Theoretical Foundations for Compensations in Flow Composition Languages *

Roberto Bruni
Dipartimento di Informatica
Università di Pisa, Italia
bruni@di.unipi.it

Hernán Melgratti
Dipartimento di Informatica
Università di Pisa, Italia
melgratt@di.unipi.it

Ugo Montanari
Dipartimento di Informatica
Università di Pisa, Italia
ugo@di.unipi.it

ABSTRACT

A key aspect when aggregating business processes and web services is to assure transactional properties of process executions. Since transactions in this context may require long periods of time to complete, traditional mechanisms for guaranteeing atomicity are not always appropriate. Generally the concept of long running transactions relies on a weaker notion of atomicity based on compensations. For this reason, programming languages for service composition cannot leave out two key aspects: *compensations*, i.e. ad hoc activities that can undo the effects of a process that fails to complete, and *transactional boundaries* to delimit the scope of a transactional flow. This paper presents a hierarchy of transactional calculi with increasing expressiveness. We start from a very small language in which activities can only be composed sequentially. Then, we progressively introduce parallel composition, nesting, programmable compensations and exception handling. A running example illustrates the main features of each calculus in the hierarchy.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent programming—*Distributed programming*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms

Languages, Theory

*Research supported by the FET-GC Project IST-2001-32747 Agile, by the MIUR Project COFIN 2001013518 CoMeta, and by the MURST-CNR 1999 Project *Software Architectures on Cooperative WAN*.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

Keywords

Transactions, compensations, process description languages

1. INTRODUCTION

The ultimate goal of web services technologies is to allow the distribution, delivery and interoperability of heterogeneous components over the Internet. Applications achieve interoperability by adhering to standard protocols that provide uniform ways to describe services (namely WSDL), to look for particular services (i.e., UDDI), and to access services (i.e., SOAP). In this way standards facilitate the interaction of different services, not only within an organization but also across organization boundaries. Nevertheless, these standards do not provide yet any support to describe complex interactions between several applications. Recently, many proposals have addressed the problem of aggregating services, giving birth to a family of XML-based *composition languages* (also known as *choreography* or *orchestration* languages), such as BPML [6], XLANG [21], WSFL [16], BPEL4WS [5] and WSCI [24].

Choreography languages allow the definition of complex services in terms of the interactions among simpler services. In general, composition can be specified in one of the following styles [16]: (i) *flow composition* (also known as *hierarchical patterns*) and (ii) *interaction based composition* (also known as *conversational patterns* or *global models*). Flow composition is reminiscent of *workflow systems* [13], where composed services are described by a process that states precisely the flow of both control and data between the component parts. Instead, conversational based languages are aimed at describing the interaction protocols or patterns that services should follow in order to achieve a specific goal. In this case, any service declares the ways in which it can be engaged in a larger process.

Usually flow composition is associated with a centralized coordination mechanism (the flow engine) that monitors the order in which activities are run, while interaction models are related to distributed orchestration, in which participants are responsible for adhering to a specific protocol.

Most proposals for orchestration languages contain a large amount of primitives, allowing for both styles of composition. (For a discussion about the several patterns or primitives they provide we refer to [2]). Since the official specifications of composition languages for web services mainly consist in an informal textual description of their constructors, many recent efforts have attempted to formalize different subsets of such proposals (see for instance [7, 3, 22]).

At the same time, foundational models for concurrency has been proposed also as foundational models for web services orchestration ([18, 1]). Nevertheless, there are several fundamental aspects when describing composed services, in particular the transactional properties of flows, that have received little attention in the area of *process description languages* (PDLS) like the π-calculus or CCS. These properties are somehow orthogonal to usual PDL's operators, and although they could be possibly encoded into standard operators, the extension of PDLS with this kind of abstractions are desirable (at least) for the following reasons: (i) to lay the foundations for the formal definition of orchestration languages; (ii) to facilitate the comparison of different semantics; (iii) to reason about the relation among operators; (iv) as the starting point for comparing the expressive power of newly proposed primitives (i.e., whether they can be conveniently defined in term of usual operators or not) and for studying the properties preserved by different encodings; and (v) to give insights about implementation details.

In this paper, we study primitives for long running transactions in flow composition languages, and in particular in structured control flows, i.e. flows defined in terms of a fixed set of primitives, like sequencing and branching. We provide a formal semantics for a hierarchy of transactional languages with increasing expressiveness and we prove that the semantics is adequate to the modelled features.

Transactional aspects in composed web services have been mainly inherited from workflow languages. The key idea is that valid executions of a transactional business process (or of a part of it) are those that "complete" all involved activities. Nevertheless, since the execution of a business process may require a very long period of time in order to complete (perhaps some hours or days), traditional mechanisms for assuring atomicity, such as locking of resources, are regarded as not suitable. Since the seminal work of Sagas [12], the key mechanism for dealing with long running transactions is that of "*compensating activities*". Instead of relying on locking and roll-back mechanisms to perfectly undo incomplete executions and avoid interference among transactions, a more relaxed form of atomicity is granted by associating processes with activities that can recover partial executions.

Although compensations can be regarded as an exception handling mechanism [17], the distinctive feature is that compensation handlers are dynamically built during the execution of processes. Consider the saga given in Figure 1, where the transactional process P consists in the sequential execution of the activities A_1, A_2 and A_3, that can be compensated respectively by B_1, B_2 and B_3. Suppose now that activity A_1 completes successfully while activity A_2 fails. In this case, after A_2 fails, the compensation B_1 (corresponding to the successfully completed activities) is run to undo as much as possible the effects of A_1, because the transaction failed as a whole. Note that B_2 is not executed, because A_2 has not completed. Instead, if both A_1 and A_2 succeed while A_3 fails, then the compensations will be executed in the reverse order, i.e. first B_2 and then B_1.

After Sagas, several workflow models have been proposed in literature for equipping processes with different (compensation-based) transactional capabilities, such as nesting and forward recovery (for a general overview see [20]). Contrastingly, the study of PDLS with compensations have been less numerous. An extension of the asynchronous π-calculus, called πt-calculus, with transactional contexts has been in-

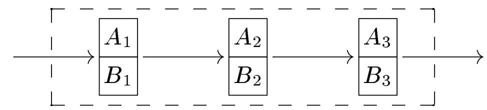

Figure 1: A sequential saga.

troduced in [4]. The πt-calculus formalizes the close relation between exception handling and compensations. Nevertheless, this approach is not aimed at capturing the order in which compensations should be activated, i.e. there is not a strong relation between compensations and the control flow of the original processes. For instance, if activity A_3 fails during the execution of flow depicted in Figure 1, compensations B_1 and B_2 are activated concurrently.

A different approach is taken by StAC [9], where compensations are installed to be executed in the reverse order w.r.t. that of completion of original activities. StAC is a language in the spirit of process algebras like CSP or CCS with exception handling mechanisms and compensations inspired by BPBeans, a framework for modelling business processes integrated to WebSphere [23]. Although being (to the best of our knowledge) the first process calculus where compensations are closely related to the control flow of the executed process[1], there are several aspects in StAC that deserve further investigation. For instance, compensations in StAC should be explicitly activated through special primitives, i.e. they are not related to the failure or success of the activities of processes, as usually expected in workflows and composition languages, e.g. BPEL4WS. Moreover, to reason about StAC processes, it is necessary to know the low level description of activities. In fact, there is an interplay between data structures used by activities and the control flow of processes. Finally, StAC provides a large number of operators including the imperative fragment of a programming language, whose operational semantics has been given in terms of an even richer intermediate language, called StAC_i [10]. In this way, operators in StAC can only be understood by analyzing their encodings into StAC_i operators. Due to the complex definition of the operational semantics of StAC_i, it is difficult to reason about the interplay among exception handling, compensations, nesting and parallel composition in StAC. Moreover, some usual behaviors of compensations (for instance, the failure of a branch in a parallel composition requiring the compensation of both branches) are only achieved by combining several operators, making the semantics in [10] not entirely satisfactory. (Recent ongoing work by Butler, Ferreira and Hoare aims to define a clean trace semantics for a subset of StAC.)

In this work we intend to give a more compact description of StAC-like languages: in the spirit of PDLS, we are aimed at providing a minimal set of operators with orthogonal meaning and, in particular, we are interested on marking the distinction between compensations and exception handling mechanisms. Moreover we attempt to provide our operators with the meaning most frequently used in composition languages. Additionally, we relate the behavior of whole processes with the success or failure of atomic activities.

[1]We are aware of previous formal approaches to define compensations, such as ACTA [11] in the context of database transactions and the work done by C.A.R Hoare [14], but they are not process calculi

In order to achieve these goals, we start from a very small language formalizing Sagas. First, we show a language corresponding to its sequential version (i.e., allowing only the sequential composition of activities inside a saga). Then we introduce the parallel composition and discuss different alternatives in defining the semantics for the compensation of parallel activities. After that, we add the possibility of defining nested transactions. Finally, we present some extensions, such as programmable compensations, exception handling and forward recovery. Each language in the resulting hierarchy comes with a clean big-step semantics and an adequacy result for such semantics.

As a running example, we select a business process for ordering goods. It is simple enough to require a process that fits in one line, yet it is expressive enough to show how the primitives can enhance business process design.

Structure of the paper. In Section 2 we present the core language for sequential sagas, which has primitives for compensated activities $A \div B$, sequential composition $P;Q$ and saga scope $\{\![P]\!\}$. In Section 3 we extend the core language with parallel composition $P|Q$, and in Section 4 we extend parallel sagas with nesting $\{\![S]\!\}$. In Section 5 we discuss how the language for nested sagas can be further extended with additional features like programmable compensations, exception handling, choices, and dependencies. Concluding remarks and future work are in Section 6.

2. SEQUENTIAL SAGAS

As mentioned before, Sagas [12] is one of the first proposals for dealing with long running transactions in database applications. A *sequential saga* (i.e., a long lived transaction) is a sequence of atomic activities (called *subtransactions*, *activities* or *steps*) that should be executed completely. The parallel execution of several sagas can interleave steps in any way, but any single step is guaranteed to be atomic. Subtransactions are atomic in the sense that either they are successfully executed (*committed*) or no effect is observed when the execution fails (*aborted*). In addition, no intermediate states computed by an activity are visible to other activities. Activities are transactions with short duration, and therefore they can rely on traditional mechanisms to assure the usual ACID properties (i.e., *Atomicity, Consistency, Isolation* and *Durability*). Additionally, any activity A_i in a saga has a compensating activity B_i that can be activated to "undo" the effects of a successful execution of A_i upon a later failure. (We remind that, in this context, the term "undo" does not mean to exactly reverse the effects by restoring the original state, but just to perform an ad hoc activity that moves the system to a sound state).

Any partial execution of a saga is undesirable, and if it occurs, it must be compensated for. A saga involving $A_1, ..., A_n$ (where each A_i has a compensation B_i) is guaranteed to execute either the entire series $A_1; ...; A_n$ or the compensated sequence $A_1; ...; A_j; B_j; ...; B_1$ for some $j < n$. The first case stands for the successful execution of the whole saga, i.e., when all activities in the sequence complete. In the second case, the activity A_{j+1} fails, and all activities already completed $(A_1; ...; A_j)$ are recovered by executing the corresponding compensations, in reverse order $(B_j; ...; B_1)$.

In this section we introduce a compensation language for sequential sagas. The semantics is intended to describe the behavior of top-level processes but not the low-level computations performed by atomic activities. The only assumption made on subtransactions is that their executions end either successfully or with a failure.

We rely on an infinite set $\mathcal{A}$ of names for atomic activities, ranged over by $A, B, \ldots$. Moreover, we will consider a special nil activity $0 \notin \mathcal{A}$ that always completes and has no effect.

DEFINITION 1 (SEQUENTIAL SAGAS). *The set of all sequential sagas is given by the following grammar:*

$$\begin{array}{llcl} (\text{STEP}) & X & ::= & 0 \mid A \mid A \div B \\ (\text{PROCESS}) & P & ::= & X \mid P;P \\ (\text{SAGA}) & S & ::= & \{\![P]\!\} \end{array}$$

A sequential saga S consists in a sequential process P. Each step in P corresponds either to an activity A or a compensated activity $A \div B$, where A is the activity of the normal flow and B its compensation. The term 0 represents the inert process, and $P;P$ stands for the sequential composition of processes.

We define the semantics of sagas up-to structural congruence over processes and steps given by the following axioms:

$$\begin{array}{rcll} A \div 0 & \equiv & A & (\text{NULL COMPENSATION}) \\ 0;P & \equiv & P;0 \;\equiv\; P & (\text{NULL PROCESS}) \\ (P;Q);R & \equiv & P;(Q;R) & (\text{ASSOC. OF SEQ. COMP.}) \end{array}$$

For simplicity we will consider all (instances of) activities in a saga named differently. This does not mean we do not allow the same activity to be executed more than once in a saga, but we consider any execution as a different instance of it and, hence distinguishable from all other instances.

DEFINITION 2 (ACTIVITIES OF A SAGA). *The set of activities of a saga S is defined as $\mathcal{A}(S) = \{A \mid A$ occurs in $S\}$.*

2.1 Big Step Semantics

As described above, the execution of a saga S either commits — i.e., every activity executes successfully — or it aborts and all completed steps are compensated for. This model implicitly assumes that compensations always succeed. In order to relax this assumption, we allow also compensations to fail. In this case, a saga S has an abnormal termination. Abnormal termination could be managed by suitable exception handling mechanisms. (We informally discuss exception handling in Section 5.2). Thus, the set of possible results for the execution of a saga is $\mathcal{R} = \{\boxdot, \boxtimes, \boxast\}$, where $\boxdot$ stands for *commit*, $\boxtimes$ for (compensated) *abort*, and $\boxast$ for *abnormal termination*. We let $\Box$ range over $\mathcal{R}$.

The execution of a sequential saga is described in terms of the results obtained by performing their constituent activities. As we are not interested on the low-level behavior of individual tasks, we rely on the abstract description of their executions, stating whether they complete successfully or abort. This information is given by a context Γ. Formally, Γ is a partial function over $\mathcal{A}$ that maps any activity to the result obtained with its execution, i.e., $\Gamma : \mathcal{A} \rightharpoonup \{\boxtimes, \boxdot\}$. Note that activities can only commit or abort (they do not terminate abnormally). We denote a particular function Γ as $A_1 \mapsto \Box_1, \ldots, A_n \mapsto \Box_n$, where $A_i \neq A_j$ for all $i \neq j$ (i.e., ',' stands for the disjoint union of partial functions).

The semantics of a sequential saga S is given by the relation $\Gamma \vdash S \xrightarrow{\alpha} \Box$ defined by the inference rules in Figure 2. The notation $\Gamma \vdash S \xrightarrow{\alpha} \Box$ denotes that the execution of S produces $\Box$ when the atomic activities behave like Γ. The

(ZERO)
$$\Gamma \vdash \langle 0, \beta\rangle \xrightarrow{0} \langle \boxdot, \beta\rangle$$

(S-ACT)
$$A \mapsto \boxdot, \Gamma \vdash \langle A \div B, \beta\rangle \xrightarrow{A} \langle \boxdot, B;\beta\rangle$$

(S-CMP)
$$\frac{\Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \boxdot, 0\rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}$$

(F-CMP)
$$\frac{\Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta\rangle \xrightarrow{\alpha} \langle \boxast, 0\rangle}$$

(S-STEP)
$$\frac{\Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha} \langle \boxdot, \beta''\rangle \quad \Gamma \vdash \langle Q, \beta''\rangle \xrightarrow{\alpha'} \langle \Box, \beta'\rangle}{\Gamma \vdash \langle P;Q, \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box, \beta'\rangle}$$

(A-STEP)
$$\frac{\Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha} \langle \sigma, 0\rangle}{\Gamma \vdash \langle P;Q, \beta\rangle \xrightarrow{\alpha} \langle \sigma, 0\rangle} \quad \sigma \in \{\boxtimes, \boxast\}$$

(SAGA)
$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \Box, \beta\rangle}{\Gamma \vdash \{\![P]\!\} \xrightarrow{\alpha} \Box}$$

Figure 2: Semantics of sequential sagas.

observation α describes the actual flow of control occurring when executing S under the context Γ. The flow α is a process whose activities have no compensations.

The auxiliary relation $\Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha} \langle \Box, \beta'\rangle$ describes the behavior of a process P within a saga that already installed the compensation β (β stands for a process without compensations). Γ and α are analogous to the previous case. When P is executed inside a saga, it can either commit, abort, or fail, but additionally, it can change the compensations, for instance by installing new activities, like in rule (S-ACT).

Rule (ZERO) states that 0 always commits without changing the installed compensation. Rule (S-ACT) stands for the successful execution of the compensated activity $A \div B$ when A commits. In this case, the observation is A (i.e., the only executed activity), the obtained result is $\boxdot$, while the compensation is updated by installing B in front of β. Note that A is the last executed activity, hence the first to be compensated for if the next activity in the saga fails.

Rules (S-CMP) and (F-CMP) describe the execution of $A \div B$ when A fails in a saga that has already installed β. Both rules activate the compensation procedure by executing β (premises of the rules). Note that neither A nor B are really executed. In fact, since A is an atomic activity that aborts, A has no effects and hence, it is not compensated for. For this reason the observation α is just the flow observed by executing β. In particular, (S-CMP) describes the case in which the compensation procedure completes successfully. Rule (F-CMP) stands for the case in which the compensation procedure fails. In this case, the process finishes abnormally (the corresponding result is $\boxast$). Since all steps in β have trivial nil compensations, the execution of β cannot produce $\boxast$. For the same reason, the execution of β installs no significant compensations, and hence any execution that ends with $\boxtimes$ or $\boxast$ must have 0 as compensation.

Rule (S-STEP) describes the behavior of a process $P;Q$ when the step P commits. In such case the remaining process Q is executed by taking into account the compensation produced after the execution of P. The observation for the whole process $P;Q$ corresponds to the sequential composition of α, i.e. the observation of executing P, and α', i.e. the flow corresponding to the execution of Q. The final result is that obtained when executing Q.

Rule (A-STEP) handles the case in which $P;Q$ is stopped because P ends with abort or abnormal termination. Note that the compensation is activated when P reaches the abort.

Last rule (SAGA) states that the execution of a saga $\{\![P]\!\}$ is the activation of P with no installed compensations.

Although a more concise set of rules could be used to describe the semantics, we choose this presentation for convenience when extending the language in the next sections.

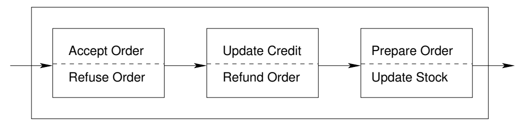

Figure 3: A sequential saga for handling orders.

EXAMPLE 1 (SEQUENTIAL SAGAS). *Figure 3 shows a sequential saga for dealing with purchase orders. It consists on three activities composed sequentially. The first activity (Accept order) handles a request from a client and it is compensated by Refuse order, which will contact the client to notify her/him that the order was canceled. The second step (Update Credit) charge the amount of the order to the balance of the client. This activity could fail, for instance when the client has not enough credit to proceed, activating the compensation, i.e., executing Refuse order. Instead, if it succeeds, then the compensation Refund order is also installed. Refund order is responsible for updating the balance with the amount detracted previously. Last activity (Prepare Order) handles the packaging of the order and update the stock. Its compensation (Update Stock) will increment the stock with the proper values.*

The following result states that the execution of a saga corresponds to the intuitive notion we gave initially.

THEOREM 1 (ADEQUACY). *Let $S \equiv \{\![A_1 \div B_1; \ldots; A_n \div B_n]\!\}$ be a saga. Then:*

(COMPLETION) $\Gamma \vdash S \xrightarrow{\alpha} \boxdot$ *iff* $\forall i \leq n : A_i \mapsto \boxdot \in \Gamma$ *and* $\alpha \equiv A_1; \ldots; A_n$*;*

(SUCCESSFUL COMPENSATION) $\Gamma \vdash S \xrightarrow{\alpha} \boxtimes$ *iff* $\exists k, 1 \leq k \leq n \ \wedge\ A_k \mapsto \boxtimes \in \Gamma \wedge\ \forall i < k : (A_i \mapsto \boxdot, B_i \mapsto \boxdot \subseteq \Gamma)$ *and* $\alpha \equiv A_1; \ldots; A_{k-1}; B_{k-1}; \ldots; B_1$.

(FAILED COMPENSATION) $\Gamma \vdash S \xrightarrow{\alpha} \boxast$ *iff* $\exists j, k,\ 1 \leq k \leq n \wedge 1 \leq j < k$ *s.t.* $A_k \mapsto \boxtimes \in \Gamma \wedge B_j \mapsto \boxtimes \in \Gamma \wedge (\forall i < k : A_i \mapsto \boxdot \in \Gamma) \wedge (\forall h \in [j+1, k-1] : B_h \mapsto \boxdot \in \Gamma)$ *and* $\alpha \equiv A_1; \ldots; A_{k-1}; B_{k-1}; \ldots; B_{j+1}$.

It is clear from the above theorem that the last compensation B_n is never activated. Nevertheless we allow such kind of definitions because they can be useful when specifying more complex sagas in the following sections.

3. PARALLEL SAGAS

In order to allow several activities to be executed concurrently, the language of sequential sagas is extended with the operator '|', denoting the parallel composition of processes.

(S-PAR)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta''\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, \beta'|\beta''; \beta\rangle}$$

(F-PAR-NAÏVE-1)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \boxtimes, 0\rangle \quad \Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\alpha''} \langle \Box_1, \beta''\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha|\alpha');\alpha''} \langle \Box_2, 0\rangle} \qquad \Box_2 = \begin{cases} \boxtimes & \text{if } \Box_1 = \boxdot \\ \boxast & \text{otherwise} \end{cases}$$

(F-PAR-NAÏVE-2)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxast, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle \beta', 0\rangle \xrightarrow{\alpha''} \langle \Box, 0\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha|\alpha');\alpha''} \langle \boxast, 0\rangle}$$

(F-PAR-NAÏVE-3)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxast, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \sigma, 0\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha|\alpha')} \langle \boxast, 0\rangle} \qquad \textit{with } \sigma \in \{\boxtimes, \boxast\}$$

(F-PAR-NAÏVE-4)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle \beta'; \beta, 0\rangle \xrightarrow{\alpha''} \langle \Box_1, 0\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{(\alpha|\alpha');\alpha''} \langle \Box_2, 0\rangle} \qquad \Box_2 = \begin{cases} \boxtimes & \text{if } \Box_1 = \boxdot \\ \boxast & \text{otherwise} \end{cases}$$

Figure 4: Naïve semantics of parallel composition.

DEFINITION 3 (PARALLEL SAGAS). *The set of all parallel sagas is defined by the grammar:*

$$\begin{array}{llll} \text{(STEP)} & X & ::= & 0 \mid A \mid A \div B \\ \text{(PROCESS)} & P & ::= & X \mid P;P \mid P|P \\ \text{(SAGA)} & S & ::= & \{[P]\} \end{array}$$

In addition to the structural axioms for sequential sagas, we require $'|'$ to be associative and commutative with unit 0. We let sequential composition have higher priority than parallel, i.e. $P;Q|R;S$ stands for $(P;Q)|(R;S)$.

Note that the dependencies among activities are described by a structured flow, and in particular synchronizations between processes take place only when composing sequentially. We will not consider descriptions based on links dependencies like those allowed in WSFL until Section 5.5.

As for sequential sagas, a computation of a parallel saga is successful only when all its activities commit, while the whole saga should be compensated for when an activity fails. Also we like compensations to be performed in the reverse order of the normal flow. Composition languages usually express this requirement by stating that all compensation handlers for completed activities run in the reverse order of completion. In our approach the compensations of concurrent activities are concurrent, because we want a semantics where compensations do not depend on the particular interleaving of executed concurrent activities.

We first give a semantics where parallel branches are completely independent (Section 3.1). This semantics is simple but not entirely satisfactory when modelling real problems, because it does not allow to force the failure in one branch as soon as a failure is detected in the other branch. A more complex semantics is then given in Section 3.2, which can deal properly with this kind of optimization.

3.1 Naïve definition for the semantics of '|'

In a first attempt at defining the semantics of the parallel composition we add the rules in Figure 4 to those of sequential sagas. Note that transition labels α can now take the form $\gamma|\gamma'$, where γ and γ' are uncompensated processes. In this way we quotient out all possible interleaving executions of $\gamma|\gamma'$. We recall that activities are atomic steps, and therefore there is no interaction among them. For this reason any interleaving of $\gamma|\gamma'$ is a valid execution of the process. Hence, the parallel branches of $P|Q$ are executed independently, i.e. each branch performs until completion in its own thread, which has no initial compensation.

The first rule (S-PAR) handles the successful execution of $P|Q$, i.e. every activity commits and both P and Q produce $\boxdot$ as result. The compensation for the whole process is updated by installing the parallel composition of β' and β'' at the top, i.e. the compensation of parallel processes corresponds to the parallel compensation of its branches. The observed flow is the parallel composition of the flows for P and Q.

The remaining four rules handle the cases where at least one activity aborts. If both branches fail during the normal flow but their compensation completes, then the execution ends by activating the original compensation β (rule F-PAR-NAÏVE-1). The result is $\boxtimes$ when β finishes without problems. On the contrary, if β aborts, then the whole process terminates abnormally (producing $\boxast$).

Rules (F-PAR-NAÏVE-2) and (F-PAR-NAÏVE-3) stand for the cases in which P terminates abnormally, i.e. some activity in the normal flow of P aborts activating its compensation procedure, which also fails. In such cases, the original compensation β is not executed, because it should follow the compensation of P that has failed. The behavior is similar to the sequential case, where the compensation procedure stops when some activity aborts. In particular, rule (F-PAR-NAÏVE-2) handles the situation in which the remaining process Q has completed successfully and it is compensated for. For this reason the compensation β' installed by Q is activated. The final result is in any case $\boxast$, and the observed flow corresponds to the parallel execution of both branches followed by the execution of the compensation β'. Instead (F-PAR-NAÏVE-3) describes the cases in which Q has already

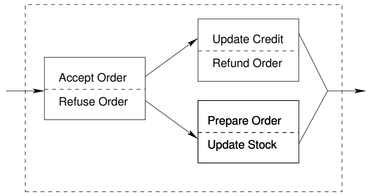

Figure 5: A parallel saga for handling orders.

been compensated for (i.e., the result is $\boxtimes$ or $\boxast$), and therefore no further compensation is activated. Also, in this case the process $P|Q$ terminates abnormally.

Last rule (F-PAR-NAÏVE-4) describes the behavior when one of the branches finishes successfully and the other has been aborted and properly compensated for. Hence. the execution done by the successful branch needs to be reversed (by running β') before activating the original compensation β. If the whole compensation (i.e., the execution of $\beta'; \beta$) finishes with success, then the final result is $\boxtimes$, otherwise the whole process terminates abnormally.

EXAMPLE 2 (PARALLEL SAGAS). *The second and third activities in Example 1 could be performed in parallel as shown in Figure 5. Nevertheless, in case some activity aborts, we would like all completed steps to be compensated for.*

Although given rules allow a failed branch to start its compensation as soon as it aborts, the successful branch is forced to execute until completion, even when it will be compensated for. Consider the following parallel saga:

$$S \equiv \{\![A_1 \div B_1;\ A_2 \div B_2 \ \mid \ C_1 \div D_1]\!\}$$

and the context in which all activities but C_1 commit, i.e. $\Gamma = A_1 \mapsto \boxdot, A_2 \mapsto \boxdot, C_1 \mapsto \boxtimes, B_1 \mapsto \boxdot, B_2 \mapsto \boxdot$, the only possible computation for S produces as result $\boxtimes$ with observation $(A_1; A_2|0); B_2; B_1$.

In a real execution of S where C_1 fails while A_1 is still executing, it would be desirable to avoid the execution of both A_2 and its compensation B_2 by starting the compensation procedure as soon as A_1 finishes. (We remind that activities are atomic and hence their execution cannot be stopped once they have started). In general, when several processes execute concurrently and some activity aborts, then the whole saga aborts and every completed activity should be compensated for. Hence, it would be desirable to stop all processes before completion and to start the compensation procedure of partially executed branches as soon as a concurrent activity aborts. The following section presents a semantics for parallel sagas that allows such kind of behaviors.

3.2 Parallel Sagas Revised

To handle partial executions of successful branches during an aborted execution of a saga, we introduce two new kinds of results for a process running in a saga: (i) $\overline{\boxtimes}$ denoting that the process has been forced to compensate and then it has been compensated for successfully, and (ii) $\overline{\boxast}$ denoting that the process has been forced to compensate but the compensation procedure has failed. Let $\square, \sigma_1, \sigma_2$ range over $\overline{\mathcal{R}} = \mathcal{R} \cup \{\overline{\boxtimes}, \overline{\boxast}\}$. Moreover, we use the binary operator $\wedge$ over $\overline{\mathcal{R}}$ to express the result obtained by combining the execution of two parallel branches. The associative and commutative operator $\wedge$ is defined in the following table (because of commutativity we omit half of the table).

$\wedge$	$\boxdot$	$\boxtimes$	$\boxast$	$\overline{\boxtimes}$	$\overline{\boxast}$
$\boxdot$	$\boxdot$				
$\boxtimes$	—	$\boxtimes$			
$\boxast$	—	$\boxast$	$\boxast$		
$\overline{\boxtimes}$	—	$\boxtimes$	$\boxast$	$\overline{\boxtimes}$	
$\overline{\boxast}$	—	$\boxast$	$\boxast$	$\overline{\boxast}$	$\overline{\boxast}$

Note that $\wedge$ is not defined when one operand is $\boxdot$ and the other not. In fact, it is not possible for a branch to commit when the other aborts or fails: in the process $P|Q$, when P commits but Q does not, P is forced to compensate. The other interesting cases are the two last rows on the table, in which one of the branches is forced to compensate (producing either $\overline{\boxtimes}$ or $\overline{\boxast}$). If the remaining branch really fails (i.e., it reduces to a configuration with result $\boxtimes$ or $\boxast$) then the parallel composition actually fails. Otherwise —if it is also forced to compensate— then the whole process has been forced to compensate.

The semantics for parallel sagas is given in Figure 6. All rules for sequential sagas remain unchanged but A-STEP, whose side condition considers also the new kinds of results for σ, and four new rules are added.

Rules (S-PAR), (F-PAR) and (C-PAR) specify the behavior of parallel composition. As for the naïve semantics, parallel branches are run in parallel without initial compensations. If both branches commit (rule (S-PAR)), then the original compensation β is updated with the compensations β' and β'' installed by both branches. In particular, if the whole process $P|Q$ has to be compensated, then β' and β'' are activated in parallel and β is started only when they finish.

If some branch has activated its compensation procedure, then also the other branch is required to be compensated for. If one of the branches fails during the compensation procedure (rule (F-PAR)), then the final result for $P|Q$ will be a (possibly forced) abnormal termination (i.e., $\boxast$ or $\overline{\boxast}$). In this case the compensation β installed before the execution of $P|Q$ is not even activated.

Finally, rule (C-PAR) handles the case in which both P and Q are successfully compensated for. In such case, also the previously installed compensation β is run.

The new rule (FORCED-ABT) handles the forced compensation of a process P, i.e. P can activate the compensation procedure before starting its execution that will produce a forced termination $\overline{\boxtimes}$ or $\overline{\boxast}$. Nevertheless, by rule (SAGA), the execution of a saga ends only when P produces $\boxdot$, $\boxtimes$ or $\boxast$. Hence, a valid execution forces a process to compensate only when it is a branch of a parallel composition. Moreover, in order to remove the tag of forced termination, the other branch is required to actually abort or finish abnormally. This is achieved by rules (F-PAR) and (C-PAR) that use the operator $\wedge$ to combine the results of concurrent executions.

As done for sequential sagas, we state the correspondence between the proposed semantics and the intended meaning of parallel sagas. We start by defining some notions that are needed to formalize the correspondence.

DEFINITION 4 (FORWARD FLOW). *The* forward flow $|S|$ *of a parallel saga S is obtained by removing all compensations from S, i.e. terms $A \div B$ are replaced by A.*

In defining the order of a saga, we assume all activities to have a compensation ($A \equiv A \div 0$). Since activities in a saga

(ZERO)
$$\Gamma \vdash \langle 0, \beta \rangle \xrightarrow{0} \langle \boxdot, \beta \rangle$$

(S-ACT)
$$A \mapsto \boxdot, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{A} \langle \boxdot, B; \beta \rangle$$

(S-CMP)
$$\frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, 0 \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle}$$

(F-CMP)
$$\frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \boxast, 0 \rangle}$$

(S-STEP)
$$\frac{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \boxdot, \beta'' \rangle \quad \Gamma \vdash \langle Q, \beta'' \rangle \xrightarrow{\alpha'} \langle \Box, \beta' \rangle}{\Gamma \vdash \langle P; Q, \beta \rangle \xrightarrow{\alpha;\alpha'} \langle \Box, \beta' \rangle}$$

(A-STEP)
$$\frac{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \sigma, 0 \rangle}{\Gamma \vdash \langle P; Q, \beta \rangle \xrightarrow{\alpha} \langle \sigma, 0 \rangle} \quad \sigma \in \{\boxtimes, \boxast, \overline{\boxtimes}, \overline{\boxast}\}$$

(S-PAR)
$$\frac{\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, \beta' \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \boxdot, \beta'' \rangle}{\Gamma \vdash \langle P|Q, \beta \rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, \beta'|\beta''; \beta \rangle}$$

(F-PAR)
$$\frac{\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \sigma_1, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha} \langle \sigma_2, 0 \rangle}{\Gamma \vdash \langle P|Q, \beta \rangle \xrightarrow{\alpha|\alpha'} \langle \sigma_1 \wedge \sigma_2, 0 \rangle} \quad \begin{cases} \sigma_1 \in \{\boxast, \overline{\boxast}\} \\ \sigma_2 \in \{\boxtimes, \boxast, \overline{\boxtimes}, \overline{\boxast}\} \end{cases}$$

(C-PAR)
$$\frac{\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \sigma_1, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \sigma_2, 0 \rangle \quad \Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\gamma} \langle \Box_1, 0 \rangle}{\Gamma \vdash \langle P|Q, \beta \rangle \xrightarrow{(\alpha|\alpha');\gamma} \langle \sigma_1 \wedge \sigma_2 \wedge \Box_2, 0 \rangle} \quad \sigma_1, \sigma_2 \in \{\boxtimes, \overline{\boxtimes}\} \text{ and } \Box_2 = \begin{cases} \overline{\boxtimes} & \text{if } \Box_1 = \boxdot \\ \overline{\boxast} & \text{otherwise} \end{cases}$$

(FORCED-ABT)
$$\frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \Box_1, 0 \rangle}{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box_2, 0 \rangle} \quad \Box_2 = \begin{cases} \overline{\boxtimes} & \text{if } \Box_1 = \boxdot \\ \overline{\boxast} & \text{otherwise} \end{cases}$$

(SAGA)
$$\frac{\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \sigma, \beta \rangle}{\Gamma \vdash \{[P]\} \xrightarrow{\alpha} \sigma} \quad \sigma \in \{\boxdot, \boxtimes, \boxast\}$$

Figure 6: Semantics of parallel sagas.

are named differently, we univocally identify the compensation of an activity $A \div B$ with $A^{-1} = B$.

DEFINITION 5 (ORDER OF A SAGA). *Let S be a parallel saga, the* order of a saga S *is the least transitive relation $\prec_S$ on $\mathcal{A}(S)$ s.t.:*

1. *if $A \div B$ occurs in S then $A \prec_S B$;*
2. *if $P; Q$ then $A \prec_S B$ $\forall A \in \mathcal{A}(|P|)$ and $\forall B \in \mathcal{A}(|Q|)$;*
3. *if $A, B \in \mathcal{A}(S)$ and $A \prec_S B$ then $B^{-1} \prec_S A^{-1}$.*

Given $\mathtt{A} \subseteq \mathcal{A}(S)$, we write $\prec_{S|\mathtt{A}}$ for the order $\prec_S$ restricted to the elements of $\mathtt{A}$. We will use $A \prec_S \{A_1, \ldots, A_n\}$ (and $\{A_1, \ldots, A_n\} \prec_S A$) if $\exists i$ s.t. $A \prec_S A_i$ (resp. $A_i \prec_S A$).

The adequacy is now expressed by three theorems.

THEOREM 2 (COMPLETION). *Given a parallel saga S, $\Gamma \vdash S \xrightarrow{\alpha} \boxdot$ iff $\forall A \in \mathcal{A}(|S|) : A \mapsto \boxdot \in \Gamma$ and $\alpha \equiv |S|$.*

THEOREM 3 (SUCCESSFUL COMPENSATION). *Let S be a parallel saga. $\Gamma \vdash S \xrightarrow{\alpha} \boxtimes$ iff there exists a non empty set $\mathcal{F}_\mathcal{A} \subseteq \mathcal{A}(|S|)$ of failed activities (i.e., $\forall A \in \mathcal{F}_\mathcal{A}, A \mapsto \boxtimes \in \Gamma$) s.t. $\forall A, B \in \mathcal{F}_\mathcal{A}$: $A \not\prec_S B$ and the following conditions hold:*

1. *$\prec_\alpha = \prec_{S|\mathcal{A}(\alpha)}$, i.e. the observed flow respects the flow given by S;*
2. *if $A \in \mathcal{A}(\alpha)$ then $A \mapsto \boxdot \in \Gamma$, i.e. all observed activities commit;*
3. *if $A \in \mathcal{F}_\mathcal{A}$ then $A \notin \mathcal{A}(\alpha)$, i.e. failed activities are not observed;*
4. *if $A \in \mathcal{A}(|S|)$ and $A \prec_S \mathcal{F}_\mathcal{A}$ then $A \in \mathcal{A}(\alpha)$, i.e. all activities that precede $\mathcal{F}_\mathcal{A}$ are executed successfully;*
5. *if $A \in \mathcal{A}(|S|)$ and $\mathcal{F}_\mathcal{A} \prec_S A$ then $A \notin \mathcal{A}(\alpha)$, i.e. all activities after $\mathcal{F}_\mathcal{A}$ in the forward flow are not run;*
6. *if $A \in \mathcal{A}(|S|)$ and $A \in \mathcal{A}(\alpha)$ then $A^{-1} \in \mathcal{A}(\alpha)$, i.e. all executed activities are compensated successfully.*

THEOREM 4 (ABNORMAL TERMINATION). *Given a parallel saga S. Then $\Gamma \vdash S \xrightarrow{\alpha} \boxast$ iff there exist a non empty set of failed activities $\mathcal{F}_\mathcal{A} \subseteq \mathcal{A}(|S|)$ s.t. $\forall A_1, A_2 \in \mathcal{F}_\mathcal{A}$: $A_1 \not\prec_S A_2$, and a non empty set of failed compensations $\mathcal{F}_\mathcal{C} \subseteq \mathcal{A}(S)$ s.t. $\mathcal{F}_\mathcal{C} \cap \mathcal{A}(|S|) = \emptyset$ and $\forall B_1, B_2 \in \mathcal{F}_\mathcal{C}$: $B_1 \not\prec_S B_2$, and the following conditions hold: 1 — 5 as in Theorem 3 and*

6'. *if $A \in \mathcal{A}(|S|)$ and $A \in \mathcal{A}(\alpha)$ and $A^{-1} \prec_S \mathcal{F}_\mathcal{C}$ then $A^{-1} \in \mathcal{A}(\alpha)$, i.e. activities whose compensations precede $\mathcal{F}_\mathcal{C}$ are compensated successfully;*

7. *if $A^{-1} \in \mathcal{F}_\mathcal{C}$ then $A^{-1} \notin \mathcal{A}(\alpha)$, i.e. failed compensations are not executed;*
8. *if $A \in \mathcal{A}(|S|)$ and $A \in \mathcal{A}(\alpha)$ and $\mathcal{F}_\mathcal{C} \prec_S A^{-1}$ then $A^{-1} \notin \mathcal{A}(\alpha)$, i.e. activities whose compensations follows $\mathcal{F}_\mathcal{C}$ are not compensated.*

Above results are a generalization of Theorem 1. In fact, the order of a sequential saga is a total order, and constraints in Theorems 2–4 reduce to conditions in Theorem 1.

4. ADDING NESTING TO SAGAS

Nesting has been introduced in database transactions to localize failures within a transaction and to allow partial rollbacks [19]. Basically, a nested transaction is decomposed into a hierarchy of activities called *subtransactions*. The root of the hierarchy is usually referred to as the *top-level transaction*. In this scheme, any subtransaction executes independently and concurrently with respect to its parent and siblings, deciding autonomously to commit or abort. When a transaction aborts all its subtransactions should abort and consequently all committed subtransactions must be rolled back. Nevertheless, a top-level transaction can commit even though some subtransactions have aborted.

DEFINITION 6 (NESTED SAGAS). *Nested sagas are defined by the following grammar:*

$$\begin{array}{llll} \text{(STEP)} & X & ::= & 0 \mid A \mid A \div B \\ \text{(PROCESS)} & P & ::= & X \mid P; P \mid P|P \mid S \\ \text{(SAGA)} & S & ::= & \{[P]\} \end{array}$$

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle}{\Gamma \vdash \langle \{\![P]\!\}, \beta\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'; \beta\rangle}\ (\textsc{sub-cmt}) \qquad \frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}{\Gamma \vdash \langle \{\![P]\!\}, \beta\rangle \xrightarrow{\alpha} \langle \boxdot, \beta\rangle}\ (\textsc{sub-abt}) \qquad \frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxast, 0\rangle}{\Gamma \vdash \langle \{\![P]\!\}, \beta\rangle \xrightarrow{\alpha} \langle \boxast, 0\rangle}\ (\textsc{sub-fail})$$

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \overline{\boxast}, 0\rangle}{\Gamma \vdash \langle \{\![P]\!\}, \beta\rangle \xrightarrow{\alpha} \langle \overline{\boxast}, 0\rangle}\ (\textsc{sub-forced-1}) \qquad \frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \overline{\boxtimes}, 0\rangle \quad \Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\gamma} \langle \Box_1, 0\rangle}{\Gamma \vdash \langle \{\![P]\!\}, \beta\rangle \xrightarrow{\alpha;\gamma} \langle \Box_2, 0\rangle}\ \Box_2 = \begin{cases} \overline{\boxtimes} & \text{if } \Box_1 = \boxdot \\ \overline{\boxast} & \text{if } \Box_1 \in \{\boxtimes, \boxast\} \end{cases}\ (\textsc{sub-forced-2})$$

Figure 7: Semantics of nested Sagas

The additional rules for nested sagas are in Figure 7. The main idea is that a subtransaction $\{\![P]\!\}$ executes P in an independent thread without initial compensation. The successful completion of $\{\![P]\!\}$ (rule SUB-CMT) is analogous to the case of a successful activity (rule S-ACT). When the subtransaction commits, the compensation β' computed by P is installed on top of the compensations.

Rule (SUB-ABT) describes the silent abortion of a subtransaction. As aforementioned, nesting is intended to allow the commit of a transaction even when some activities fail. That is, if an activity in P fails while running $\{\![P]\!\}$, and the executed activities of P are successfully compensated for, then the abort is hidden to the parent. For this reason the result associated with $\{\![P]\!\}$ is $\boxdot$ even though P aborts. (We discuss another possibility for handling this situation in Section 5.3). The observed flow corresponds to the execution of P and the original compensation β is not modified.

Instead, $\{\![P]\!\}$ ends abnormally when P has an abnormal termination (SUB-FAIL). This result is propagated until the top-level transaction, which will finish abnormally. (Section 5.2 introduces local handlers for abnormal termination).

The three rules described above do not allow a subtransaction to be stopped and compensated for when a concurrent activity aborts.

Consider the saga $S \equiv \{\![\ \{\![P]\!\}\ \mid\ A \div B]\!\}$ and a context $\Gamma = A \mapsto \boxtimes, \Gamma'$ such that $\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta\rangle$ and $\Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\alpha'} \langle \boxtimes, 0\rangle$, the whole saga S should abort because A aborts. With the rules seen until now we can build only the two derivations shown in Figure 8. The branch $\{\![P]\!\}$ is forced to abort either before (Figure 8(a)) or after (Figure 8(b)) the whole execution of P. Nevertheless, if P is a composed process, for instance a sequence, it is not possible to stop the execution of P once it starts. To allow the activation of the compensation procedure in subtransactions as soon as possible, we add the last two rules in Figure 7, which handle the interruption and compensation of a subtransaction. Rule (SUB-FORCED-1) handles the failure of the forced compensation. In this case the compensation β previously installed is not activated since the compensation procedure fails. On the contrary, rule (SUB-FORCED-2) activates β when P is compensated successfully.

EXAMPLE 3 (NESTED SAGAS). *The organization has a reward program in which users accumulate points when they purchase. The activity* Add points *updates the reward balance of a user. This activity aborts when the buyer is not part of the reward program. Clearly, we do not like the whole process to abort when the user is not registered, for this reason we model this activity as a nested transaction (see Figure 9). The compensation* Subtract points *undoes the step* Add points *when some activity in the top level flow fails.*

To formalize the adequacy theorems we need some preliminary definitions.

DEFINITION 7 (SUBTRANSACTIONS). *The set of all subtransactions of S are $\mathcal{S}(S) = \{\{\![P]\!\} \mid \{\![P]\!\}$ is a proper subterm of $S\}$ while the top-level subtransactions are $\mathcal{S}_{top}(S) = \{S' | S' \in \mathcal{S}(S) \wedge \forall S'' \in \mathcal{S}(S) : S' \notin \mathcal{S}(S'')\}$. The set of all top-level activities of S is $\mathcal{A}_{top}(S) = \{A \mid A \in \mathcal{A}(|S|)$ and A does not occur in a subterm $\{\![P]\!\}\}$.*

The definition of order of a saga considers only top activities and subtransactions (i.e., A and B range over top activities and subtransactions, and A^{-1} denotes also compensations of subtransactions seen as symbolic atoms). Again, we break the adequacy results in three theorems.

THEOREM 5 (COMPLETION). *Let S be a nested saga. Then $\Gamma \vdash S \xrightarrow{\alpha} \boxdot$ iff*

1. *if $A \in \mathcal{A}_{top}(|S|)$ then $A \mapsto \boxdot \in \Gamma$;*
2. *if $S' \in \mathcal{S}_{top}(S)$ then $\Gamma \vdash \langle S', 0\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta\rangle$ for some α', β.*

THEOREM 6 (SUCCESSFUL COMPENSATION). *Let S be a nested saga. Then $\Gamma \vdash S \xrightarrow{\alpha} \boxtimes$ iff there exists a non empty set of failed activities $\mathcal{F}_{\mathcal{A}} \subseteq \mathcal{A}_{top}(|S|)$ s.t. $\forall A, B \in \mathcal{F}_{\mathcal{A}}$: $A \not\prec_S B$, and the following conditions hold:*

1. *if $A \in \mathcal{A}(\alpha)$ then $A \mapsto \boxdot \in \Gamma$;*
2. *if $A \in \mathcal{F}_{\mathcal{A}}$ then $A \notin \mathcal{A}(\alpha)$;*
3. *if $A \in \mathcal{A}_{top}(|S|)$ and $A \prec_S \mathcal{F}_{\mathcal{A}}$ then $A \in \mathcal{A}(\alpha)$, i.e. all top activities that precede $\mathcal{F}_{\mathcal{A}}$ are executed successfully;*
4. *if $S' \in \mathcal{S}_{top}(S)$ and $S' \prec_S \mathcal{F}_{\mathcal{A}}$ then $\Gamma \vdash \langle S', 0\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta\rangle$, and $\Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\gamma} \langle \boxdot, 0\rangle$ for some α', β and γ, i.e. all top subtransactions before $\mathcal{F}_{\mathcal{A}}$ (and their compensations) are successful;*
5. *if $A \in \mathcal{A}_{top}(|S|)$ and $\mathcal{F}_{\mathcal{A}} \prec_S A$ then $A \notin \mathcal{A}(\alpha)$, i.e. all activities in the forward flow after $\mathcal{F}_{\mathcal{A}}$ are not executed;*
6. *if $S' \in \mathcal{S}_{top}(S)$ and $\mathcal{F}_{\mathcal{A}} \prec_S S'$ then $\forall A \in \mathcal{A}(S')$: $A \notin \mathcal{A}(\alpha)$, i.e. activities of subtransactions following $\mathcal{F}_{\mathcal{A}}$ are not executed;*
7. *if $A \in \mathcal{A}(|S|)$ and $A \in \mathcal{A}(\alpha)$ then $A^{-1} \in \mathcal{A}(\alpha)$, i.e. all executed activities are compensated successfully.*

THEOREM 7 (ABNORMAL TERMINATION). *Let S be a nested saga. Then $\Gamma \vdash S \xrightarrow{\alpha} \boxast$ iff there exist: (i) a set of failed activities $\mathcal{F}'_{\mathcal{A}} \subseteq \mathcal{A}(|S|)$ and (ii) a set of abnormal terminated subtransactions $\mathcal{F}_{\mathcal{S}} \subseteq \mathcal{S}_{top}(S)$ s.t.: $\mathcal{F}_{\mathcal{A}} = \mathcal{F}'_{\mathcal{A}} \cup \mathcal{F}_{\mathcal{S}}$*

$$
\dfrac{
  \dfrac{\dfrac{}{\Gamma \vdash \langle 0,0\rangle \xrightarrow{0} \langle \boxdot,0\rangle}\,\textsc{zero}}{\Gamma \vdash \langle \{\![P]\!\},0\rangle \xrightarrow{0} \langle \overline{\boxtimes},0\rangle}\,\textsc{forced-abt}
  \quad
  \dfrac{\dfrac{}{\Gamma \vdash \langle 0,0\rangle \xrightarrow{0} \langle \boxdot,0\rangle}\,\textsc{zero}}{A \mapsto \boxtimes, \Gamma' \vdash \langle A \div B,0\rangle \xrightarrow{0} \langle \boxtimes,0\rangle}\,\textsc{s-cmp}
  \quad
  \dfrac{}{\Gamma \vdash \langle 0,0\rangle \xrightarrow{0} \langle \boxdot,0\rangle}\,\textsc{zero}
}{
  \dfrac{\Gamma \vdash \langle \ \{\![P]\!\} \ \mid A \div B \ ,0\rangle \xrightarrow{0} \langle \boxtimes,0\rangle}{\Gamma \vdash \langle \{\![\ \{\![P]\!\} \ \mid A \div B \ ]\!\},0\rangle \xrightarrow{0} \langle \boxdot,0\rangle}\,\textsc{sub-abt}
}\,\textsc{c-par}
$$

(a) $\{\![P]\!\}$ is not activated

$$
\dfrac{
  \dfrac{
    \dfrac{\vdots}{\Gamma \vdash \langle \{\![P]\!\},0\rangle \xrightarrow{\alpha} \langle \boxdot,\beta\rangle}
    \quad
    \dfrac{\dfrac{\vdots}{\Gamma \vdash \langle \beta,0\rangle \xrightarrow{\gamma} \langle \boxtimes,0\rangle}}{\Gamma \vdash \langle 0,\beta\rangle \xrightarrow{\gamma} \langle \overline{\boxast},0\rangle}\,\textsc{forced-abt}
  }{\Gamma \vdash \langle \{\![P]\!\};0,0\rangle \xrightarrow{\alpha;\gamma} \langle \overline{\boxast},0\rangle}\,\textsc{s-step}
  \quad
  \dfrac{\dfrac{}{\Gamma \vdash \langle 0,0\rangle \xrightarrow{0} \langle \boxdot,0\rangle}\,\textsc{zero}}{A \mapsto \boxtimes, \Gamma' \vdash \langle A \div B,0\rangle \xrightarrow{0} \langle \boxtimes,0\rangle}\,\textsc{s-cmp}
}{
  \dfrac{\Gamma \vdash \langle \ \{\![P]\!\} \ \mid A \div B \ ,0\rangle \xrightarrow{\alpha;\gamma|0} \langle \boxast,0\rangle}{\Gamma \vdash \langle \{\![\ \{\![P]\!\} \ \mid A \div B \ ]\!\},0\rangle \xrightarrow{\alpha;\gamma|0} \langle \boxast,0\rangle}\,\textsc{sub-fail}
}\,\textsc{f-par}
$$

(b) P is completely executed

Figure 8: Possible executions of $S \equiv \{\![\ \{\![P]\!\} \ \mid \ A \div B]\!\}$ when $\Gamma = A \mapsto \boxtimes, \Gamma'$.

is not empty and $\forall A_1, A_2 \in \mathcal{F}_{\mathcal{A}}$*:* $A_1 \not\prec_S A_2$*, (iii) a set of failed compensations* $\mathcal{F}'_{\mathcal{C}} \subseteq \mathcal{A}_{top}(S)$ *s.t* $\mathcal{F}'_{\mathcal{C}} \cap \mathcal{A}(|S|) = \emptyset$*, (iv) a set of precommitted subtransactions with failed compensations* $\mathcal{F}_{\mathcal{P}} \subseteq \mathcal{S}_{top}(S)$ *s.t.* $\mathcal{F}_{\mathcal{C}} = \mathcal{F}'_{\mathcal{C}} \cup {\mathcal{F}_{\mathcal{S}}}^{-1} \cup {\mathcal{F}_{\mathcal{P}}}^{-1}$ *is not empty and* $\forall A_1, A_2 \in \mathcal{F}_{\mathcal{C}}$*:* $A_1 \not\prec_S A_2$*, and the following conditions hold: conditions 1,3,5,6 as in Theorem 6 and*

2'. *if* $A \in \mathcal{F}'_{\mathcal{A}}$ *then* $A \notin \mathcal{A}(\alpha)$

4'. *if* $S' \in \mathcal{S}_{top}(S)$*,* $S' \prec_S \mathcal{F}_{\mathcal{A}}$*, and* ${S'}^{-1} \prec_S \mathcal{F}_{\mathcal{C}}$ *then* $\Gamma \vdash \langle S',0\rangle \xrightarrow{\alpha'} \langle \boxdot,\beta\rangle$ *and* $\Gamma \vdash \langle \beta,0\rangle \xrightarrow{\gamma} \langle \boxdot,0\rangle$

7'. *if* $A \in \mathcal{A}_{top}(|S|)$ *and* $A \in \mathcal{A}(\alpha)$ *and* $A^{-1} \prec_S \mathcal{F}_{\mathcal{C}}$ *then* $A^{-1} \in \mathcal{A}(\alpha)$*, i.e. activities whose compensations precede* $\mathcal{F}_{\mathcal{C}}$ *are compensated successfully.*

8. *if* $S' \in \mathcal{F}_{\mathcal{S}}$ *then* $\Gamma \vdash \langle S',0\rangle \xrightarrow{\alpha'} \langle \sigma,0\rangle$ *with* $\sigma \in \{\boxast, \overline{\boxast}\}$*, i.e. failed subtransactions terminate abnormaly;*

9. *if* $S' \in \mathcal{F}_{\mathcal{P}}$ *then* $S' \prec_S \mathcal{F}_A$*,* $\Gamma \vdash \langle S',0\rangle \xrightarrow{\alpha'} \langle \boxdot,\beta\rangle$ *and* $\Gamma \vdash \langle \beta,0\rangle \xrightarrow{\gamma} \langle \boxtimes,0\rangle$*, i.e. failed precommitted subtransactions complete successfully but their installed compensations fail.*

10. *if (*$S' \in \mathcal{S}_{top}(S)$ *and* $\mathcal{F}_{\mathcal{C}} \prec_S {S'}^{-1}$ *and* $A \in \mathcal{A}(S')$*) or (*$A \in \mathcal{A}_{top}(S)$ *and* $\mathcal{F}_{\mathcal{C}} \prec_S A^{-1}$*) then* $A^{-1} \notin \mathcal{A}(\alpha)$*, i.e. compensations after* $\mathcal{F}_{\mathcal{C}}$ *are skipped.*

Previous results do not characterize precisely the order of the observation α (as in previous sections). Instead they state the set of executed steps and the result they produce.

5. ADDITIONAL FEATURES

This section presents further extensions of nested sagas.

5.1 Programmable compensations

The compensation mechanism described until now is usually referred to as *implicit* or *default* compensation. In addition, some composition languages (such as BPEL4WS) allow

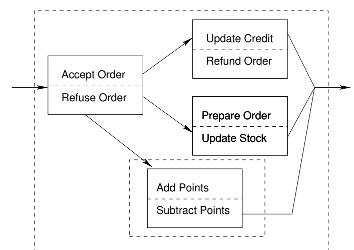

Figure 9: A nested saga for handling orders.

the programmer to explicitly define the compensation procedure associated with a completed subtransaction. In our case, the syntax of steps should include terms with the following form: $S \div P$. Consider the long running transaction $S \equiv \{\![\ \{\![A_1 \div B_1; A_2 \div B_2]\!\} \div P \ ; A_3]\!\}$, which should behave as follows: when A_1 commits and A_2 aborts, the default mechanism should compensate A_1 by activating B_1. Instead, if A_1 and A_2 commit while A_3 aborts, the programmed compensation P (and not the default $B_2; B_1$) should be run.

The difference between default and programmable compensations is that the former are always flat processes without compensations whose executions always produce results like $\langle \Box, 0\rangle$, while the latter can produce $\langle \Box, \beta\rangle$. Let $P \equiv C_1 \div D_1; \{\![Q]\!\} \div Q'; C_2 \div D_2$ in S above. Clearly, if A_3 fails, then P is activated and it can commit, abort or terminate abnormally, but it may also generate compensations. In particular, if P commits, then it produces $\langle \boxdot, D_2; Q'; D_1\rangle$ but the rules used in previous sections, which assume compensations not to generate new compensations (i.e. S-CMP, F-CMP, C-PAR, FORCED-ABT and SUB-FORCED-2), will not handle this expected behavior.

One alternative for dealing with programmable compensations is to restrict their syntax to allow only basic activities or processes without compensations. Similarly, without imposing a syntactical restriction, to make compensations to behave as their forward flow, as follow.

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle}{\Gamma \vdash \langle \{[P]\} \div Q, \beta\rangle \xrightarrow{\alpha} \langle \boxdot, |Q|;\beta\rangle} \quad (\textsc{pgm-cmp})$$

This rule is similar to (SUB-CMT) in Figure 7 that installs the default compensation β'. Differently, rule (PGM-CMP) discards β' and replaces it by the forward flow $|Q|$ of Q. We recall that the forward flow of a process is obtained by replacing in P each term $Q \div Q^{-1}$ by Q. This will assure that the execution of a compensation never generates new compensations neither terminates abnormally.

On the contrary, it could be possible to take into account compensations produced by compensations (as done in StAC) and to install and use them to repeatedly compensate a process. For instance by adding the following rule

$$\frac{\Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha} \langle \boxtimes, \beta''\rangle \quad \Gamma \vdash \langle \beta'', 0\rangle \xrightarrow{\alpha'} \langle \Box, \beta'\rangle \quad \beta'' \not\equiv 0}{\Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box, \beta'\rangle} \quad (\textsc{repeated-comp})$$

Consider the process $\{[\ \{[P]\} \div A_0\ ; A_1 \div (B_1 \div C_1);\ R\ ]\}$ and an execution in which $\{[P]\}$ commits and installs A_0 as a compensation. Then, A_1 commits and installs $B_1 \div C_1$ on top. Suppose now that R fails and starts the compensation procedure by executing $B_1 \div C_1$. If B_1 commits, C_1 will be installed and could be activated later on, for instance when A_0 aborts. Note that this kind of definitions generates an upward flow of control when a compensation fails, i.e. the failure of A_0 activates C_1. In our approach, where compensations are used to undo committed steps, the meaning of such a construction is quite obscure. Basically, it would mean that a successful execution of A_1 can be undone by running B_1, which can be in turn compensated with C_1. In particular if they are perfect compensations, i.e. they remove all the effects, the term $A_1 \div (B_1 \div C_1)$ leaves all the effects of A_1 when the compensation procedure fails. Moreover it is difficult to figure out real cases in which repeated compensation is really necessary. In our opinion the failure of a compensation can be modelled more naturally by exploiting an exception handling mechanism like the one presented in Section 5.2. For this reason, we prefer rule (PGM-CMP) instead of (REPEATED-COMP) for handling programmable compensations.

5.2 Exception handling

A basic exception handling mechanism can be added to the presented languages by interpreting the result $\langle \boxast, \beta\rangle$ as a process that raises an exception. At the syntactic level, we can consider exception handlers introduced by steps like **try** S **with** P, where S is a saga and P a generic process. The behavior for such processes is defined in Figure 10.

The first rule handles the case in which S_1 finishes without raising an exception. As usual, the exception handler P_2 is discarded, and the compensation created by S_1 is installed on top of the stack. The second rule describes the activation of the handler P_2 when S_1 raises an exception. Note that P_2 starts with the original compensation β. The last rule handles the activation of the compensation handler when the abnormal termination is reached during a forced compensation. The handler is also run in this case because it is intended to finish the compensation procedure. Note that the final result is always a forced termination.

$$\frac{\Gamma \vdash \langle S, 0\rangle \xrightarrow{\alpha} \langle \Box, \beta'\rangle}{\Gamma \vdash \langle \textbf{try } S \textbf{ with } P, \beta\rangle \xrightarrow{\alpha} \langle \Box, \beta';\beta\rangle} \quad \textit{with } \Box \in \{\boxdot, \boxtimes, \overline{\boxtimes}\} \quad (\textsc{no-excp})$$

$$\frac{\Gamma \vdash \langle S, 0\rangle \xrightarrow{\alpha} \langle \boxast, 0\rangle \quad \Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha'} \langle \Box, \beta'\rangle}{\Gamma \vdash \langle \textbf{try } S \textbf{ with } P, \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box, \beta'\rangle} \quad (\textsc{excp})$$

$$\frac{\Gamma \vdash \langle S, 0\rangle \xrightarrow{\alpha} \langle \overline{\boxast}, 0\rangle \quad \Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha'} \langle \Box_1, \beta'\rangle}{\Gamma \vdash \langle \textbf{try } S \textbf{ with } P, \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box_2, \beta'\rangle} \quad \textit{with } \Box_2 = \begin{cases} \overline{\boxast} & \text{if } \Box_1 = \boxast \\ \overline{\boxtimes} & \text{otherwise} \end{cases} \quad (\textsc{forced-excp})$$

Figure 10: Semantics of exception handling

Although the described mechanism is naïve, it illustrates the interplay between both concepts: compensations undo partial executions of transactions, while exception handling deals with incomplete compensations.

5.3 Alternatives to aborted subtransactions

In the nested model we presented in Section 4, the behavior of a parent transaction does not depend on the completion/abortion of its subtransactions. In fact, rule SUB-ABT hides to the parent transaction the fact that one (or more) of its subtransactions have not been executed (i.e., compensated). Nevertheless, workflow systems usually allow the possibility of specifying forward recovery strategies for a process that fails to commit, such as the retry of the activity or the execution of an alternative process.

These aspects can be modelled by extending the language with new primitive steps **try** S **or** P whose behavior can be described with the following rules

$$\frac{\Gamma \vdash \langle S, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle}{\Gamma \vdash \langle \textbf{try } S \textbf{ or } P, \beta\rangle \xrightarrow{\alpha} \langle \boxdot, \beta';\beta\rangle} \quad (\textsc{s-alt})$$

$$\frac{\Gamma \vdash \langle S, 0\rangle \xrightarrow{\alpha} \langle \Box, 0\rangle}{\Gamma \vdash \langle \textbf{try } S \textbf{ or } P, \beta\rangle \xrightarrow{\alpha} \langle \Box, 0\rangle} \quad \textit{with } \Box \in \{\boxast, \overline{\boxtimes}, \overline{\boxast}\} \quad (\textsc{f-alt})$$

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle \quad \Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha'} \langle \Box, \beta'\rangle}{\Gamma \vdash \langle \textbf{try } S \textbf{ or } P, \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box, \beta'\rangle} \quad (\textsc{t-alt})$$

This mechanism is similar to the exception handling described above. Nevertheless, while exception handling is used during backward computation (for failed compensations) alternative procedures are used as a forward recovery mechanism. For this reason, an alternative is activated only when the subtransaction aborts. Moreover, by rule (F-ALT), which shift the forced abortion $\overline{\boxtimes}$ to the parent level, alternatives are not executed during a forced termination. In fact, alternatives are intended to be used while executing towards a completion not during the compensation procedure.

5.4 Choices

The recovery capability introduced above allows the sequential search of one process that executes successfully. Some composition languages (like BPML [6]) allow alterna-

(S-CHOICE)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \Box, 0\rangle}{\Gamma \vdash \langle P \boxplus Q, \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, \beta'; \beta\rangle} \qquad \textit{with}\ \Box \in \{\boxtimes, \overline{\boxtimes}\}$$

(F-CHOICE)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \sigma_1, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha} \langle \sigma_2, 0\rangle}{\Gamma \vdash \langle P \boxplus Q, \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \sigma_1 \wedge \sigma_2, 0\rangle} \qquad \begin{cases} \sigma_1 \in \{\boxast, \overline{\boxast}\} \\ \sigma_2 \in \{\boxtimes, \boxast, \overline{\boxtimes}, \overline{\boxast}\} \end{cases}$$

(C-CHOICE)

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \sigma_1, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \sigma_2, 0\rangle \quad \Gamma \vdash \langle \beta, 0\rangle \xrightarrow{\gamma} \langle \Box_1, 0\rangle}{\Gamma \vdash \langle P \boxplus Q, \beta\rangle \xrightarrow{(\alpha|\alpha');\gamma} \langle \sigma_1 \wedge \sigma_2 \wedge \Box_2, 0\rangle} \qquad \sigma_1, \sigma_2 \in \{\boxtimes, \overline{\boxtimes}\}\ \text{and}\ \Box_2 = \begin{cases} \overline{\boxtimes} & \text{if } \Box_1 = \boxdot \\ \overline{\boxast} & \text{otherwise} \end{cases}$$

Figure 11: Semantics for the choice of a successful branch: $P \boxplus Q$

tives to be explored in parallel (this kind of choice is known as *discriminator*). Once one branch finishes successfully all the remaining alternatives are stopped and compensated for. We will write these processes as $P \boxplus Q$, and we assume $\boxplus$ associative and commutative. Inference rules are in Figure 11. The last two rules (i.e., abnormal termination (F-CHOICE) and abort ((C-CHOICE)) are analogous to those for parallel composition. Differently, a choice $P \boxplus Q$ succeeds only when one branch commits and the other has been successfully (possible forced) compensated for (rule S-CHOICE). A computation cannot go forward when one branch terminates abnormally because the state of the system is inconsistent. Hence, the successful branch is forced to compensate and the whole process $P \boxplus Q$ ends abnormally (F-CHOICE).

These kind of choices, where the selection takes place once one of the branches has completed successfully, are quite different to usual internal or external choices. Internal choices $P \sqcap Q$ can be defined straightforwardly by defining a unique rule and requiring $\sqcap$ to be associative and commutative.

(INT-CHOICE)

$$\frac{\Gamma \vdash \langle P, \beta\rangle \xrightarrow{\alpha} \langle \Box, \beta'\rangle}{\Gamma \vdash \langle P \sqcap Q, \beta\rangle \xrightarrow{\alpha} \langle \Box, \beta'\rangle}$$

External choices are related to the notion of synchronization of events that make the description of flows not structured. We analyze more in detailed the synchronization between flows in the following section.

5.5 Link dependencies

This section discusses the synchronization of concurrent flows. Consider the flow depicted in Figure 12, and let

$$P \equiv A_1 \div B_1; (A_2 \div B_2; A_4 \div B_4 | A_3 \div B_3; A_5 \div B_5); A_6 \div B_6$$

with the additional constraint stating that A_4 must be executed after A_3, written $\mathtt{link}(\mathtt{A}_3, \mathtt{A}_4)$. Hence, any valid execution α of P must hold both the order given by P and the additional constraints $A_3 \prec_\alpha A_4$. Although all languages agree on this meaning for links (or synchronization) while computing forward, it is less clear which is the desired behavior when compensating. For instance, StAC (which provides an operator for parallel composition with synchronization over a name set) ignores all synchronizations when computing backward. For instance, if A_6 fails during the execution of P, then, according to the compensation policy of StAC, the compensation procedure could activate B_3 before the termination of B_4. In our opinion this semantics has a main drawback in that the encoding of sequential composition as a synchronization between parallel flows has a different meaning when compensating. Consider the sequential process $P \equiv A_1 \div B_1; A_2 \div B_2; Q$, and $P' \equiv A_1 \div B_1 \mid A_2 \div B_2; Q$ with $\mathtt{link}(\mathtt{A}_1, \mathtt{A}_2)$. It is clear that requiring $A_1 \prec_\alpha A_2$ does not make any execution α of P a valid execution of P': we also need $B_2 \prec_\alpha B_1$.

The following definitions formalize the notion of valid execution for a structured flow process with links.

DEFINITION 8 (ORDER OF A SAGA WITH LINKS). *Let S be a parallel saga and $L = \{\mathtt{link}(A_i, A_j) | A_i, A_j \in \mathcal{A}(S)\}$ be a set of links. The order $\prec_{S,L}$ is the least transitive and antisymmetric relation (if defined) satisfying: (i) $\prec_S \subseteq \prec_{S,L}$, and (ii) $\forall \mathtt{link}(A_i, A_j) \in L$, $A_i \prec_{S,L} A_j$ and $A_j^{-1} \prec_{S,L} A_i^{-1}$.*

Clearly, when L introduces cycles in the control flow the order is not defined. The following definition singles out those executions that satisfy a set of well-defined links.

DEFINITION 9 (VALID EXECUTION WITH LINKS). *Let S be a parallel saga and L a set of links s.t. $\prec_{S,L}$ is defined. An order α is a valid execution of S with links L iff $\Gamma \vdash S \xrightarrow{\alpha'} \Box$ and $\alpha = \prec_{\alpha',L}$.*

The definition above simply states that a valid execution of a process with links is an execution of the process that also satisfies the dependency constraints.

6. CONCLUSION AND FUTURE WORKS

We have presented several primitives and mechanisms for the specification and execution of long running transactions in flow composition languages. In this context, the key issue is the backward compensation of completed activities upon abortion at a later stage of the transaction: compensations must be programmable and installable.

Starting from the formalization of sequential sagas with a minimal set of primitives, we have progressively enriched the language with primitives for dealing with parallelism, nesting, exception handling and choices. In particular, parallel composition and nesting requires a careful analysis of the mechanisms used for notifying failures to siblings and forcing their abortion. We have given to each language a neat big step semantics and proved its adequacy with respect to the informal requirements of each different kind of sagas.

This work follows a thread (started not earlier than four years ago) concerning the formalization of transactions via

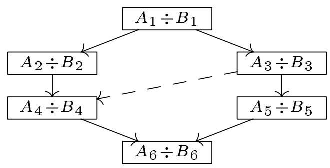

Figure 12: A flow with links.

process description languages. Works in the literature roughly fall into one of the categories below:

Extensions of well-known calculi that exploits the original characteristics of the calculus to describe flow or interaction based composition.(Like dialects of the π-calculus [4], of the Join [8], and of the object calculus [15]).

Languages for the description of business processes. They are generally graphical or XML-based languages that do not provide a formal / unambiguous definition of their semantics. All well-known (proposed) standards fall into this category, for instance XLANG [21], WSFL [16], BPEL4WS [5].

Formal definition of compensations for flow composition languages. To the best of our knowledge StAC is the only proposal in this category. Nevertheless, since the use of operators and transaction scopes are not well-disciplined in StAC, it allows writing processes with obscure meaning.

This work is a next step in the line initiated by StAC. Our goal is to formalize the relation of compensation with ordinary primitives in flow languages and to highlight alternative meanings for them (such as forced termination of concurrent processes vs. independent completion of threads, silent abort of subtransactions vs. forward recovery). Nevertheless, there are several aspects in this work that still require investigation. For instance, we do not include usual imperative features, such as state (or variables), control structures like branching or iteration, neither data communication between activities (i.e. parameter passing). We abstract away from the fact that compensations usually require appropriate data when activated. Dependency links are first step towards this direction, but undoubtedly, more work is needed to formally explain the "state" seen by compensations. We leave these issues as future work.

Acknowledgments. We thank Michael Butler and Carla Ferreira for the discussions about StAC semantics that motivated this work, and Dan Hirsch for helpful comments.

7. REFERENCES

[1] W. Aalst, M. Dumas, and A. Hofstede. Web service composition languages: Old wine in new bottles? *Proc. of EUROMICRO'03*, pp. 298—307. IEEE Computer Society, 2003.

[2] W. Aalst, M. Dumas, and A. Hofstede, and P. Wohed, Analysis of web services composition languages: The case of BPEL4WS. *Proc. of ER'03*, vol. 2813 of *LNCS*, pp. 200–215. Springer, 2003.

[3] B. Benatallah and R. Hamadi. A Petri net-based model for web service composition. *Proc. of ADC'03*, pp. 191–200. Australian Computer Society, 2003.

[4] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. *Proc. of FMOODS'03*, vol. 2884 of *LNCS*, pp. 124–138. Springer, 2003.

[5] BPEL Specification. Version 1.1. Available at `http://www.ibm.com/developerworks/library/ws-bpel`.

[6] Business process modelling language (BPML). `http://www.bpmi.org`.

[7] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. *Proc. of WS-FM'04*. To appear as ENTCS.

[8] R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending Join. *Proc. of IFIP-TCS'04*, pp. 569–582. Kluwer, 2004.

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

[10] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. *Proc. of Coordination'04*, vol. 2949 of *LNCS*, pp. 87–104. Springer, 2004.

[11] P. Chrysanthis and K. Ramamritham. *Transaction Models for Advanced Applications*, ACTA: The SAGA Continues, pp. 349—397. Morgan Kaufmann, 1992.

[12] H. Garcia-Molina and K. Salem. Sagas. *Proc. of ACM SIGMOD'87*, pp. 249–259. ACM Press, 1987.

[13] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[14] C. Hoare. Long-running transactions. Slides for the Second Microsoft .NET Crash Course 2002. `http://research.microsoft.com/Collaboration/University/Europe/Events/dotnetcc/Version2`.

[15] A. Hosking, S. Jagannathan, J. Vitek, and A. Welc. A semantic framework for designer transactions. *Proc. of ESOP'04*, vol. 2986 of *LNCS*, pp. 249–263. Springer, 2004.

[16] F. Leymann. The WSFL Guide. Available at `http://www.ibm.com/software/solutions/webservices/documentation.html`.

[17] M. Mazzara and R. Lucchi. A framework for generic error handling in business processes. *Proc. WS-FM'04*. To appear as ENTCS.

[18] L. G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[19] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Dept. of Electrical Eng. and Computer Sci., MIT, 1981.

[20] A. Sheth and D. Worah. Transactions in transactional workflows. *Advanced Transaction Models and Architectures*, pp. 3–34. Kluwer, 1997.

[21] S. Thatte. XLANG: Web Services for Business Process Design. Available at `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`.

[22] M. Viroli. Towards a formal foundation to orchestration languages. *Proc. of WS-FM'04*. To appear as ENTCS.

[23] WebSphere Software Platform. IBM. Available at `http://www.ibm.com/software/websphere`.

[24] WSCI Specification. Version 1.0. Available at `http://www.w3.org/TR/wsci`.