas initially start executing in a commit state.

The next definition formally introduces the labelled transition system for processes in the small-step semantics:

Definition 7. *The LTS semantics of (sequential) sagas is the least LTS $(\mathcal{S}, L, \mathcal{T})$ generated by the rules in Figure 51–53, whose set of states is $\mathcal{S} = \mathcal{S}_C \cup \mathcal{S}_P \cup \mathcal{S}_S$ and whose set of labels is $L = \mathcal{A} \cup \{\tau\}$.*

We will write transitions $t \in \mathcal{T}$ as $t : \Gamma \vdash s \xrightarrow{\lambda} s'$ for states s, s', a label $\lambda \in L$ and a context Γ.

The semantics exploits some auxiliary notation. The predicate dn_σ checks the completion of (the forward execution of) a compensable process. The subscript σ stands for $\boxdot$ or $\boxtimes$ and means that the process is either evaluated in a commit or an abort context. The predicate dn_σ is inductively defined as:

$$\begin{array}{llll} dn_\sigma([C]) & \triangleq \mathtt{tt} & dn_\sigma(A \div B) & \triangleq \mathtt{ff} \\ dn_\sigma(P\$C) & \triangleq dn_\sigma(P) & dn_\sigma(P;Q) & \triangleq dn_\sigma(P) \end{array}$$

Note that for sequential processes dn is independent of the subscript,

(C-ACT)

$$\frac{}{\Gamma \vdash A \xrightarrow{A} \mathbf{nil}}$$

(C-SEQ1)

$$\frac{\Gamma \vdash C \xrightarrow{\lambda} C' \wedge \neg dn(C')}{\Gamma \vdash C; D \xrightarrow{\lambda} C'; D}$$

(C-SEQ2)

$$\frac{\Gamma \vdash C \xrightarrow{\lambda} C' \wedge dn(C')}{\Gamma \vdash C; D \xrightarrow{\lambda} D}$$

Figure 51: LTS for sequential compensations

this will change when introducing parallel composition. Analogously, we define a predicate dn on compensations,

$$dn(\mathbf{nil}) \triangleq \mathtt{tt} \qquad dn(A) \triangleq \mathtt{ff} \qquad dn(C; C') \triangleq dn(C)$$

together with a function $cmp(P)$ that extracts the installed compensation from a process P that is "done":

$$\begin{array}{rcl} cmp([C]) & \triangleq & C \\ cmp(P; Q) & \triangleq & cmp(P) \\ cmp(P\$C) & \triangleq & \begin{cases} C & \text{if } dn(cmp(P)) \\ cmp(P); C & \text{if } \neg dn(cmp(P)) \end{cases} \end{array}$$

When a compensable process P is done, we use the shorthand $cm?(P) \triangleq \neg dn(cmp(P))$ (*i.e.*, $cm?(P)$ holds when there is some compensation to run).

For sagas we define the predicate dn similar to compensations:

$$dn(\mathbf{nil}) \triangleq \mathtt{tt} \qquad dn(A) \triangleq \mathtt{ff} \qquad dn(\{\!|P|\!\}) \triangleq \mathtt{ff} \qquad dn(S; T) \triangleq dn(S)$$

The rules in Figure 51 handle compensations. As we assume a compensation is always successful, only rule C-ACT is needed for basic activities. Rules C-SEQ1 and C-SEQ2 exploit the "done" predicate to avoid reaching states such as $\mathbf{nil}; C$.

(S-ACT)

$$\frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \boxdot, A \div B \xrightarrow{A} \boxdot, [B]}$$

(F-ACT)

$$\frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \boxdot, A \div B \xrightarrow{\tau} \boxtimes, [\mathbf{nil}]}$$

(SEQ)

$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge \neg dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxdot, P'; Q}$$

(S-SEQ)

$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxdot, Q\$cmp(P')}$$

(A-SEQ)

$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxtimes, P'}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxtimes, P'}$$

(STEP)

$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge \neg dn_{\sigma'}(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', P'\$C}$$

(AS-STEP1)

$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge dn_{\sigma'}(P') \wedge cm?(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', [cmp(P'); C]}$$

(AS-STEP2)

$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge dn_{\sigma'}(P') \wedge \neg cm?(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', [C]}$$

(COMP)

$$\frac{\Gamma \vdash C \xrightarrow{\lambda} C'}{\Gamma \vdash \boxtimes, [C] \xrightarrow{\lambda} \boxtimes, [C']}$$

Figure 52: LTS for sequential compensable processes

For compensable processes a basic activity A of a compensation pair $A \div B$ can either commit or abort, depending on the context: if A commits then B is installed (S-ACT); if A fails, then there is nothing to be compensated (F-ACT). A sequential composition $P; Q$ acts according to how P acts (SEQ). If P finishes successfully (S-SEQ), then Q will run under the installed compensation $cmp(P')$. If P aborts (A-SEQ) the continuation Q is discarded. The process $P\$C$ acts according to P (STEP). When P finishes, its compensation is installed on top of C (AS-STEP1). The rule AS-STEP2 ensures that a **nil** is not installed on top of a compensation. Compensations are executed via rule COMP.

The rules A-SACT and F-SACT for basic activities A are as expected, as well as SSEQ, S-SSEQ and A-SSEQ for sequential composition of sagas. A transaction $\{\!|P|\!\}$ can be executed as long as either it is still running forward (SAGA) or it has already aborted and compensates (SAGA and A-SAGA1). If the transaction did not abort, *i.e.*, it is in a commit state, and it finishes, the compensation is discarded and the next state is $\boxdot, \mathbf{nil}$ (S-SAGA). If the saga aborts but is able to compensate, then it reaches a good state (A-SAGA2).

Note that the rules of our LTS exploit lookahead in the following sense: In several rules (*e.g.* C-SEQ1, SEQ, STEP) we use the previously defined predicates on the right-hand sides of the premises. We do however not consider this a drawback as respective rules complement each other. For example rules SEQ and S-SEQ differ in the premises only in whether the predicate $dn_{\boxdot}(P')$ is false or true.

The formal correspondence between the LTS semantics and the denotational semantics of policies #1–5 is an immediate consequence of our main result and will be deferred to Section 5.2 (see Corollary 6).

Example 8. *Consider the booking of a trip from Example 1 ignoring the extra step for purchasing a ticket and with serialized activities. Let*

$$bT \triangleq \mathsf{rT} \div \mathsf{cR} \;;\; \mathsf{bF} \div \mathsf{cF} \;;\; \mathsf{bH} \div \mathsf{cH} \;;\; \mathsf{cC} \div \mathit{skip}$$

Assume that the booking of the hotel fails, e.g., there are no rooms available, and let Γ *map* bH *to* $\boxtimes$ *and the other actions to* $\boxdot$*. A possible derivation is*

(S-SACT)
$$\frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \boxdot, A \xrightarrow{A} \boxdot, \mathbf{nil}}$$
(F-SACT)
$$\frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \boxdot, A \xrightarrow{\tau} \boxtimes, \mathbf{nil}}$$
(SSEQ)
$$\frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \sigma', S' \wedge \neg dn(S')}{\Gamma \vdash \sigma, S; T \xrightarrow{\lambda} \sigma', S'; T}$$
(S-SSEQ)
$$\frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \boxdot, S' \wedge dn(S')}{\Gamma \vdash \sigma, S; T \xrightarrow{\lambda} \boxdot, T}$$
(A-SSEQ)
$$\frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \boxtimes, S' \wedge dn(S')}{\Gamma \vdash \sigma, S; T \xrightarrow{\lambda} \boxtimes, S'}$$
(SAGA)
$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge \neg dn_{\sigma'}(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \sigma', \{\!|P'|\!\}}$$
(S-SAGA)
$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, \{\!|P|\!\} \xrightarrow{\lambda} \boxdot, \mathbf{nil}}$$
(A-SAGA1)
$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \boxtimes, P' \wedge dn_{\boxtimes}(P') \wedge cm?(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \boxtimes, \{\!|P'|\!\}}$$
(A-SAGA2)
$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \boxtimes, P' \wedge dn_{\boxtimes}(P') \wedge \neg cm?(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \boxdot, \mathbf{nil}}$$

Figure 53: LTS for sequential sagas

$\boxdot, \{\!|bT|\!\} \xrightarrow{\mathsf{rT}} \xrightarrow{\mathsf{bF}} \xrightarrow{\tau} \xrightarrow{\mathsf{cF}} \xrightarrow{\mathsf{cR}} \boxdot, \mathbf{nil}$, *because*

$$
\begin{array}{lll}
\boxdot, bT & \xrightarrow{\mathsf{rT}} & \boxdot, (\mathsf{bF} \div \mathsf{cF}\ ;\ \mathsf{bH} \div \mathsf{cH}\ ;\ \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} \qquad \text{S-SEQ,S-ACT} \\
 & \xrightarrow{\mathsf{bF}} & \boxdot, ((\mathsf{bH} \div \mathsf{cH}\ ;\ \mathsf{cC} \div \mathit{skip})\$\mathsf{cF})\$\mathsf{cR} \qquad \text{STEP,S-SEQ,S-ACT} \\
 & \xrightarrow{\tau} & \boxtimes, [\mathsf{cF}; \mathsf{cR}] \qquad \text{AS-STEP1, AS-STEP2, A-SEQ, F-ACT} \\
 & \xrightarrow{\mathsf{cF}} & \boxtimes, [\mathsf{cR}] \qquad \text{COMP,C-SEQ2,C-ACT} \\
 & \xrightarrow{\mathsf{cR}} & \boxtimes, [\mathbf{nil}] \qquad \text{COMP,C-ACT}
\end{array}
$$

Rule A-SAGA2 *then returns the system to a consistent state again.*

5.2 Extension to Concurrency

In this section we extend the LTS semantics to handle parallel Sagas under policy #5. First, we extend the runtime syntax as follows:

$$
\begin{array}{lrcl}
(\text{COMP}) & C & ::= & A \mid C;C \mid \mathbf{nil} \mid C|C \\
(\text{PROCESS}) & P & ::= & X \mid P;P \mid P\$C \mid [C] \mid P_\sigma|_{\sigma'}P \\
(\text{SAGA}) & S & ::= & A \mid S;S \mid \{\!|P|\!\} \mid \mathbf{nil} \mid S_\sigma|_{\sigma'}S
\end{array}
$$

We add parallel composition to compensations. Moreover we use subscripts for the parallel composition of processes or sagas ${}_\sigma|_{\sigma'}$ such that $\sigma, \sigma' \in \Omega$. If a thread is denoted with $\boxdot$, it can still move forward and commit. A thread denoted with $\boxtimes$ either aborted or was interrupted, so it can compensate. If a thread is denoted with a $\boxdot$ then also every parallel composition contained as a subprocess in this thread must have a $\boxdot$. Similarly if the global state is a $\boxdot$, any parallel composition in this state has subscripts $\boxdot$. We consider $P_\boxdot|_\boxdot Q$ part of the normal syntax, not just of the runtime syntax, and usually write just $P|Q$. We sometimes use $\|$ instead of ${}_\sigma|_{\sigma'}$ if the values of σ, σ' are irrelevant.

Definition 8. *The LTS semantics of parallel sagas is the least LTS $(\mathcal{S}, L, \mathcal{T})$ generated by the rules in Figure 51–53 together with the rules in Figure 54 (symmetric rules* C-PAR-R, PAR-R, INT-L *and* SPAR-R *are omitted).*

The semantics exploits some auxiliary notation. First, the binary function $\sqcap : \Omega \times \Omega \to \Omega$ is defined such that $\sigma \sqcap \sigma' = \boxdot$ iff $\sigma = \sigma' = \boxdot$. It is

(C-PAR-L)
$$\frac{\Gamma \vdash C \xrightarrow{\lambda} C'}{\Gamma \vdash C|D \xrightarrow{\lambda} C'|D}$$
(PAR-L)
$$\frac{\Gamma \vdash \sigma_1, P \xrightarrow{\lambda} \sigma_1', P'}{\Gamma \vdash \sigma, P_{\sigma_1}|_{\sigma_2} Q \xrightarrow{\lambda} \sigma \sqcap \sigma_1', {P'}_{\sigma_1'}|_{\sigma_2} Q}$$
(INT-R)
$$\frac{Q \rightsquigarrow Q'}{\Gamma \vdash \boxtimes, P_\sigma|_{\boxdot} Q \xrightarrow{\tau} \boxtimes, P_\sigma|_{\boxtimes} Q'}$$
(SPAR-L)
$$\frac{\Gamma \vdash \sigma_1, S \xrightarrow{\lambda} \sigma_1', S'}{\Gamma \vdash \sigma, S_{\sigma_1}|_{\sigma_2} T \xrightarrow{\lambda} \sigma_1' \sqcap \sigma_2, {S'}_{\sigma_1'}|_{\sigma_2} T}$$

Figure 54: LTS rules for parallel Sagas (symmetric rules omitted for brevity)

easy to check that $\sqcap$ is associative and commutative. Then, the predicates dn_σ, dn and the function cmp are extended to parallel composition:

$$\begin{array}{rcl} dn(C|C') & \triangleq & dn(C) \wedge dn(C') \\ dn_\sigma(P_{\sigma_1}|_{\sigma_2} Q) & \triangleq & dn_\sigma(P) \wedge dn_\sigma(Q) \wedge (\sigma = \sigma_1 = \sigma_2) \\ dn(S_{\sigma_1}|_{\sigma_2} T) & \triangleq & dn(S) \wedge dn(T) \\ cmp(P \parallel Q) & \triangleq & cmp(P)|cmp(Q) \end{array}$$

Note that for concurrency we have to consider also the global state for processes when defining the dn predicate. The compensable process $P_{\sigma_1}|_{\sigma_2} Q$ is done when both P and Q are done and both subscripts are the same and coincide with the global state σ. If they are different, then one of them has still to be interrupted such that it can start executing its compensation.

In Figure 54 the additional rules needed for parallel Sagas are displayed. The rules C-PAR-L/C-PAR-R define just the ordinary interleaving of compensations. The rules PAR-L/PAR-R are analogous, but the subscript determines the modality of execution. A thread can move forward when it is in a commit state. If a thread aborts, the failure is annotated

also in the global state by taking $\sigma \sqcap \sigma'_1$. Note that a commit thread can still move forward even if the global mode is abort.

The rules INT-L/INT-R use an "extract" predicate $P \rightsquigarrow P'$ to interrupt a commit thread if the global process is in abort mode. In $P \rightsquigarrow P'$, the process P' is a possible result of interrupting P (see Figure 55). Interruption is not necessarily the same as extracting the compensation. For example interrupting a parallel composition means interrupting one branch. As a special case, note the interrupt of a sequential composition: we distinguish whether P is a parallel composition (predicate $par(P)$ is true) or not. This is motivated by the intention to adhere to the Petri net semantics, where $(P|Q); R$ can be interrupted discarding R but without necessarily interrupting P or Q. The predicate $\rightsquigarrow$ is deterministic except for the last two cases: in $P|Q$ we can decide which thread we want to interrupt first.

For the parallel composition of sagas (SPAR-L/SPAR-R) we just remark that in the case of fault of one thread we let the other threads execute as much as possible and just record the global effect in the σ component of the state.

Example 9. *Consider the running example of a trip booking. Let* $bT' \triangleq \mathsf{rT} \div \mathsf{cR}\ ;\ ((\mathsf{bF} \div \mathsf{cF}\ ;\ \mathsf{bH} \div \mathsf{cH}) \mid \mathsf{cC} \div \mathit{skip})$*, and assume that the credit card check fails while the other actions are successful. We have, e.g.*

$$
\begin{array}{lll}
 & \boxdot, bT' & \\
\xrightarrow{\mathsf{rT}} & \boxdot, ((\mathsf{bF} \div \mathsf{cF}\ ;\ \mathsf{bH} \div \mathsf{cH}) \mid \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} & \text{S-SEQ,S-ACT} \\
\xrightarrow{\mathsf{bF}} & \boxdot, ((\mathsf{bH} \div \mathsf{cH})\$\mathsf{cF} \mid \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} & \text{STEP,PAR-L, S-SEQ,S-ACT} \\
\xrightarrow{\tau} & \boxtimes, ((\mathsf{bH} \div \mathsf{cH})\$\mathsf{cF}\ {}_{\boxdot}|_{\boxtimes}\ [\mathbf{nil}])\$\mathsf{cR} & \text{STEP, PAR-R, F-ACT} \\
\xrightarrow{\tau} & \boxtimes, [(\mathsf{cF} \mid \mathbf{nil}); \mathsf{cR}] & \text{AS-STEP1,INT-L,E-STEP3} \\
\xrightarrow{\mathsf{cF}} & \boxtimes, [\mathsf{cR}] & \text{COMP,C-SEQ2,C-PAR-L,C-ACT} \\
\xrightarrow{\mathsf{cR}} & \boxtimes, [\mathbf{nil}] & \text{COMP,C-ACT}
\end{array}
$$

5.3 Operational Correspondence

In this section we will show a weak bisimilarity between our novel LTS semantics and the Petri net semantics for policy #5 of Chapter 4.

(E-COMP)

$$\frac{}{[C] \rightsquigarrow [C]}$$

(E-PAIR)

$$\frac{}{A \div B \rightsquigarrow [\mathbf{nil}]}$$

(E-SEQ1)

$$\frac{P \rightsquigarrow P' \wedge \neg par(P)}{P;Q \rightsquigarrow P'}$$

(E-SEQ2)

$$\frac{par(P)}{P;Q \rightsquigarrow P}$$

(E-STEP1)

$$\frac{P \rightsquigarrow P' \wedge \neg dn_{\boxtimes}(P')}{P\$C \rightsquigarrow P'\$C}$$

(E-STEP2)

$$\frac{P \rightsquigarrow P' \wedge dn_{\boxtimes}(P') \wedge cm?(P')}{P\$C \rightsquigarrow [cmp(P');C]}$$

(E-STEP3)

$$\frac{P \rightsquigarrow P' \wedge dn_{\boxtimes}(P') \wedge \neg cm?(P')}{P\$C \rightsquigarrow [C]}$$

(E-PAR1)

$$\frac{P \rightsquigarrow P'}{P|Q \rightsquigarrow P'_{\boxtimes}|_{\boxdot}Q}$$

(E-PAR2)

$$\frac{Q \rightsquigarrow Q'}{P|Q \rightsquigarrow P_{\boxdot}|_{\boxtimes}Q'}$$

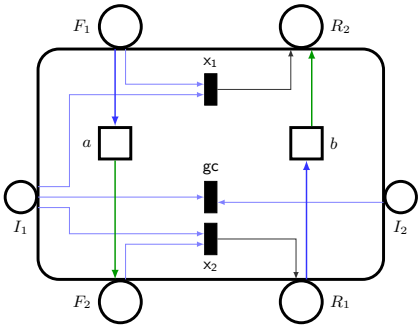
Figure 55: Predicate $P \rightsquigarrow P'$ for interrupting a process

In the previous chapter Sagas processes are encoded in safe Petri nets by structural induction (see Figure 56). The net for a saga has just three places to interact with the environment: F_1 starts its flow, F_2 signals successful termination, and E raises a fault. Each compensable process has six places to interact with the environment: a token in F_1 triggers the forward flow, to end in F_2; a token in R_1 starts the compensation, to end in R_2; a token in I_1 indicates the arrival of an interrupt from the outside; a token in I_2 informs the environment that a fault occurred.

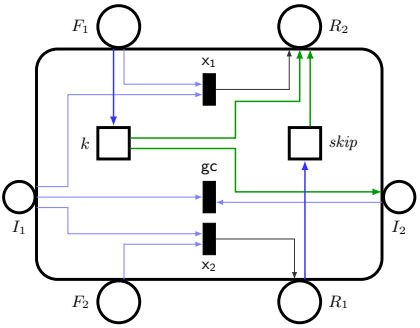
For sagas, a computation starting in F_1 will lead either to F_2 or to E, while for compensable processes we expect to have the following kinds of computations:

Successful (forward) computation: from marking F_1 the net reaches F_2

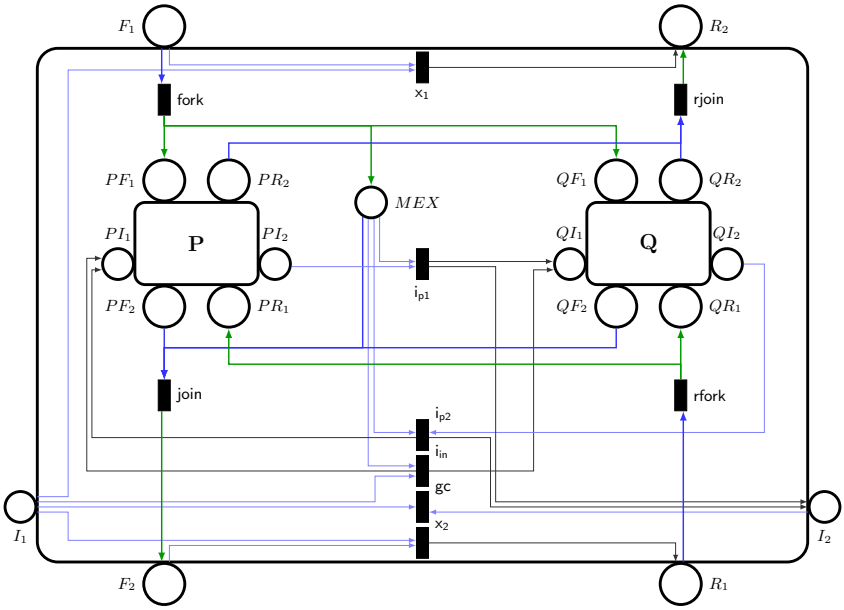
Compensating (backward) computation: from R_1 the net reaches R_2.



(a) Compensation pair $A \div B$



(b) Failing compensation pair



(c) Parallel composition

Figure 56: Petri net encoding

Aborted computation: from F_1 the net reaches $R_2 + I_2$.

Interrupted computation: from $F_1 + I_1$ the net reaches R_2.

For the reader's convenience the nets for compensable processes are depicted in Figure 56. The encoding introduces several auxiliary transitions, *e.g.* to fork and join the control flow, to catch an interrupt and reverse the flow, or for the disposal of the interrupt in case the process already produced a fault.

We introduce now the notion of weak bisimilarity for two LTS. In the following, we write $p \stackrel{\hat{\tau}}{\Rightarrow} q$ if $(p, q) \in (\stackrel{\tau}{\rightarrow})^*$ (the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}$). Moreover, for $\mu \neq \tau$ we write $p \stackrel{\hat{\mu}}{\Rightarrow} q$ if there exists p', q' such that $p' \stackrel{\mu}{\rightarrow} q'$ and $(p, p'), (q, q') \in (\stackrel{\tau}{\rightarrow})^*$.

Definition 9. *Let (S_1, L, T_1) and (S_2, L, T_2) be two LTSs over the same set of labels L. A relation $\mathbf{R} \subseteq S_1 \times S_2$ is a* weak bisimulation *if whenever $(s_1, s_2) \in \mathbf{R}$, then:*

1. *if $s_1 \stackrel{\mu}{\rightarrow}_1 s_1'$ then there exists s_2' such that $s_2 \stackrel{\hat{\mu}}{\Rightarrow}_2 s_2'$ and $(s_1', s_2') \in \mathbf{R}$; and*
2. *if $s_2 \stackrel{\mu}{\rightarrow}_2 s_2'$ then there exists s_1' such that $s_1 \stackrel{\hat{\mu}}{\Rightarrow}_1 s_1'$ and $(s_1', s_2') \in \mathbf{R}$.*

The largest weak bisimulation is called weak bisimilarity *and denoted by $\approx$.*

We shall let the marking graph of the net N_P play the role of (S_1, L, T_1) and (the fragment of) the LTS reachable from process P play the role of (S_2, L, T_2), so that $\approx$ relates markings of N_P with processes P' reachable from P. More precisely, we take the marking graph of the Petri nets assuming the only observable actions are those corresponding to activities $a \in \mathcal{A}$; all the other transitions are labelled with τ.

We have seen that the Petri net semantics associates to a compensable process P a corresponding net N_P that exchanges tokens with the context via six places: two for the forward control flow (F_1 and F_2), two for the backward compensation flow (R_1 and R_2) and two for interrupts (I_1 and I_2). The places F_1, R_1, I_1 are used to receive tokens in input from the environment, while the places F_2, R_2, I_2 are used to output tokens to the environment. Nets are usually considered up-to isomorphism,

therefore the names of their places and transitions are not important, as long as the same structure is maintained. However, to establish the behavioural correspondence between our LTS for P and the marking graph of the net N_P we need to fix a particular naming of the elements in N_P. Moreover, the same activity can occur many times in a process and every instance corresponds to a different element of the net. One way to eliminate any ambiguity is to annotate processes with the names of the places to be used for building the interface of the corresponding net (before the translation takes place).[1] This is formalised in the Appendix A, where the proof of the main theorem is also included. The proof of the main theorem requires some ingenuity to fix the correspondence between net markings and process terms. Here, we just mention that we write $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$ meaning that process P (and all its sub-processes) has been annotated in such a way that the names of the places in the "public interface" of the net N_P are $F_1, F_2, R_1, R_2, I_1, I_2$. Consider as an example the encoded transaction of Example 6. We replace *throww* with a compensation pair $T \div \mathit{skip}$ such that T is mapped to $\boxtimes$ in the context. Then the tagged version of the compensable process $A \div B; T \div \mathit{skip}$ is

$$\begin{array}{l}((A@\langle PF_1, F_3\rangle \div B@\langle R_3, R_2\rangle)@\langle PF_1, F_3, R_3, R_2, I_1, I_2\rangle;\\ (T@\langle F_3, PF_2\rangle \div \mathit{skip}@\langle R_1, R_3\rangle)@\langle F_3, PF_2, R_1, R_3, I_1, I_2\rangle)\\ @\langle PF_1, PF_2, R_1, R_2, I_1, I_2\rangle\end{array}$$

Theorem 11. *Let N_P be the Petri net associated with the tagged compensable process $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$. Then, $F_1 \approx (\boxdot, P)$.*

As an immediate consequence of the theorem and the main result of Chapter 4 the correspondence to the denotational semantics given in Chapter 3 follows. We write $[\![S]\!]$ to denote the set of traces for a saga S and use subscripts i to indicate the traces are according to policy #i.

Definition 10. *For any sagas S we let $(\!|S|\!)$ denote the set of (maximal)* weak

[1] An equivalent tagging procedure is to first translate the process to a net and then annotate the process with the names of the places that have been used in the net.

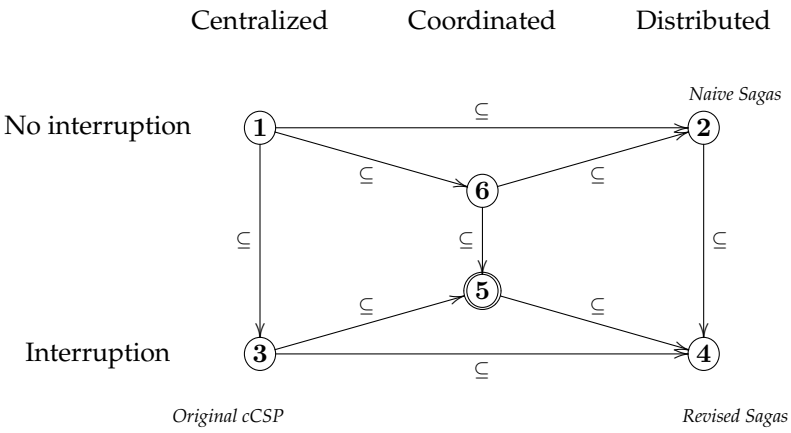


Figure 57: Compensation policies (arrows stand for trace inclusion)

traces *in our LTS semantics:*

$$
\begin{aligned}
(\!|S|\!) \triangleq \quad & \{\, a_1 ... a_n \langle \checkmark \rangle \mid \quad \exists S_1, ..., S_n, \sigma_1, ..., \sigma_{n-1}. \\
& \qquad \odot, S \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, S_1 \stackrel{\hat{a_2}}{\Rightarrow} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \odot, S_n \not\rightarrow \,\} \\
& \cup \\
& \{\, a_1 ... a_n \langle ! \rangle \mid \quad \exists S_1, ..., S_n, \sigma_1, ..., \sigma_{n-1}. \\
& \qquad \odot, S \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, S_1 \stackrel{\hat{a_2}}{\Rightarrow} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxtimes, S_n \not\rightarrow \,\}
\end{aligned}
$$

Actually, under the assumption that compensation cannot fail, only successful traces are present in $(\!|S|\!)$ (as well as in $[\![S]\!]_i$ for any $i \in [1, 5]$). This is not necessarily the case for the extension in Subsection 5.5.2.

Corollary 6. *For any sagas $S = \{\!|P|\!\}$ we have $[\![S]\!]_5 = (\!|S|\!)$. Moreover, if P is sequential then $[\![S]\!]_i = (\!|S|\!)$ for $i \in [1, 5]$.*

5.4 Dealing with Different Policies

In this section we show that we can easily tune the LTS semantics for parallel sagas to match and improve other compensation policies discussed in Chapter 3. In Figure 57 we show the relation between the different

(E-COMP)

$$\frac{}{[C] \rightsquigarrow [C]}$$

(E-STEP1)

$$\frac{P \rightsquigarrow P' \wedge \neg dn_{\boxtimes}(P')}{P\$C \rightsquigarrow P'\$C}$$

(E-STEP2)

$$\frac{P \rightsquigarrow P' \wedge dn_{\boxtimes}(P') \wedge cm?(P')}{P\$C \rightsquigarrow [cmp(P'); C]}$$

(E-STEP3)

$$\frac{P \rightsquigarrow P' \wedge dn_{\boxtimes}(P') \wedge \neg cm?(P')}{P\$C \rightsquigarrow [C]}$$

(E-PAR1)

$$\frac{P \rightsquigarrow P'}{P|Q \rightsquigarrow P'_{\boxtimes}|_{\boxdot}Q}$$

(E-PAR2)

$$\frac{Q \rightsquigarrow Q'}{P|Q \rightsquigarrow P_{\boxdot}|_{\boxtimes}Q'}$$

Figure 58: Remaining rules for predicate $\rightsquigarrow$ in policy #6

policies. We will present here policies #6 and #3 as different versions of the already presented LTS semantics.

5.4.1 Notification and distributed compensation

To remove the possibility to interrupt a sibling process before it ends its execution we just redefine the "extract" predicate by removing most cases, so that the interrupt is possible only when the process is "done". The remaining rules are shown in Figure 58. We remove the rules for compensation pairs (E-PAIR) and sequential composition (E-SEQ1 and E-SEQ2).

Now, the rule INT is only applicable if the interrupted process consists of an installed compensation $[C]$. Moreover, we only change the subscripts, not the process: since any interrupted thread is "done" we never inhibit sibling forward activities upon a fault.

Proposition 3. *Let $(\!|\cdot|\!)_6$ denote the set of weak traces generated by policy #6 above. Then, for any sagas $S = \{\![P]\!\}$ we have $[\![S]\!]_1 \subseteq (\!|S|\!)_6 \subseteq [\![S]\!]_2$. Moreover, for some processes P the inclusion is strict, while for all sequential processes*

$[\![P]\!]_1 = (\!|P|\!)_6 = [\![P]\!]_2$.

Proof. Let $S = \{[P]\}$.

The proof of the inclusion $[\![S]\!]_1 \subseteq (\!|S|\!)_6$ is by induction on the structure of P. Actually, we prove a stronger result, namely that for any compensable process P:

1. for any $(p\langle\checkmark\rangle, q) \in [\![P]\!]_1$ then $\exists P_1, ..., P_n, P'_1, ..., P'_m$.
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, P_n \not\rightarrow$ with $p = a_1 a_2 ... a_n$;
   - $\boxtimes, [cmp(P_n)] \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\rightarrow$ with $q = b_1 ... b_m\langle\checkmark\rangle$.
2. for any $(p\langle!\rangle, q) \in [\![P]\!]_1$ then $\exists P_1, ..., P_n, P'_1, ..., P'_m$.
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \boxdot, P_{n-1} \stackrel{\hat{a_n}}{\Rightarrow} \boxtimes, P_n$ with $p = a_1 a_2 ... a_n$;
   - $\boxtimes, P_n \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\rightarrow$ with $q = b_1 ... b_m\langle\checkmark\rangle$.

(Remind that no trace in $[\![P]\!]_1$ can end with $\langle?\rangle$.)

The more interesting case is that of parallel composition, where we just notice that we can postpone the application of rule INT as long as possible, *i.e.*, until each branch is done.

The proof of the inclusion $(\!|S|\!)_6 \subseteq [\![S]\!]_2$ is analogous, but the stronger inductive argument we use is the following. For any compensable process P:

1. for any runs
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, P_n \not\rightarrow$
   - $\boxtimes, [cmp(P_n)] \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\rightarrow$

   we have $(p\langle\checkmark\rangle, q) \in [\![P]\!]_2$ for $p = a_1 a_2 ... a_n$ and $q = b_1 ... b_m\langle\checkmark\rangle$.
2. for any run $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \sigma_2, P_2 \cdots \stackrel{\hat{a_n}}{\Rightarrow} \sigma_n, P_n \not\rightarrow$ with $\sigma_n = \boxtimes$, then there exists some $k \leq n$ such that we have $(p\langle!\rangle, q\langle\checkmark\rangle) \in [\![P]\!]_2$ for $p = a_1 a_2 ... a_k$ and $q = a_{k+1} ... a_n$.
3. for any runs
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \stackrel{\hat{a_h}}{\Rightarrow} \boxdot, P_h$

- $P_h \rightsquigarrow Q$
- $\boxtimes, Q \stackrel{a_{h+1}\hat{}}{\Rightarrow} \boxtimes, P_{h+1} \stackrel{a_{h+2}\hat{}}{\Rightarrow} \boxtimes, P_{h+2} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxtimes, P_n \not\rightarrow$

then there exists some $k \leq n$ such that we have $(p\langle!\rangle, q\langle\checkmark\rangle) \in [\![P]\!]_2$ for $p = a_1a_2...a_k$ and $q = a_{k+1}...a_n$.

The more interesting case is proving the second assertion for the parallel composition. Let $P = P'|P''$. Since $\sigma_n = \boxtimes$ and $P_n \not\rightarrow$, it must be the case that $P_n = P'_n{\boxtimes}|_{\boxtimes}P''_n$ for some P'_n and P''_n such that $dn_{\boxtimes}(P'_n)$, $dn_{\boxtimes}(P''_n)$, $\neg cm?(P'_n)$ and $\neg cm?(P''_n)$. Let j be the unique index such that $\sigma_j = \boxdot$ and $\sigma_{j+1} = \boxtimes$. Then, either rule PAR-L or PAR-R must have been applied to prove $\boxdot, P_j \stackrel{a_j}{\rightarrow} \boxtimes, P_{j+1}$. Without loss of generality, assume PAR-L has been used, $P_j = P'_j{\boxdot}|_{\boxdot}P''_j$, $P_{j+1} = P'_{j+1}{\boxtimes}|_{\boxdot}P''_{j+1}$ with $\boxdot, P'_j \stackrel{a_j}{\rightarrow} \boxtimes, P'_{j+1}$ and $P''_{j+1} = P''_j$. Now let i be the unique index such that $P_i = P'_i{\boxtimes}|_{\boxdot}P''_i$ and $P_{i+1} = P'_{i+1}{\boxtimes}|_{\boxtimes}P''_{i+1}$ (with $j \leq i \leq n$). Either rule PAR-R has been used, or rule INT-R to prove $\boxtimes, P_i \stackrel{a_i}{\rightarrow} \boxtimes, P_{i+1}$.

- If PAR-R has been used, then it means that $\boxdot, P''_i \stackrel{a_i}{\rightarrow} \boxtimes, P''_{i+1}$. Then, each step in the sequence must have been obtained by using either PAR-L or PAR-R. Hence the sequence is just made by the interleaving of two analogous subsequences, one from $\boxdot, P'$ to $\boxtimes, P'_n$ and one from $\boxdot, P''$ to $\boxtimes, P''_n$. Then, by inductive hypothesis we have $(p'\langle!\rangle, q'\langle\checkmark\rangle) \in [\![P']\!]_2$ and $(p''\langle!\rangle, q''\langle\checkmark\rangle) \in [\![P'']\!]_2$ for suitable p', q', p'', q'' and the set of possible interleavings of $p'q'$ with $p''q''$ certainly contains the sequence $a_1a_2...a_n$. Thus, we take $k = n$ and we are done.

- If INT-R has been used, then we distinguish three cases.

  If $P''_i = [C]$, then $dn_{\boxdot}(P''_i)$ and the overall sequence is just made by the interleaving of one sequence from $\boxdot, P'$ to $\boxtimes, P'_n$ and one obtained by concatenating a run $\boxdot, P''$ to $\boxdot, P''_i$ and one from $\boxtimes, [C]$ to $\boxtimes, P''_n$ and we conclude by inductive hypothesis similarly to the case when PAR-R was used.

  If $P''_i = Q''_i|R''_i$ or $P''_i = Q''_i\$C$ the overall sequence is just made by the interleaving of one sequence from $\boxdot, P'$ to $\boxtimes, P'_n$ and one obtained by concatenating a run $\boxdot, P''$ to $\boxdot, P''_i$ and one from $\boxtimes, P''_{i+1}$ to $\boxtimes, P''_n$ and we conclude by inductive hypothesis similarly to previous cases.

As a counterexample to the equality, we take the booking of a trip bT' from Example 9 and assume the booking of the hotel fails: the (weak) trace $\mathsf{rT}\ \mathsf{bFcF}\ \mathsf{cC}\mathit{skip}\ \mathsf{cR}\langle\checkmark\rangle$ belongs to $(\!|S|\!)_6$ but not to $[\![S]\!]_1$, while the trace $\mathsf{rT}\ \mathsf{cC}\mathit{skip}\ \mathsf{bFcF}\ \mathsf{cR}\langle\checkmark\rangle$ belongs to $[\![S]\!]_2$ but not to $(\!|S|\!)_6$.

The fact that the semantics coincide on sequential processes is a trivial consequence of the fact that $[\![S]\!]_1 \subseteq (\!|S|\!)_6 \subseteq [\![S]\!]_2$ and $[\![S]\!]_1 = [\![S]\!]_2$ when S is sequential. □

Example 10. *We show an example execution for the small-step semantics for policy #6 using the process of Example 9. Assume this time that the booking of the hotel fails. A possible computation is:*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\boxdot, bT'} \\
\xrightarrow{\mathsf{rT}} & \boxdot, ((\mathsf{bF} \div \mathsf{cF}\ ;\ \mathsf{bH} \div \mathsf{cH}) \mid \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} & \text{S-SEQ,S-ACT} \\
\xrightarrow{\mathsf{bF}} & \boxdot, ((\mathsf{bH} \div \mathsf{cH})\$\mathsf{cF} \mid \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} & \text{STEP,PAR-L, S-SEQ,S-ACT} \\
\xrightarrow{\tau} & \boxtimes, ([\mathsf{cF}]\ {}_{\boxtimes}|_{\boxdot}\ \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} & \text{STEP, PAR-L, F-ACT} \\
\xrightarrow{\mathsf{cF}} & \boxtimes, ([\mathbf{nil}]\ {}_{\boxtimes}|_{\boxdot}\ \mathsf{cC} \div \mathit{skip})\$\mathsf{cR} & \text{STEP,PAR-L,COMP,C-ACT} \\
\xrightarrow{\mathsf{cC}} & \boxtimes, ([\mathbf{nil}]\ {}_{\boxtimes}|_{\boxdot}\ [\mathit{skip}])\$\mathsf{cR} & \text{STEP,PAR-R,S-ACT} \\
\xrightarrow{\tau} & \boxtimes, [(\mathbf{nil} \mid \mathit{skip})\ ;\ \mathsf{cR}] & \text{AS-STEP1,INT-R,E-COMP} \\
\xrightarrow{\mathit{skip}} & \boxtimes, [\mathsf{cR}] & \text{COMP,C-SEQ2,C-PAR-R,C-ACT} \\
\xrightarrow{\mathsf{cR}} & \boxtimes, [\mathbf{nil}] & \text{COMP,C-ACT}
\end{array}
$$

Note that it is not possible to interrupt the right branch before executing the forward action cC*. However we can execute the left compensation* cF *even though not every branch was interrupted yet.*

5.4.2 Interruption and centralized compensation

To move from distributed to centralized execution we simply strengthen the premise in PAR-L (and PAR-R):

$$
\text{(PAR-L)}\quad \frac{(\sigma_1 = \boxdot \vee dn_{\boxtimes}(P_{\sigma_1}|_{\sigma_2}Q) \vee \neg dn_{\boxtimes}(P)) \qquad \Gamma \vdash \sigma_1, P \xrightarrow{\lambda} \sigma_1', P'}{\Gamma \vdash \sigma, P_{\sigma_1}|_{\sigma_2}Q \xrightarrow{\lambda} \sigma \sqcap \sigma_1', P'_{\sigma_1'}|_{\sigma_2}Q}
$$

Thus a process can only be executed if it is either moving forward ($\sigma_1 = \boxdot$) or the complete parallel composition finished in a failing case

($dn_{\boxtimes}(P_{\sigma_1}|_{\sigma_2}Q)$) or the thread has not yet finished its execution in a failing case ($\neg dn_{\boxtimes}(P)$).

Proposition 4. *Let $(\!|\cdot|\!)_3$ denote the set of weak traces generated by policy #3 above. Then, for any sagas $S = \{[P]\}$ we have $[\![S]\!]_3 = (\!|S|\!)_3$.*

Proof. The proof of the two inclusions is similar to that of Proposition 3.

The inclusion $[\![S]\!]_3 \subseteq (\!|S|\!)_3$ is shown by induction on the structure of P. For each kind of trace we have to show a possible derivation in the small-step semantics.

1. for any $(p\langle\checkmark\rangle, q) \in [\![P]\!]_3$ then $\exists P_1, ..., P_n, P'_1, ..., P'_m$.
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, P_n \not\to$ with $p = a_1a_2...a_n$;
   - $\boxtimes, [cmp(P_n)] \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\to$ with $q = b_1...b_m\langle\checkmark\rangle$.
2. for any $(p\langle!\rangle, q) \in [\![P]\!]_3$ then $\exists P_1, ..., P_n, P'_1, ..., P'_m$.
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \sigma, P_{n-1} \stackrel{\hat{a_n}}{\Rightarrow} \boxtimes, P_n$ such that $dn_{\boxtimes}(P_n)$ and $\neg dn_{\boxtimes}(P_i)$ for all $i < n$ with $p = a_1a_2...a_n$;
   - $\boxtimes, P_n \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\to$ with $q = b_1...b_m\langle\checkmark\rangle$.
3. for any $(p\langle?\rangle, q) \in [\![P]\!]_3$ then $\exists P_1, ..., P_n, P'_1, ..., P'_m$.
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \boxdot, P_{n-1} \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, P_n$ with $p = a_1a_2...a_n$;
   - $\boxtimes, P'_n \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\to$ with $P_n \rightsquigarrow^+ P'_n$ such that $dn_{\boxtimes}(P'_n)$ and $q = b_1...b_m\langle\checkmark\rangle$. (We postpone the interrupt until any forward action has been executed, then possibly multiple interrupts have to be issued.)

Now consider the inclusion $(\!|S|\!)_3 \subseteq [\![S]\!]_3$, we have to show by induction:

1. for any runs
   - $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \boxdot, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \boxdot, P_2 \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, P_n \not\to$
   - $\boxtimes, [cmp(P_n)] \stackrel{\hat{b_1}}{\Rightarrow} \boxtimes, P'_1 \stackrel{\hat{b_2}}{\Rightarrow} \boxtimes, P'_2 \cdots \stackrel{\hat{b_m}}{\Rightarrow} \boxtimes, P'_m \not\to$

   we have $(p\langle\checkmark\rangle, q) \in [\![P]\!]_3$ for $p = a_1a_2...a_n$ and $q = b_1...b_m\langle\checkmark\rangle$.

2. for any run $\boxdot, P \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, P_1 \stackrel{\hat{a_2}}{\Rightarrow} \sigma_2, P_2 \cdots \stackrel{\hat{a_n}}{\Rightarrow} \sigma_n, P_n \not\rightarrow$ with $\sigma_n = \boxtimes$, then there exists some $k \leq n$ such that we have $(p\langle!\rangle, q\langle\checkmark\rangle) \in [\![P]\!]_3$ for $p = a_1 a_2 ... a_k$ and $q = a_{k+1} ... a_n$.

The more interesting case is proving the second assertion for the parallel composition. Let $P = P'|P''$. Since $\sigma_n = \boxtimes$ and $P_n \not\rightarrow$, it must be the case that $P_n = P'_n \boxtimes | \boxtimes P''_n$ for some P'_n and P''_n such that $dn_{\boxtimes}(P'_n)$, $dn_{\boxtimes}(P''_n)$, $\neg cm?(P'_n)$ and $\neg cm?(P''_n)$. Let j be the unique index such that $\sigma_j = \boxdot$ and $\sigma_{j+1} = \boxtimes$. Then, either rule PAR-L or PAR-R must have been applied to prove $\boxdot, P_j \stackrel{a_j}{\rightarrow} \boxtimes, P_{j+1}$. Without loss of generality, assume PAR-L has been used, $P_j = P'_j \boxdot | \boxdot P''_j$, $P_{j+1} = P'_{j+1} \boxtimes | \boxdot P''_{j+1}$ with $\boxdot, P'_j \stackrel{a_j}{\rightarrow} \boxtimes, P'_{j+1}$ and $P''_{j+1} = P''_j$. Note that P'_{j+1} can now only move if it is not finished yet. We consider two cases:

$dn_{\boxtimes}(P'_{j+1})$ Then only P''_{j+1} can move. Let i be the unique index such that $\neg dn_{\boxtimes}(P_i)$ and $dn_{\boxtimes}(P_{i+1})$. Note that this implies $\neg dn_{\boxtimes}(P''_i)$, so P'' can still move forward until $i+1$. Thus until $i+1$ any observable action was a forward action. We take $k = i$ and are done.

$\neg dn_{\boxtimes}(P'_{j+1})$ The case is similar though now also P'_{j+1} can move. However in any case we reach an index i such that $\neg dn_{\boxtimes}(P_i)$ and $dn_{\boxtimes}(P_{i+1})$ and take $k = i$.

□

Note that by combining the above changes in Section 5.4.1 and 5.4.2 we recover policy #1.

5.5 Possible extensions

In Section 5.4 we have shown how the rules of our LTS semantics can be easily adjusted to cover other compensation policies. In this section we sketch how to extend the LTS semantics (independently from the preferred policy) to take other aspects into account.

5.5.1 Choice and iteration

Our first extension adds choice and iteration operators to the basic syntax for processes

$$P ::= \ldots \mid P + P \mid P^*$$

$$\underset{\text{(E-CHOICE)}}{dn_\sigma(P+Q) \triangleq dn_\sigma(P) \wedge dn_\sigma(Q)} \qquad \underset{\text{(E-ITER)}}{dn_\sigma(P^*) \triangleq dn_\sigma(P)}$$

$$\frac{}{P+Q \rightsquigarrow [\mathbf{nil}]} \qquad \frac{}{P^* \rightsquigarrow [\mathbf{nil}]}$$

(CHOICE-L)
$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \sigma, P'}{\Gamma \vdash \boxdot, P+Q \xrightarrow{\lambda} \sigma, P'}$$

(CHOICE-R)
$$\frac{\Gamma \vdash \boxdot, Q \xrightarrow{\lambda} \sigma, Q'}{\Gamma \vdash \boxdot, P+Q \xrightarrow{\lambda} \sigma, Q'}$$

(E-ITER)
$$\frac{}{\Gamma \vdash \boxdot, P^* \xrightarrow{\tau} \boxdot, [\mathbf{nil}]}$$

(S-ITER)
$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge \neg dn(P')}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxdot, P'; P^*}$$

(A-ITER)
$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxtimes, P'}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxtimes, P'}$$

(S-ITER2)
$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn(P')}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxdot, P^* \$ cmp(P')}$$

Figure 59: LTS for choice and iteration

The corresponding rules are in Figure 59. In a process $P + Q$ one option is nondeterministically executed while the alternative is dropped. For iteration we exploit the fact that our LTS allows τ as a label. Thus a process P^* either executes a τ and finishes or acts as the sequential composition $P; P^*$. Note that, while it is easy to account for choice and iteration in the denotational semantics, the extension is harder for the Petri net semantics. For example, let us consider the sequential process $(A \div A' + B \div B')^*; \mathit{throww}$. At any iteration, either A or B is executed and thus either A' or B' is installed. When the iteration is closed, the installed compensation may be any arbitrary sequence of A' or B', an information that cannot be recorded in the state of a finite (safe) Petri net.

Example 11. *We consider the example of an eStore. First we accept the order, then we either pack an item or, if it is not in Store, we abort. We repeat this several times (until all items are packed or we aborted) and then book a courier. We formalise this behaviour as follows:*

$$eS \triangleq \mathsf{aO} \div \mathsf{cO}; (\mathsf{pI} \div \mathsf{uI} + \mathsf{niS} \div \mathit{skip})^*; \mathsf{bC} \div \mathsf{cC}$$

where niS *fails and the other actions are successful. Then we can have the fol-*

(C-ACT)
$$\frac{A \mapsto_\Gamma \sigma}{\Gamma \vdash \boxdot, A \xrightarrow{A} \sigma, \mathbf{nil}}$$
(F-C-SEQ)
$$\frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxtimes, C'}{\Gamma \vdash \boxdot, C; D \xrightarrow{\lambda} \boxtimes, C'}$$
(C-PAR-L)
$$\frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \sigma, C'}{\Gamma \vdash \boxdot, C|D \xrightarrow{\lambda} \sigma, C'|D}$$

Figure 60: Additional or changed rules for failing compensations

lowing computation:

$$\begin{array}{lll}
\boxdot, eS & \xrightarrow{\mathsf{aO}} & \boxdot, ((\mathsf{pI} \div \mathsf{uI} + \mathsf{niS} \div \mathit{skip})^*; \mathsf{bC} \div \mathsf{cC})\$\mathsf{cO} \qquad \text{S-SEQ, S-ACT} \\
& \xrightarrow{\mathsf{pI}} & \boxdot, ((\mathsf{pI} \div \mathsf{uI} + \mathsf{niS} \div \mathit{skip})^*; \mathsf{bC} \div \mathsf{cC})\$\mathsf{uI}\$\mathsf{cO} \\
& & \qquad \text{STEP,S-ITER2,CHOICE-L, S-ACT} \\
& \xrightarrow{\mathsf{niS}} & \boxtimes, [\mathsf{uI}; \mathsf{cO}] \qquad \text{AS-STEP1,AS-STEP2,F-ITER,CHOICE-R, F-ACT}
\end{array}$$

In the computation after packing one item the next one is not in store. Thus the process aborts and we have to compensate.

5.5.2 Failing compensations

One important contribution of [BMM05] was the ability to account for the failure of compensations. Here we discuss the changes needed to extend our LTS semantics accordingly.

Compensations. We extend the state in LTS for compensations with $\Omega = \{\boxdot, \boxtimes\}$, modify the sources / targets from C to $\boxdot, C$ in the rules we have presented, change the rule C-ACT and finally add the rules F-C-SEQ, C-PAR-L and C-PAR-R that record the execution of a faulty compensation in the target of the transition. The new rules are shown in Figure 60

Compensable processes. We extend the state in LTS for processes to $\Omega^{\circledast} = \{\boxdot, \boxtimes, \circledast\}$, where the symbol $\circledast$ denotes the fault of a compensation, i.e., a non recoverable crash. As a matter of notation for meta-

(COMP-1)
$$\frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxdot, C'}{\Gamma \vdash \delta, [C] \xrightarrow{\lambda} \delta, [C']}$$
(COMP-2)
$$\frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxtimes, C'}{\Gamma \vdash \delta, [C] \xrightarrow{\lambda} \circledast, [C']}$$
(C-STEP)
$$\frac{\Gamma \vdash \boxtimes, P \xrightarrow{\lambda} \circledast, P'}{\Gamma \vdash \boxtimes, P\$C \xrightarrow{\lambda} \circledast, P'}$$

Figure 61: New rules for Sagas with failing compensations

variables, we let $\sigma, \ldots \in \Omega$ and $\delta \in \{\boxtimes, \circledast\}$. When executing a compensation $[C]$, we must take into account the possibility of a crash (COMP-1 and COMP-2). Moreover, if we generate a crash, previously installed local compensations will not be executed (C-STEP). The new rules are given in Figure 61.

Note that in the premises of rules COMP-1 and COMP-2 we intentionally put $\boxdot$ in the source of the transition, because the LTS for compensations has only such states as sources of transitions. The other rules for sequential Sagas stay as before.

For parallel composition we redefine the predicate dn such that

$$dn_{\boxdot}(P_{\boxdot}|_{\boxdot}Q) \triangleq dn_{\boxdot}(P) \wedge dn_{\boxdot}(Q) \quad dn_{\delta}(P_{\delta_1}|_{\delta_2}Q) \triangleq dn_{\delta}(P) \wedge dn_{\delta}(Q)$$

where $\delta, \delta_1, \delta_2 \in \{\boxtimes, \circledast\}$. The rules PAR-L/PAR-R are as before however for any meta-variable we allow also $\circledast$ as a possible value, *i.e.*, $\sigma, \sigma_1, \sigma_2, \sigma'_1, \sigma'_2 \in \Omega^{\circledast}$. Thus we have to extend the operation $\sqcap$ such that $\circledast \sqcap \sigma = \circledast$. The rules INT-L/INT-R are also applicable in a global $\circledast$ state.

The rules guarantee that in case of a crash parallel branches can execute their compensations as far as possible, only previously installed compensations (i.e., before the parallel composition,) are not reachable anymore.

Example 12. *Consider the computation of Example 9. We assume that also the*

cancelling of the flight fails, then te computation changes as follows:

$$\begin{array}{ll} \boxdot, bT' \xrightarrow{\mathsf{rT}} \xrightarrow{\mathsf{bF}} \xrightarrow{\tau} & \\ \xrightarrow{\tau} \quad \boxtimes, [(\mathsf{cF} \mid \mathbf{nil}); \mathsf{cR}] & \text{AS-STEP1,INT-R} \\ \xrightarrow{\mathsf{cF}} \quad \circledast, [\mathbf{nil} \mid \mathbf{nil}] & \text{COMP-2,F-C-SEQ,C-PAR-L,C-ACT} \end{array}$$

The failing compensation causes a crash. Thus the remaining compensation cR *cannot be executed.*

5.6 Tool support

In this section we present a tool support for the LTS semantics presented in the Sections 5.1, 5.2 and 5.4. The tool is implemented in Maude (see Section 2.6) and can be found at [Sou13].

The implementation is structured in three modules, a functional module `SMALLSTEP-SYNTAX` and two system modules `SMALLSTEP-SEM` and `SMALLi` for the respective policy #i.

The first module defines the syntax for the three basic sorts `Sagaprocess`, `Cprocess` and `Compensation`. It uses the runtime syntax of Section 5.2. We use the natural numbers to denote basic activities and add the special action `tau`. The sort for actions is called `Name` or `Names` for lists of actions. Final symbols can be `ok` for $\boxdot$ and `fail` for $\boxtimes$. The module then defines auxiliary functions such as the predicate `dn` for both compensations and compensable processes as well as the function `cmp` extracting a compensation from a process that is "done".

The module `SMALLSTEP-SEM` defines the sort `Environment` as a set of actions such that an action is successful if it is contained in the environment. The sort `State` can have three possible definitions, one for each basic sort for processes, as well as a definition adding the environment to the state:

```
op <_,_,_> : FinalSymbol Cprocess Names
                                -> State [frozen].
op <_,_,_> : FinalSymbol Sagaprocess Names
                                -> State [frozen].
op <_,_> : Compensation Names -> State [frozen].
op _|-_ : Environment State   -> State [frozen] .
```

The last item `Names` in any state contains the observed flow of actions including `tau`. Note that as described in [VMO06] to model SOS semantics in Maude we have to make sure that the state changes in every single step. Otherwise the system continues rewriting for subprocesses ignoring possible nondeterminism. The observed names adds exactly one observed name in each step thus guaranteeing that our system works correctly.

On the states we define rewrite rules modelling the transitions of Figures 51–54. Exemplary take rules `S-ACT` and `F-ACT` for compensation pairs. They are modelled as the following two conditional rewrite rules:

```
crl [s-act] : E |- < ok , A / B , L >      =>
      E |- < ok , [B] , L A >        if A in E .
crl [f-act] : E |- < ok , A / B , L >      =>
      E |- < fail , [nil] , L tau >  if not(A in E) .
```

Depending on the success of action `A`, *i.e.*, whether `A` is included in the environment `E`, the two rules result in two different states. If `A` is successful then its compensation `B` is installed, the overall state remains `ok` and `A` is added to the observed flow. If `A` aborts, the empty compensation `nil` is installed, the overall state changes to `fail` and we observe a `tau` for the failure.

Note that the rules for parallel composition are specific for the different policies and thus defined in `SAGAi` for each policy #i. Moreover we define the special operation `extract` that is used for the predicate $\rightsquigarrow$ defined in Figure 55. As the predicate is nondeterministic we use rewrite rules for its implementation.

In Figure 62 we present an example computation in the small-step semantics using the tool. The initial state is

```
(1,2) |- < ok,{1 / 2 ok || ok 3 / 4},nnil >
```

i.e., the environment includes the two activities 1 and 2, the overall state is `ok`, then there is the transaction consisting of a parallel composition of two compensation pairs, and the initially empty flow `nnil` for labels. As action 3 aborts the saga fails. There are three possible results. In the first solution we observe only two `tau`. In this case first the right

```
Maude> search in SMALL5 :
       (1,2) |- < ok,{1 / 2 ok || ok 3 / 4},nnil >
          =>! S:State .

Solution 1 (state 4)
states: 7  rewrites: 577 in 8ms cpu (6ms real)
                     (72125 rewrites/second)
S:State --> (1,2) |- < ok,snil,tau tau >

Solution 2 (state 8)
states: 10  rewrites: 766 in 8ms cpu (8ms real)
                      (95750 rewrites/second)
S:State --> (1,2) |- < ok,snil,1 tau tau 2 >

Solution 3 (state 9)
states: 10  rewrites: 766 in 8ms cpu (8ms real)
                      (95750 rewrites/second)
S:State --> (1,2) |- < ok,snil,tau 1 tau 2 >

No more solutions.
states: 10  rewrites: 766 in 8ms cpu (9ms real)
                      (95750 rewrites/second)
```

Figure 62: Example of a computation using the tool for the small-step semantics

branch aborts and then the left branch is immediately interrupted. In the other two cases the left branch executes its forward action and thus its compensation has to run. Note that before the compensation the two `tau` have to be observed, first for the failure, then for the interrupt. In each case the final result is again in a commit state due to the rule A-SAGA2.

5.7 Conclusion

We presented an LTS semantics for the Sagas calculus. Using a weak bisimulation we investigated the correspondence with previously defined Petri net and denotational semantics. Moreover, with small changes

we can deploy a different policy for the execution of concurrent compensable processes. We have shown possible extensions for the semantics enriching first the syntax and then the LTS itself.

Note that a small-step semantics for cCSP was defined in [BR05]. It relies on the centralized compensation policy, but is otherwise similar to our approach. Using a synchronizing step at the end of the forward flow the success or failure of the transaction is published, in case of a failure the compensations are executed as normal saga processes (outside the transaction scope). On the other hand in our approach the information about a failure is kept in the state and compensations are executed inside the saga.

Chapter 6

Dynamic logic for long-running transactions

In this chapter we shift our focus to long-running transactions. On the one hand we focus on verification and suggest a logic to define program properties. We propose an extension of dynamic logic, a temporal logic reasoning over complete programs instead of single steps. The choice of extending dynamic logic is not incidental: we have been inspired by the interesting literature using deontic logic for error handling [BWM01, CM09]. On the other hand we move from an abstract model closer to standard programming paradigms, where we study concurrent programs based on compensation pairs using assignments instead of regular actions. Moreover we move from comparing different policies for compensation handling to the verification of programs in one particular policy The main result establishes some sufficient conditions under which a compensable program is guaranteed to always restore a correct state after a fault.

We start by giving some background on dynamic logic including related work regarding error handling in Section 6.1.3 and concurrency in Section 6.1.4. In Section 6.2 we present how we can interpret concurrent programs and define formulas over them. Section 6.3 extends this logic with compensations and transactions. In Section 6.4 we introduce tool

support and in Section 6.5 we extend the language to handle interruption of parallel branches.

The contents of this chapter was first published in [BFK12]. It is presented here with more examples, a new definition of (strong) serializability (Definitions 28, 29, 35, 36), an extended discussion regarding valid formulas (Propositions 6, 7) and including extended proofs. Sections 6.4 and 6.5 are original to this work.

6.1 Background on Dynamic Logic

In this section we give a summary of the basic concepts of first order dynamic logic [Har79]. It was introduced to reason directly about programs, using classical first order predicate logic and modal logics combined with the algebra of regular events. We start by introducing programs, then we switch to first-order dynamic logic. In the second part of this section, we shall briefly overview related work on the two main concepts for our extension of dynamic logic, namely deontic formalisms for error handling and concurrency.

6.1.1 Programs

Let $\Sigma = \{f, g, \ldots, p, q \ldots\}$ be a finite first-order vocabulary where f, g range over Σ-function symbols and p, q over Σ-predicates. Each element of Σ has a fixed arity, and we denote by Σ_n the subset of symbols with arity $n > 0$. Moreover let $V = \{x_0, x_1, \ldots\}$ be a countable set of variables. Let $Trm(V) = \{t_1, \ldots, t_n, \ldots\}$ be the set of terms over the signature Σ with variables in V. We use the grammar

$$\phi ::= p(t_1, \cdots, t_n) \mid \top \mid \bot \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi$$

to denote the set $Pred(V)$ where $p \in \Sigma_n$ a predicate and $t_1, \cdots t_n \in Trm(V)$.

Definition 11 (Activities)**.** *Let $x_1, \cdots, x_n \in V$ and $t_1, \cdots, t_n \in Trm(V)$. A basic activity $a \in Act(V)$ is a multiple assignment $x_1, \cdots, x_n := t_1, \cdots, t_n$.*

As special cases, we write a single assignment as $x := t \in Act(V)$ (with $x \in V$ and $t \in Trm(V)$) and the empty assignment for the inaction $skip$.

Example 13 (e-Store). *Instead of using the running example in this chapter we use several different one. We take an e-Store as our first example (see Figure 63 for the complete presentation). Activity $acceptOrder \in Act(V)$ is defined as a multiple assignment to variables $stock$ and $card$ such that $acceptOrder \triangleq stock, card := stock - 1, unknown$. This activity decreases the items in stock by one (the item being sold is no longer available), and resets the current state of the credit card to unknown.*

Basic activities can be combined in different ways. While it is possible to consider while programs (with sequential composition, conditional statements and while loops), we rely on the more common approach based on the so-called regular programs.

Definition 12 (Programs). *A program α is any term generated by the grammar:*

$$\alpha, \beta \ ::= \ a \ \mid \ \phi? \ \mid \ \alpha\,;\beta \ \mid \ \alpha + \beta \ \mid \ \alpha^*$$

A program is either: a basic activity $a \in Act(V)$; a test operator $\phi?$ for $\phi \in Pred(V)$; a sequential composition $\alpha\,;\beta$ of programs α and β; a nondeterministic choice $\alpha + \beta$ between programs α and β; or an iteration α^*.

To define the semantics of programs we introduce a computational domain.

Definition 13 (Computational Domain). *Let Σ be a first-order vocabulary. A first-order structure $\mathcal{D} = (D, I)$ is called the domain of computation such that: D is a non-empty set, called the carrier, and I is a mapping assigning:*

- *to every n-ary function symbol $f \in \Sigma$ a function $f^I : D^n \to D$;*
- *to every n-ary predicate $p \in \Sigma$ a predicate $p^I : D^n \to Bool$.*

A state is a function $s : V \to D$ that assigns to each variable an element of D. The set of all states is denoted by $State(V)$. As usual, we denote by $s[x \mapsto v]$ the state s' such that $s'(x) = v$ and $s'(y) = s(y)$ for $y \neq x$. Now we can extend the interpretation to terms in a given state.

Definition 14 (Term Valuation). *The valuation val of a term $t \in Trm(V)$ in a state $s \in State(V)$ is defined by:*

$$\begin{array}{lcl} val(s,x) & \triangleq & s(x) \text{ if } x \in V; \\ val(s,f(t_1,\ldots,t_n)) & \triangleq & f^I(val(s,t_1),\ldots,val(s,t_n)). \end{array}$$

Basic activities (and thus programs) are interpreted as relations on states. For basic activities this means evaluating the assignments in the current state replacing the old values of the variables.

Definition 15 (Interpretation of Activities). *Let the activity $a \in Act(V)$ be defined by a multiple assignment $x_1,\cdots,x_n := t_1,\cdots,t_n$. The valuation $\rho \in 2^{State(V)\times State(V)}$ of activity a is defined by:*

$$\rho(a) \triangleq \{(s,s') \mid s' = s[x_1 \mapsto val(s,t_1),\cdots,x_n \mapsto val(s,t_n)]\}$$

Definition 16 (Interpretation of Programs). *We extend the interpretation ρ of basic activities to programs in the following manner:*

$$\begin{array}{lcl} \rho(\alpha\,;\beta) & \triangleq & \{(s,r) \mid (s,w) \in \rho(\alpha) \wedge (w,r) \in \rho(\beta)\} \\ \rho(\alpha \,+\, \beta) & \triangleq & \rho(\alpha) \cup \rho(\beta) \\ \rho(\phi?) & \triangleq & \{(s,s) \mid s \models \phi\} \\ \rho(\alpha^*) & \triangleq & \{(s,s) \mid s \in State(V)\} \cup \\ & & \{(s,r) \mid (s,w) \in \rho(\alpha) \wedge (w,r) \in \rho(\alpha)^*\} \end{array}$$

Sequential composition is defined using the composition of relations. The union is used for nondeterministic choice. The iteration is defined as the choice of executing a program zero or more times.

6.1.2 First-order dynamic logic

To reason about program correctness, first order dynamic logic relies on the following syntax for logical formulas.

Definition 17 (Formulas). *The set of formulas $Fml(V)$ is defined by the following grammar:*

$$\begin{array}{lcl} \varphi,\psi & ::= & p(t_1,\cdots,t_n) \mid \top \mid \neg\varphi \mid \varphi\wedge\psi \mid \forall x.\varphi \mid \langle\alpha\rangle\varphi \mid \\ & & \qquad\qquad \bot \mid \varphi\vee\psi \mid \varphi\to\psi \mid \exists x.\varphi \mid [\alpha]\varphi \end{array}$$

for $p \in \Sigma_n$ a predicate, $t_1,\cdots t_n \in Trm(V)$, $x \in V$ and $\alpha \in Prog(V)$.

The second line reports formulas that can be derived using the operators in the first line. The notion of satisfaction for logic formulas is straightforward for first order operators. The program possibility $\langle\alpha\rangle\varphi$ states that *it is possible that after executing program α, φ is true*. The necessity operator is dual to the possibility. It is defined as $[\alpha]\varphi \triangleq \neg\langle\alpha\rangle\neg\varphi$ stating that *it is necessary that after executing program α, φ is true*.

Definition 18 (Formula Validity)**.** *The satisfiability of a formula φ in a state $s \in State(V)$ of a computational domain $\mathcal{D} = (D, I)$ is defined by:*

s	$\models$	$p(t_1, \ldots, t_n)$	*iff*	$p^I(val(s, t_1), \ldots, val(s, t_n))$
s	$\models$	$\top$		*for all* $s \in S$
s	$\models$	$\neg\varphi$	*iff*	*not* $s \models \varphi$
s	$\models$	$\varphi \wedge \psi$	*iff*	$s \models \varphi$ *and* $s \models \psi$
s	$\models$	$\forall x.\varphi$	*iff*	$s[x \mapsto d] \models \varphi$ *for all* $d \in D$
s	$\models$	$\langle\alpha\rangle\varphi$	*iff*	*there is a state* r *such that* $(s, r) \in \rho(\alpha)$ *and* $r \models \varphi$

A formula is valid in a domain $\mathcal{D}$ if it is satisfiable in all states over $\mathcal{D}$, it is valid if it is valid in all domains.

A valid formula for Example 13 is $stock > 0 \rightarrow [acceptOrder]stock \geq 0$ (assuming that the computational domain are the natural numbers).

The following equivalences are taken from [Har79] as properties for dynamic logic. We will revisit them at the end of the next chapter and regard their validity considering our extension of the logic.

Proposition 5 (cf. [Har79])**.** *The following are valid formulas of first order dynamic logic.*

$\langle\alpha\rangle(\varphi \vee \psi)$	$\leftrightarrow$	$\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi$	$[\alpha](\varphi \wedge \psi)$	$\leftrightarrow$	$[\alpha]\varphi \wedge [\alpha]\psi$
$\langle\alpha\rangle\varphi \wedge [\alpha]\psi$	$\rightarrow$	$\langle\alpha\rangle(\varphi \wedge \psi)$	$[\alpha](\varphi \rightarrow \psi)$	$\rightarrow$	$([\alpha]\varphi \rightarrow [\alpha]\psi)$
$\langle\alpha\rangle(\varphi \wedge \psi)$	$\rightarrow$	$\langle\alpha\rangle\varphi \wedge \langle\alpha\rangle\psi$	$[\alpha]\varphi \vee [\alpha]\psi$	$\rightarrow$	$[\alpha](\varphi \vee \psi)$
$\langle\alpha + \beta\rangle\varphi$	$\leftrightarrow$	$\langle\alpha\rangle\varphi \vee \langle\beta\rangle\varphi$	$[\alpha + \beta]\varphi$	$\leftrightarrow$	$[\alpha]\varphi \wedge [\beta]\varphi$
$\langle\alpha\,;\beta\rangle\varphi$	$\leftrightarrow$	$\langle\alpha\rangle\langle\beta\rangle\varphi$	$[\alpha\,;\beta]\varphi$	$\leftrightarrow$	$[\alpha][\beta]\varphi$
$\langle\alpha^*\rangle\varphi$	$\leftrightarrow$	$\varphi \vee \langle\alpha\,;\alpha^*\rangle\varphi$	$[\alpha^*]\varphi$	$\leftrightarrow$	$\varphi \wedge [\alpha\,;\alpha^*]\varphi$
$\langle\phi?\rangle\psi$	$\leftrightarrow$	$\phi \wedge \psi$	$[\phi?]\psi$	$\leftrightarrow$	$\phi \rightarrow \psi$

Below we show how in previous approaches dynamic logic is either extended with deontic formalisms or concurrency. Most of these approaches use propositional dynamic logic [HKT00]. It is more abstract than first order dynamic logic. The interpretation of basic activities is an abstract relation on states, often Kripke frames are used. Thus there is no need for the valuation of terms and the computational domain. However for our purpose we consider the use of variables more suitable and realistic.

6.1.3 Deontic Formalisms for Error Handling

In this section we overview previous approaches that combine dynamic logic with deontic logic [VW51] by introducing operators for permission, obligation and prohibition. While the original deontic logic reasons on predicates, in combination with dynamic logic these operators are applied to actions.

Meyer [Mey88] proposed the use of a violation condition V that describes an undesirable situation, so that the violation condition corresponds to validity of proposition V. The prohibition operator is defined such that $s \models F\alpha$ iff $s \models [\alpha]V$, *i.e.*, it is forbidden to do α in s iff all executions of α terminate in a violation. Obligation and permission are defined based on prohibition. The main problem with Meyer's work is that dependency between permission and prohibition raises paradoxes. Paradoxes of deontic logics are valid logical formulas that go against common sense, *e.g.*, Ross' paradox states if a letter ought *to be sent*, then a letter ought *to be sent or burnt*.

While Meyer focused on permission of states, Meyden [vdM96] defined permission on the possible executions of an action. He extended models for dynamic logic with a relation P on states. An execution of an action α is permitted if every (internal) state transition of α is in P. This implies that if an execution of an action is permitted also each of its subactions must be permitted. This avoids, for example, Ross' paradox. Meyden's definition of violation is however not very different from Meyer's. As shown in [Bro03] there is a correspondence between the two

definitions.

In [BWM01] Broersen *et al.* define permission of an action as a proposition over states and actions. Each model contains a valuation function that maps permission propositions over atomic actions to sets of states. Contrary to the previous approaches permission is in fact based on the action itself. For compound actions permission is defined in terms of possible traces and its subactions. Broersen extends this approach in [Bro03] including also concurrency.

Castro and Maibaum [CM09] refine Broersen's approach. Their definition of violation is very similar, however actions can be combined differently. While previous approaches focused on free choice and sequence for the combination of atomic actions, the authors define the domain of actions as an atomic boolean algebra. Actions can be combined using choice, intersection (*i.e.* concurrency) and a locally restricted form of complement. This allows them to show not only that their logic is sound and complete, but moreover it is decidable and compact.

6.1.4 Concurrency

There are only a few approaches adding concurrency to dynamic logic. The first, concurrent dynamic logic [Pel87], interprets programs as a collection of reachability pairs, such that for each initial state it assigns a set of final states. In case of parallel composition the sets of final states are joined, while for choice the reachability pairs are joined. In this approach the formula $\langle\alpha\rangle\varphi$ holds in states s such that a reachability pair (s, U) for the interpretation of α exists and each state $s' \in U$ satisfies φ. In particular, the axiom $\langle\alpha \cap \beta\rangle\varphi \leftrightarrow \langle\alpha\rangle\varphi \wedge \langle\beta\rangle\varphi$ is valid, *i.e.*, actions are independent of each other.

For Broersen [Bro03] this is an undesirable property. He considers only true concurrency, *i.e.* executing actions in parallel has a different effect than interleaving them. The interpretation uses the intersection for concurrency. Moreover he considers an open concurrency interpretation, which is not applicable to first order dynamic logic (that follows a closed action interpretation).

In [BS08] the authors define a dynamic logic for CCS where the interpretation of concurrency is based on the labelled transition system of CCS. Thus concurrency is either interpreted as interleaving or the so-called handshake of CCS. This is to our knowledge the first article applying dynamic logic to a process calculus.

As we have seen, concurrency in dynamic logic is often interpreted as a simultaneous execution. This interpretation is not suited for the kind of systems we want to model, where concurrent programs describe activities that can be executed concurrently, or even in parallel, but do not have to happen simultaneously. A possible approach would be to only allow parallel composition of independent processes, *i.e.*, processes that do not interfere with each other. This requirement is quite strong and excludes most long running transactional systems. In the area of transactional concurrency control, extensive work has been done on the correctness criterion for the execution of parallel transactions. Proposed criteria include, linearizability [HW90], serializability [Pap79], etc. These criteria are more realistic, since they allow some interference between concurrent transactions. Therefore, for our interpretation of concurrency we use a notion of serializability (less restrictive than linearizability), stating that: the only acceptable interleaved executions of activities from different transactions are those that are equivalent to some sequential execution. Serializability is presented formally in Definitions 28 and 35.

6.2 Concurrent programs

We will consider an extension of first-order dynamic logic. We keep the definitions for the term algebra and variables from Section 6.1. Our definition of basic activities is extended to take into account a validity formula.

Definition 19 (Basic Activities)**.** *A basic activity $a \in Act(V)$ is a multiple assignment together with a quantifier and program-free formula $E(a) \in Fml(V)$ that specifies the conditions under which activity a leads to an error state.*

Formula $E(a)$ can be seen as expressing some precondition on activity a: if formula $E(a)$ holds on state s, executing a will cause an error. We

exploit $E(a)$ to classify state transitions as failed or successful, depending on whether the error condition holds or not on a given state. We could use instead for each activity a an explicit set of error transitions or error states, but those sets (either of error transitions or states) can be derived from formula $E(a)$. Another point worth discussing is the evaluation of $E(a)$ as before or after the execution of the activity a. Evaluating $E(a)$ in advance ensures erroneous state transitions do not occur, leaving the system in a correct state (the last correct state before the error). Whereas moving the evaluation of $E(a)$ after the execution of a would cause the state transition to occur in order to determine if $E(a)$ holds. Note that the empty assignment *skip* is always successful.

Consider once more activity $acceptOrder$ from Example 13. A possible error condition could be $E(acceptOrder) \triangleq stock \leq 0$, that checks if there is any item available in stock.

The definition of concurrent programs uses standard dynamic operators, including parallel composition and a test operator (over program- and quantifier-free formulas).

Definition 20 (Concurrent Programs)**.** *The set $Prog(V)$ of programs is defined by the following grammar:*

$$\alpha, \beta \ ::= \ a \ \mid \ \phi? \ \mid \ \alpha \,;\, \beta \ \mid \ \alpha \ + \ \beta \ \mid \ \alpha \parallel \beta \ \mid \ \alpha^*$$

Let the computational domain be as in Definition 13, as well as the valuation of terms as in Definition 14. The interpretation of basic activities (and thus programs) differs from the usual interpretation of dynamic logic. First we distinguish between activities that succeed or fail. Second we use traces instead of the relation on states. This is due to the combination of possible failing executions and nondeterminism. We will explain this further when introducing the interpretation of concurrent programs.

Definition 21 (Steps)**.** *Let $\ell \in \{a, -a, \phi?\}$, for $a \in Act(V)$ a multiple assignment $x_1, \cdots, x_n := t_1, \cdots, t_n$ and $\phi \in Pred(V)$ a program- and quantifier-free formula. We call a* step $[\![\ell]\!] \subseteq State(V) \times (Act(V) \cup Pred(V)) \times State(V)$ *the set of triples defined by:*

$$
\begin{array}{lcll}
[\![a]\!] & \triangleq & \{s\, a\, s' & \mid s \models \neg E(a) \wedge \\
 & & & \quad s' = s[x_1 \mapsto val(s,t_1), \cdots, x_n \mapsto val(s,t_n)]\} \\
[\![-a]\!] & \triangleq & \{s\, a\, s & \mid s \models E(a)\} \\
[\![\phi?]\!] & \triangleq & \{s\, \phi?\, s & \mid s \models \phi\}.
\end{array}
$$

We say that in $[\![a]\!]$ the a stands for a good or successful behaviour of the activity while $-a$ in $[\![-a]\!]$ represents failure of the activity.

Example 14 (e-Store)**.** *Consider activity $acceptOrder$ from Example 13 and its error formula $E(acceptOrder)$. In this setting, we have that*

$$
\begin{array}{lcl}
[\![acceptOrder]\!] & \triangleq & \{s\ acceptOrder\ s' \mid \\
 & & \quad s' = s[stock \mapsto s(stock) - 1, card \mapsto unknown] \\
 & & \quad \wedge\ s \models stock > 0\} \\
[\![-acceptOrder]\!] & \triangleq & \{s\ acceptOrder\ s \mid s \models stock \leq 0\}
\end{array}
$$

Definition 22 (Traces)**.** *A trace τ is defined as a sequence of sets of triples $[\![\ell]\!]$.*

We will use $[\![]\!]$ for the singleton containing the empty (but defined) trace; when combined with another trace it acts like the identity. Moreover we use the notation $[\![\ell.\tau]\!] = [\![\ell]\!][\![\tau]\!]$ for traces with length ≥ 1. Note that if $[\![\ell]\!] = \emptyset$ the trace is not defined. When composing traces there is no restriction whether adjoining states have to match or not. The system is in general considered to be open, *i.e.*, even within one trace between two actions there might be something happening in parallel changing the state. When we build the closure of the system traces that do not match are discarded.

A closed trace is a trace where adjoining states match. We define a predicate $closed$ on traces such that $closed(s\, \ell\, s') = \top$ and $closed(s\, \ell\, s'.\tau) = closed(\tau) \wedge (s' = first(\tau))$. For closed traces we can define functions $first$ and $last$ that return the first and the last state of the trace. Note that one open trace usually corresponds to a set of closed traces, but with a fixed starting state the execution is deterministic.

Example 15. *Consider a small example with two basic actions $a \triangleq x := x - 1$ and $b \triangleq y := y - 1$ where $E(a) \triangleq x < 1$ and $E(b) \triangleq y < 1$. The open trace $[\![a.b]\!]$ stands for a multitude of traces including those where the final state*

of $[\![a]\!]$ and the first of $[\![b]\!]$ do not match. Take for example a triple $s\,a\,s' \in [\![a]\!]$ with $y = 0$ in both s and s' (as a does not change y). Then there is no triple in $[\![b]\!]$ that matches s' as it satisfies b's error formula. The closure will rule out such traces. If we take now a fixed initial state s where $x = 2$ and $y = 5$, the computation is deterministic and we reach a state s' where $x = 1$ and $y = 4$.

Definition 23 (Interpretation of Basic Activities)**.** *The valuation ρ of an activity $a \in Act(V)$ is defined by:*

$$\rho(a) \quad \triangleq \quad [\![a]\!] \cup [\![-a]\!]$$

With this semantic model an activity a may have "good" (committed) or "bad" (failed) traces, depending on whether the initial state does not satisfy $E(a)$ or it does. As it is clear from the interpretation of basic activities, if the error condition holds, then it forbids the execution of an activity. Therefore failed transitions do not cause a state change.

As we mentioned for basic activities, we use traces instead of a state relation for the interpretation of programs. To motivate this decision, consider the behaviour of a compensable program. If it is successful, the complete program will be executed. If it fails, the program is aborted and the installed compensations will be executed. Thus we need to distinguish between successful and failing executions. Hence we need to extend the error formula E. But extending E to programs does not suffice as programs introduce nondeterminism, *i.e.*, a program with choice may both succeed and fail in the same state. Using E for programs would however only tell us that the program might fail, not which execution actually does fail. For a trace we can state whether this execution fails or not.

Definition 24 (Error Formulas of Traces)**.** *We lift error formulas from activities to traces τ by letting $E(\tau)$ be inductively defined as:*

$$\begin{array}{llll} E([\![]\!]) & \triangleq & \bot & E([\![a]\!]\tau) \quad \triangleq \quad E(\tau) \\ E([\![-a]\!]\tau) & \triangleq & \top & E([\![\phi?]\!]\tau) \quad \triangleq \quad E(\tau) \end{array}$$

Take the two actions a and b from Example 15. We have that $E([\![a.b]\!]) = E([\![b]\!])$ is false while $E([\![a.-b]\!]) = E([\![-b]\!])$ is true.

We exploit error formulas for defining the sequential composition $\tau_\alpha \circ \tau_\beta$ of two traces. If the first trace raises an error ($E(\tau_\alpha)$ is true), the

execution is aborted and thus not combined with the second trace. If the first trace succeeds ($E(\tau_\alpha)$ is false) sequential composition is defined as usual, *i.e.* we append the second trace to the first trace.

$$\tau_\alpha \circ \tau_\beta \triangleq \begin{cases} \tau_\alpha & \text{if } E(\tau_\alpha) \\ \tau_\alpha\ \tau_\beta & \text{if } \neg E(\tau_\alpha) \end{cases}$$

Abusing the notation we use the same symbol $\circ$ to compose sets of traces. Consider once more the two activities a and b from Example 15. The sequential composition $[\![a]\!] \circ [\![b]\!]$ is defined as $[\![a]\!][\![b]\!] = [\![a.b]\!]$. On the other hand the sequential composition $[\![-a]\!] \circ [\![b]\!]$ is defined as $[\![-a]\!]$.

Next, to build the trace for parallel composition of two basic programs we would consider the interleaving of any combination of traces:

$$\begin{array}{rcl@{\qquad}rcl} [\![]\!] \parallel \tau_2 & \triangleq & \{\tau_2\} & [\![\ell_1]\!]\tau_1 \parallel [\![\ell_2]\!]\tau_2 & \triangleq & \{[\![\ell_1]\!]\tau \mid \tau \in (\tau_1 \parallel [\![\ell_2]\!]\tau_2)\} \\ \tau_1 \parallel [\![]\!] & \triangleq & \{\tau_1\} & & \cup & \{[\![\ell_2]\!]\tau \mid \tau \in ([\![\ell_1]\!]\tau_1 \parallel \tau_2)\} \end{array}$$

We use once more activities a and b from Example 15. Then $[\![a]\!] \parallel [\![b]\!]$ is defined as both $[\![a.b]\!]$ and $[\![b.a]\!]$. Note that we ignore whether an action is successful or not. Thus $[\![-a]\!] \parallel [\![b]\!]$ is defined as $[\![-a.b]\!]$ and $[\![b. - a]\!]$.

Now we can define the interpretation of concurrent programs:

Definition 25 (Interpretation of Concurrent Programs)**.** *We extend the interpretation ρ from basic activities to concurrent programs in the following manner:*

$$\begin{array}{lcl} \rho(\phi?) & \triangleq & [\![\phi?]\!] \\ \rho(\alpha\,;\beta) & \triangleq & \{\tau_\alpha \circ \tau_\beta \mid \tau_\alpha \in \rho(\alpha) \wedge \tau_\beta \in \rho(\beta)\} \\ \rho(\alpha\ +\ \beta) & \triangleq & \rho(\alpha) \cup \rho(\beta) \\ \rho(\alpha \parallel \beta) & \triangleq & \{\tau \mid \tau_\alpha \in \rho(\alpha) \wedge \tau_\beta \in \rho(\beta) \wedge \tau \in \tau_\alpha \parallel \tau_\beta\} \\ \rho(\alpha^*) & \triangleq & [\![]\!] \cup \rho(\alpha) \circ \rho(\alpha^*) \end{array}$$

Test $\phi?$ is interpreted as the identity trace for the states that satisfy formula ϕ. The interpretation of sequential programs is the sequential composition of traces. Failed transitions are preserved in the resulting set as executions of α that have reached an erroneous state and cannot evolve. Choice is interpreted as the union of trace sets of programs α and

β. The interpretation of parallel composition is the set of any possible interleaving of the traces for both branches. Iteration is defined recursively (by taking the least fixpoint of the equation).

When defining the notion of (strong) serializability (see Definitions 28 and 29) we will rely on the associativity and commutativity of parallel composition, which is stated by the following lemma.

Lemma 1. *For any programs α_1, α_2 and α_3 we have:*

1. $\rho(\alpha_1 ; (\alpha_2 ; \alpha_3)) = \rho((\alpha_1 ; \alpha_2) ; \alpha_3)$*;*
2. $\rho(\alpha_1 + (\alpha_2 + \alpha_3)) = \rho((\alpha_1 + \alpha_2) + \alpha_3)$ *and* $\rho(\alpha_1 + \alpha_2) = \rho(\alpha_2 + \alpha_1)$*;*
3. $\rho(\alpha_1 \parallel (\alpha_2 \parallel \alpha_3)) = \rho((\alpha_1 \parallel \alpha_2) \parallel \alpha_3)$ *and* $\rho(\alpha_1 \parallel \alpha_2) = \rho(\alpha_2 \parallel \alpha_1)$.

Proof. For each equation over the sets of traces, the proof goes by showing separately the double inclusion of one set into the other. In all cases—sequential, choice and parallel composition—given any trace of one set (e.g., $\tau \in \rho(\alpha_1 + \alpha_2)$), it is shown that the same trace belong to the other set (e.g., that $\tau \in \rho(\alpha_2 + \alpha_1)$) by relying on the properties of the corresponding operator at the level of traces (e.g., set union). □

To build the closure of the system, *i.e.*, a program α, we define the set of all closed traces for α such that $closure(\alpha) \triangleq \{\tau \mid \tau \in \rho(\alpha) \wedge closed(\tau)\}$.

Next we show in an example the application of Definition 25.

Example 16 (e-Store)**.** *Take program $eStore$ defined as in Figure 63. To keep the notation compact, we abbreviate activities using initials, e.g., we write* aO *for $acceptOrder$,* aC *for $acceptCard$, etc. This program describes a simple online shop and it starts with an activity that removes from the stock the ordered items. Since for most orders the credit cards are not rejected, and to decrease the delivery time, the client's card processing and courier booking can be done in parallel. In this example, the activities running in parallel may interfere with each other as $bookCourier$ will fail once the card is rejected. As the order of the execution for the parallel composition is not fixed after rejecting the credit card we issue a throw (defined as the empty assignment that always fails). In the interpretation of program $eStore$ we can first distinguish the traces where $acceptOrder$ succeeds and where it fails. In the latter case no other action is executable. In the successful case the parallel composition is executed where both branches may succeed or fail and we include any possible interleaving. Note that the traces $[\![-\mathsf{aC}]\!]$, $[\![-\mathsf{rC}]\!]$ and $[\![throw]\!]$ are not defined, as their condition is not satisfied by*

$$
\begin{array}{lcl}
eStore & \triangleq & acceptOrder\,; \\
 & & ((acceptCard \,+\, rejectCard\,; throw) \parallel bookCourier)
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathsf{aO} & \triangleq & stock, card := & E(\mathsf{aO}) & \triangleq & stock \leq 0 \\
 & & stock - 1, unkown & & & \\
\mathsf{aC} & \triangleq & card := accepted & E(\mathsf{aC}) & \triangleq & false \\
\mathsf{rC} & \triangleq & card := rejected & E(\mathsf{rC}) & \triangleq & false \\
\mathsf{bC} & \triangleq & courier := booked & E(\mathsf{bC}) & \triangleq & card = rejected \\
throw & \triangleq & skip & E(throw) & \triangleq & true
\end{array}
$$

$$
\begin{array}{l}
\rho(\mathsf{aO}\,; ((\mathsf{aC} \,+\, \mathsf{rC}\,; throw) \parallel \mathsf{bC})) = \\
\quad [\![\mathsf{aO.aC.bC}]\!] \cup [\![\mathsf{aO.bC.aC}]\!] \cup \\
\quad [\![\mathsf{aO.aC.} - \mathsf{bC}]\!] \cup [\![\mathsf{aO.} - \mathsf{bC.aC}]\!] \cup \\
\quad [\![\mathsf{aO.rC.} - throw.\mathsf{bC}]\!] \cup [\![\mathsf{aO.bC.rC.} - throw]\!] \cup \\
\quad [\![\mathsf{aO.rC.bC.} - throw]\!] \cup [\![\mathsf{aO.rC.} - throw.\, - \mathsf{bC}]\!] \cup \\
\quad [\![\mathsf{aO.} - \mathsf{bC.rC.} - throw]\!] \cup [\![\mathsf{aO.rC.} - \mathsf{bC.} - throw]\!] \cup \\
\quad [\![-\mathsf{aO}]\!]
\end{array}
$$

Figure 63: *eStore* example.

any possible state. Building the closure for these traces we can rule out some possibilities, namely $[\![\mathsf{aO.aC.} - \mathsf{bC}]\!]$, $[\![\mathsf{aO.} - \mathsf{bC.aC}]\!]$, $[\![\mathsf{aO.} - \mathsf{bC.rC.} - throw]\!]$, $[\![\mathsf{aO.rC.} - throw.\mathsf{bC}]\!]$ *and* $[\![\mathsf{aO.rC.bC.} - throw]\!]$ *would be excluded. In the first three cases the error formula of action* bC *is not fulfilled, thus it should not fail. In the other two cases the card was rejected before the booking of the courier, thus* $E(\mathsf{bC})$ *is fulfilled and the action should fail. A possible closed trace is:*

$$
\begin{array}{l}
closed(\; s\,\mathsf{aO}\,s' \,.\, s'\mathsf{aC}\,s'' \,.\, s''\mathsf{bC}\,s''' \;\mid \\
\quad s' = s[stock \mapsto s(stock) - 1, card \mapsto unkown] \wedge \\
\quad s'' = s'[card \mapsto accepted] \wedge \\
\quad s''' = s''[courier \mapsto booked] \wedge \\
\quad s \models stock > 0 \wedge s'' \models \neg card = rejected \;)
\end{array}
$$

For formulas we include two modal operators related to program success, where success of a program is interpreted as not reaching an erroneous state. The modal operator success $\mathsf{S}(\alpha)$ states that *every way of executing* α *is successful*, so program α must never reach an erroneous state. The modal operator weak success $\mathsf{S_W}(\alpha)$ states that *some way of execut-*

ing α is successful. The failure modal operator $\mathsf{F}(\alpha)$ is a derived operator, $\mathsf{F}(\alpha) \triangleq \neg\mathsf{S_W}(\alpha)$, and states that *every way of executing α fails*. Note that both weak success and program possibility ensure program termination, while success and program necessity do not.

Definition 26 (Formulas)**.** *The set of formulas $Fml(V)$ is defined by the grammar:*

$$\begin{array}{rcl} \varphi,\psi & ::= & p(t_1,\cdots,t_n) \mid \top \mid \neg\varphi \mid \varphi\wedge\psi \mid \forall x.\varphi \mid \langle\alpha\rangle\varphi \mid \\ & & \qquad\qquad \mathsf{S}(\alpha) \mid \mathsf{S_W}(\alpha) \mid \\ & & \qquad\qquad \bot \mid \varphi\vee\psi \mid \varphi\to\psi \mid \exists x.\varphi \mid [\alpha]\varphi \mid \\ & & \qquad\qquad\qquad\quad \mathsf{F}(\alpha) \end{array}$$

for $p \in \Sigma_n$ a predicate, $t_1,\cdots t_n \in Trm(V)$, $x \in V$ and $\alpha \in Prog(V)$.

Definition 27 (Formula Validity)**.** *The validity of a formula φ in a state $s \in State(V)$ of a computational domain $\mathcal{D} = (D, I)$ is defined by:*

$$\begin{array}{lcl} s \models p(t_1,\ldots,t_n) & \textit{iff} & p^I(val(s,t_1),\ldots,val(s,t_n)) \\ s \models \top & & \textit{for all } s \in S \\ s \models \neg\varphi & \textit{iff} & \textit{not } s \models \varphi \\ s \models \varphi\wedge\psi & \textit{iff} & s \models \varphi \textit{ and } s \models \psi \\ s \models \forall x.\varphi & \textit{iff} & s[x \mapsto d] \models \varphi \textit{ for all } d \in D \\ s \models \langle\alpha\rangle\varphi & \textit{iff} & \exists\,\tau \in closure(\alpha) \textit{ such that } first(\tau) = s \\ & & \textit{and } last(\tau) \models \varphi \\ s \models \mathsf{S}(\alpha) & \textit{iff} & \textit{for all } \tau \in closure(\alpha), \\ & & first(\tau) = s \textit{ implies } \neg E(\tau) \\ s \models \mathsf{S_W}(\alpha) & \textit{iff} & \exists\,\tau \in closure(\alpha) \textit{ such that} \\ & & first(\tau) = s \textit{ and } \neg E(\tau) \\ s \models \mathsf{F}(\alpha) & \textit{iff} & s \models \neg\mathsf{S_W}(\alpha) \end{array}$$

The new modal operators for success and failure are defined according to the description given above.

Example 17 (e-Store)**.** *Considering the example of Figure 63 a possible formula would be $\mathsf{F}(acceptOrder)$, that is only satisfiable in some states. However, the following are valid formulas for any state:*

$$stock \leq 0 \to \mathsf{F}(acceptOrder)$$
$$stock > 0 \to \mathsf{S_W}(acceptOrder)$$
$$[acceptOrder]card = accepted \to courier = booked$$

The first formula states that if $stock \leq 0$, *program* $acceptOrder$ *always fails. While the second formula says that if* $stock > 0$, *there are some successful executions of* $acceptOrder$. *The last formula describes a condition that is always true after the execution of program* $acceptOrder$.

As discussed in Section 6.1, we want an interpretation of concurrency where concurrent activities do not have to happen simultaneously. For example, packing the items in a client's order and booking a courier to deliver that same order are independent activities that can be run in parallel. If, contrary to the example just mentioned, parallel activities are not independent, then concurrency becomes more complex: as activities may interfere with each other, this may lead to unexpected results. In this work parallel composition is not restricted to independent programs (such that the set of variables updated and read by those programs is disjoint). However, without any restriction the logic is too liberal to prove any good properties. To address this issue, we use a notion of serializability (see Definition 28) to determine the correctness of concurrent programs. Every trace of serializable concurrent composition of programs can be matched to a trace of a serial execution of those same programs, such that both traces have the same initial and final state.

Notice, that because programs α and β may both fail we cannot require that concurrent traces are equivalent to some sequential execution of α and β. Therefore, in our setting, for each concurrent trace of a serializable program $\alpha \parallel \beta$ there are traces of α and β, such that the composition of those traces is closed and matches the initial and final state of the concurrent trace.

We will use the abbreviation $\tau \bowtie \tau'$ for traces τ and τ' denoting

$$\tau \bowtie \tau' \triangleq closed(\tau) \wedge closed(\tau') \wedge first(\tau) = first(\tau') \wedge last(\tau) = last(\tau')$$

Definition 28 (Serializable Concurrent Programs)**.** *A set of* n *concurrent programs* $\alpha_1, ..., \alpha_n$ *forms a* serializable set *if for any trace* $\tau \in closure(\alpha_1 \parallel \cdots \parallel \alpha_n)$ *there exist* $\nu_1 \in \rho(\alpha_1), ..., \nu_n \in \rho(\alpha_n)$ *and a permutation* $\iota : [1, n] \to [1, n]$ *such that* $\tau \bowtie (\nu_{\iota(1)} \cdots \nu_{\iota(n)})$.

A concurrent program α *is* serializable *if all of its subterms of the form* $\alpha_1 \parallel \cdots \parallel \alpha_n$ *we have that* $\alpha_1, ..., \alpha_n$ *form a serializable set.*

In transactional concurrency control serializability implicitly imposes that the order of execution of activities within a transaction must be preserved. The above serializability definition does not impose any relation on the order of activities for the parallel and serial execution. To take this ordering into consideration we define a stronger notion of serializability where concurrent execution must maintain the order of the serial execution. For that we use a projection function $pr_{\alpha}(\tau)$ that extracts from trace τ the sequence of activities that belong to program α. This function is used to ensure that each trace of the concurrent process $\alpha \parallel \beta$ follows an order of execution of some individual process.

Definition 29 (Strong Serializable Concurrent Programs)**.** *A set of n concurrent programs $\alpha_1, ..., \alpha_n$ forms a* strong serializable *set if for every trace $\tau \in closure(\alpha_1 \parallel \cdots \parallel \alpha_n)$ there exist $\nu_1 \in \rho(\alpha_1), ..., \nu_n \in \rho(\alpha_n)$ and a permutation $\iota : [1, n] \to [1, n]$ such that $pr_{\alpha_i}(\tau) = \nu_i$ for all $i \in [1, n]$, and $\tau \bowtie (\nu_{\iota(1)} \cdots \nu_{\iota(n)})$.*

A concurrent program α is strong serializable *if all of its subterms of the form $\alpha_1 \parallel \cdots \parallel \alpha_n$ we have that $\alpha_1, ..., \alpha_n$ form a strong serializable set.*

Note that both serializability notions are defined as implications, because logical equivalence cannot be ensured in the general case: the fact that activities may be executed independently does not ensure, in the presence of interference, that those activities may be executed concurrently.

We revisit the formulas for first order dynamic logic presented in Proposition 5. Most are still valid, but, as the interpretation for programs has changed, some are not. In fact, the following modal formulas over sequential programs are not valid in our interpretation of programs:

$$\langle \alpha \,;\, \beta \rangle \varphi \quad \leftrightarrow \quad \langle \alpha \rangle \langle \beta \rangle \varphi \qquad [\alpha \,;\, \beta] \varphi \quad \leftrightarrow \quad [\alpha][\beta] \varphi$$

The reason is that sequential composition will abort in the occurrence of a failure. Therefore, it is no longer possible to reason compositionally about these formulas for sequential programs. Note that we cannot state any properties regarding the new operators and sequential composition. A program α may change the success or failure of any following program β regardless of its success before the execution of α. Consider for instance

activity a from Example 15 and a state s where $x = 1$. Then it holds that $s \models S(a)$, but not $s \models S(a\,;a)$.

We show some useful logical equivalences involving the novel modal operators. We are particularly considering their interplay with parallel composition.

Proposition 6. *Let α, β be two serializable programs. The following are valid formulas in the presented dynamic logic:*

$$
\begin{array}{lcl}
\langle \alpha \parallel \beta \rangle \varphi & \leftrightarrow & \langle \alpha \rangle \langle \beta \rangle \varphi \vee \langle \beta \rangle \langle \alpha \rangle \varphi \\
[\alpha \parallel \beta] \varphi & \leftrightarrow & [\alpha][\beta]\varphi \wedge [\beta][\alpha]\varphi \\
\mathsf{S}(\alpha + \beta) & \leftrightarrow & \mathsf{S}(\alpha) \wedge \mathsf{S}(\beta) \\
\mathsf{S}(\alpha^*) & \leftrightarrow & \mathit{true} \wedge \mathsf{S}(\alpha; \alpha^*) \\
\mathsf{S}(\alpha \parallel \beta) & \rightarrow & \mathsf{S}(\alpha\,;\beta) \wedge \mathsf{S}(\beta\,;\alpha) \\
\mathsf{S_W}(\alpha + \beta) & \leftrightarrow & \mathsf{S_W}(\alpha) \vee \mathsf{S_W}(\beta) \\
\mathsf{S_W}(\alpha^*) & \leftrightarrow & \mathit{true} \vee \mathsf{S_W}(\alpha; \alpha^*) \\
\mathsf{S_W}(\alpha \parallel \beta) & \rightarrow & \mathsf{S_W}(\alpha\,;\beta) \vee \mathsf{S_W}(\beta\,;\alpha) \\
\mathsf{S_W}(\alpha \parallel \beta) & \rightarrow & \mathsf{S_W}(\alpha) \wedge \mathsf{S_W}(\beta) \\
\mathsf{F}(\alpha + \beta) & \leftrightarrow & \mathsf{F}(\alpha) \wedge \mathsf{F}(\beta) \\
\mathsf{F}(\alpha^*) & \leftrightarrow & \mathit{false} \wedge \mathsf{F}(\alpha\,;\alpha^*) \\
\mathsf{F}(\alpha \parallel \beta) & \rightarrow & \mathsf{F}(\alpha\,;\beta) \wedge \mathsf{F}(\beta\,;\alpha) \\
\mathsf{F}(\alpha \parallel \beta) & \rightarrow & \mathsf{F}(\alpha) \vee \mathsf{F}(\beta)
\end{array}
$$

Proof. See B.2. □

Proposition 7. *Let α, β be two strong serializable programs. The following are valid formulas in the presented dynamic logic:*

$$\mathsf{S}(\alpha \parallel \beta) \quad \leftrightarrow \quad \mathsf{S}(\alpha) \wedge \mathsf{S}(\beta) \qquad \mathsf{F}(\alpha \parallel \beta) \quad \leftrightarrow \quad \mathsf{F}(\alpha) \vee \mathsf{F}(\beta)$$

Proof. See B.2. □

The above formulas show that, by imposing a strong serializability correctness criteria on concurrent programs, it is possible to reason about these kind of programs in a compositional manner.

We prove in B.2 the remaining formulas for first order dynamic logic presented in Proposition 5 as well as Proposition 6 and give counterexamples for sequential composition used in program possibility and necessity.

$$
\begin{array}{lcl}
\mathit{BuyFlight} & \triangleq & \mathit{initVars}\,;(\{\![\mathit{Reservation}_1]\!\} \parallel \cdots \parallel \{\![\mathit{Reservation}_n]\!\}) \\
\mathit{Reservation}_i & \triangleq & \mathit{reserveFlight}_i \div \mathit{cancelReservation}_i\,; \\
 & & \mathit{payReservation}_i \div \mathit{skip}
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathsf{iV} & \triangleq & \mathit{flight} := \bot & E(\mathsf{iV}) & \triangleq & \mathit{false} \\
\mathsf{rF}_i & \triangleq & \mathit{seats}_i, \mathit{status}_i := \mathit{seats}_i - 1, \mathit{reserved} & E(\mathsf{rF}_i) & \triangleq & \mathit{seats}_i \leq 0 \\
\mathsf{cR}_i & \triangleq & \mathit{seats}_i, \mathit{status}_i := \mathit{seats}_i + 1, \mathit{cancelled} & E(\mathsf{cR}_i) & \triangleq & \mathit{false} \\
\mathsf{pR}_i & \triangleq & \mathit{status}_i, \mathit{flight} := \mathit{payed}, i & E(\mathsf{pR}_i) & \triangleq & \neg \mathit{flight} = \bot
\end{array}
$$

Figure 64: *BuyFlight* example.

6.3 Compensable programs

This section defines compensable programs. While the basic building blocks of concurrent programs are basic activities, in the case of compensable programs the basic building blocks are *compensation pairs*. A compensation pair $a \div \overline{a}$ is composed by two activities, such that if the execution of the first activity is successful, compensation $\overline{a}$ is installed. Otherwise, if activity a fails, no compensation is stored. Compensation pairs can then be composed using similar operators as for concurrent programs. The transaction operator $\{\![\cdot]\!\}$ converts a compensable program into an ordinary one by discarding stored compensations for successful executions and running compensations for failed executions.

Definition 30 (Compensable Programs)**.** *The set $\mathit{Cmp}(V)$ of compensable programs is defined by the following grammar (for $a, \overline{a} \in \mathit{Act}(V)$):*

$$
\begin{array}{lcl}
\alpha & ::= & \ldots \mid \{\![\delta]\!\} \\
\delta, \gamma & ::= & a \div \overline{a} \mid \delta\,;\gamma \mid \delta + \gamma \mid \delta \parallel \gamma \mid \delta^*
\end{array}
$$

Example 18 (Flight Reservation: Speculative Execution)**.** *An example of a compensable program is shown in Figure 64. This example specifies a flight reservation system that launches n concurrent reservation processes. However, only one of those flight reservations will be purchased. This strategy of executing in parallel several processes that fulfil a similar task is called* speculative execution *[USH95]. Each parallel process can be seen as an attempt to complete a*

task. When one attempt succeeds, the other attempts may be abandoned. Then, compensations can be used to cancel the effect of the abandoned attempts. In our example, when one of the processes successfully pays the flight reservation, the remaining processes will cancel the reservations already made.

Next, we describe process $BuyFlight$ in more detail. The first activity called $initVars$ initializes the shared variable $flight$. This variable stores the identification of the flight purchased, so initially is undefined ($flight = \bot$). After initialization, n copies of transaction $\{[Reservation_i]\}$ are executed in parallel. Each copy represents an attempt to book a flight with a different travel agency. Process $Reservation_i$ starts by decreasing the number of seats available by one and setting the status of the reservation to $reserved$. Next, activity $payReservation_i$ will try to buy flight i. If no other flight was already bought, a payment is made thus completing the reservation. Otherwise, activity $payReservation_i$ fails and the compensation is executed, cancelling the reservation already made.

In order to ensure the overall correctness of a compensable program, the backward program of a compensation pair $a \div \overline{a}$ must satisfy a condition: $\overline{a}$ must successfully revert all forward actions of activity a. In the following we do not require that $\overline{a}$ exactly undoes all assignments of a and thus reverts the state to the exact initial state of a. Instead, we require that it performs some compensation actions that lead to a "sufficiently" similar state to the initial one, where a was performed. The way to determine if two states are similar depends on the system under modelling, so we do not enforce any rigid definition. Still, we propose a concrete notion that may characterize a widely applicable criterion of correctness (but other proposals may be valid as well).

The notion we give is parametric w.r.t. a set of (integer) variables whose content we wish to monitor.

Definition 31 (Difference over X)**.** *Let X be a set of integer variables. We call $s\backslash_X s'$ the difference over X between two states, i.e., the set of changes occurred over the variables in X, when moving from s to s'. Formally, we define $(s\backslash_X s')(x) = s'(x) - s(x)$ for any $x \in X$ (and let $(s\backslash_X s')(x) = 0$ otherwise).*

The criterion for evaluating the correctness of a trace in the presence of faults is then to focus on the difference between the initial state and the final one. For example, the difference $s\backslash_X s$ is null, the same holds for any empty trace.

Definition 32 (Correct Compensation Pair)**.** *Let X be a set of integer variables. Activity $\overline{a} \in Act(V)$ is a correct compensation over X of a iff for all traces $s\,a\,s' \in \rho(a)$ with $s \models \neg E(a)$, then for all $t'\,\overline{a}\,t \in \rho(\overline{a})$ we have $t' \models \neg E(\overline{a})$ and $s\backslash_X s' = t\backslash_X t'$.*

Example 19 (Hotel Booking)**.** *The cancellation of a hotel booking may require the payment of a cancellation fee. This can be modelled by a compensation pair $bookHotel \div cancelHotel$, where forward activity $bookHotel$ books a room and sets the amount to be paid, while the compensation $cancelHotel$ cancels the reservation and charges the cancellation fee.*

$$\begin{array}{lcl} bookHotel & \triangleq & rooms, price, fee := rooms - 1, 140\$, 20\$ \\ cancelHotel & \triangleq & rooms, price, fee := rooms + 1, fee, 0\$ \end{array}$$

In this example $cancelHotel$ does not completely revert all of $bookHotel$ actions, and in fact it is likely that room cancellation imposes some fees. However, if we are only interested in the consistency of the overall number of available rooms, we can take $X = \{rooms\}$ and take the difference over X as a measure of correctness. The idea is that a correct compensation should make available all the rooms that were booked but later cancelled.

In Definition 33 below each compensable program is interpreted as a set of pairs of traces $(\tau, \overline{\tau})$, where τ is a trace of the forward program and $\overline{\tau}$ is a trace that compensates the actions of the forward program. Ideally, τ defines one trace of the compensable program in absence of faults, while τ "followed" by $\overline{\tau}$ defines one trace of the compensable program when a fault occurs in τ. However, as we are going to see, there are some subtleties to be taken into account.

Definition 33 (Interpretation of Compensable Programs)**.** *We define the interpretation ρ_c of compensable programs in the following manner:*

$$
\begin{array}{lcl}
\rho_c(a \div \overline{a}) & \triangleq & \{(\tau,\overline{\tau}) \mid \tau \in \rho(a) \wedge \overline{\tau} \in \rho(\overline{a}) \wedge \neg E(\tau)\} \cup \\
 & & \{(\tau,[\![]\!]) \mid \tau \in \rho(a) \wedge E(\tau)\} \\
\rho_c(\delta \,+\, \gamma) & \triangleq & \rho_c(\delta) \cup \rho_c(\gamma) \\
\rho_c(\delta\,;\gamma) & \triangleq & \{(\tau \circ \nu, \overline{\nu} \circ \overline{\tau}) \mid (\tau,\overline{\tau}) \in \rho_c(\delta) \wedge (\nu,\overline{\nu}) \in \rho_c(\gamma) \wedge \neg E(\tau)\} \cup \\
 & & \{(\tau,\overline{\tau}) \mid (\tau,\overline{\tau}) \in \rho_c(\delta) \wedge E(\tau)\} \\
\rho_c(\delta \parallel \gamma) & \triangleq & \{(\tau,\overline{\tau}) \mid (\nu,\overline{\nu}) \in \rho_c(\delta) \wedge (\mu,\overline{\mu}) \in \rho_c(\gamma) \wedge \\
 & & \qquad \tau \in \nu \parallel \mu \;\wedge\; \overline{\tau} \in \overline{\nu} \parallel \overline{\mu}\} \\
\rho_c(\delta^*) & \triangleq & \{([\![]\!],[\![]\!])\} \;\cup\; \rho_c(\delta\,;\delta^*) \\
\rho(\{\!|\delta|\!\}) & \triangleq & \{\tau \mid (\tau,\overline{\tau}) \in \rho_c(\delta) \wedge \neg E(\tau)\} \cup \\
 & & \{cl(\tau) \circ \overline{\tau} \mid (\tau,\overline{\tau}) \in \rho_c(\delta) \wedge E(\tau)\}
\end{array}
$$

Compensation pairs are interpreted as the union of two sets. The first set represents the successful traces of forward activity a, paired with the traces of compensation activity $\overline{a}$. The second set represents the failed traces, paired with the empty trace as its compensation. Sequential composition of compensable programs $\delta\,;\gamma$ is interpreted by two sets, one where the successful termination of δ allows the execution of γ, the other where δ fails and therefore γ cannot be executed. As for the compensations of a sequential program, their traces are composed in the reverse order of their forward programs. Parallel composition interleaves the forward and compensation traces, separately.

The interpretation of a transaction $\{\!|\delta|\!\}$ includes two sets: the first set discards compensable traces for all successful traces; while the second set deals with failed traces by composing each failed trace with the correspondent compensation trace. Notice that for this trace composition to be defined, it is necessary to clear any faulty activities in the failed forward trace. Therefore, in defining the interpretation of a transaction $\{\!|\delta|\!\}$, we exploit a function cl that clears failing activities from a run:

$$
cl([\![]\!]) \;\triangleq\; [\![]\!] \qquad cl([\![\ell]\!]\tau) \;\triangleq\; \begin{cases} [\![E(a)?]\!]\; cl(\tau) & \text{if } \ell = -a \\ [\![\ell]\!]\; cl(\tau) & \text{otherwise} \end{cases}
$$

where $E(a)?$ is the test for the error formula of activity a. It is always defined in s as activity a only aborts if $s \models E(a)$, thus it is in general like a *skip*. In fact, if τ is faulty but successfully compensated by $\overline{\tau}$, then we

$$
\begin{array}{lcl}
\rho_c(Reservation_i) & = & \rho_c(\mathsf{rF}_i \div \mathsf{cR}_i \,;\, \mathsf{pR}_i \div skip) \\
& = & ([\![\mathsf{rF}_i.\mathsf{pR}_i]\!], [\![skip.\mathsf{cR}_i]\!]) \cup \\
& & ([\![\mathsf{rF}_i. - \mathsf{pR}_i]\!], [\![\mathsf{cR}_i]\!]) \cup \\
& & ([\![-\mathsf{rF}_i]\!], [\![]\!]) \\
\\
\rho(\{\![Reservation_i]\!\}) & = & [\![\mathsf{rF}_i.\mathsf{pR}_i]\!] \cup \\
& & [\![\mathsf{rF}_i.E(\mathsf{pR}_i)?.\mathsf{cR}_i]\!] \cup \\
& & [\![E(\mathsf{rF}_i)?]\!]
\end{array}
$$

$$
\begin{array}{lcll}
\rho(BuyFlight) & = & \rho(\mathsf{iV} \,;\, \{\![Reservation_1]\!\} \parallel \{\![Reservation_2]\!\}) \text{ (for } n = 2) & \\
& = & [\![\mathsf{iV}]\!].(& \\
& & \quad [\![\mathsf{rF}_1.\mathsf{pR}_1]\!] \parallel [\![\mathsf{rF}_2.E(\mathsf{pR}_2)?.\mathsf{cR}_2]\!] \cup & \text{(flight 1 purchased)} \\
& & \quad [\![\mathsf{rF}_1.\mathsf{pR}_1]\!] \parallel [\![E(\mathsf{rF}_2)?]\!] \cup & \\
& & \quad [\![\mathsf{rF}_1.E(\mathsf{pR}_1)?.\mathsf{cR}_1]\!] \parallel [\![\mathsf{rF}_2.\mathsf{pR}_2)]\!] \cup & \text{(flight 2 purchased)} \\
& & \quad [\![E(\mathsf{rF}_1)?]\!] \parallel [\![\mathsf{rF}_2.\mathsf{pR}_2]\!] \cup & \\
& & \quad [\![E(\mathsf{rF}_1)?]\!] \parallel [\![E(\mathsf{rF}_2)?]\!] & \text{(no flight available)} \\
& &) &
\end{array}
$$

Figure 65: Interpretation of the $BuyFlight$ process.

want to exhibit an overall non faulty run, so that we cannot just take $\tau \circ \overline{\tau}$ (if $E(\tau)$ then obviously $E(\tau \circ \overline{\tau})$). The following lemma states that cl does not alter the first nor the last state of a trace:

Lemma 2. *For any trace τ: If $\neg E(cl(\tau))$, then $first(\tau) = first(cl(\tau))$ and $last(\tau) = last(cl(\tau))$.*

Example 20 (Flight Reservation)**.** *Figure 65 shows step by step the interpretation of the BuyFlight process from Example 18. First, it presents the interpretation of compensable process $Reservation_i$ where each forward run is paired with a compensation trace that reverts all its successfully terminated activities. Next, the interpretation of transaction $\{\![Reservation_i]\!\}$ can be built and it considers three possible outcomes. In the first case, flight i is purchased, so the compensation for that reservation can be discarded. In the second case, another reservation was previously purchased, so reservation i has to be compensated for. In the third case, reservation i cannot be made because there are no seats available. Lastly, Figure 65 shows a speculative execution of two $Reservation$ transactions. The set of all possible interleaving is quite large, even after discarding traces that are not satisfied by any possible state. An example of a trace to be discarded is $[\![\mathsf{iV}.\mathsf{rF}_2.E(\mathsf{pR}_2)?.\mathsf{cR}_2.\mathsf{rF}_1.\mathsf{pR}_1]\!]$, because the execution*

$$\begin{array}{lcl} \mathit{HotelBooking} & \triangleq & \mathit{bookHotel} \div \mathit{cancelHotel}\,; \\ & & (\mathit{acceptBooking} \div \mathit{skip} \;+\; \mathit{cancelBooking} \div \mathit{skip}\,; \\ & & \;\; \mathit{throw} \div \mathit{skip})) \end{array}$$

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{bH} & \triangleq & \mathit{rooms}, \mathit{status}, \mathit{price}, \mathit{fee} := & E(\mathsf{bH}) & \triangleq & \mathit{rooms} \leq 0 \\ & & \;\; \mathit{rooms} - 1, \mathit{booked}, 140, 20 & & & \\ \mathsf{cH} & \triangleq & \mathit{rooms}, \mathit{price}, \mathit{fee} := \mathit{rooms} + 1, \mathit{fee}, 0 & E(\mathsf{cH}) & \triangleq & \mathit{false} \\ \mathsf{aB} & \triangleq & \mathit{status} := \mathit{confirmed} & E(\mathsf{aB}) & \triangleq & \mathit{false} \\ \mathsf{cB} & \triangleq & \mathit{status} := \mathit{cancelled} & E(\mathsf{cB}) & \triangleq & \mathit{false} \end{array}$$

$$\begin{array}{l} \rho_c(\mathsf{bH} \div \mathsf{cH}\,; (\mathsf{aB} \div \mathit{skip} \;+\; \mathsf{cB} \div \mathit{skip}\,; \mathit{throw} \div \mathit{skip})) = (\llbracket \mathsf{bH}.\mathsf{aB} \rrbracket, \llbracket \mathit{skip}.\mathsf{cH} \rrbracket) \cup \\ \qquad\qquad (\llbracket \mathsf{bH}.\mathsf{cB}.- \mathit{throw} \rrbracket, \llbracket \mathit{skip}.\mathsf{cH} \rrbracket) \cup \\ \qquad\qquad (\llbracket - \mathsf{bH} \rrbracket, [\![]\!]) \end{array}$$

Figure 66: *HotelBooking* example.

of iV *sets the variable flight to* $\bot$*. Therefore,* $E(\mathsf{pR}_2)$ *could not be true (as no other reservation has been purchased at this point in time). An acceptable interleaving would be* $\llbracket \mathsf{iV}.\mathsf{rF}_2.\mathsf{rF}_1.\mathsf{pR}_1.E(\mathsf{pR}_2)?.\mathsf{cR}_2 \rrbracket$ *where process* $\mathit{Reservation}_2$ *manages to make the flight reservation before process* $\mathit{Reservation}_1$*, but is overtaken by* $\mathit{Reservation}_1$ *in the flight payment. Notice that these set of traces are equivalent (same initial and final state) to the sequential execution of processes* $\mathit{Reservation}_1$ *and* $\mathit{Reservation}_2$*. We can conclude that* $\mathit{BuyFlight}$ *is serializable, because any given interleaved trace is equivalent to a sequential trace that starts by executing the process that succeeds in paying the reservation.*

Example 21 (Hotel Booking)**.** *Figure 66 shows another example of a compensable program, that specifies a hotel booking system. In this example the activity* $\mathit{bookHotel}$ *updates several variables: it decreases the number of available rooms, sets the booking status to* booked*, while the price and cancellation fee are set to predefined values. Next, there is a choice between confirming or cancelling the booking. After* $\mathit{cancelBooking}$ *the process is aborted by executing* throw*. The interpretation of* $\mathit{HotelBooking}$ *pairs each forward trace with a compensation trace. For example, the first pair of traces represents a successful execution, where after booking a room the client accepts that reservation. In this case the stored compensation reverts both the acceptance of the booking and the booking itself. Likewise Example 19, the compensation activity* $\mathit{cancelHotel}$ *does not revert completely its main activity: it reverts the room booking, charges the cancellation fee, and it leaves the booking status unchanged.*

$$HotelTransactions \triangleq \{\![HotelBooking_1]\!\} \parallel \{\![HotelBooking_2]\!\}$$

$$\rho(\{\![HotelBooking]\!\}) = [\![\mathsf{bH.aB}]\!] \cup [\![\mathsf{bH.cB}.E(throw)?.skip.\mathsf{cH}]\!] \cup [\![E(\mathsf{bH})?]\!]$$

$$\begin{aligned}
&\rho(HotelTransactions) = \\
&\quad [\![\mathsf{bH}_1.\mathsf{aB}_1]\!] \parallel [\![\mathsf{bH}_2.\mathsf{aB}_2]\!] \cup \\
&\quad [\![E(\mathsf{bH}_1)?]\!] \parallel [\![E(\mathsf{bH}_2)?]\!] \cup \\
&\quad [\![\mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!] \parallel [\![\mathsf{bH}_2.\mathsf{cB}_2.E(throw_2)?.skip_2.\mathsf{cH}_2]\!] \cup \\
&\quad [\![\mathsf{bH}_1.\mathsf{aB}_1]\!] \parallel [\![\mathsf{bH}_2.\mathsf{cB}_2.E(throw_2)?.skip_2.\mathsf{cH}_2]\!] \cup \\
&\quad [\![\mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!] \parallel [\![\mathsf{bH}_2.\mathsf{aB}_2]\!] \cup \\
&\quad [\![E(\mathsf{bH}_1)?]\!] \parallel [\![\mathsf{bH}_2.\mathsf{cB}_2.E(throw_2)?.skip_2.\mathsf{cH}_2]\!] \cup \\
&\quad [\![\mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!] \parallel [\![E(\mathsf{bH}_2)?]\!] \cup \\
&\quad [\![\mathsf{bH}_1.\mathsf{aB}_1]\!] \parallel [\![E(\mathsf{bH}_2)?]\!] \cup \\
&\quad [\![E(\mathsf{bH}_1)?]\!] \parallel [\![\mathsf{bH}_2.\mathsf{aB}_2]\!]
\end{aligned}$$

Figure 67: *HotelTransactions* example.

Figure 67 shows the parallel composition of two HotelBooking transactions. The program HotelTransactions shows how the interleaved execution of processes may lead to interferences. Let $\theta_1 = [\![\mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!]$ *that describes traces where the booking succeeds and is later cancelled by the client, and* $\theta_2 = [\![E(\mathsf{bH}_2)?]\!]$ *that describes traces where no rooms are available and therefore no activity can be executed. Take the interleaving of traces of* θ_1 *and* θ_2 *on an initial state* s *such that* $s \models rooms = 1$*. In this setting,* θ_1 *has to be executed first and consequently activity* bH_1 *books the last room available. There is no serial execution of* θ_1 *and* θ_2 *(these sets of traces cannot be sequentially composed), since after the execution of a trace of* θ_1 *activity* bH_2 *should succeed (the last room becomes available again after the execution of* cB_1*). However, the following interleaved execution is possible* $[\![\mathsf{bH}_1.E(\mathsf{bH}_2)?.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!]$*, because when* bH_2 *is executed there are no rooms available. This shows that HotelTransactions is not serializable, since some interleaved traces do not correspond to a serial execution.*

The aim of compensable programs is that the overall recoverability of a system can be achieved through the definition of local recovery actions. As the system evolves, those local compensation actions are dynamically composed into a program that reverts all actions performed until then. Therefore, it is uttermost important that the dynamically built compensation trace does indeed revert the current state to the initial state. Next,

we define the notion of a correct compensable program, where any failed forward trace can be compensated to a state equivalent to the initial state.

Definition 34 (Correct Compensable Program)**.** *Let X be a set of integer variables. A compensable program δ has a correct compensation over X if for all pairs of traces $(\tau, \overline{\tau}) \in \rho_c(\delta)$, if $closed(\tau)$ and $closed(\overline{\tau})$ then it holds that $first(\tau)\backslash_X last(\tau) = last(\overline{\tau})\backslash_X first(\overline{\tau})$.*

Serializability is extended from basic programs to compensable programs:

Definition 35 (Serializable Compensable Programs)**.** *A set of n compensable programs $\delta_1, ..., \delta_n$ forms a* serializable set *if for all $(\tau, \overline{\tau}) \in \rho_c(\delta_1 \parallel \cdots \parallel \delta_n)$ with $closed(\tau)$ and $closed(\overline{\tau})$ there exist $(\nu_1, \overline{\nu}_1) \in \rho_c(\delta_1), ..., (\nu_n, \overline{\nu}_n) \in \rho_c(\delta_n)$ and a permutation $\iota : [1, n] \to [1, n]$ such that $\tau \bowtie (\nu_{\iota(1)} \cdots \nu_{\iota(n)})$ and $\overline{\tau} \bowtie (\overline{\nu}_{\iota(n)} \cdots \overline{\nu}_{\iota(1)})$.*

A compensable program δ is serializable *if all of its subterms of the form $\delta_1 \parallel \cdots \parallel \delta_n$ we have that $\delta_1, ..., \delta_n$ form a serializable set.*

Similarly to basic programs we can define strong serializability for compensable programs:

Definition 36 (Strong Serializable Compensable Programs)**.** *A set of n compensable programs $\delta_1, ..., \delta_n$ forms a* strong serializable set *if for all $(\tau, \overline{\tau}) \in \rho_c(\delta_1 \parallel \cdots \parallel \delta_n)$ with $closed(\tau)$ and $closed(\overline{\tau})$ there exist $(\nu_1, \overline{\nu}_1) \in \rho_c(\delta_1), ...,$ $(\nu_n, \overline{\nu}_n) \in \rho_c(\delta_n)$ and a permutation $\iota : [1, n] \to [1, n]$ such that $pr_{\delta_i}(\tau) = \nu_i$ for all $i \in [1, n]$, $pr_{\delta_i}(\overline{\tau}) = \overline{\nu}_i$ for all $i \in [1, n]$, $\tau \bowtie (\nu_{\iota(1)} \cdots \nu_{\iota(n)})$ and $\overline{\tau} \bowtie (\overline{\nu}_{\iota(n)} \cdots \overline{\nu}_{\iota(1)})$.*

A compensable program δ is strong serializable *if all of its subterms of the form $\delta_1 \parallel \cdots \parallel \delta_n$ we have that $\delta_1, ..., \delta_n$ form a strong serializable set.*

The following theorem shows the soundness of our language, since it proves that compensation correctness is ensured by construction: the composition of correct compensable programs results in a correct compensable program.

Theorem 12. *Let X be a set of integer variables and δ a serializable compensable program where every compensation pair is correct over X, then δ is correct over X.*

Proof. See B.1. □

As in general serializability is hard to prove we suggest the simpler definition of apartness. If the set of variables updated and consulted by two programs are disjoint, those programs can be concurrently executed since they do not interfere with each other (a similar approach was taken in [CFV08]). Two compensable programs δ and γ are apart if they do not update or read overlapping variables, then δ and γ can be executed concurrently and their resulting traces can be merged. The final state of a concurrent execution of apart programs can be understood as a join of the resulting states of each program.

Formulas for basic programs can be easily extended to compensable programs as they can be applied to the forward program and stored compensations can be ignored.

Definition 37 (Formulas)**.** *The set of formulas $Fml(V)$ from Definition 26 is extended as follows:*

$$\begin{array}{rcl} \varphi, \psi & ::= & \ldots \mid \langle\delta\rangle\varphi \mid [\delta]\varphi \mid \mathsf{S}(\delta) \mid \mathsf{S}_{\mathsf{W}}(\delta) \mid \mathsf{F}(\delta) \mid \\ & & \mathsf{C}(\delta, X) \mid \mathsf{C}_{\mathsf{W}}(\delta, X). \end{array}$$

The modal operator $\mathsf{C}(\delta, X)$ states that every failure of δ is compensable regarding a set of variables X. A weak compensable operator $\mathsf{C}_{\mathsf{W}}(\delta, X)$ states that some failures of δ are compensable for a set of variables X.

To define formula validity, we extend the notion of closed traces from basic programs to compensable programs as shown below:

$$closure(\delta) \triangleq \{(\tau, \overline{\tau}) \mid (\tau, \overline{\tau}) \in \rho_c(\delta) \wedge closed(\tau) \wedge closed(\overline{\tau})\}.$$

Definition 38 (Formula Validity)**.** *We extend Definition 27 with the new modal operators for compensable programs.*

$$
\begin{array}{lcll}
s & \models & \langle\delta\rangle\varphi & \textit{iff} \quad \exists\,(\tau,\overline{\tau}) \in \textit{closure}(\delta) \textit{ such that } \textit{first}(\tau) = s \\
 & & & \textit{and } \textit{last}(\tau) \models \varphi \\
s & \models & \mathsf{S}(\delta) & \textit{iff} \quad \textit{for all traces } (\tau,\overline{\tau}) \in \textit{closure}(\delta) \textit{ if } \textit{first}(\tau) = s \\
 & & & \textit{then } \neg E(\tau) \\
s & \models & \mathsf{S}_{\mathsf{W}}(\delta) & \textit{iff} \quad \exists(\tau,\overline{\tau}) \in \textit{closure}(\delta) \textit{ such that } \textit{first}(\tau) = s \\
 & & & \textit{and } \neg E(\tau) \\
s & \models & \mathsf{F}(\delta) & \textit{iff} \quad s \models \neg\mathsf{S}_{\mathsf{W}}(\delta) \\
s & \models & \mathsf{C}(\delta, X) & \textit{iff} \quad \textit{for all traces } (\tau,\overline{\tau}) \in \textit{closure}(\delta) \textit{ if } \textit{first}(\tau) = s \\
 & & & \textit{then } \textit{first}(\tau)\backslash_X \textit{last}(\tau) = \textit{last}(\overline{\tau})\backslash_X \textit{first}(\overline{\tau}) \\
s & \models & \mathsf{C}_{\mathsf{W}}(\delta, X) & \textit{iff} \quad \exists(\tau,\overline{\tau}) \in \textit{closure}(\delta) \textit{ such that } \textit{first}(\tau) = s \\
 & & & \textit{and } \textit{first}(\tau)\backslash_X \textit{last}(\tau) = \textit{last}(\overline{\tau})\backslash_X \textit{first}(\overline{\tau})
\end{array}
$$

Considering the hotel booking program in Figure 66, the formula

$$\langle \mathit{HotelBooking} \rangle \mathit{status} = \mathit{confirmed}$$

holds for any state where $\mathit{rooms} > 0$. Moreover $\mathsf{C}(\mathit{HotelBooking}, \{\mathit{rooms}\})$ holds for any state. For the flight booking program in Figure 64 we can require that, if there are seats available a flight is booked, which can be written as the formula:

$$\mathit{seats}_1 > 0 \vee \mathit{seats}_2 > 0 \vee \cdots \vee \mathit{seats}_n > 0 \;\rightarrow\; [\mathit{BuyFlight}]\, \neg \mathit{flight} = \bot$$

Next, we list some other valid formulas for program $\mathit{BuyFlight}$:

$$
\begin{array}{l}
[\mathit{BuyFlight}]\ \mathit{status}_i = \mathit{payed} \;\leftrightarrow\; \mathit{flight} = i \\
[\mathit{BuyFlight}]\ \mathit{status}_i = \mathit{cancelled} \;\rightarrow\; \mathit{flight} \neq i \wedge \mathit{flight} \neq \bot \\
\mathsf{C}(\mathit{Reservation}_i, \{\mathit{seats}_i\})
\end{array}
$$

6.4 Tool support

We present a tool for the dynamic logic of Section 6.3 and experiment with examples from the previous sections.

For our tool we prefer to use a language that adequately represents concurrent change and works as a semantic and logical framework, for this reason we chose Maude (see Section 2.6).

The implementation is structured in modules, the three main modules are `SYNTAX`, `PROGRAMS` and `SEMANTICS`. The complete definition of the modules can be found at [Sou13]. We use predefined modules such as `NAT` and `INT` for natural and integer numbers that are used for variables and terms. Moreover the parametric module `SET` is used for sets of variables, traces or trace pairs. The module `MAP` standing for functions or dictionaries defines a state in our setting as a partial function from variables to integers. It enriches the codomain with a special element called `undefined` that is returned if a value has no mapping. Note that variables can only have integer values.

In the module `SYNTAX` we define the syntax for programs and compensable programs. They are given as in Definitions 20 and 30 except of iteration. Though in this case we could define an infinite set of traces we cannot handle them in the semantics. Instead we define the n-th power of a program as executing the program n times. Next the module defines formulas of dynamic logic as in Definitions 26 and 37. We use a subsort for base formulas without programs. Note that we do not include quantifiers since we cannot handle them in the semantics. Terms can be variables, integers or, for combining these, the addition of terms. Predicates are equality of terms and the less-than predicate. An action is a list of assignments of terms to variables and a base formula. The empty assignment is denoted `skip`. As a last item the module defines traces and trace pairs.

In the module `PROGRAMS` the functions ρ and ρ_c are defined together with any auxiliary functions needed like the sequential composition $\circ$ of traces or the interleaving. Note that for the interpretation of programs we would not need an exact definition of actions. This becomes necessary once we build the closure of a trace. Figure 68[1] shows the interpretation of the program from the eStore example in Figure 63. While the first lines of the result correspond to our previous result, the tool also returns the traces we omitted previously. These traces would be impossible in the closure, *e.g.* the tool returns traces where *throw* is successful.

The last module `SEMANTICS` defines the interpretation of formulas.

[1]For the sake of readability we improved the layout of the result.

```
Maude> red rho(aO ; ((aC O (rC ; throw)) | bC)) .
reduce in EXAMPLE : rho(aO ; bC | (aC O rC ; throw)) .
rewrites: 664 in 5503762061ms cpu (3ms real)
(0 rewrites/second)
result NeTraceSet: [- aO],
  [aO] [aC] [bC],   [aO] [bC] [aC],
  [aO] [aC] [- bC], [aO] [- bC] [aC],
  [aO] [rC] [- throw] [bC],
  [aO] [bC] [rC] [- throw],
  [aO] [rC] [bC] [- throw],
  [aO] [rC] [- bC] [- throw],
  [aO] [- bC] [rC] [- throw],
  [aO] [rC] [- throw] [- bC],
  [aO] [bC] [- aC],   [aO] [- aC] [bC],
  [aO] [- aC] [- bC], [aO] [- bC] [- aC],
  [aO] [bC] [- rC],   [aO] [- rC] [bC],
  [aO] [- rC] [- bC], [aO] [- bC] [- rC],
  [aO] [rC] [throw] [bC],
  [aO] [rC] [bC] [throw],
  [aO] [bC] [rC] [throw],
  [aO] [rC] [throw] [- bC],
  [aO] [rC] [- bC] [throw],
  [aO] [- bC] [rC] [throw]
```

Figure 68: Interpretation of a program

We start by giving the valuation functions for terms and predicates in a given state. The valuation of terms returns the kind `Int` instead of the sort. The reason is that a variable that is not yet defined in the current state cannot be reduced. For predicates if one of the terms is not reduceable the valuation function should return `false`. However, for the less than predicate returning false does not imply that the terms are greater or equal. Thus we omit this case for this predicate, but leave it for the equality.

The valuation function is extended to assignments. We use it as well to evaluate a trace from a given state. In the previous sections we used the closure of traces to denote the actually valid executions of a program in a state. For our tool this notion is too general as there might be an infi-

nite set of possible executions. However given a state and a trace the valuation is deterministic, *i.e.*, there is only one possible result. If the trace is not valid we return the special item `nostate`. This different valuation of the closure restricts the expressiveness of the operators C and C_W. While in the original semantics the first state of the compensation can be any state this has to be fixed in the tool, the last state of the forward trace is the obvious choice here.

The module uses auxiliary functions for the formulas including programs in order to sift through the complete set of possible traces. For example for the strong success operator $\mathsf{S}(\alpha)$ we build the set of traces for α and have to check for every trace if it is valid then there is no error. At the end the module defines the validity of formulas as in the Definitions 27 and 38.

In Figure 69 we present the application of the tool to the example of Figure 67. We show a formula in the state where only one room is available. After every execution of the program, *i.e.*, after both transactions have been completed, we want to show that not both can have booked the room. The corresponding formula is $[\mathit{HotelTransactions}]\neg(\mathit{status}_1 = \mathit{confirmed}) \wedge \neg(\mathit{status}_2 = \mathit{confirmed})$. The formula shows that after the execution of the program the status of the two transactions cannot be *confirmed* for both. Note that variables can only be integers, thus the status of the booking can have values `0` for *booked*, `1` for *confirmed* and `-1` for *cancelled*. The tool first builds the complete set of traces for the program, then checks the formula after each valid execution. The result as expected is `true`.

As we have seen in the first example we can simulate computations using the tool. Moreover given a defined state we can verify properties for finite programs. Among these properties we can prove are the success of a program or whether it has a correct compensation. This information can be used to validate and improve systems including long-running transactions and compensations.

```
reduce in EXAMPLE : rooms |-> 1 |-
    [ {
        act( def(rooms, add(rooms, -1)) def(status, 0)
             def(price, 140)            def(fee, 20),
             le(rooms, 0) + eq(rooms, 0))
        % act(def(rooms, add(rooms, 1)) def(price, fee)
              def(fee, 0) , ff) ;
       (act(def(status, 1), ff) % act(skip, ff)
        O
        act(def(status, -1), ff) % act(skip, ff) ;
        act(skip, tt) % act(skip, ff))
      }
    |
      {
        act(def(rooms, add(rooms, -1)) def(status2, 0)
            def(price2, 140)           def(fee2, 20),
             le(rooms, 0) + eq(rooms, 0))
        % act(def(rooms, add(rooms, 1)) def(price2, fee2)
              def(fee2, 0), ff) ;
       (act(def(status2, 1), ff) % act(skip, ff)
        O
        act(def(status2, -1), ff) % act(skip, ff) ;
        act(skip, tt) % act(skip, ff))
      }
    ]   - (eq(status, 1) & eq(status2, 1)) .
rewrites: 873171 in 672072792ms cpu (6800ms real)
(1 rewrites/second)
result Bool: true
```

Figure 69: Interpretation of a formula

6.5 Including Interruption

As we pointed out in Chapter 3 it is important to include interruption in concurrent compensable programs. However in our current setting a program may execute several futile actions in case of an error as each sibling in a parallel composition always completes its forward execution before compensating. While this largely simplifies our interpretation and the technical treatment, it is not very likely that it can be enforced in actual applications. We will show in this section how to extend our approach to deal with interruption for compensable programs.

Our first idea was to change the interleaving operator to interrupt traces in case of an error. However it soon became evident that this is not sufficient. While we can interrupt the forward flow by construction there is no link to the compensation. Then, how can we know what has to be compensated? Thus we need to establish an immediate link between forward and backward flow.

We solve this problem in a way that is close to the approach proposed in [LZ09], where the order of the backward flow is exactly the reverted forward flow. The idea is to rely on a unique repository for the installed compensations, so that the total order of installation is recorded. We achieve this by changing the interpretation of compensable programs from pairs of traces to traces of pairs. Elements in a pair can either be the empty trace $[\![]\!]$ or a singleton $[\![\ell]\!]$. We use π to denote a trace of pairs, λ for a single pair, ϵ for a trace of length zero and pr_l and pr_r for the left and right projection. We will use the projection functions as well for whole traces of pairs π to extract the forward or backward flow respectively, *i.e.*, we apply the projection to each pair and build again a "classical" trace from the result. Furthermore the function rpr_r will return a projection of the backward flow in the reverse order. The error formula for a pair λ returns the error formula for the first element, *i.e.*, $E(pr_l(\lambda))$. This can be lifted to the whole trace. We combine traces of pairs similarly to the combination of normal traces. The sequential composition checks whether the first trace contains an error and depending on the result either appends the second trace or not.

$$
\begin{array}{lcl}
\epsilon \,{}_{\flat}\|_{\mathsf{ff}}\ \pi_2 & \triangleq & \{\pi_2\} \\
\epsilon \,{}_{\flat}\|_{\mathsf{tt}}\ \pi_2 & \triangleq & \mathit{Pre}(\{\pi_2\}) \\
\lambda_1\pi_1\ {}_{\mathsf{tt}}\|_{\mathsf{tt}}\ \lambda_2\pi_2 & \triangleq & \mathit{Pre}(\lambda_1\pi_1\ \|\ \lambda_2\pi_2) \\
\lambda_1\pi_1\ {}_{\mathsf{ff}}\|_{\mathsf{ff}}\ \lambda_2\pi_2 & \triangleq & \{\lambda_1\pi \mid \pi \in (\pi_1\ {}_{\mathsf{ff}}\|_{E(\lambda_1)}\ \lambda_2\pi_2)\} \\
 & \cup & \{\lambda_2\pi \mid \pi \in (\lambda_1\pi_1\ {}_{E(\lambda_2)}\|_{\mathsf{ff}}\ \pi_2)\} \\
\lambda_1\pi_1\ {}_{\mathsf{tt}}\|_{\mathsf{ff}}\ \lambda_2\pi_2 & \triangleq & \{\lambda_1\pi \mid \pi \in (\pi_1\ {}_{\mathsf{tt}}\|_{E(\lambda_1)}\ \lambda_2\pi_2)\} \\
 & \cup & \{\lambda_2\pi \mid \pi \in (\lambda_1\pi_1\ {}_{\mathsf{tt}}\|_{\mathsf{ff}}\ \pi_2)\}
\end{array}
$$

Figure 70: Definition of interleaving with interrupt

Next we define the interleaving of traces of pairs including interruption. We introduce boolean subscripts for both arguments of the parallel composition operator, where ff is used for a normal or aborting execution and tt for a process that can be interrupted. The new definition is displayed in Figure 70. Let T be a set of traces. We define the set of prefixes of T as $\mathit{Pre}(T) \triangleq \{t \mid t\ t' \in T\}$. Moreover we use $\|$ without subscripts for the interleaving as defined on page 136. Note that interleaving is commutative.

We have two base cases where one branch is ϵ. In the first case the remaining process was not interrupted, then the definition is equivalent to the classical interleaving returning the trace itself. In the other case the process received an interrupt, then we return the set of all prefixes for the remaining trace. Next both processes were interrupted, *i.e.*, both subscripts are tt, but none finished executing yet. In this case we build the set of traces using the classical interleaving $\|$ without subscripts and then build the closure over prefixes. That way we include the full trace as well as any partial one. In the last two cases at least one process was not interrupted, *i.e.*, one subscript is ff. The definition is similar to the classical interleaving, though if there is an abort the corresponding subscript of the other process is updated. We use the error formula of the current step to inform the sibling branch (replacing the original subscript). Note that once a branch received an interrupt, *i.e.*, its subscript is tt, it remains that way.

Now we can give the new definition for the interpretation of com-

pensable programs:

Definition 39 (Interpretation of Compensable Programs with Interrupt)**.** *We define the interpretation ρ_i of compensable programs with interrupt in the following manner:*

$$
\begin{array}{lcl}
\rho_i(a \div \overline{a}) & \triangleq & \{\langle \tau, \overline{\tau} \rangle \mid \tau \in \rho(a) \wedge \overline{\tau} \in \rho(\overline{a}) \wedge \neg E(\tau)\} \cup \\
 & & \{\langle \tau, [\![]\!] \rangle \mid \tau \in \rho(a) \wedge E(\tau)\} \\
\rho_i(\delta \; + \; \gamma) & \triangleq & \rho_i(\delta) \cup \rho_i(\gamma) \\
\rho_i(\delta \,;\gamma) & \triangleq & \{\pi_1 \circ \pi_2 \mid \pi_1 \in \rho_i(\delta) \wedge \pi_2 \in \rho_i(\gamma)\} \\
\rho_i(\delta \parallel \gamma) & \triangleq & \{\pi \mid \pi_1 \in \rho_i(\delta) \wedge \pi_2 \in \rho_i(\gamma) \;\wedge \pi \in \pi_1 \parallel_{\mathtt{ff}} \pi_2\} \\
\rho_i(\delta^*) & \triangleq & \{\langle [\![]\!], [\![]\!] \rangle\} \;\cup\; \rho_i(\delta \,; \delta^*) \\
\rho(\{\![\delta]\!\}) & \triangleq & \{pr_l(\pi) \mid \pi \in \rho_i(\delta) \wedge \neg E(\pi)\} \cup \\
 & & \{cl(pr_l(\pi)) \circ rpr_r(\pi) \mid \pi \in \rho_i(\delta) \wedge E(\pi)\}
\end{array}
$$

As expected, the interpretation is very similar to Definition 33. Note that for the parallel composition we start with the subscripts for both siblings being false. For a transaction in case of an error we first build the forward trace, clear the errors and then append the reverted backward trace.

Example 22 (Travel Agency)**.** *In Figure 71 we present an extension of Example 21 using the compensable program HotelBooking from Figure 66. Additionally to the booking of a hotel room the program now also books a flight at the same time. We first show the interpretation ρ_i for each single process. In each case there is one successful trace and two failing traces where either the booking is cancelled or there are not enough resources (rooms or seats). Note that this program is an example of apartness for a parallel composition. Figure 72 shows a subset of the interpretation of the concurrent program. The first part shows any possible interleaving where both booking processes are successful. In the next part the hotel booking fails while the flight booking succeeds. As the interpretation shows the right sibling does not have to be fully executed but can be interrupted at any time.*

Now we can transfer some of the previous definitions to compensable programs with interrupts. We introduce the set of closed traces for a compensable program δ similarly to the definition without interrupt.

$$
\begin{array}{lcl}
closure(\delta) & \triangleq & \{(\tau, \overline{\tau}) \mid \pi \in \rho_i(\delta) \wedge \tau = pr_l(\pi) \wedge \\
 & & \qquad \overline{\tau} = rpr_r(\pi) \wedge closed(\tau) \wedge closed(\overline{\tau})\}.
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Booking} & \triangleq & \mathit{HotelBooking} \parallel \mathit{FlightBooking} \\
\mathit{FlightBooking} & \triangleq & \mathit{bookFlight} \div \mathit{cancelFlight}\,; \\
 & & (\mathit{acceptBFlight} \div \mathit{skip} + \\
 & & \mathit{cancelBFlight} \div \mathit{skip}\,;\, \mathit{throw} \div \mathit{skip}))
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathsf{bF} & \triangleq & \mathit{seats}, \mathit{statusF}, \mathit{priceF}, \mathit{feeF} := & E(\mathsf{bF}) & \triangleq & \mathit{seats} \leq 0 \\
 & & \quad \mathit{seats} - 1, \mathit{booked}, 210, 50 & & & \\
\mathsf{cF} & \triangleq & \mathit{seats}, \mathit{priceF}, \mathit{feeF} := \mathit{seats} + 1, \mathit{feeF}, 0 & E(\mathsf{cF}) & \triangleq & \mathit{false} \\
\mathsf{aBF} & \triangleq & \mathit{statusF} := \mathit{confirmed} & E(\mathsf{aBF}) & \triangleq & \mathit{false} \\
\mathsf{cBF} & \triangleq & \mathit{statusF} := \mathit{cancelled} & E(\mathsf{cBF}) & \triangleq & \mathit{false}
\end{array}
$$

$$
\begin{array}{l}
\rho_i(\mathsf{bH} \div \mathsf{cH}\,;(\mathsf{aB} \div \mathit{skip} \;+\; \mathsf{cB} \div \mathit{skip}\,;\mathit{throw} \div \mathit{skip})) \;= \\
\qquad (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!] \rangle . \langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!] \rangle) \\
\quad \cup \quad (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!] \rangle . \langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!] \rangle . \langle [\![-\mathit{throw}]\!], [\![]\!] \rangle) \\
\quad \cup \quad (\langle [\![-\mathsf{bH}]\!], [\![]\!] \rangle)
\end{array}
$$

$$
\begin{array}{l}
\rho_i(\mathsf{bF} \div \mathsf{cF}\,;(\mathsf{aBF} \div \mathit{skip} \;+\; \mathsf{cBF} \div \mathit{skip}\,;\mathit{throw} \div \mathit{skip})) \;= \\
\qquad (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!] \rangle . \langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!] \rangle) \\
\quad \cup \quad (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!] \rangle . \langle [\![\mathsf{cBF}]\!], [\![\mathit{skip}]\!] \rangle . \langle [\![-\mathit{throw}]\!], [\![]\!] \rangle) \\
\quad \cup \quad (\langle [\![-\mathsf{bF}]\!], [\![]\!] \rangle)
\end{array}
$$

Figure 71: Extended *Booking* example.

Definition 40 (Correct Compensable Program with Interrupt)**.** *Let X be a set of integer variables. A compensable program δ has a correct compensation over X if for all traces of pairs $\pi \in \rho_i(\delta)$, if $\tau = pr_l(\pi)$ with $closed(\tau)$ and $\overline{\tau} = rpr_r(\pi)$ with $closed(\overline{\tau})$ then $first(\tau) \backslash_X last(\tau) = last(\overline{\tau}) \backslash_X first(\overline{\tau})$.*

By transferring the definition for correctness we can state as well a new version of Theorem 12. Note that due to the different semantics for compensable programs serializability is not needed to ensure correctness.

Theorem 13. *Let X be a set of integer variables and δ a compensable program with interrupt where every compensation pair is correct over X, then δ is correct over X.*

Proof. See B.3. □

Definition 38 for formula validity can be reused for compensable programs with interrupt due to the similar definition of the closure operator.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\rho_i(\mathsf{bH} \div \mathsf{cH}\,;(\mathsf{aB} \div \mathit{skip} \;+\; \mathsf{cB} \div \mathit{skip}\,;\mathit{throw} \div \mathit{skip})} \\
\multicolumn{2}{l}{\quad \| } \\
\multicolumn{2}{l}{\quad \mathsf{bF} \div \mathsf{cF}\,;(\mathsf{aBF} \div \mathit{skip} \;+\; \mathsf{cBF} \div \mathit{skip}\,;\mathit{throw} \div \mathit{skip}))} \\
= & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle).\langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!]\rangle \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{aB}]\!], [\![\mathit{skip}]\!]\rangle) \\
\\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{bH}]\!], [\![\mathsf{cH}]\!]\rangle.\langle [\![\mathsf{cB}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathit{throw}]\!], [\![]\!]\rangle) \\
\\
\cup & (\langle [\![-\mathsf{bH}]\!], [\![]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![-\mathsf{bH}]\!], [\![]\!]\rangle.\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle) \\
\cup & (\langle [\![-\mathsf{bH}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![-\mathsf{bH}]\!], [\![]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![-\mathsf{bH}]\!], [\![]\!]\rangle) \\
\cup & (\langle [\![\mathsf{bF}]\!], [\![\mathsf{cF}]\!]\rangle.\langle [\![\mathsf{aBF}]\!], [\![\mathit{skip}]\!]\rangle.\langle [\![-\mathsf{bH}]\!], [\![]\!]\rangle) \\
\cup & \ldots
\end{array}
$$

Figure 72: Interpretation of *Booking* example

Consider the compensable program $Booking$ from Example 22, we want to ensure that a hotel is only booked if also a flight was booked and vice versa. This property can be defined with the following formula:

$$[\{\![Booking]\!\}]statusH = confirmed \leftrightarrow statusF = confirmed$$

Other valid formulas are:

$$C(Booking, \{rooms, seats\})$$
$$\langle\{\![Booking]\!\}\rangle statusH = cancelled \rightarrow priceF < 50$$

The first one states that the program $Booking$ has a correct compensation regarding the variables $rooms$ and $seats$. The second formula states a property that can only be valid for compensable programs with interrupt. It says that when the hotel was cancelled there are traces where we do not have to pay the fee for cancelling the flight. That can only happen if either there are no available seats or the execution of the flight booking was interrupted.

6.6 Conclusion

In this chapter we presented a concurrent language including compensations and long-running transactions and formalised its behaviour. We introduced a dynamic logic for this language to reason about program correctness regarding the special requirements of compensations. It allows us to specify properties for verification. Moreover we state in the main theorem under which conditions a compensable program always restores a correct state.

Chapter 7

Conclusion

In this thesis we presented new approaches to the formal specification and analysis of long-running transactions. We centered our research around four directions: Expressiveness, modularity, implementation and verification. In the first part we focused on the first three items by looking at workflow based calculi. In [BBF+05] four existing policies for handling compensations in concurrent programs are described. As we explained in the thesis none of them is entirely satisfactory. We deduced a new policy that improves existing ones by allowing the activation of compensations autonomously from siblings but only after an actual error occured. We introduced this policy both with and without interruption and described it formally using three different but coherent semantics.

First we presented a denotational semantics that allowed us to compare the new policy to existing ones. Figure 73 shows the relationship between the different policies where we use double lines to mark those proven in this thesis. The numbers along the arrows refer to the theorems proving the inclusion.

Moreover we introduced two operational semantics. The first one is based on an encoding into Petri nets that allows us to exploit their well-developed theory. The other operational semantics is a small-step semantics based on labelled transition systems. It can easily be extended to include additional syntax and modified to represent other policies.

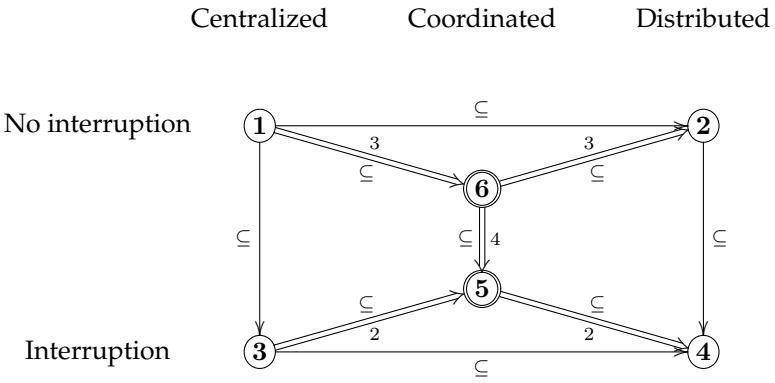


Figure 73: Compensation policies (arrows stand for trace inclusion)

Policies	(1)	(2)	(3)	(4)	(5)	(6)
big-step	X	Naive Sagas	X	Revised Sagas		
denotational	X	X	cCSP	X	X	X
Petri nets	X	X	X	X	X	X
small-step	X		X		X	X

Figure 74: Overview semantics and policies

An overview regarding the different policies and semantics is shown in Figure 74. We mark with an X or the name of the existing calculus which policy is represented in which semantics. The existing semantics described in [BBF+05] are colored in black where the dashed frames represent the correspondence of the respective two calculi. The remaining marks show the achievements presented in the thesis. Note that the frame around the semantics for the fifth policy reflects the correspondence proven between the different semantics.

In the second part of the thesis we focused on verification, but with-

out dismissing the other directions. We introduced a logical framework to reason on long-running transactions. It is based on an extension of dynamic logic to include concurrency and compensations. Within this logic we are able to define a notion of correctness of a compensable program given sufficient conditions. It is to our knowledge the first logic for long-running transactions.

Throughout this thesis we used Maude to implement different tools. They helped us in testing and improving the theory by solving larger and more complex problems.

7.1 Novel research directions

The research carried out in this thesis provides a basis for new research directions:

- We showed possible extensions of the core language of our calculus for the small-step semantics. While it may be easy to include them in the denotational semantics this seems more difficult in the Petri net encoding. Because of the intricate mechanism to broadcast interrupts it is rather complicated to extend the Petri nets. While it is possible to include choice and failing compensations with several additions, iteration requires a more sophisticated type of Petri nets like Dynamic nets [AB09]. When unfolding the iteration we have to keep track of the sequence of actions in order to activate compensations in the right order. A possible solution could be to dynamically add parts to the net.

- From the operational semantics using the Petri net encoding we can derive an event structure semantics [NPW79]. We distinguish the causality of events $e \preceq e'$, *i.e.*, one event e has to happen before another event e', and the conflict of events $e \neq e'$, *i.e.*, an event e cannot happen if already another event e' happened. This can be used together with partial order verification methods to address the scalability of the analysis.

- We presented different tools for each of the semantics. They provide a sort of library for implementing larger examples and case studies. Moreover exploiting the LTL model checker of Maude can contribute to a new link to verification

- Regarding the logic an interesting aspect would be the adaptation of the interpretation of the underlying language to be closer to actual programming paradigms. That implies a better inclusion of interruption, but also distribution. A possibility might be to to link the logic to the denotational semantics. Regarding the discussion in Section 6.3 it would be interesting to learn if anything can be adopted from the areas of research regarding speculative execution of (simple) multi-threaded programs and optimistic parallel simulation. As another item, by developing suitable equivalences over states it would be possible to reduce the complexity of the analysis, and facilitate the development of automatic reasoning tools.

Appendix A

Proof of Bisimilarity Theorem 11

We state with a lemma a sort of well typed states in the labelled transition system:

Lemma 3. *Starting from a state $\boxdot, P$ where P is built from the basic syntax for processes (without runtime syntax) the following states are not reachable in the smallstep-semantics (for any A, B, P_1, P_2):*

$$\boxtimes, A \div B$$
$$\boxtimes, P_1; P_2$$

Moreover for any subterm of a reachable state of the following form

$$P_1; P_2 \qquad P_1 \$ C \qquad [C_1; C_2]$$

it holds that $\neg dn(P_1)$ and $\neg dn(C_1)$.

The lemma is shown by rule induction.

The above lemma implies the following:

Corollary 7. *For any subterm P' of a reachable state σ, P it holds that $dn(P')$ implies $P' \equiv [C] \wedge cmp(P') = C$ or $P' \equiv [C_1]_\sigma|_\sigma \ldots_\sigma|_\sigma [C_n] \wedge cmp(P') = \prod_{0 \leq i \leq n} C_i$.*

Thus a process that is done is either equivalent to a compensation or a parallel composition of compensations.

$$\frac{}{\mathbf{nil}@\langle R, R\rangle \in \mathbf{WTC}}$$

$$\frac{disj(R_1, R_2)}{A@\langle R_1, R_2\rangle \in \mathbf{WTC}}$$

$$\frac{C_T = C@\langle R_1, R_3\rangle \in \mathbf{WTC} \quad D_T = D@\langle R_3, R_2\rangle \in \mathbf{WTC} \quad tags(C_T) \cap tags(D_T) = \{R_3\}}{(C_T; D_T)@\langle R_1, R_2\rangle \in \mathbf{WTC}}$$

$$\frac{C_T \in \mathbf{WTC} \quad D_T \in \mathbf{WTC} \quad disj(tags(C_T), tags(D_T), \mathcal{K})}{(C_T | D_T)@\mathcal{K} \in \mathbf{WTC}}$$

Figure 75: Set of well-tagged compensations $\mathbf{WTC}$

We state an additional theorem for compensations, the proof is similar to the one for Theorem 11 but much simpler and is thus omitted:

Theorem 14. *Given the Petri Net N_C generated by a compensation C with external places R_1, R_2 the following holds:*

$$R_1 \approx C@\langle R_1, R_2\rangle$$

The proof is similar to the one for Theorem 11.

We remind that our intention is to prove that any computation in the Petri net associated with a compensable process P from its initial marking is weakly bisimilar to the state $\boxdot, P$ in the LTS semantics. The proof will be carried on by coinduction. First we need to fix some notation. Without loss of generality we assume that all activities are distinct (we disregard $throww$); if necessary we introduce subscripts in order to distinguish two otherwise identical actions. Remember that a Petri Net is a triple (P, T, F) with P a set of places, T a set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ the flow relation. We will use ${}^\bullet t$ for the preset of transition t and $t^\bullet$ for the postset.

We tag processes with a list of names. Basic activities and compensations are tagged with two names, while processes are tagged with six. This list of names will correspond to the outer interface of the corre-

$$\frac{disj(F_1, F_2, R_1, R_2, I_1, I_2)}{(A@\langle F_1, F_2\rangle \div B@\langle R_1, R_2\rangle)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \in \mathbf{WTP}}$$

$$\frac{\begin{array}{c} P_T = P@\langle F_1, F_3, R_3, R_2, I_1, I_2\rangle \in \mathbf{WTP} \\ Q_T = Q@\langle F_3, F_2, R_1, R_3, I_1, I_2\rangle \in \mathbf{WTP} \\ tags(P_T) \cap tags(Q_T) = \{F_3, R_3, I_1, I_2\} \end{array}}{(P_T; Q_T)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \in \mathbf{WTP}}$$

$$\frac{P_T \in \mathbf{WTP} \quad Q_T \in \mathbf{WTP} \quad disj(tags(P_T), tags(Q_T), \mathcal{I})}{(P_T \parallel Q_T)@\mathcal{I} \in \mathbf{WTP}}$$

$$\frac{C_T = C@\langle R_1, R_2\rangle \in \mathbf{WTC} \quad tags(C_T) \cap \{F_1, F_2, I_1, I_2\} = \emptyset}{[C_T]@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \in \mathbf{WTP}}$$

$$\frac{\begin{array}{c} P_T = P@\langle F_1, F_2, R_1, R_3, I_1, I_2\rangle \in \mathbf{WTP} \\ C_T = C@\langle R_3, R_2\rangle \in \mathbf{WTC} \\ tags(P_T) \cap tags(C_T) = R_3 \end{array}}{(P_T \$ C_T)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \in \mathbf{WTP}}$$

Figure 76: Set of well-tagged processes **WTP**

sponding Petri net. For basic activities and compensations the names are in- and outgoing place. For a process there are two names for the forward process, two for the reverse and two for handling and propagating interrupts. We use variables $\mathcal{I}$ for tags for processes denoted $P@\mathcal{I}$ and $\mathcal{K}$ for tags of length two. Abusing the notation, we write $\mathcal{K}$ and $\mathcal{I}$ for denoting both the tag (i.e. a list) and the set of underlying names. The set of tags for a tagged process is inductively defined as follows:

$$\begin{array}{lcl} tags((A@\mathcal{K}_1 \div B@\mathcal{K}_2)@\mathcal{I}) & = & \mathcal{K}_1 \cup \mathcal{K}_2 \cup \mathcal{I} \\ tags((P_T; Q_T)@\mathcal{I}) & = & \mathcal{I} \cup tags(P_T) \cup tags(Q_T) \\ tags((P_T \parallel Q_T)@\mathcal{I}) & = & \mathcal{I} \cup tags(P_T) \cup tags(Q_T) \\ tags([C_T]@\mathcal{I}) & = & \mathcal{I} \cup tags(C_T) \\ tags((P_T \$ C_T)@\mathcal{I}) & = & \mathcal{I} \cup tags(P_T) \cup tags(C_T) \end{array}$$

Moreover we use a predicate *disj* defined on lists of names that is true if all names are pairwise disjoint. With this we can define the sets of well-tagged compensations (Figure 75) and processes (Figure 76). Note that

(C-ACT-TAG)

$$\frac{}{\Gamma \vdash A@\langle R_1, R_2\rangle \xrightarrow{A} \mathbf{nil}@\langle R_2, R_2\rangle}$$

(C-PAR-L-TAG)

$$\frac{\Gamma \vdash C_T \xrightarrow{\lambda} C'_T}{\Gamma \vdash (C_T|D_T)@\mathcal{K} \xrightarrow{\lambda} (C'_T|D_T)@\mathcal{K}}$$

(C-SEQ1-TAG)

$$\frac{\Gamma \vdash C_T \xrightarrow{\lambda} C'_T \wedge \neg dn(C'_T) \wedge (C'_T; D_T)@\mathcal{I}' \in \mathbf{WTC}}{\Gamma \vdash (C_T; D_T)@\mathcal{I} \xrightarrow{\lambda} (C'_T; D_T)@\mathcal{I}'}$$

(C-SEQ2-TAG)

$$\frac{\Gamma \vdash C_T \xrightarrow{\lambda} C'_T \wedge dn(C'_T)}{\Gamma \vdash (C_T; D_T)@\mathcal{I} \xrightarrow{\lambda} D_T}$$

Figure 77: Semantics for tagged compensations

for a compensation $[C]$ in the corresponding outermost tag the first two names might be equivalent.

In Figure 77 – 79 the rules of the semantics given in Sections 5.1 and 5.2 are extended to tagged processes. In general for a process $P_T = P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$ that is moving forward the first element of the tag changes, while moving backward the third changes. For parallel composition this applies to the branches. In case of a fault the complete tag may change though the names are a subset of the previous ones. We can state the following two lemmas:

Lemma 4. *For every transition* $\Gamma \vdash \sigma, P_T \xrightarrow{\lambda} \sigma', Q_T$ *it holds that if* $P_T \in \mathbf{WTP}$ *then also* $Q_T \in \mathbf{WTP}$.

The proof is by rule induction.

Lemma 5. *For every transition* $\Gamma \vdash \sigma, P_T \xrightarrow{\lambda} \sigma', Q_T$ *it holds that* $tags(Q_T) \subseteq tags(P_T)$.

The proof is by rule induction.

Note that the predicate dn can be easily extended to tagged processes, however for cmp and the extract predicate $\rightsquigarrow$ names have to be adjusted,

(S-ACT-TAG)

$$\frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \begin{array}{c}\boxdot, (A_T \div B_T)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \\ \xrightarrow{A} \\ \boxdot, [B_T]@\langle F_2, F_2, R_1, R_2, I_1, I_2\rangle\end{array}}$$

(F-ACT-TAG)

$$\frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \begin{array}{c}\boxdot, (A_T \div B_T)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \\ \xrightarrow{\tau} \\ \boxtimes, [\mathbf{nil}]@\langle F_1, F_1, R_2, R_2, I_1, I_2\rangle\end{array}}$$

(SEQ-TAG)

$$\frac{\Gamma \vdash \boxdot, P_T \xrightarrow{\lambda} \boxdot, P'_T \wedge \neg dn(P'_T) \wedge (P'_T; Q_T)@\mathcal{I}' \in \mathbf{WTP}}{\Gamma \vdash \boxdot, (P_T; Q_T)@\mathcal{I} \xrightarrow{\lambda} \boxdot, (P'_T; Q_T)@\mathcal{I}'}$$

(S-SEQ-TAG)

$$\frac{\Gamma \vdash \boxdot, P_T \xrightarrow{\lambda} \boxdot, P'_T \wedge dn(P'_T) \wedge (Q_T\$cmp(P'_T))@\mathcal{I}' \in \mathbf{WTP}}{\Gamma \vdash \boxdot, (P_T; Q_T)@\mathcal{I} \xrightarrow{\lambda} \boxdot, (Q_T\$cmp(P'_T))@\mathcal{I}'}$$

(A-SEQ-TAG)

$$\frac{\Gamma \vdash \boxdot, P_T \xrightarrow{\lambda} \boxtimes, P'_T}{\Gamma \vdash \boxdot, (P_T; Q_T)@\mathcal{I} \xrightarrow{\lambda} \boxtimes, P'_T}$$

(STEP-TAG)

$$\frac{\Gamma \vdash \sigma, P_T \xrightarrow{\lambda} \sigma_2, P'_T \wedge \neg dn(P'_T) \wedge (P'_T\$C_T)@\mathcal{I}' \in \mathbf{WTP}}{\Gamma \vdash \sigma, (P_T\$C_T)@\mathcal{I} \xrightarrow{\lambda} \sigma_2, (P'_T\$C_T)@\mathcal{I}'}$$

(AS-STEP1-TAG)

$$\frac{\begin{array}{c}\Gamma \vdash \sigma, P_T \xrightarrow{\lambda} \sigma_2, P'_T \wedge dn(P'_T) \wedge cm?(P'_T) \wedge \\ (cmp(P'_T); C_T)@\mathcal{K} \in \mathbf{WTC} \wedge (P'_T\$C_T)@\mathcal{I}' \in \mathbf{WTP}\end{array}}{\Gamma \vdash \sigma, (P_T\$C_T)@\mathcal{I} \xrightarrow{\lambda} \sigma_2, [(cmp(P'_T); C_T)@\mathcal{K}]@\mathcal{I}'}$$

(AS-STEP2-TAG)

$$\frac{\begin{array}{c}\Gamma \vdash \sigma, P_T \xrightarrow{\lambda} \sigma_2, P'_T \wedge dn(P'_T) \wedge \neg cm?(P'_T) \wedge \\ (P'_T\$C_T)@\mathcal{I}' \in \mathbf{WTP}\end{array}}{\Gamma \vdash \sigma, (P_T\$C_T)@\mathcal{I} \xrightarrow{\lambda} \sigma_2, [C_T]@\mathcal{I}'}$$

Figure 78: Semantics for tagged Sagas

(COMP-TAG)

$$\frac{\Gamma \vdash C_T \xrightarrow{\lambda} C'@\langle R_1', R_2'\rangle}{\Gamma \vdash \begin{array}{c} \boxtimes, [C_T]@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \\ \xrightarrow{\lambda} \\ \boxtimes, [C'@\langle R_1', R_2'\rangle]@\langle F_1, F_2, R_1', R_2', I_1, I_2\rangle \end{array}}$$

(PAR-L-TAG)

$$\frac{\Gamma \vdash \sigma_1, P_T \xrightarrow{\lambda} \sigma_1', P_T'}{\Gamma \vdash \sigma, P_{T\sigma_1}|_{\sigma_2} Q_T@\mathcal{I} \xrightarrow{\lambda} \sigma \sqcap \sigma_1', P'_{T\sigma_1'}|_{\sigma_2} Q_T@\mathcal{I}}$$

(INT-R-TAG)

$$\frac{Q_T \rightsquigarrow Q'_T}{\Gamma \vdash \boxtimes, (P_{T\sigma}|_{\boxdot} Q_T)@\mathcal{I} \xrightarrow{\tau} \boxtimes, (P_{T\sigma}|_{\boxtimes} Q'_T)@\mathcal{I}}$$

Figure 79: Semantics for tagged Sagas, continued

such that they are still well-defined. Exemplary we give the adaptation for the extract predicate in Figure 80 (where $\mathcal{I} = \langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$ and $\mathcal{I}' = \langle F_1', F_2', R_1', R_2', I_1', I_2'\rangle$).

For every process $P_T \in \mathbf{WTP}$ we can inductively define the corresponding net. For a compensation pair $(A_T \div B_T)@\mathcal{I}$ this corresponds to the net displayed in Figure 30 with $\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle = \mathcal{I}$. Others can be equally defined. For new names for places and transitions in the generated net we use the subscript $\mathcal{I}$ of the tagged process (which is unique for $P_T = P@\mathcal{I} \in \mathbf{WTP}$). Take a parallel composition $(P_T|Q_T)@\mathcal{I}$ with the corresponding net as in Figure 33. As for compensation pairs $\mathcal{I}$ defines the interface $F_1, F_2, R_1, R_2, I_1, I_2$ of the net. The new place is defined as $MEX_{\mathcal{I}}$ and new transitions are defined like $\mathsf{fork}_{\mathcal{I}}, \mathsf{join}_{\mathcal{I}}$. Note that the names $PF_1, PF_2, \ldots$ are defined inductively by $P_T = P@\mathcal{I}'$ and $Q_T = Q@\mathcal{I}''$.

Theorem 11. Let N_P be the Petri net associated with the tagged compensable process $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$. Then, $F_1 \approx (\boxdot, P)$.

Proof. The proof is by coinduction.

Certain transitions in the net for processes do not infer any actual change in its executions w.r.t. weak bisimulation. In a way they could be

$$\frac{}{(A_T \div B_T)@\mathcal{I} \rightsquigarrow [\mathbf{nil}]@\langle F_2, F_2, R_2, R_2, I_1, I_2\rangle} \quad \frac{}{[C_T]@\mathcal{I} \rightsquigarrow [C_T]@\mathcal{I}}$$

$$\frac{P_T \rightsquigarrow P'_T \wedge \neg par(P_T)}{(P_T; Q_T)@\mathcal{I} \rightsquigarrow P'_T} \quad \frac{par(P_T)}{(P_T; Q_T)@\mathcal{I} \rightsquigarrow P_T}$$

$$\frac{P_T \rightsquigarrow P'_T \wedge \neg dn_{\boxtimes}(P'_T) \wedge (P'_T\$C_T)@\mathcal{I}' \in \mathbf{WTP}}{(P_T\$C_T)@\mathcal{I} \rightsquigarrow (P'_T\$C_T)@\mathcal{I}'}$$

$$\frac{P_T \rightsquigarrow P'@\mathcal{I}' \wedge dn_{\boxtimes}(P'@\mathcal{I}') \wedge cm?(P'@\mathcal{I}')}{(P_T\$C@\langle R'_2, R_2\rangle)@\mathcal{I} \rightsquigarrow ([cmp(P'@\mathcal{I}'); C@\langle R'_2, R_2\rangle])@\langle F'_1, F'_2, R'_1, R_2, I'_1, I'_2\rangle}$$

$$\frac{P_T \rightsquigarrow P'@\mathcal{I}' \wedge dn_{\boxtimes}(P'@\mathcal{I}') \wedge \neg cm?(P'@\mathcal{I}')}{(P_T\$C@\langle R'_2, R_2\rangle)@\mathcal{I} \rightsquigarrow [C@\langle R'_2, R_2\rangle]@\langle F'_1, F'_2, R'_1, R_2, I'_1, I'_2\rangle}$$

$$\frac{P_T \rightsquigarrow P'_T}{P_T|Q_T@\mathcal{I} \rightsquigarrow P'_T{\boxtimes}|_{\boxdot}Q_T@\mathcal{I}} \quad \frac{Q_T \rightsquigarrow Q'_T}{P_T|Q_T@\mathcal{I} \rightsquigarrow P_T{\boxdot}|_{\boxtimes}Q'_T@\mathcal{I}}$$

Figure 80: Predicate $P \rightsquigarrow P'$ for interrupting a tagged process

considered silent actions. These transitions are

$$\mathbf{Aux} = \{\mathsf{fork}_{\mathcal{I}}, \mathsf{join}_{\mathcal{I}}, \mathsf{rfork}_{\mathcal{I}}, \mathsf{rjoin}_{\mathcal{I}}, \mathsf{ip1}_{\mathcal{I}}, \mathsf{ip2}_{\mathcal{I}}, \mathsf{gc}_{\mathcal{I}}\}$$

for any $\mathcal{I}$. We will use a function that maps a marking to the set of (weakly) equivalent markings:

$$\begin{array}{rcl} M\downarrow & = & \{M' \mid M \xrightarrow{t} M' \wedge t \in \mathbf{Aux}\} \\ & \cup & \{M' \mid M' \xrightarrow{t} M \wedge t \in \{\mathsf{fork}_{\mathcal{I}}, \mathsf{join}_{\mathcal{I}}, \mathsf{rfork}_{\mathcal{I}}, \mathsf{rjoin}_{\mathcal{I}}\}\} \end{array}$$

We use a function **MP** to map a tagged process to a possible marking in its defined net. The function depends on the current state, i.e., whether the process can still commit or has already failed. Figures 81 – 83 show how **MP** is defined.

We define a relation $\approx_{N_P}$ for any well-tagged process P_T where N_P is the net generated by P_T such that

$$\begin{array}{rcl} \approx_{N_P} & = & \{\boxdot, Q_T,\ \mathbf{MP}_{\boxdot}(Q_T)\} \\ & \cup & \{\boxtimes, Q_T,\ \mathbf{MP}_{\boxtimes}(Q_T) \oplus I_2\} \\ & \cup & \{\boxtimes, Q'_T,\ \mathbf{MP}_{\boxtimes}(Q'_T)\} \end{array}$$

$$
\begin{array}{lcl}
\mathbf{MP}(\mathbf{nil}@\langle R, R\rangle) & = & R \\
\mathbf{MP}(A@\langle R_1, R_2\rangle) & = & R_1 \\
\mathbf{MP}((C_T; D_T)@\mathcal{K}) & = & \mathbf{MP}(C_T) \\
\mathbf{MP}((C_T | D_T)@\mathcal{K}) & = & \mathbf{MP}(C_T) \oplus \mathbf{MP}(D_T)
\end{array}
$$

Figure 81: Function **MP** for compensations

$$
\begin{array}{ll}
\mathbf{MP}_{\boxdot}((A_T \div B_T)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle) & = F_1 \\
\mathbf{MP}_{\boxdot}([C_T]@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle) & = F_2 \\
\mathbf{MP}_{\boxdot}((P_T; Q_T)@\mathcal{I}) & = \mathbf{MP}_{\boxdot}(P_T) \\
\mathbf{MP}_{\boxdot}((P_T \$ C_T)@\mathcal{I}) & = \mathbf{MP}_{\boxdot}(P_T) \\
\mathbf{MP}_{\boxdot}((P_T | Q_T)@\mathcal{I}) & = \mathbf{MP}_{\boxdot}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus MEX_{\mathcal{I}}
\end{array}
$$

Figure 82: Function **MP** for compensable processes in a successful state

where $\boxdot, Q_T$ and $\boxtimes, Q_T$ are any states reachable from $\boxdot, P_T$ and Q'_T such that $\boxdot, Q_T$ reachable from $\boxdot, P_T$ and $Q_T \rightsquigarrow Q'_T$. We have to show that for a well-tagged process $P_T \in \mathbf{WTP}$ and N_P the net generated from P_T the relation $\approx_{N_P}$ is a weak bisimulation.

1. $\boxdot, (A_T \div B_T)@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \approx_{N_P} F_1$

   We consider two cases depending on whether A_T succeeds or fails. In the first case we can only apply rule S-ACT-TAG leading to the state $\boxdot, [B_T]@\langle F_2, F_2, R_1, R_2, I_1, I_2\rangle$ with label A. The net can also do only one transition, according to its definition in Figure 30. Performing transition A leads to F_2. It is trivial to see that these are again in the relation $\approx_{N_P}$.

   For the case where A_T aborts in the smallstep semantics we can only apply rule F-ACT-TAG. We reach $\boxtimes, [\mathbf{nil}]@\langle F_1, F_1, R_2, R_2, I_1, I_2\rangle$. The corresponding net is the one shown in Figure 31. Also here there is only one possible transition namely from F_1 to $R_2 \oplus I_2$, which is exactly the marking related to the final state. Note that the transition K in the Petri Net is not observable, i.e., a τ as in the smallstep-semantics.

2. $\boxdot, (P_T; Q_T)@\mathcal{I} \approx_{N_{P;Q}} \mathbf{MP}_{\boxdot}(P_T)$

   Depending on the behaviour of P_T there are three different cases.

$$
\begin{array}{ll}
\mathbf{MP}_{\boxtimes}([C_T]@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle) = \mathbf{MP}(C_T) & \\
\mathbf{MP}_{\boxtimes}((P_T\$C_T)@\mathcal{I}) & = \mathbf{MP}_{\boxtimes}(P_T) \\
\mathbf{MP}_{\boxtimes}((P_T{\boxtimes}|_{\boxtimes}Q_T)@\mathcal{I}) & = \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxtimes}(Q_T) \\
\mathbf{MP}_{\boxtimes}((P_T{\boxdot}|_{\boxtimes}Q_T)@\mathcal{I}) & = \mathbf{MP}_{\boxdot}(P_T) \oplus \mathbf{MP}_{\boxtimes}(Q_T) \oplus PI_1 \\
\mathbf{MP}_{\boxtimes}((P_T{\boxtimes}|_{\boxdot}Q_T)@\mathcal{I}) & = \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \\
\mathbf{MP}_{\boxtimes}((P_T{\boxdot}|_{\boxdot}Q_T)@\mathcal{I}) & = \mathbf{MP}_{\boxdot}(P_T) \oplus PI_1 \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \\
 & \quad \text{if } P_T = P@\mathcal{I}' \wedge Q_T = Q@\mathcal{I}''
\end{array}
$$

Figure 83: Function $\mathbf{MP}$ for compensable processes in a failing state

SEQ-TAG By induction hypothesis $\boxdot, P_T \approx_{N_P} \mathbf{MP}_{\boxdot}(P_T)$. Since $\boxdot, P_T \xrightarrow{\lambda} \boxdot, P_T'$ it must be the case that $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} M'$ and $\boxdot, P_T' \approx_{N_P} M'$ and therefore $M' = \mathbf{MP}_{\boxdot}(P_T')$. Since $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxdot}(P_T')$ we have that $\mathbf{MP}_{\boxdot}(P_T; Q_T@\mathcal{I}) = \mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxdot}(P_T') = \mathbf{MP}_{\boxdot}(P_T'; Q_T@\mathcal{I}')$ and $\boxdot, (P_T'; Q_T)@\mathcal{I}' \approx_{N_{P;Q}} \mathbf{MP}_{\boxdot}(P_T'; Q_T@\mathcal{I}')$

A-SEQ-TAG By induction hypothesis $\boxdot, P_T \approx_{N_P} \mathbf{MP}_{\boxdot}(P_T)$. Since $\boxdot, P_T \xrightarrow{\lambda} \boxtimes, P_T'$ it must be the case that $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} M'$ and $\boxtimes, P_T' \approx_{N_P} M'$ and therefore $M' = \mathbf{MP}_{\boxtimes}(P_T') \oplus I_2$. Since $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxtimes}(P_T') \oplus I_2$ we have that $\mathbf{MP}_{\boxdot}(P_T; Q_T@\mathcal{I}) = \mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxtimes}(P_T') \oplus I_2$ and $\boxtimes, P_T' \approx_{N_{P;Q}} \mathbf{MP}_{\boxtimes}(P_T') \oplus I_2$.

The most interesting one is P_T finishing successfully. In the small-step semantics this means applying S-SEQ-TAG. Applying the hypothesis to P we know that $\boxdot, P_T \approx_{N_P} \mathbf{MP}_{\boxdot}(P_T)$. Since $\boxdot, P_T \xrightarrow{\lambda} \boxdot, P_T'$ it must be the case that $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} M'$ and $\boxdot, P_T' \approx_{N_P} M'$ and therefore $M' = \mathbf{MP}_{\boxdot}(P_T')$. Moreover by Corollary 7 we can conclude that either $P_T' \equiv [C_T]@\mathcal{I}$ and $cmp(P_T') = C$ or $P' \equiv [C_{1T}]_\sigma|_\sigma \ldots_\sigma|_\sigma [C_{nT}]@\mathcal{I}$ and $cmp(P') = \prod_{0 \le i \le n} C_{iT}$ (including the tags). Then for $\mathbf{MP}_{\boxdot}(P_T; Q_T@\mathcal{I}) = \mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxdot}(P_T') \xrightarrow{\tau^*} \mathbf{MP}_{\boxdot}(Q_T) = \mathbf{MP}_{\boxdot}((Q_T\$(cmp(P_T')))@\mathcal{I}')$ and $\boxdot, (Q_T\$(cmp(P_T')))@\mathcal{I}' \approx_{N_{P;Q}} \mathbf{MP}_{\boxdot}((Q_T\$(cmp(P_T')))@\mathcal{I}')$.

Note that for sequential composition $P; Q$ (as well as for compensation pairs) a state $\boxtimes, P; Q$ is not considered as such a state cannot

be reached according to Lemma 3.

3. $\boxdot, (P_T \$ C_T)@\mathcal{I} \approx_{N_{P\$C}} \mathbf{MP}_{\boxdot}(P_T)$ (this means $\sigma = \boxdot$ with reference to rules STEP, AS-STEP1, AS-STEP2)

   STEP-TAG like SEQ-TAG for $\sigma_2 = \boxdot$ and like A-SEQ-TAG for $\sigma_2 = \boxtimes$

   AS-STEP1-TAG From Corollary 7 it follows that $dn(P'_T) \rightarrow P'_T \equiv [C'_T]@\mathcal{I} \vee P' \equiv [C_{1T}]_\sigma|_\sigma \ldots _\sigma|_\sigma [C_{nT}]@\mathcal{I}$

   For $\sigma_2 = \boxdot$: Since $\mathcal{K}$ and $\mathcal{I}'$ are determined by P'_T, we can conclude $\mathbf{MP}_{\boxdot}(P_T \$ C_T@\mathcal{I}) = \mathbf{MP}_{\boxdot}(P_T) \stackrel{\hat{\lambda}}{\rightarrow} \mathbf{MP}_{\boxdot}(P'_T) = \mathbf{MP}_{\boxdot}([(cmp(P'_T); C_T)@\mathcal{K}]@\mathcal{I}')$ and

   $$\boxdot, [(cmp(P'_T); C_T)@\mathcal{K}]@\mathcal{I}' \approx_{N_{P\$C}} \mathbf{MP}_{\boxdot}([(cmp(P'_T); C_T)@\mathcal{K}]@\mathcal{I}')$$

   . In the case where P'_T is a parallel composition additional τ steps are necessary for transition join.

   For $\sigma_2 = \boxtimes$ similar, In the case where P'_T is a parallel composition additional τ steps are necessary for transition gc. Note that $done(P'_T)$ implies that any branch was either interrupted or aborted. (In fact this should not be possible, only if $\sigma = \boxtimes$.)

   AS-STEP2-TAG for $\sigma_2 = \boxdot$ like AS-STEP1-TAG

   For $\sigma_2 = \boxtimes$ the predicate $dn(cmp(P'_T))$ implies $cmp(P'_T) = \mathbf{nil}@\langle R, R\rangle$ or some parallel composition of this.

4. $\boxtimes, (P_T \$ C_T)@\mathcal{I} \approx_{N_{P\$C}} \mathbf{MP}_{\boxtimes}(P_T)$ (this means $\sigma = \sigma_2 = \boxtimes$ with reference to rules STEP, AS-STEP1, AS-STEP2) Cases similar to commit case.

5. $\boxtimes, [C_T]@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle \approx_{N_{[C]}} \mathbf{MP}(C_T) \oplus I_2$

   As a consequence of Theorem 14.

   In the commit case both net and smallstep semantics cannot move.

6. $\boxdot, (P_{T\boxdot}|_{\boxdot} Q_T)@\mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxdot}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus MEX_{\mathcal{I}}$

   Considering PAR-L-TAG the assumption implies that $\sigma_1 = \sigma_2 = \boxdot$. By induction hypothesis $\boxdot, P_T \approx_{N_P} \mathbf{MP}_{\boxdot}(P_T)$. In the first case $\boxdot, P_T \stackrel{\lambda}{\rightarrow} \boxdot, P'_T$, then it must be the case that $\mathbf{MP}_{\boxdot}(P_T) \stackrel{\hat{\lambda}}{\rightarrow} M'$ and $\boxdot, P'_T \approx_{N_P} M'$ and therefore $M' = \mathbf{MP}_{\boxdot}(P'_T)$. Since $\mathbf{MP}_{\boxdot}(P_T) \stackrel{\hat{\lambda}}{\rightarrow} \mathbf{MP}_{\boxdot}(P'_T)$ we have $\mathbf{MP}_{\boxdot}(P_{T\boxdot}|_{\boxdot} Q_T@\mathcal{I}) = \mathbf{MP}_{\boxdot}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus$

$MEX_{\mathcal{I}} \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxdot}(P'_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus MEX_{\mathcal{I}} = \mathbf{MP}_{\boxdot}(P'_{T\boxdot}|_{\boxdot} Q_T @ \mathcal{I})$
and $\boxdot, (P'_{T\boxdot}|_{\boxdot} Q_T) @ \mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxdot}(P'_{T\boxdot}|_{\boxdot} Q_T @ \mathcal{I})$

In the second case $\boxdot, P_T \xrightarrow{\lambda} \boxtimes, P'_T$, then it must be the case that $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} M'$ and $\boxtimes, P'_T \approx_{N_P} M'$ and therefore $M' = \mathbf{MP}_{\boxtimes}(P'_T) \oplus PI_2$. Since $\mathbf{MP}_{\boxdot}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxtimes}(P'_T) \oplus PI_2$ we have that

$$\begin{aligned}
&\mathbf{MP}_{\boxdot}(P_{T\boxdot}|_{\boxdot} Q_T @ \mathcal{I}) = \mathbf{MP}_{\boxdot}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus MEX_{\mathcal{I}} \\
&\xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxtimes}(P'_T) \oplus PI_2 \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus MEX_{\mathcal{I}} \\
&\xrightarrow{\tau} \mathbf{MP}_{\boxtimes}(P'_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \oplus I_2 \\
&= \mathbf{MP}_{\boxtimes}(P'_{T\boxtimes}|_{\boxdot} Q_T @ \mathcal{I}) \oplus I_2
\end{aligned}$$

and
$\boxtimes, (P'_{T\boxtimes}|_{\boxdot} Q_T) @ \mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxtimes}(P'_{T\boxtimes}|_{\boxdot} Q_T @ \mathcal{I}) \oplus I_2$
The symmetric case where Q moves (PAR-R-TAG) is analogous.

7. $\boxtimes, (P_{T\boxtimes}|_{\boxtimes} Q_T) @ \mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxtimes}(Q_T) \oplus I_2$
Considering PAR-L-TAG the assumption implies that $\sigma_1 = \sigma_2 = \boxtimes$. By induction hypothesis $\boxtimes, P_T \xrightarrow{\lambda} \boxtimes, P'_T$, then it must be the case that $\mathbf{MP}_{\boxtimes}(P_T) \oplus PI_2 \xrightarrow{\hat{\lambda}} M'$ and $\boxtimes, P'_T \approx_{N_P} M'$ and therefore $M' = \mathbf{MP}_{\boxtimes}(P'_T) \oplus PI_2$. Since $\mathbf{MP}_{\boxtimes}(P_T) \xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxtimes}(P'_T)$ we have that

$$\begin{aligned}
&\mathbf{MP}_{\boxtimes}(P_{T\boxtimes}|_{\boxtimes} Q_T @ \mathcal{I}) = \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxtimes}(Q_T) \oplus I_2 \\
&\xrightarrow{\hat{\lambda}} \mathbf{MP}_{\boxtimes}(P'_T) \oplus \mathbf{MP}_{\boxtimes}(Q_T) \oplus I_2 \\
&= \mathbf{MP}_{\boxtimes}(P'_{T\boxtimes}|_{\boxtimes} Q_T @ \mathcal{I}) \oplus I_2
\end{aligned}$$

and
$\boxtimes, (P'_{T\boxtimes}|_{\boxtimes} Q_T) @ \mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxtimes}(P'_{T\boxtimes}|_{\boxtimes} Q_T @ \mathcal{I}) \oplus I_2$

8. $\boxtimes, (P_{T\boxtimes}|_{\boxdot} Q_T) @ \mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \oplus I_2$
For PAR-L-TAG similar to previous case, PAR-R-TAG similar to case 6.

INT-TAG: By the induction hypothesis $\boxdot, Q_T \approx_{N_Q} \mathbf{MP}_{\boxdot}(Q_T)$ and $\boxtimes, Q'_T \approx_{N_Q} \mathbf{MP}_{\boxtimes}(Q'_T)$ for any $Q_T \rightsquigarrow Q'_T$. We have to show by induction over Q_T that $\mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \xrightarrow{\tau^*} \mathbf{MP}_{\boxtimes}(Q'_T)$, considering the different cases for the "extract" predicate. Most cases can be

solved quite trivially by either taking transitions x_1 or x_2 included in every net or by applying the induction hypothesis. The most interesting case is where $Q_T = Q_{1T}; Q_{2T}$ and $Q_{1T} = PP_T|QQ_T$ is a parallel composition. Note that as Q_T is in a commit state also each branch in Q_{1T} is in a commit state. Thus

$$\begin{aligned}&\mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 = \mathbf{MP}_{\boxdot}(PP_T) \oplus \mathbf{MP}_{\boxdot}(QQ_T) \oplus MEX_{\mathcal{I}} \oplus QI_1\\ &\xrightarrow{\mathsf{i}_{\mathsf{in}}} \mathbf{MP}_{\boxdot}(PP_T) \oplus \mathbf{MP}_{\boxdot}(QQ_T) \oplus PPI_1 \oplus QQI_1\\ &= \mathbf{MP}_{\boxtimes}(PP_{T\boxdot}|_{\boxdot}QQ_T) = \mathbf{MP}_{\boxtimes}(Q_{1T})\end{aligned}$$

taking the transition i_{in} in the net. In the case where Q_T is a parallel composition the net first fires transition i_{in} and then we can apply the induction hypothesis.

Now since $\mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \xrightarrow{\tau^*} \mathbf{MP}_{\boxtimes}(Q'_T)$ then

$$\begin{aligned}&\boxtimes, (P_{T\boxtimes}|_{\boxdot}Q_T)@\mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxdot}(Q_T) \oplus QI_1 \oplus I_2\\ &\xrightarrow{\tau^*} \mathbf{MP}_{\boxtimes}(P_T) \oplus \mathbf{MP}_{\boxtimes}(Q'_T) \oplus I_2\\ &= \mathbf{MP}_{\boxtimes}(P_{T\boxtimes}|_{\boxtimes}Q'_T) \oplus I_2 \approx_{N_{P|Q}} \boxtimes, (P_{T\boxtimes}|_{\boxtimes}Q'_T)@\mathcal{I}\end{aligned}$$

9. $\boxtimes, (P_{T\boxdot}|_{\boxdot}Q_T)@\mathcal{I} \approx_{N_{P|Q}} \mathbf{MP}_{\boxtimes}((P_{T\boxdot}|_{\boxdot}Q_T)@\mathcal{I})$

   For PAR-R-TAG/PAR-L-TAG similar to case 6, for INT-R-TAG/INT-L-TAG proceed as in the previous case.

□

Appendix B

Proofs for Chapter 6

B.1 Proof of Theorem 12

In the following we let $null$ denote the null distance (over any X), i.e., $null(x) \triangleq 0$ for any $x \in X$. We will use the operation $\oplus$ to denote the union of two distances between states (over the same X), formally defined as:

$$(d_1 \oplus d_2)(x) \triangleq d_1(x) + d_2(x)$$

Then we can state the following lemma

Lemma 6. *Let $s, s', s'' \in State(V)$. The following equality holds:*

$$s\backslash_X s' \oplus s'\backslash_X s'' = s\backslash_X s''$$

Proof. Trivially, for any $x \in X$:

$$\begin{aligned}(s\backslash_X s' \oplus s'\backslash_X s'')(x) &= (s\backslash_X s')(x) + (s'\backslash_X s'')(x) \\ &= s'(x) - s(x) + s''(x) - s'(x) \\ &= s''(x) - s(x) = (s\backslash_X s'')(x)\end{aligned}$$

□

Theorem 15 (from page 150)**.** *Let X be a set of integer variables and δ a serializable compensable program where every compensation pair is correct over X, then δ is correct over X.*

Proof. We proceed by induction on the structure of δ.

1. $\delta = a \div \overline{a}$.
   For any $(\tau, \overline{\tau}) \in \rho_c(a \div \overline{a})$ with $\neg E(\tau)$ we conclude by applying the hypothesis that the compensation pair $a \div \overline{a}$ is correct over X. For any failed trace $(\tau, \overline{\tau}) \in \rho_c(a \div \overline{a})$ with $E(\tau)$, we have that $\tau = s\, a\, s$ for some state s such that $s \models E(a)$. Furthermore, the compensation trace for a failed basic activity is empty. It is easy to see that $\mathit{first}(\tau)\backslash_X \mathit{last}(\tau) = \mathit{null}$, which concludes the proof for this case.

2. $\delta = \delta_1 + \delta_2$.
   Follows trivially by applying the induction hypothesis on δ_1 and δ_2.

3. $\delta = \delta_1 \,;\, \delta_2$.
   We need to distinguish two cases, according to the definition of $\rho_c(\delta_1 \,;\, \delta_2)$.

   - For any pair $(\tau \circ \nu, \overline{\nu} \circ \overline{\tau}) \in \rho_c(\delta_1 \,;\, \delta_2)$ such that $(\tau, \overline{\tau}) \in \rho_c(\delta_1)$, $(\nu, \overline{\nu}) \in \rho_c(\delta_2)$, and $\neg E(\tau)$, then we want to prove that if $\mathit{closed}(\tau \circ \nu)$ and $\mathit{closed}(\overline{\nu} \circ \overline{\tau})$ then $\mathit{first}(\tau \circ \nu)\backslash_X \mathit{last}(\tau \circ \nu) = \mathit{last}(\overline{\nu} \circ \overline{\tau})\backslash_X \mathit{first}(\overline{\nu} \circ \overline{\tau})$.
     As we consider closed traces we can conclude that also $\mathit{closed}(\tau)$ and $\mathit{closed}(\nu)$ hold with $\mathit{last}(\tau) = \mathit{first}(\nu)$ and the same holds for the compensation. Thus we can apply the induction hypothesis getting $\mathit{first}(\tau)\backslash_X \mathit{last}(\tau) = \mathit{last}(\overline{\tau})\backslash_X \mathit{first}(\overline{\tau})$ and $\mathit{first}(\nu)\backslash_X \mathit{last}(\nu) = \mathit{last}(\overline{\nu})\backslash_X \mathit{first}(\overline{\nu})$. We build the union of these two sets, *i.e.*,
     $\mathit{first}(\tau)\backslash_X \mathit{last}(\tau) \oplus \mathit{first}(\nu)\backslash_X \mathit{last}(\nu) = \mathit{last}(\overline{\nu})\backslash_X \mathit{first}(\overline{\nu}) \oplus \mathit{last}(\overline{\tau})\backslash_X \mathit{first}(\overline{\tau})$.
     As $\mathit{last}(\tau) = \mathit{first}(\nu)$ and $\mathit{last}(\overline{\nu}) = \mathit{first}(\overline{\tau})$ we can conclude by Lemma 6 that $\mathit{first}(\tau)\backslash_X \mathit{last}(\nu) = \mathit{last}(\overline{\tau})\backslash_X \mathit{first}(\overline{\nu})$ which is equivalent to $\mathit{first}(\tau \circ \nu)\backslash_X \mathit{last}(\tau \circ \nu) = \mathit{last}(\overline{\nu} \circ \overline{\tau})\backslash_X \mathit{first}(\overline{\nu} \circ \overline{\tau})$.
   - For any $(\tau, \overline{\tau}) \in \rho_c(\delta_1 \,;\, \delta_2)$ such that $(\tau, \overline{\tau}) \in \rho(\delta_1)$ and $E(\tau)$, then the result follows immediately from the induction hypothesis on δ_1.

4. $\delta = \delta_1 \parallel \delta_2$ (with δ_1, δ_2 serializable)[1]
   For any $(\tau, \overline{\tau}) \in \rho_c(\delta_1 \parallel \delta_2)$ with $(\nu, \overline{\nu}) \in \rho_c(\delta_1)$, $(\mu, \overline{\mu}) \in \rho_c(\delta_2)$

[1]Note that for simplicity we give the proof for the binary parallel composition, but the more general case of n-ary parallel composition of a serializable set of processes is along the same line.

$\tau \in \nu \parallel \mu, \overline{\tau} \in \overline{\nu} \parallel \overline{\mu}$, we need to show that if $closed(\tau)$ and $closed(\overline{\tau})$ then $first(\tau)\backslash_X last(\tau) = last(\overline{\tau})\backslash_X first(\overline{\tau})$.

Note that in general neither $closed(\nu)$ and $closed(\overline{\nu})$ nor $closed(\mu)$ and $closed(\overline{\mu})$ hold, because the interleavings ν and $\overline{\nu}$ are defined independently of this.

According to serializability there exist traces $(\nu', \overline{\nu}') \in \rho_c(\delta_1)$ and $(\mu', \overline{\mu}') \in \rho_c(\delta_2)$ with several different possibilities for a sequential representation (though at least one holds). Without loss of generality we assume that $\tau \bowtie (\nu'\mu')$ and $\overline{\tau} \bowtie (\overline{\mu}'\,\overline{\nu}')$. (The other representations can be treated similarly.)

From $closed(\nu'\,\mu')$ we know that $closed(\nu')$, $closed(\mu')$ and $last(\nu') = first(\mu')$. The same holds for $closed(\overline{\mu}'\,\overline{\nu}')$. Thus we can apply the induction hypothesis. We obtain $first(\nu')\backslash_X last(\nu') = last(\overline{\nu}')\backslash_X first(\overline{\nu}')$ and $first(\mu')\backslash_X last(\mu') = last(\overline{\mu}')\backslash_X first(\overline{\mu}')$. As for the sequential case we build the union of the two sets $first(\nu')\backslash_X last(\nu') \oplus first(\mu')\backslash_X last(\mu') = last(\overline{\mu}')\backslash_X first(\overline{\mu}') \oplus last(\overline{\nu}')\backslash_X first(\overline{\nu}')$. From the previous equivalences and Lemma 6 we obtain $first(\nu')\backslash_X last(\mu') = last(\overline{\mu}')\backslash_X first(\overline{\nu}')$ which is equivalent to $first(\nu'\,\mu')\backslash_X last(\nu'\,\mu') = last(\overline{\mu}'\,\overline{\nu}')\backslash_X first(\overline{\mu}'\,\overline{\nu}')$. From the equivalences for serializability we can conclude $first(\tau)\backslash_X last(\tau) = last(\overline{\tau})\backslash_X first(\overline{\tau})$.

5. $\delta = \delta_1^*$
   By the induction hypothesis we know that the theorem holds for δ_1, we have to show that it holds also for any $(\tau, \overline{\tau}) \in \rho_c(\delta)$ in the iteration. We do this by induction on the depth of recursion.

   In the base case for $([\![], [\![]) \in \rho_c(\delta)$ obviously the theorem holds.

   In the induction step we must show $first(\tau)\backslash_X last(\tau) = last(\overline{\tau})\backslash_X first(\overline{\tau})$ for $(\tau, \overline{\tau}) \in \rho_c(\delta_1; \delta_1^*)$. We can apply the induction hypothesis of the theorem to δ_1 and the induction hypothesis over the number of iterations to δ_1^*. Then the proof is similar to the case of sequential composition.

□

B.2 Proof of Propositions 5 and 6

Proposition 5 (from page 129)**.** *The following are valid formulas of first order dynamic logic (see also [Har79]).*

$$
\begin{array}{lcllcl}
\langle\alpha\rangle(\varphi \vee \psi) & \leftrightarrow & \langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi & [\alpha](\varphi \wedge \psi) & \leftrightarrow & [\alpha]\varphi \wedge [\alpha]\psi \\
\langle\alpha\rangle\varphi \wedge \langle\alpha\rangle\psi & \rightarrow & \langle\alpha\rangle(\varphi \wedge \psi) & [\alpha](\varphi \rightarrow \psi) & \rightarrow & ([\alpha]\varphi \rightarrow [\alpha]\psi) \\
\langle\alpha\rangle(\varphi \wedge \psi) & \rightarrow & \langle\alpha\rangle\varphi \wedge \langle\alpha\rangle\psi & [\alpha]\varphi \vee [\alpha]\psi & \rightarrow & [\alpha](\varphi \vee \psi) \\
\langle\alpha \,+\, \beta\rangle\varphi & \leftrightarrow & \langle\alpha\rangle\varphi \vee \langle\beta\rangle\varphi & [\alpha \,+\, \beta]\varphi & \leftrightarrow & [\alpha]\varphi \wedge [\beta]\varphi \\
\langle\alpha^*\rangle\varphi & \leftrightarrow & \varphi \vee \langle\alpha\,;\alpha^*\rangle\varphi & [\alpha^*]\varphi & \leftrightarrow & \varphi \wedge [\alpha\,;\alpha^*]\varphi \\
\langle\varphi?\rangle\psi & \leftrightarrow & \varphi \wedge \psi & [\varphi?]\psi & \leftrightarrow & \varphi \rightarrow \psi \\
\langle\alpha\,;\beta\rangle\varphi & \not\leftrightarrow & \langle\alpha\rangle\langle\beta\rangle\varphi & [\alpha\,;\beta]\varphi & \not\leftrightarrow & [\alpha][\beta]\varphi
\end{array}
$$

Proof. The formulas for program possibility and necessity involving choice, iteration and test operators are valid, since the interpretation of these operators follows closely the standard interpretation of first order dynamic logic.

In the following we present counter examples for the formulas for sequential composition that do not hold with the new interpretation.

1. $\langle\alpha\,;\beta\rangle\varphi \not\rightarrow \langle\alpha\rangle\langle\beta\rangle\varphi$

   Take programs:

   - α such that for every state s we have $s \models F(\alpha)$ and $s \models \langle\alpha\rangle\varphi$,
   - β such that for every state s' we have $s' \not\models \langle\beta\rangle\varphi$ and $\mathit{last}(\rho(\alpha)) \subseteq \mathit{first}(\rho(\beta))$.

   Because every trace of α fails, we have that $\rho(\alpha\,;\beta) = \rho(\alpha)$. Take a state s. Since $s \models \langle\alpha\rangle\varphi$ we can conclude that $s \models \langle\alpha\,;\beta\rangle\varphi$. From construction of program β we have that every trace of α can be composed with a trace of β. Since for every state s' we have that $s' \not\models \langle\beta\rangle\varphi$, we can conclude that $s \not\models \langle\alpha\rangle\langle\beta\rangle\varphi$.

2. $\langle\alpha\rangle\langle\beta\rangle\varphi \not\rightarrow \langle\alpha\,;\beta\rangle\varphi$

   Take a program:

   - α such that for every state s we have $s \models F(\alpha)$ and $s \not\models \langle\alpha\rangle\varphi$,
   - β such that for every state s' we have $s' \models \langle\beta\rangle\varphi$ and $\mathit{last}(\rho(\alpha)) \subseteq \mathit{first}(\rho(\beta))$.

Take a state s. We know that $s \models \langle\alpha\rangle\langle\beta\rangle\varphi$. Because α always fails, we know that $\rho(\alpha\,;\beta) = \rho(\alpha)$. Since $s \not\models \langle\alpha\rangle\varphi$, we can conclude that $s \not\models \langle\alpha\,;\beta\rangle\varphi$.

3. $[\alpha\,;\beta]\varphi \not\to [\alpha][\beta]\varphi$

   Take programs:

   - α such that for every state s we have $s \models F(\alpha)$ and $s \models [\alpha]\varphi$,
   - β such that for every state s' we have $s' \models [\beta]\neg\varphi$ and $last(\rho(\alpha)) \subseteq first(\rho(\beta))$.

   Because every trace of α fails, we have that $\rho(\alpha\,;\beta) = \rho(\alpha)$. Take a state s. Since $s \models [\alpha]\varphi$ we can conclude that $s \models [\alpha\,;\beta]\varphi$. From construction of program β we have that every trace of α can be composed with a trace of β. Since $s' \models [\beta]\neg\varphi$ for all s', we can conclude that $s \models [\alpha][\beta]\neg\varphi$.

4. $[\alpha][\beta]\varphi \not\to [\alpha\,;\beta]\varphi$.

   A counter example can be defined as in the previous cases.

□

Proposition 6 (from page 142)**.** *Let α, β be two serializable programs. The following are valid formulas in the presented dynamic logic:*

$\langle\alpha \parallel \beta\rangle\varphi$	$\leftrightarrow$	$\langle\alpha\rangle\langle\beta\rangle\varphi \vee \langle\beta\rangle\langle\alpha\rangle\varphi$	$[\alpha \parallel \beta]\varphi$	$\leftrightarrow$	$[\alpha][\beta]\varphi \wedge [\beta][\alpha]\varphi$
$\mathsf{S}(\alpha + \beta)$	$\leftrightarrow$	$\mathsf{S}(\alpha) \wedge \mathsf{S}(\beta)$	$\mathsf{S}(\alpha^*)$	$\leftrightarrow$	$\mathit{true} \wedge \mathsf{S}(\alpha;\alpha^*)$
$\mathsf{S}(\alpha \parallel \beta)$	$\rightarrow$	$\mathsf{S}(\alpha\,;\beta) \wedge \mathsf{S}(\beta\,;\alpha)$			
$\mathsf{S_W}(\alpha + \beta)$	$\leftrightarrow$	$\mathsf{S_W}(\alpha) \vee \mathsf{S_W}(\beta)$	$\mathsf{S_W}(\alpha^*)$	$\leftrightarrow$	$\mathit{true} \vee \mathsf{S_W}(\alpha;\alpha^*)$
$\mathsf{S_W}(\alpha \parallel \beta)$	$\rightarrow$	$\mathsf{S_W}(\alpha\,;\beta) \vee \mathsf{S_W}(\beta\,;\alpha)$	$\mathsf{S_W}(\alpha \parallel \beta)$	$\rightarrow$	$\mathsf{S_W}(\alpha) \wedge \mathsf{S_W}(\beta)$

Proof.

1. (a) $\langle\alpha \parallel \beta\rangle\varphi \to \langle\alpha\rangle\langle\beta\rangle\varphi \vee \langle\beta\rangle\langle\alpha\rangle\varphi$

   From the hypothesis we know that it exists a trace $\tau \in closure(\alpha \parallel \beta)$ such that $last(\tau) \models \varphi$. Because α and β are serializable programs, there exists traces $\nu \in \rho(\alpha)$ and $\mu \in \rho(\beta)$ such that $\tau \bowtie \nu\mu$ or $\tau \bowtie \mu\nu$. Assuming that $\tau \bowtie \nu\mu$, then $last(\nu\mu) \models \varphi$. So, it can be concluded that $\langle\alpha\rangle\langle\beta\rangle\varphi$.

(b) $\langle\alpha\rangle\langle\beta\rangle\varphi \vee \langle\beta\rangle\langle\alpha\rangle\varphi \rightarrow \langle\alpha \parallel \beta\rangle\varphi$
Follows immediately from the hypothesis and interpretation of parallel composition.

2. (a) $[\alpha \parallel \beta]\varphi \rightarrow [\alpha][\beta]\varphi \wedge [\beta][\alpha]\varphi$
Similar to the proof of the case 1a.

(b) $[\alpha][\beta]\varphi \wedge [\beta][\alpha]\varphi \rightarrow [\alpha \parallel \beta]\varphi$
We want to prove that for every trace $\tau \in \mathit{closure}(\alpha \parallel \beta)$ is such that $\mathit{last}(\tau) \models \varphi$. Because α and β are serializable programs, there exists traces $\nu \in \rho(\alpha)$ and $\mu \in \rho(\beta)$ such that $\tau \bowtie \nu\mu$ or $\tau \bowtie \mu\nu$. From the hypothesis we know that $\mathit{last}(\nu\mu) \models \varphi$ or $\mathit{last}(\mu\nu) \models \varphi$. Therefore, we can conclude that $\mathit{last}(\tau) \models \varphi$.

3. $\mathsf{S}(\alpha + \beta) \leftrightarrow \mathsf{S}(\alpha) \wedge \mathsf{S}(\beta)$
Follows immediately from the interpretation of choice.

4. $\mathsf{S}(\alpha^*) \leftrightarrow \mathit{true} \wedge \mathsf{S}(\alpha; \alpha^*)$

5. $\mathsf{S}(\alpha \parallel \beta) \rightarrow \mathsf{S}(\alpha\,;\beta) \wedge \mathsf{S}(\beta\,;\alpha)$

Follows immediately from the hypothesis and interpretation of parallel composition.

6. $\mathsf{S_W}(\alpha + \beta) \leftrightarrow \mathsf{S_W}(\alpha) \vee \mathsf{S_W}(\beta)$

Follows immediately from the interpretation of choice.

7. $\mathsf{S_W}(\alpha^*) \leftrightarrow \mathit{true} \vee \mathsf{S_W}(\alpha; \alpha^*)$

8. $\mathsf{S_W}(\alpha \parallel \beta) \rightarrow \mathsf{S_W}(\alpha\,;\beta) \vee \mathsf{S_W}(\beta\,;\alpha)$

From the hypotheses we have that it exists a trace $\tau \in \mathit{closure}(\alpha \parallel \beta)$ such that $\neg E(\tau)$. Because α and β are serializable, there must exist traces $\nu \in \rho(\alpha)$ and $\mu \in \rho(\beta)$ such that either $\tau \bowtie \nu\mu$ or $\tau \bowtie \mu\nu$. Therefore, $\neg E(\nu\mu)$ or $\neg E(\mu\nu)$. So we can conclude that $\mathsf{S_W}(\alpha\,;\beta)$ or $\mathsf{S_W}(\beta\,;\alpha)$.

9. $\mathsf{S_W}(\alpha \parallel \beta) \rightarrow \mathsf{S_W}(\alpha) \wedge \mathsf{S_W}(\beta)$

It follows immediately from formulas $\mathsf{S_W}(\alpha \parallel \beta) \rightarrow \mathsf{S_W}(\alpha\,;\beta) \wedge \mathsf{S_W}(\beta\,;\alpha)$ and $\mathsf{S_W}(\alpha\,;\beta) \rightarrow \mathsf{S_W}(\alpha) \wedge \mathsf{S_W}(\beta)$.

□

Proposition 7. *Let α, β be two strong serializable programs. The following are valid formulas in the presented dynamic logic:*

$$\mathsf{S}(\alpha \parallel \beta) \quad \leftrightarrow \quad \mathsf{S}(\alpha) \wedge \mathsf{S}(\beta) \qquad \mathsf{F}(\alpha \parallel \beta) \quad \leftrightarrow \quad \mathsf{F}(\alpha) \vee \mathsf{F}(\beta)$$

Proof. For α, β strong serializable programs we need to prove that $\mathsf{S}(\alpha) \wedge \mathsf{S}(\beta) \rightarrow \mathsf{S}(\alpha \parallel \beta)$. The implication on the other direction follows from the interpretation of parallel composition.

Lets assume that $\mathsf{S}(\alpha)$ and $\mathsf{S}(\beta)$, and it exists a trace $\tau \in \mathit{closure}(\alpha \parallel \beta)$ such that $E(\tau)$. Because α and β are serializable, there must exist traces $\nu \in \rho(\alpha)$ and $\mu \in \rho(\beta)$ such that either $\tau \bowtie \nu\mu$ or $\tau \bowtie \mu\nu$. Then we can conclude that either $E(\nu\mu)$ or $E(\mu\nu)$. Therefore, an error was raised in ν, μ, or both. either $E(\nu)$ or $E(\mu)$. Which contradicts our hypothesis that $\mathsf{S}(\alpha)$ and $\mathsf{S}(\beta)$. □

B.3 Proof of Theorem 13

We start by giving a stronger version of correctness and prove the theorem with strong correctness. We then state a lemma that implies the original Theorem 13.

Definition 41 (Strong Correctness)**.** *Let X be a set of integer variables. A compensable program δ has a strong correct compensation over X if for all traces of pairs $\pi \in \rho_i(\delta)$ with $\pi = \lambda_1\lambda_2 \ldots \lambda_n$ then then for all $\lambda_i = \langle \tau_i, \overline{\tau_i} \rangle$, $i \in \{1, \ldots, n\}$ it holds that $\mathit{first}(\tau_i) \backslash_X \mathit{last}(\tau_i) = \mathit{last}(\overline{\tau_i}) \backslash_X \mathit{first}(\overline{\tau_i})$.*

The new definition states that every pair in a trace has to be correct, then the program has a strong correct compensation. Note that we do not require closure. Now we restate the theorem.

Theorem 16. *Let X be a set of integer variables and δ a compensable program with interrupt where every compensation pair is correct over X, then δ is strong correct over X.*

Proof. By structural induction over compensable program δ.

1. $\delta = a \div \overline{a}$

   This part of the proof is equivalent to the one of Theorem 12 (as compensation pairs are interpreted by single pairs).

2. $\delta = \delta_1 + \delta_2$

   Follows trivially by applying the induction hypothesis to δ_1 and δ_2.

3. $\delta = \delta_1 ; \delta_2$

   We show that for any trace $\pi_1 \circ \pi_2 \in \delta$ with $\pi_1 \in \delta_1$ and $\pi_2 \in \delta_2$ the theorem holds. The composition operator $\circ$ is defined depending on $E(\pi_1)$. If the error formula is false the two traces are combined. By applying the induction hypothesis we know that π_1 and π_2 consist only of correct pairs, thus their combination also consists of only correct pairs. If the error formula is true only π_1 is returned. Using the induction hypothesis this case is trivially true.

4. $\delta = \delta_1 \parallel \delta_2$

   Let $\pi_1 \in \rho_i(\delta_1)$ and $\pi_2 \in \rho_i(\delta_2)$. We want to show for every case of the interleaving of the two traces that the resulting trace consists only of correct pairs using the induction hypothesis for both π_1 and π_2.

   Note that for any case of the interleaving only pairs of π_1 and π_2 are reused, no new pairs are introduced. As π_1 and π_2 contain only correct pairs also any combination consists only of correct pairs (including prefixes). This inference will lead to the theorem.

5. $\delta = \delta_1^*$

   By the induction hypothesis we know that the theorem holds for δ_1, we have to show that it holds also for any $\pi \in \rho_i(\delta)$ in the iteration. We do this by induction on the depth of recursion.

   In the base case for $\langle [\![]\!], [\![]\!] \rangle \in \rho_i(\delta)$ obviously the theorem holds.

   In the induction step we have to show that each pair is correct in the trace of pairs $\pi \in \rho_i(\delta_1; \delta_1^*)$. We can apply the induction hypothesis of the theorem to δ_1 and the induction hypothesis over the number of iterations to δ_1^*. Then the proof is similar to sequential composition.

□

Now we state a lemma that will lead to our final result:

Lemma 7. *Let $\pi = \lambda_1 \dots \lambda_n$ be a trace of pairs. If for each $\lambda_i = \langle \tau_i, \overline{\tau_i} \rangle$, $i \in \{1, \dots, n\}$, it holds that $first(\tau_i) \backslash_X last(\tau_i) = last(\overline{\tau_i}) \backslash_X first(\overline{\tau_i})$ then for $\mu = pr_l(\pi)$ with $closed(\mu)$ and $\nu = rpr_r(\pi)$ with $closed(\nu)$ it holds that*

$$first(\mu) \backslash_X last(\mu) = last(\nu) \backslash_X first(\nu).$$

Proof. By induction on the length of π. In the base case, the empty trace, the lemma is trivially true.

For the induction step let $\pi = \pi' \lambda$ with $\lambda = \langle \tau, \overline{\tau} \rangle$ such that

$$first(\tau) \backslash_X last(\tau) = last(\overline{\tau}) \backslash_X first(\overline{\tau}).$$

Moreover from the induction hypothesis we know that there are $\mu = pr_l(\pi')$ with $closed(\mu)$ and $\nu = rpr_r(\pi')$ with $closed(\nu)$ such that $first(\mu) \backslash_X last(\mu) = last(\nu) \backslash_X first(\nu)$.

We build the union of the two distances for the forward and backward flow, *i.e.*, we compute separately the two distance $first(\tau) \backslash_X last(\tau) \oplus first(\mu) \backslash_X last(\mu)$ and $last(\nu) \backslash_X first(\nu) \oplus last(\overline{\tau}) \backslash_X first(\overline{\tau})$. From the assumption that $closed(\mu\tau)$ and $closed(\overline{\tau}\nu)$ as $pr_l(\pi) = \mu\tau$ and $rpr_r(\pi) = \overline{\tau}\nu$ we know that $last(\mu) = first(\tau)$ and $last(\overline{\tau}) = first(\nu)$. Using Lemma 6 we can conclude that $first(\tau) \backslash_X last(\mu) = last(\nu) \backslash_X first(\overline{\tau})$. This proves the lemma. □

Theorem 17 (from page 160)**.** *Let X be a set of integer variables and δ a compensable program with interrupt where every compensation pair is correct over X, then δ is correct over X.*

Proof. From Theorem 16 we know that a compensable program δ is strong correct over X if every compensation pair is correct over X. Thus every trace in $\rho_i(\delta)$ is of the form $\pi = \lambda_1 \dots \lambda_n$ where for all $\lambda_i = \langle \tau_i, \overline{\tau_i} \rangle$, $i \in \{1, \dots, n\}$ it holds that $first(\tau_i) \backslash_X last(\tau_i) = last(\overline{\tau_i}) \backslash_X first(\overline{\tau_i})$. Applying Lemma7 we get that for $\mu = pr_l(\pi)$ with $closed(\mu)$ and $\nu = rpr_r(\pi)$ with $closed(\nu)$ it holds that $first(\mu) \backslash_X last(\mu) = last(\nu) \backslash_X first(\nu)$ which proves the theorem. □

References

[AB09] Andrea Asperti and Nadia Busi. Mobile Petri nets. *Mathematical Structures in Computer Science*, 19(6):1265–1278, 2009.

[ABDZ07] Lucia Acciai, Michele Boreale, and Silvano Dal-Zilio. A Concurrent Calculus with Atomic Transactions. In *ESOP*, pages 48–63, 2007.

[Act13] ActiveVOS. `http://www.activevos.com/`, June 2013.

[Apa13] Apache ODE. `http://ode.apache.org/`, June 2013.

[AR06] Baver Acu and Wolfgang Reisig. Compensation in Workflow Nets. In *ICATPN*, pages 65–83, 2006.

[BBF+05] Roberto Bruni, Michael J. Butler, Carla Ferreira, C. A. R. Hoare, Hernán Melgratti, and Ugo Montanari. Comparing Two Approaches to Compensable Flow Composition. In *CONCUR*, pages 383–397, 2005.

[BF00] Michael J. Butler and Carla Ferreira. A Process Compensation Language. In *IFM*, pages 61–76, 2000.

[BF04] Michael J. Butler and Carla Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In *COORDINATION*, pages 87–104, 2004.

[BFK12] Roberto Bruni, Carla Ferreira, and Anne Kersten Kauer. First-Order Dynamic Logic for Compensable Processes. In *COORDINATION*, pages 104–121, 2012.

[BH00] Martin Berger and Kohei Honda. The Two-Phase Commitment Protocol in an Extended π-Calculus. *Electronic Notes in Theoretical Computer Science*, 39(1):21 – 46, 2000. EXPRESS.

[BHF04] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A Trace Semantics for Long-Running Transactions. In *25 Years Communicating Sequential Processes*, pages 133–150, 2004.

[BK12] Roberto Bruni and Anne Kersten Kauer. LTS Semantics for Compensation-based Processes. In *TGC*, pages 112–128, 2012.

[BKLS10] Roberto Bruni, Anne Kersten, Ivan Lanese, and Giorgio Spagnolo. A New Strategy for Distributed Compensations with Interruption in Long-Running Transactions. In *WADT*, pages 42–60, 2010.

[BM03] Roberto Bruni and José Meseguer. Generalized Rewrite Theories. In *ICALP*, pages 252–266, 2003.

[BM06] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[BM07a] Maria Grazia Buscemi and Hernán C. Melgratti. Transactional Service Level Agreement. In *TGC*, pages 124–139, 2007.

[BM07b] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP*, pages 18–32, 2007.

[BMM04a] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Flat Committed Join in Join. *Electronic Notes in Theoretical Computer Science*, 104:39 – 59, 2004.

[BMM04b] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Nested Commits for Mobile Calculi: Extending Join. In *IFIP TCS*, pages 563–576, 2004.

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

[BMM11] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. cJoin: Join with communicating transactions. *Mathematical Structures in Computer Science*, page 46 pages, 2011. To Appear.

[BMÖ12] Kyungmin Bae, José Meseguer, and Peter Csaba Ölveczky. Formal Patterns for Multi-rate Distributed Real-Time Systems. In *FACS*, pages 1–18, 2012.

[BPE13] OASIS Web Services Business Process Execution Language (WS-BPEL). `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, June 2013.

[BR05] Michael J. Butler and Shamim Ripon. Executable Semantics for Compensating CSP. In *WS-FM*, pages 243–256, 2005.

[Bro03] Jan Broersen. *Modal Action Logics for Reasoning about Reactive Systems*. PhD thesis, Faculteit der Exacte Wetenschappen, Vrije Universiteit Amsterdam, 2003.

[BS08] Mario R. F. Benevides and L. Menasché Schechter. A Propositional Dynamic Logic for CCS Programs. In *WoLLIC*, pages 83–97, 2008.

[BWM01] Jan Broersen, Roel Wieringa, and John-Jules Ch. Meyer. A Fixed-point Characterization of a Deontic Logic of Regular Action. *Fundamenta Informaticae*, 48(2-3):107–128, 2001.

[BZ09] Mario Bravetti and Gianluigi Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.

[Car09] Marco Carbone. Session-based Choreography with Exceptions. *Electronic Notes in Theoretical Computer Science*, 241:35–55, 2009.

[CDE+99] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a Formal Meta-tool. In *World Congress on Formal Methods*, pages 1684–1703, 1999.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude*, volume 4350 of *Lecture Notes in Computer Science*, 2007.

[CFV08] Luís Caires, Carla Ferreira, and Hugo Torres Vieira. A Process Calculus Analysis of Compensations. In *TGC*, pages 87–103, 2008.

[CGV+02] Mandy Chessell, Catherine Griffin, David Vines, Michael J. Butler, Carla Ferreira, and Peter Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758, 2002.

[CHY08] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Interactional Exceptions in Session Types. In *CONCUR*, pages 402–417, 2008.

[CLW12] Zhenbang Chen, Zhiming Liu, and Ji Wang. Failure-divergence semantics and refinement of long running transactions. *Theoretical Computer Science*, 455:31–65, 2012.

[CM09] Pablo F. Castro and T. S. E. Maibaum. Deontic action logic, atomic boolean algebras and fault-tolerance. *Journal of Applied Logic*, 7(4):441–466, 2009.

[CP13] Christian Colombo and Gordon J. Pace. Recovery within long-running transactions. *ACM Computing Surveys*, 45(3):28, 2013.

[DK04] Vincent Danos and Jean Krivine. Reversible Communicating Systems. In *CONCUR*, pages 292–307, 2004.

[DK05] Vincent Danos and Jean Krivine. Transactions in RCCS. In *CONCUR*, pages 398–412, 2005.

[DKS07] Vincent Danos, Jean Krivine, and Pawel Sobocinski. General Reversibility. *Electronic Notes in Theoretical Computer Science*, 175(3):75–86, 2007.

[dVKH10a] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Communicating Transactions - (Extended Abstract). In *CONCUR*, pages 569–583, 2010.

[dVKH10b] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Liveness of Communicating Transactions (Extended Abstract). In *APLAS*, pages 392–407, 2010.

[EMS03] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL Model Checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2003.

[ER12] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[ES08] Christian Eisentraut and David Spieler. Fault, Compensation and Termination in WS-BPEL 2.0 - A Comparative Analysis. In *WS-FM*, pages 107–126, 2008.

[FG96] Cédric Fournet and Georges Gonthier. The Reflexive CHAM and the Join-Calculus. In *POPL*, pages 372–385, 1996.

[FLR+11] Carla Ferreira, Ivan Lanese, António Ravara, Hugo Torres Vieira, and Gianluigi Zavattaro. Advanced Mechanisms for Service Combination and Transactions. In *Results of the SENSORIA Project*, volume 6582 of *Lecture Notes in Computer Science*, pages 302–325, 2011.

[FM07] Jeffrey Fischer and Rupak Majumdar. Ensuring Consistency in Long Running Transactions. In *ASE*, pages 54–63, 2007.

[Gla90] Rob J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Free University, 1990.

[GLG+06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *ICSOC*, pages 327–338, 2006.

[GLMZ09] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic Error Handling in Service Oriented Applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.

[GMM88] Roberto Gorrieri, Sergio Marchetti, and Ugo Montanari. A^2CCS: A Simple Extension of CCS for Handling Atomic Actions. In *CAAP*, pages 258–270, 1988.

[GMM90] Roberto Gorrieri, Sergio Marchetti, and Ugo Montanari. $A^2$2CCS: Atomic Actions for CCS. *Theoretical Computer Science*, 72(2-3):203–223, 1990.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, pages 249–259, 1987.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[Har79] David Harel. *First-Order Dynamic Logic*, volume 68. Springer, 1979.

[Hen08] Joe Hendrix. *Decision Procedures for Equationally Based Reasoning*. PhD thesis, University of Illinois, 2008.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Jol13] Jolie Programming Language. `http://www.jolie-lang.org/`, June 2013.

[KSMA03] Nirman Kumar, Koushik Sen, José Meseguer, and Gul Agha. A Rewriting Based Model for Probabilistic Distributed Object Systems. In *FMOODS*, pages 32–46, 2003.

[Lan10] Ivan Lanese. Static vs Dynamic SAGAs. In *ICE*, pages 51–65, 2010.

[LMS10] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing Higher-Order Pi. In *CONCUR*, pages 478–493, 2010.

[LMS12] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Controlled Reversibility and Compensations. In *RC*, pages 233–240, 2012.

[LMSS11] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling Reversibility in Higher-Order Pi. In *CONCUR*, pages 297–311, 2011.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, pages 33–47, 2007.

[LPT08] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Formal Account of WS-BPEL. In *COORDINATION*, pages 199–215, 2008.

[LVF10] Ivan Lanese, Cátia Vaz, and Carla Ferreira. On the Expressive Power of Primitives for Compensation Handling. In *ESOP*, pages 366–386, 2010.

[LZ05] Cosimo Laneve and Gianluigi Zavattaro. Foundations of Web Transactions. In *FoSSaCS*, pages 282–298, 2005.

[LZ09] Ivan Lanese and Gianluigi Zavattaro. Programming Sagas in SOCK. In *SEFM*, pages 189–198, 2009.

[LZ13] Ivan Lanese and Gianluigi Zavattaro. Decidability Results for Dynamic Installation of Compensation Handlers. In *COORDINATION*, pages 136–150, 2013.

[Mau13a] Maude 2.6. `http://maude.cs.uiuc.edu/`, June 2013.

[Mau13b] Maude Manual. `http://maude.cs.uiuc.edu/maude1/manual/`, June 2013.

[Mes92] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes12] José Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7-8):721–781, 2012.

[Mey88] John-Jules Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic*, 29(1):109–136, 1988.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[ML06] Manuel Mazzara and Ivan Lanese. Towards a Unifying Theory for Web Services Composition. In *WS-FM*, pages 257–272, 2006.

[MM90] José Meseguer and Ugo Montanari. Petri Nets Are Monoids. *Information and Computation*, 88(2):105–155, 1990.

[MR11] José Meseguer and Grigore Rosu. The Rewriting Logic Semantics Project: A Progress Report. In *FCT*, pages 1–37, 2011.

[NPW79] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri Nets, Event Structures and Domains. In *Semantics of Concurrent Computation*, pages 266–284, 1979.

[NSM01] Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL Proof Translator (A Practical Approach to Formal Interoperability). In *TPHOLs*, pages 329–345, 2001.

[ÖBM10] Peter Csaba Ölveczky, Artur Boronat, and José Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In *FMOODS/FORTE*, pages 47–62, 2010.

[ÖM02] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[Ora13] Oracle BPEL Process Manager. `http://www.oracle.com/technology/bpel/`, June 2013.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[Pel87] David Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34:450–479, 1987.

[PU07] Iain C. C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *The Journal of Logic and Algebraic Programming*, 73(1-2):70–96, 2007.

[PZQ+06] Geguang Pu, Huibiao Zhu, Zongyan Qiu, Shuling Wang, Xiangpeng Zhao, and Jifeng He. Theoretical Foundations of Scope-Based Compensable Flow Language for Web Service. In *FMOODS*, pages 251–266, 2006.

[RB09] Shamim Ripon and Michael J. Butler. PVS Embedding of cCSP Semantic Models and Their Relationship. *Electronic Notes in Theoretical Computer Science*, 250(2):103–118, 2009.

[Rei85] W. Reisig. *Petri Nets*. Springer, 1985.

[RVMOC12] Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 81(7-8):851–897, 2012.

[Sou13] Maude source code. `https://github.com/annekauer/thesis.git`, October 2013.

[Spa10] Giorgio Spagnolo. Analisi e confronto di politiche di compensazione distribuita nell'ambito di transazioni a lunga durata. Master's thesis, Università di Pisa, 2010.

[USH95] Augustus K. Uht, Vijay Sindagi, and Kelley Hall. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 313–325, 1995.

[VCS08] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *ESOP*, pages 269–283, 2008.

[vdM96] Ron van der Meyden. The Dynamic Logic of Permission. *Journal of Logic and Computation*, 6(3):465–479, 1996.

[VF09] Cátia Vaz and Carla Ferreira. Towards Compensation Correctness in Interactive Systems. In *WS-FM*, pages 161–177, 2009.

[VF12] Cátia Vaz and Carla Ferreira. On the analysis of compensation correctness. *The Journal of Logic and Algebraic Programming*, 81(5):585–605, 2012.

[VFR08] Cátia Vaz, Carla Ferreira, and António Ravara. Dynamic Recovering of Long Running Transactions. In *TGC*, pages 201–215, 2008.

[VMO06] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in Maude. *The Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.

[VW51] Georg Henrik Von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

[WEMM12] Martin Wirsing, Jonas Eckhardt, Tobias Mühlbauer, and José Meseguer. Design and Analysis of Cloud-Based Architectures with KLAIM and Maude. In *WRLA*, pages 54–82, 2012.