Fault, Compensation and Termination in WS-BPEL 2.0 — A Comparative Analysis

Christian Eisentraut and David Spieler

Department of Computer Science, Saarland University,
Campus Saarbrücken, 66123 Saarbrücken, Germany
{eisentraut,spieler}@cs.uni-sb.de

Abstract. One of the most challenging aspects in Web Service composition is guaranteeing transactional integrity. This is usually achieved by providing mechanisms for fault, compensation and termination (FCT) handling. WS-BPEL 2.0, the de-facto standard language for Business Process Orchestration provides powerful scope-based FCT-handling mechanisms. However, the lack of a formal semantics makes it difficult to understand and implement these constructs, and renders rigid analysis impossible. The general concept of compensating long-running business transactions has been studied in different formal theories, such as cCSP and Sagas, but none of them is specific to WS-BPEL 2.0. Other approaches aim at providing formal semantics for FCT-handling in WS-BPEL 2.0, but only concentrate on specific aspects. Therefore, they cannot be used for a comparative analysis of FCT-handling in WS-BPEL 2.0. In this paper we discuss the BPEL approach to FCT-handling in the light of recent research. We provide formal semantics for the WS-BPEL 2.0 FCT-handling mechanisms which aims at capturing the FCT-part of the WS-BPEL 2.0 specification in full detail. We then compare the WS-BPEL 2.0 approach to FCT-handling to existing formal theories.

1 Introduction

As a standard for Web Service Orchestration, the language WS-BPEL 2.0 [1] (in the following BPEL) has gained wide acceptance over the past years. BPEL provides primitives to specify the flow of execution and communication between a process and its communication partners. Similar to the notion of transaction in database systems, the successful completion of certain communication sequences between processes must be ensured in order not to bring the course of business between the partners into an inconsistent state. Different to transactions of database systems, transactions of orchestrations are inherently long running (Long-Running-Transactions, LRT) with typical durations from hours to days, for example because of sub-transactions that require human interaction or batch processing, and may be subject to faults of various kinds. Therefore, the use of the transaction paradigms as used in database systems, where resources are locked for exclusive access, becomes infeasible in this context. Therefore BPEL uses a concept called compensation in order to obtain a more relaxed notion of undoing partial executions. Every activity within a transaction possesses an associated compensation that is (ideally) able to revert its effects. When a transaction fails, the effects of all activities executed within that transaction so far are undone by executing their respective compensations. This makes transactions behave atomically for an external observer: they always either complete successfully or they appear to have never been started at all. It seems appropriate to execute compensations in the reverse order relative to the execution of their respective activities. In BPEL this is the default order.

The compensation approach to LRTs is based on the seminal idea of Sagas [7]. [3] provides a formalization of Sagas together with several extensions to the basic calculus, such as nested transactions, programmable compensations and exception handling, which are vital ideas of modern Web Service composition languages like BPEL. A similar approach is pursued in cCSP [5] based on CSP [8]. Different approaches to and other aspects of LRTs, which we will not consider in our paper, have been studied in [4, 11, 12].

While the theory is thus reasonably well developed, the effective and reliable use of LRT concepts is of crucial importance for Web Services. Therefore LRT and compensation are an integral part of BPEL. However, it is difficult to analyze how LRTs are realized in BPEL, since the language does not come with formal semantics. While desirable formal properties of LRTs are known, they are uncheckable with the current BPEL specification.

Nested Sagas and cCSP are two recent formal approaches to the foundations of FCT-handling. However, due to the very different styles of giving semantics that are used for BPEL and Sagas and cCSP respectively, it seems almost impossible to faithfully validate to which extent the formal semantics of the calculi and the BPEL specification coincide without formal proofs.

In this paper, we provide an formalization of fault, compensation and termination handling in BPEL, following the BPEL documents as faithful as at all possible. The resulting operational small-step semantics describes BPEL in a step-wise fashion, which is small-grained enough to be validated as a formal translation of the specification. Since both our calculus and Sagas or cCSP are formulated in a process algebraic way, our calculus furthermore seems a natural choice for a formal comparison.

It turns out that an interesting fragment of BPEL corresponds to Sagas, as far as basic transaction handling without nesting is considered. This result is especially interesting since Sagas is designed in a syntactically and semantically slim and clean way, which makes it an appropriate choice for formal analysis. As shown in [2], the same subset also coincides with a subset of cCSP. We can therefore restrict our formal comparison to Sagas.

However, more advanced features of the languages like generalized exception handling, programmable compensations etc. seem incomparable.

To keep the calculus as simple as possible, we do not consider aspects of BPEL that are orthogonal to FCT-handling like correlation sets or the binding of partner links. Furthermore, we only consider control flow primitives that are also found in Sagas. Section 3 provides a more detailed discussion of our choice.

We furthermore abstract from time and data. For a formal treatment of these aspects we refer the reader for example to [13, 10, 18, 17, 14].

In summary, this paper makes the following contributions. It (*i*) provides a detailed, low-level semantics for BPEL's FCT-handling behavior and (*ii*) uses this semantics to place BPEL in the context of more abstract foundational work.

In Section 2 we give an overview of existing approaches to the formalization of BPEL. Syntax and semantics of our calculus $BPEL_{fct}$ are presented in Section 3. We also give a short introduction to the principle of *all-or-nothing* semantics in BPEL and how it is realized in our calculus. Section 4 formally compares $BPEL_{fct}$ to nested Sagas and discusses possible extensions and limitations of our comparative approach. Section 5 concludes our work.

2 Related Work

Several formalizations of BPEL and its FCT-handling mechanisms exist. For an overview see [6]. Most algebraic approaches either consider BPEL4WS, the predecessor of BPEL, or do not give full account to all features of FCT-handling in BPEL. [13] and [9] provide feature-complete semantics for BPEL. However, their graph-based semantics seem a less natural choice for the intended comparison with the algebraic calculus Sagas.

[15, 11, 12] show how compensation handling can be reduced to event handling in the webπ-calculus. However, their approach relies on statically specified compensation handlers. Thus BPELs default compensation and all-on-nothing semantics (cf. Section 3.2) are not represented in their model. In [14] a nearly complete encoding of the BPEL scope construct into the $web\pi_{\infty}$-calculus is given, although without termination handler. That calculus is derived from the $web\pi$-calculus and consequently suffers from the problems mentioned above.

In [19], Qiu et al. introduce the calculus $BPEL$ formalizing a subset of BPEL4WS. Our calculus was inspired by $BPEL$. However, as we will show in Section 3.2, their calculus cannot deal with *all-or-nothing semantics* to its full extent.

3 The $BPEL_{fct}$ Calculus

Being a real-word language, BPEL comes with a huge number of primitives needed to design business processes, for example partner links and correlation sets for communication purposes, many different control flow primitives and FCT-handling mechanisms, to only mention a few. It is in general not easy to decouple these mechanism and analyze them in isolation. For example, the FCT-handling mechanism we want to consider cannot be separated completely from the control flow primitives. When a faulted transaction needs to be compensated, this may be done in default compensation order, which means executing the compensations in reverse order relative to the execution order of their respective activities. This order is of course dependent of the control flow primitives provided. For example consider the flow construct with condition links. This construct can be easily encoded in a graph-based language such as Petri-nets, but is difficult to encode in algebraic languages like the one we use. Sagas, the calculus we will compare BPEL to in this paper only offers control flow primitives for sequential and parallel execution. Note that parallel execution as understood in Sagas, corresponds to the flow construct without using condition links. Therefore, we also restrict the set of control flow primitives to those supported in Sagas. However, if a control flow primitive as well as the resulting default compensations can be encoded with process algebraic operators, we are confident that our calculus can be extended to comprise this control flow primitive.

3.1 Syntax

$BPEL_{fct}$ is a formalization of BPEL focused only on fault, compensation and termination (FCT) handling. We assume an infinite set of basic activities Act. The language of $BPEL_{fct}$ is defined as follows

Syntax of $BPEL_{fct}$

$$S ::= 0 \mid A \mid \tau_A \mid \tau \mid exit \mid S\,;\,S \mid S+S \mid S \parallel S \mid !_A \mid\, ! \mid \uparrow \mid \{S : S : S : S\}$$
$$P ::= \{\!|S : S|\!\} \qquad PT ::= \checkmark \mid \varnothing \mid \triangledown$$
$$C ::= S \mid P \qquad \alpha, \beta ::= (C, \alpha, \beta)^m \mid \alpha\,;\,\beta \mid \alpha \parallel \beta$$
where $A \in \mathsf{Act}$, $m \in \{n, f, n', f', c, t\}$

Initial Terms ($BPEL^0_{fct}$):

$$S^0 ::= 0 \mid A \mid exit \mid S^0\,;\,S^0 \mid S^0 + S^0 \mid S^0 \parallel S^0 \mid !_A \mid\, ! \mid \uparrow \mid (\{S^0 : S^0 : S^0 : S^0\}, 0, 0)^n$$
$$P^0 ::= (\{\!|S^0 : S^0|\!\}, 0, 0)^{n'}$$

S denotes a BPEL activity and P a process. PT, α and β are not part of BPEL and only needed for the semantics. The language $BPEL^0_{fct}$ is the subset of $BPEL_{fct}$ that can be directly translated to BPEL expressions and vice versa. It does not include the semantically necessary intermediate representations. All basic BPEL activities are abstracted to atomic activities A which are assumed to be taken from an arbitrary set of names Act. τ denotes an internal action, that is not part of BPEL, but is used in $BPEL_{fct}$ to denote the occurrence of some event that is not visible to an observer. In case this unobservable event originates from a named fault $!_A$, we write τ_A. Observable (named) faults are signalled by the action ! ($!_A$). For a external observer faults are never observable and always appear as τ or τ_A. We only need them for the inference rules. Unnamed faults ! (which appear simply as τ from an external observers perspective) and named silent actions τ_A are only introduced in the language to allow for a nice comparison of $BPEL_{fct}$ with Sagas. In order to model BPEL itself they are not needed. The set of all actions is denoted by

$\overline{\mathsf{Act}} = \mathsf{Act} \cup \{!, \tau, exit\} \cup \{!_A \mid A \in \mathsf{Act}\} \cup \{\tau_A \mid A \in \mathsf{Act}\}$. 0 can be considered the completed activity. Please note that completion/termination at process level is indicated by an element from PT. Activities can be executed in sequence (;) or in parallel ($\|$). The parallel construct corresponds to the *flow* activity of BPEL. Nondeterministic choice ($+$) allows us to abstract from orthogonal aspects such as data. All primitives so far are standard in process algebra. The other activities are introduced to model the FCT part of BPEL. We again abstract from details and use $!_A$ (!) in combination with nondeterministic choice to model faults communicated by partner Web Services as well as run-time errors. $\uparrow$ triggers the default compensation mechanisms. $\{S : S_F : S_C : S_T\}$ denotes a *scope*, consisting of four components: the main activity S, the fault handler S_F, the compensation handler S_C and the termination handler S_T, which are again arbitrary activities. When no compensation handler is specified, the default compensation handler $\uparrow$ is assumed. If no fault handler is specified, the default handler $\uparrow;!$ is assumed, which first compensates all already executed activities enclosed by the scope and then rethrows the fault to the directly enclosing scope or process.

Compensation contexts are sequences and parallel constructs of *compensation closures* $(S, \alpha, \beta)^m$. They denote installed compensations of successfully completed scopes, where S is the compensation handler itself and β is the set of compensations that can be activated by $\uparrow$ inside of S. α is used to collect compensations of scopes that successfully terminated during the execution of S. These constructs allow for all-or-nothing semantics. (cf. Section 3.2). A BPEL process can be considered as a scope without compensation and termination handler. We use $\checkmark$ to denote successful termination of a process, $\varnothing$ to signal a process abortion due to the occurrence of a fault during fault handling. Forced termination of a process (using the *exit* activity) is denoted by $\triangledown$. The language can be extended to named fault and compensation handling as it is part of BPEL. However, the mechanisms behind BPEL-FCT handling are not influenced considerably by this extensions.

3.2 Semantics

Our semantics is given in SOS-style [16] together with a set of congruence rules, that allow for a more compact presentation of our rules. Analogous to [19], the state space of a BPEL-process consists of a term plus additional information about installed compensations. In contrast to [19] we use a pair (α, β) of such contexts. This is necessary in order to allow for repeated compensations [3], called *all-or-nothing semantics* in BPEL. This allows to compensate a failed compensation. In principle, it is also possible to compensate a failed compensation of a failed compensation, etc. Now, when a compensation is triggered by $\uparrow$, we have to make sure that we execute the right compensations. If, say, a compensation handler H of a successfully completed scope T is executed and it first performs a sequence of transaction S and then afterwards decides to start the default compensation $\uparrow$, then it is not supposed to compensate S, but to execute the installed compensations for T. However, if H fails while performing S, the partial execution of H has to be compensated. In this case, the installed compensations for S have to be executed, whereas those for T must remain untouched. We call the first component α of the compensation context *accumulated compensation context*. It contains all compensation handlers that have been installed by the currently executing transaction or handler (H in our example). The second component β is called *fixed compensation context*. It contains those compensation handlers that have been installed before the compensation handler (H) has been called and which are supposed to be activated, if the compensation does not fail. So in our example it contains the installed compensations for T.

In order to achieve all-or-nothing semantics it is necessary to enclose the activation of the compensation by a scope, since in case a fault occurs during the compensation, it is then intercepted by the fault handler of the scope, which might then activate the accumulated compensations of the compensations. Without an enclosing scope, the fault would abruptly abort the compensation handling procedure and thus rendering all-nothing semantics impossible, since this would require to revert the so far executed compensations. The use of only a single compensation context seems to inherently lead to a weaker notion of compensation semantics, where either all-or-nothing semantics are enabled at the cost of sacrificing the ability to call the default compensations as part of a user-specified compensation, as it is the case in [3], or the other way round, as in [19]. Our semantics allow for all-or-nothing semantics even in the case when the compensation is triggered directly inside a fault handler without an enclosing scope (generalized all-or-nothing semantics). We will now discuss the rules of our semantics.

Let in the following $x \in \mathsf{Act} \cup \{\tau, exit\} \cup \{\tau_A \mid A \in \mathsf{Act}\} \cup \{!_A \mid A \in \mathsf{Act}\}$.

BASIC
$$\frac{}{(\alpha,\beta) \vdash x \xrightarrow{x} (\alpha,\beta) \vdash 0}$$

CHOICE
$$\frac{(\alpha,\beta) \vdash S_1 \xrightarrow{x} (\alpha',\beta') \vdash S_1'}{(\alpha,\beta) \vdash S_1 + S_2 \xrightarrow{x} (\alpha',\beta') \vdash S_1'}$$

SEQ
$$\frac{(\alpha,\beta) \vdash S_1 \xrightarrow{x} (\alpha',\beta') \vdash S_1' \qquad S_1' \not\equiv 0}{(\alpha,\beta) \vdash S_1 \,;\, S_2 \xrightarrow{x} (\alpha',\beta') \vdash S_1' \,;\, S_2}$$

SEQT
$$\frac{(\alpha,\beta) \vdash S_1 \xrightarrow{x} (\alpha',\beta') \vdash 0}{(\alpha,\beta) \vdash S_1 \,;\, S_2 \xrightarrow{x} (0 \,;\, \alpha',\beta') \vdash S_2}$$

PARL
$$\frac{(\alpha_1,\beta) \vdash S_1 \xrightarrow{x} (\alpha_1',\beta') \vdash S_1'}{((\alpha_1 \parallel \alpha_2) \,;\, \alpha,\beta) \vdash S_1 \parallel S_2 \xrightarrow{x} ((\alpha_1' \parallel \alpha_2) \,;\, \alpha,\beta') \vdash S_1' \parallel S_2}$$

PARR
$$\frac{(\alpha_2,\beta) \vdash S_2 \xrightarrow{x} (\alpha_2',\beta') \vdash S_2'}{((\alpha_1 \parallel \alpha_2) \,;\, \alpha,\beta) \vdash S_1 \parallel S_2 \xrightarrow{x} ((\alpha_1 \parallel \alpha_2') \,;\, \alpha,\beta') \vdash S_1 \parallel S_2'}$$

The inference rules so far are almost standard, except for SEQT, where we extend the accumulated compensation context by a leading 0. The reason is that in order to be applicable, rules PARL,PARR demand the structure $(\gamma_1 \parallel \gamma_2) \,;\, \gamma_3$ of the accumulated compensation context. This is ensured by the following invariant: For every transition $(\gamma,\delta) \vdash T \xrightarrow{x} (\gamma',\delta') \vdash T'$ it holds that if the accumulated compensation context of the configuration on the left hand side is of the form $\gamma \equiv (\gamma_1 \parallel \gamma_2); \gamma_3$, then this will also be the case for the resulting context γ' on the right hand side (as long as $T' \not\equiv 0$). This invariant also holds for the accumulated compensation context α of closures $(C, \alpha, \beta)^m$. The invariant can be proven by induction on the term structure, i.e. by case analysis on the applicable inference rules. The execution of business processes described by $BPEL_{fct}$ starts in a configuration $(0, 0) \vdash p$ with $p \in P^0$. Using rules CB1 and CB2, it is easy to see that the invariant holds at the beginning of an execution of a $BPEL_{fct}$ process. Using induction on the length of an execution, it can be shown that the invariant also holds during the whole execution (up to the last configuration). Therefore the rules PARR and PARL can always be applied if the term in execution is a parallel composition.

A scope S is always executed inside a closure $(S, \alpha_A, \beta_F)^m$, which stores information about the current compensation context (α_A, β_F). The effect of activities inside a scope may differ depending on the circumstances under which they are executed. The ! primitive, for instance, shows subtle differences in its effect depending on whether it is used inside a compensation handler or termination handler. Our semantics stores the information under which circumstances a term is executed in what we call the *mode* m of a closure. We refer to m as the mode of the scope or process that is directly enclosed by the closure. Possible modes are:

- normal mode n, i.e. the scope is executing its enclosed activity
- faulted mode f, i.e. a fault has happened while executing the enclosed activity
- compensating mode c, i.e. the scope is a compensation handler in execution
- terminating mode t, i.e. the scope is being terminated and is executing its termination handler

The primed variants are used when a process is in the corresponding mode instead of a scope. Let in the following $y \in \mathsf{Act} \cup \{\tau\} \cup \{\tau_A \mid A \in \mathsf{Act}\}$.

As long as no fault occurs, a scope executes its enclosed activity (cf. [1] 12.1, p. 116). Please note the use of β in SCOPE, which enables compensation handlers to have nested enclosed scopes that may trigger a compensation. This is needed in order to obtain correct *all-or-nothing* semantics of compensation handlers.

Exactly those scopes that complete successfully will install their compensation handler (cf. [1] 12.4.3 p. 122, 12.4.4.3 p.125, and 12.5 pp. 127). Those handlers are placed in front of the accumulated compensation context such that if the compensation is triggered later on, they will be executed in default compensation order.

SCOPE

$$\frac{(\alpha_A, \beta) \vdash S \xrightarrow{y} (\alpha'_A, \beta') \vdash S' \qquad S' \not\equiv 0}{(\alpha, \beta) \vdash (\{S : F : C : T\}, \alpha_A, 0)^n \xrightarrow{y} (\alpha, \beta') \vdash (\{S' : F : C : T\}, \alpha'_A, 0)^n}$$

SCOPE END

$$\frac{(\alpha_A, \beta) \vdash S \xrightarrow{y} (\alpha'_A, \beta') \vdash 0}{(\alpha, \beta) \vdash (\{S : F : C : T\}, \alpha_A, 0)^n \xrightarrow{y} ((C, 0, \alpha_A)^c \,;\, \alpha, \beta) \vdash 0}$$

SCOPE FCT

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{y} (\alpha'_A, \beta'_F) \vdash S' \qquad S' \not\equiv 0 \qquad m \in \{c, f, t\}}{(\alpha, \beta) \vdash (S, \alpha_A, \beta_F)^m \xrightarrow{y} (\alpha, \beta) \vdash (S', \alpha'_A, \beta'_F)^m}$$

SCOPE END FCT

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{y} (\alpha'_A, \beta'_F) \vdash 0 \qquad m \in \{c, f, t\}}{(\alpha, \beta) \vdash (S, \alpha_A, \beta_F)^m \xrightarrow{y} (\alpha, \beta) \vdash 0}$$

If a scope throws a fault, it is intercepted by the fault handler. Before the fault handling activities are executed, the scope's remaining enclosed activity is forced to terminate (cf. [1] 12.5, p. 127 and pp. 131-132, and 12.6, p. 135). We discuss forced termination in detail later.

SCOPE FAULT

$$\frac{(\alpha_A, \beta) \vdash S \xrightarrow{!_A} (\alpha'_A, \beta) \vdash S'}{(\alpha, \beta) \vdash (\{S : F : C : T\}, \alpha_A, 0)^n \xrightarrow{\tau_A} (\alpha, \beta) \vdash ([S'] \,;\, F, 0, \alpha'_A)^f}$$

In case that a fault handler faults itself, its activity will be terminated and the fault will be rethrown to the enclosing scope's or process' fault handler (cf. [1] 12.4.4.3, p. 126).

SCOPE FAULT F

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{!_A} (\alpha'_A, \beta_F) \vdash S'}{(\alpha, \beta) \vdash (S, \alpha_A, \beta_F)^f \xrightarrow{\tau_A} (\alpha, \beta) \vdash [S'] \,;\, !_A}$$

A faulting compensation handler will start the compensation of already executed compensation activities (enabling *generalized* all-or-nothing semantics) and then it will rethrow the fault to the initiator of the compensation (cf. [1] 12.4.4.3, p. 126).

SCOPE FAULT C

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{!_A} (\alpha'_A, \beta_F) \vdash S'}{(\alpha, \beta) \vdash (S, \alpha_A, \beta_F)^c \xrightarrow{\tau_A} (\alpha, \beta) \vdash (\uparrow \,;\, !_A, 0, \alpha'_A)^f}$$

A fault during termination handling leads to forced termination of the termination handler (cf. [1] 12.4.4.3, p. 127, and 12.6, p. 137).

SCOPE FAULT T

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{!_A} (\alpha'_A, \beta_F) \vdash S'}{(\alpha, \beta) \vdash (S, \alpha_A, \beta_F)^t \xrightarrow{\tau_A} (\alpha, \beta) \vdash [S']}$$

Compensation is realized in $BPEL_{fct}$ by executing the fixed compensation context β of the current context, i.e. the compensation handlers of the child scopes enclosed by the original scope (cf. [1] 12.4.3.2, pp. 123-124).

COMP

$$\frac{}{(\alpha, \beta) \vdash \uparrow \xrightarrow{\tau} (\alpha, 0) \vdash \beta}$$

The behavior of processes is quite similar to that of scopes. Because processes are top level terms, they are always executed in the empty context $(0,0)$.

PROCESS

$$\frac{(\alpha_A, 0) \vdash S \xrightarrow{y} (\alpha'_A, 0) \vdash S'}{(0,0) \vdash (\{\!|S : F|\!\}, \alpha_A, 0)^{n'} \xrightarrow{y} (0,0) \vdash (\{\!|S' : F|\!\}, \alpha'_A, 0)^{n'}}$$

PROCESS F

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{y} (\alpha'_A, \beta'_F) \vdash S'}{(0,0) \vdash (S, \alpha_A, \beta_F)^{f'} \xrightarrow{y} (0,0) \vdash (S', \alpha'_A, \beta'_F)^{f'}}$$

PROCESS FAULT

$$\frac{(\alpha_A, 0) \vdash S \xrightarrow{!_A} (\alpha'_A, 0) \vdash S'}{(0,0) \vdash (\{\!|S : F|\!\}, \alpha_A, 0)^{n'} \xrightarrow{\tau_A} (0,0) \vdash ([S']\ ;\ F, 0, \alpha'_A)^{f'}}$$

Successful completion in either normal mode (no fault on process level happened so far) or faulted mode (a fault happened on process level and was handled successfully by the process' fault handler) will result in successful termination ($\checkmark$). If the execution of the fault handler failed the process will end up in a failed state $\varnothing$.

PROCESS FAULT F

$$\frac{(\alpha_A, \beta_F) \vdash S \xrightarrow{!_A} (\alpha'_A, \beta_F) \vdash S'}{(0,0) \vdash (S, \alpha_A, \beta_F)^{f'} \xrightarrow{\tau_A} (0,0) \vdash \varnothing}$$

PROCESS END

$$\frac{}{(0,0) \vdash (\{\!|0 : F|\!\}, \alpha_A, \beta_F)^{n'} \xrightarrow{\tau} (0,0) \vdash \checkmark}$$

PROCESS END F

$$\frac{}{(0,0) \vdash (0, \alpha_A, \beta_F)^{f'} \xrightarrow{\tau} (0,0) \vdash \checkmark}$$

The *exit* activity of BPEL forces a process to terminate immediately. The SOS rules for *exit* are straightforward. The *exit* signal is passed through until it reaches a process, where it leads to forced termination $\triangledown$. Due to space limitations we omit the rules and only present the rule for processes:

EXIT PROCESS

$$\frac{(\alpha_A, 0) \vdash S \xrightarrow{exit} (\alpha'_A, 0) \vdash 0}{(0,0) \vdash (\{\!|S : F|\!\}, \alpha_A, 0)^{n} \xrightarrow{exit} (0,0) \vdash \triangledown}$$

In the following we present the syntactical congruence rules. Since our representation of the compensation mechanism relies on a strong structural resemblance of the process term and its compensation contexts in the closure, we may not freely commute parallel terms. For the same reason associativity does not hold. As an example consider $((\alpha_1 \parallel \alpha_2); \alpha, \beta) \vdash S_1 \parallel S_2 \not\equiv ((\alpha_1 \parallel \alpha_2); \alpha, \beta) \vdash S_2 \parallel S_1$. This ensures that S_i stays associated with the accumulated compensation context α_i, where $i \in \{1, 2\}$. In initial $BPEL_{fct}$ terms, however, parallel terms can by arbitrarily associated and commuted without changing the resulting semantics. So semantically, the parallel operator in $BPEL_{fct}$ is –as is the flow construct in BPEL– commutative and associative.

Rule CB1 has to be used to expand the leading 0 in front of the accumulated compensation context in case of the start of a new parallel flow inside the main activity of a scope or process. In detail e.g. $0; \alpha \equiv 0 \parallel 0; \alpha$. CB2 allows reduction of sequences of 0.

$$\begin{array}{llllll} \text{CB1} & 0 \equiv 0 \parallel 0 & \text{CB3} & S + (S' + S'') \equiv (S + S') + S'' & & \\ \text{CB2} & S\,;\,0 \equiv S & \text{CB4} & S + S' \equiv S' + S & \text{CB5} & S + 0 \equiv S \end{array}$$

The last set of rules we consider deals with forced termination of terms mirroring section 12.6 of [1]. When in a parallel branch a fault occurs, then all other parallel branches have to be terminated. Forced termination has to happen as soon as possible, however, in order to allow for controllable terminations of scopes, BPEL introduces the concept of termination handlers, which are activated as soon as a scope is forced to terminate. The BPEL specification allows a fault that is about to occur either to happen or to be terminated without effect (CT9) (cf. Section 4). Note that the handling of a fault or termination are not affected by forced termination (CT12, CT13). This ensures that a transaction (scope), which has faulted before the forced termination occurred, is always able to complete its fault handler and any compensation activated there.

$$\begin{array}{ll} \text{CT1} & [0] \equiv 0 \\ \text{CT2} & [\tau] \equiv 0 \\ \text{CT3} & [A] \equiv 0 \\ \text{CT4} & [\uparrow] \equiv 0 \\ \text{CT5} & [exit] \equiv 0 \\ \text{CT6} & [S + S'] \equiv 0 \\ \text{CT7} & [S\,;\,S'] \equiv [S] \end{array} \qquad \begin{array}{ll} \text{CT8} & [S \parallel S'] \equiv [S] \parallel [S'] \\ \text{CT9} & [!_A] \equiv !_A + \tau\,;\,0 \\ \text{CT10} & [(\{S : F : C : T\}, \alpha_A, 0)^n] \equiv ([S]\,;\,T, 0, \alpha_A)^t \\ \text{CT11} & [(S, \alpha_A, \beta_F)^c] \equiv [S] \\ \text{CT12} & [(S, \alpha_A, \beta_F)^f] \equiv (S, \alpha_A, \beta_F)^f \\ \text{CT13} & [(S, \alpha_A, \beta_F)^t] \equiv (S, \alpha_A, \beta_F)^t \end{array}$$

4 BPEL Is Sagas! Almost

BPEL's FCT-handling mechanisms are –as stated in [1]– inspired by Sagas. However, there are several seemingly different concepts. Even if we prescind from constructs that are not at all represented in Sagas, like links inside a flow construct, some apparent differences remain. Before we will compare the two languages, we will shortly summarize Sagas syntax and semantics.

Sagas. Sagas is equipped with a big-step semantics [3] that distinguishes three different execution results for sagas (i.e. transactions): successful termination, faulted execution with successful compensation, faulted execution with unsuccessful compensation. During an execution a trace of observable basic activities is recorded up to partial order (partial order trace). A partial order trace induces a set of traces. Following [3], we assume that every basic activity that occurs in a Sagas term has a unique name. So the same action can never occur twice in a trace.

Definition 1 ((Partial-Order) Trace). *We call elements of* $\overline{\mathsf{Act}}^*$ traces*. A partial-order trace (*pot*) is a partial order* (V, E) *where* $V \subseteq \overline{\mathsf{Act}}$*. The set of all* pot*s is denoted by* POT*.*

Note that every trace can be considered as a pot that is linear.
Notation: We write X_A to denote the component X of the tuple A, when $A = (\dots, X, \dots)$. For a partial order A we hence always assume $A = (V_A, E_A)$. We write singleton sets $\{x\}$ sometimes simply as x if no ambiguities arise. We define the following two operations on sets:
$A; B = \{(a, b) \mid a \in A, b \in B\}$, $C|_B = \{(a, b) \in C \mid a, b \in B\}$
It is sometimes useful to consider terms that are built from the operators $0, A, \|$ and ; as partial orders as follows:

- $0 = (\emptyset, \emptyset)$
- $A = (\{A\}, \emptyset)$
- $P; Q = (V_P \cup V_Q, E_P \cup E_Q \cup V_P; V_Q)$
- $P \parallel Q = (V_P \cup V_Q, E_P \cup E_Q)$

Definition 2 (Syntax of Sagas). *Sagas S are defined by the following grammar:*

$$\begin{array}{ll} X ::= 0 \mid A \mid A \div B & \textit{(STEP)} \\ P ::= X \mid P; P \mid P \parallel P & \textit{(PROCESS)} \\ S ::= \{\!|\, P \,|\!\} & \textit{(SAGA)} \end{array}$$

We denote the set of all sagas terms by Sagas.

An atomic activity B can be attached to another atomic activity A as its compensation. Please note that different to $BPEL_{fct}$ compensations cannot be composite terms and cannot be attached to composite terms. A saga formalizes the idea of long-running transactions and corresponds to scopes in $BPEL_{fct}$. In Sagas there is no language primitive to signal a fault. The success or fault of an atomic activity is determined at run-time by an environment Γ mapping every activity either to success ($\boxdot$) or fault ($\boxtimes$). The semantics of Sagas is given in terms of subsets of POT. For the complete semantic rules we refer the reader to [3]. In the original semantics faults cannot be observed and are not represented in the partial order traces. To allow for a decent comparison it is however necessary to make faults observable. This needs only a minor change to the original semantics where rule S-CMP is replaced by rule S-CMP'.

S-CMP

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

S-CMP'

$$\frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, 0 \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\tau_A; \alpha} \langle \boxtimes, 0 \rangle}$$

4.1 Comparison

- In Sagas the main transaction and the sub-transactions are represented by one construct: a saga. In BPEL, the main transaction and subtransactions are represented by two different constructs: the process and scopes. However, for the language subsets that we compare, a process behaves identical to scopes (except that it cannot be terminated, since it always is top-level).
- BPEL's scope construct represents subtransactions and is at the same time used to associate compensations to forward activities. In Sagas, subtransactions are realized via nested *sagas*. In contrast to scopes, they do not have any explicit handlers. Furthermore, every forward activity is associated with its compensations immediately during its execution. In BPEL a compensation for an activity becomes available only after the surrounding scope has successfully finished. Despite the principles of transactions in Sagas seem rather different from those used by BPEL, we will see that it is rather straightforward to express them in BPEL.
- BPEL's and Sagas's compensation policy for concurrent processes forces all branches to compensating themselves in case of a fault in one of the branches. In Sagas, the compensation phases of all branches run independently of each other. This behavior is called *distributed interruption* in [2]. As we will see, BPEL follows the *coordinated interruption* policy (cf. [2]): as soon as a fault has occurred, all parallel branches have to be compensated immediately. BPEL has an additional termination phase, that is intermediary of forward flow phase and compensation phase. Hence every faulted BPEL (sub)transaction, can be divided into three phases that take place strictly in sequence: forward flow phase(F), termination phase(T) and compensation phase(C). Representing the fault with f, we can intuitively represent the execution of the faulted scope/process by $F; f; T; C$. As we will see this policy is an inherent part of BPEL FCT-handling mechanisms.
- Furthermore, the treatment of faults during compensation in both languages is different in principle and we cannot mimic Sagas behavior in BPEL. We will treat this in Section 4.2 in greater detail.

In order to formally compare the two languages we map every saga to a BPEL process and analyze their behaviors in terms of partial orders over actions. Since we want our mapping to be recursively defined over the term structure, we cannot map Sagas terms directly to $BPEL_{fct}$ processes, since processes are always top-level constructs. We therefore use a translation function that maps arbitrary Sagas term S to a $BPEL_{fct}$ term S' that is no process itself and in a final step S' is raised to a top-level process term by enclosing it with the process as follows: $(\{\!|S' : \uparrow|\!\}, 0, 0)^{n'}$. Since $BPEL$ to its full extent is very powerful, it is likely that in principle there exists some unobvious encoding of Sagas behavior even in $BPEL_{fct}$. However, by the choice of our translation mapping, we had in mind to investigated how the predefined structures for compensation handling in BPEL work compared to those of Sagas, so our mapping defines a translation as straightforward as possible. Please remember that we assume that compensations can never fault for reasons mentioned above. Furthermore, we do not allow two faults to occur at the same scope/process level. This restriction does not influence the principle results, but saves us some additional case distinction.

Definition 3 (Translation Function)

$$\begin{aligned}
&[\![]\!]_{\Gamma} : Sagas \rightarrow BPEL^0_{fct}\\
&[\![A]\!]_{\Gamma} = A \text{ if } \Gamma A = \boxdot, \quad [\![A]\!]_{\Gamma} = \,! \text{ if } \Gamma A = \boxtimes,\\
&[\![0]\!] = 0, \quad [\![P \parallel Q]\!] = [\![P]\!] \parallel [\![Q]\!], \quad [\![P \,;\, Q]\!] = [\![P]\!] \;;\; [\![Q]\!],\\
&[\![A \div B]\!]_{\Gamma} = (\{[\![A]\!]_{\Gamma} : !_A : B : 0\}, 0, 0)^n, \quad [\![\{\!| P |\!\}]\!]_{\Gamma} = (\{[\![P]\!]_{\Gamma} : \uparrow : \uparrow : \uparrow\}, 0, 0)^n
\end{aligned}$$

We will sometimes write $[\![.]\!]$ instead of $[\![.]\!]_{\Gamma}$ if Γ is clear from the context.

The operators for sequential and parallel execution behave identical in $BPEL_{fct}$ and Sagas. Remember that parallel execution is a special case of the BPEL flow construct, which is however more powerful than the parallel operator and cannot be modelled in Sagas. If an action A fails in the environment Γ, we replace it by a nameless fault in $BPEL_{fct}$. An action/compensation pair $A \div B$ corresponds to a scope that executes A and has B as it compensation handler. When A fails, then the fault is simply rethrown by the fault handler $!_A$, which triggers already installed compensations at the level of the enclosing scope/process. The fact that faulting activities are themselves translated to a nameless fault !, while they are rethrown as a named fault in the respective fault handlers may be counterintuitive at first sight. Indeed, this is only done to allow for a nice comparison of the observable behavior and has no fundamental consequences or reasons. Nested sagas are translated into scope in a straightforward manner, where the default handlers are used where possible.

In order to compare the two calculi it is very helpful to note that the compensation handlers of both calculi can be translated bijectively into each other. Since $[\![]\!]_{cl}$ is bijective on the contexts of Sagas and the restricted variation of $BPEL_{fct}$, we will use α and $[\![\alpha]\!]_{cl}$ interchangeably when the meaning is clear from the context.

Definition 4 (Context Translation Function). *We translate Sagas contexts to $BPEL_{fct}$ contexts by the following function:*

$$\begin{aligned}
&[\![0]\!]_{cl} = 0, \quad [\![P ; Q]\!]_{cl} = [\![P]\!]_{cl} \,;\, [\![Q]\!]_{cl}, \quad [\![P \parallel Q]\!]_{cl} = [\![P]\!]_{cl} \parallel [\![Q]\!]_{cl},\\
&[\![B]\!]_{cl} = (B, 0, 0)^c
\end{aligned}$$

In Sagas, the basic building blocks of a context are primitive compensation activities. In $BPEL_{fct}$, however, the basic building blocks are closures $(A, \alpha_C, \beta_F)^c$ signalling that the enclosed compensation activity is indeed a compensation (c). In Sagas and therefore in restricted BPEL the default compensation ($\uparrow$) cannot be triggered explicitly, but only implicitly when a fault occurs. Since we assume no faults to occur during compensation, it is safe to arbitrarily let both contexts of the closure be 0.

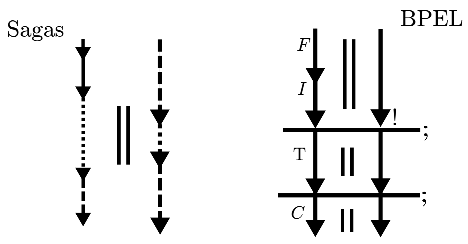

Fig. 1. Corresponding partial orders in Sagas and BPEL

***BPEL**$_{fct}$* **Big-Step Semantics** Let us in the following fix Γ and then let $BPEL'_{fct} = \{[\![S]\!]_\Gamma \mid S \in Sagas\}$. We present a big-step semantics for the subset $BPEL'_{fct}$ of our calculus that corresponds to Sagas following the translation function $[\![]\!]_\Gamma$. Our semantics will be for each term a set of structured partial-order traces as described below.

Definition 5 (Structured POT). *A* structured pot *(*$s$pot*) is a tuple* (V, E, P, D, t) *where* (V, E) *is a* pot *and* $P = (F, I, T, C, f) \in \mathcal{T}$ *and* $D \subseteq \mathcal{T}$*, where* $\mathcal{T} = (2^{Act})^4 \times \{\tau_A \mid A \in \mathsf{Act}\}$ *and* $t \in \{\boxdot, \boxtimes, \boxast\}$*. We call the set of all* $s$pot*s* sPOT*.*

To obtain a nifty comparison of the two languages, the semantics of each term will be represented by a of $s$pots, which can be translated into partial-order traces. We can then compare the two languages by comparing partial-orders traces with identical underlying sets of activities. The partial order relation consists of two parts: One that exactly corresponds to the partial order for the corresponding term in Sagas (E) and one that represents the additional edges of the partial orders, that are induced by the more restricted termination/compensation policy of the top-level process (P) and of each (sub)scope (D). Furthermore V contains all observable activities of the represented execution, success or failure or forced termination of the execution of the (sub)transaction is expressed by t (where $\boxdot$ = successful termination, $\boxtimes$ = execution with compensation due to an internal fault, $\overline{\boxtimes}$ = execution with compensation due to external termination). Please remember that we will not consider faulted compensations. In case a fault occurred, the fault name is represented by f. P with its parts F, T and C represent the partition of the activities of the top-level transaction in forward, termination and compensation flow. The set I represents activities that can either occur before or after the fault f, i.e. during the termination phase, but before the compensation phase. Such activities arise when a subscope compensates itself due to a internal fault. Then this scope is not responsive to termination. D represents the activity partitions for non-successfully terminated subscopes. In Figure 1 the po traces of the same term are presented schematically in both calculi.

How the single parts of a $s$pot exactly contribute to the pot that it represents is represented formally by the following transformation.

Definition 6. $\delta((V, E, (F, I, T, C, f), D, t)) = (V, E')$ *where*

$$E' = E \cup H \cup \bigcup_{(F,I,T,C,f)\in D} F; f; T; C \cup I; C$$

$$\textit{with } H = \begin{cases} \emptyset \ \textit{if } t = \boxdot \\ F; f; T; C \cup I; C \ \textit{otherwise} \end{cases}$$

The $s$pots for every term in $BPEL'_{fct}$ are constructed relative to an initial context compensation context $\beta = (V_\beta, E_\beta)$, which is itself a partial order trace. This intentionally strongly resembles the way the Sagas semantics is defined. We will now see how the big-steps semantics $\mathcal{S}(S, \beta)$ of a term $S \in BPEL'_{fct}$ and a context β is constructed recursively. Please note that for every $S \in BPELfct'$, the set $\mathcal{S}(S, \beta)$ contains the $s$pot where $C = V_\beta$, $E = E_\beta$ and all other sets are empty and $t = \overline{\boxtimes}$ and $f = 0$. We will not repeat this $s$pot later. Intuitively, this $s$pot describes the termination of S before starting its execution. It is the only element of $\mathcal{S}(0, \beta)$.

The following $s$pots are in $\mathcal{S}(S, \beta)$:

Case $S = [\![A \div B]\!] = (\{[\![A]\!]_\Gamma : !_A : B : 0\}, 0, 0)^n$**:**
- $\underline{\text{If } \Gamma A = \boxdot}$: $(\{A\}, (E_\beta \cup A; B; V_\beta), (\{A\}, \emptyset, \emptyset, \emptyset, \{B\} \cup V_\beta, 0), \emptyset, \boxdot)$
- $\underline{\text{If } \Gamma A = \boxtimes}$: $(\{\tau_A\} \cup V_\beta, (E_\beta \cup \tau_A; V_\beta), (\emptyset, \emptyset, \emptyset, V_\beta, \tau_A), \emptyset, \boxtimes)$
- $\underline{\text{If } \Gamma A = \overline{\boxtimes}}$: $(\{A, B\} \cup V_\beta, E_\beta \cup A; B; V_\beta, (\{A\}, \emptyset, \emptyset, \{B\} \cup V_\beta, 0), \emptyset, \overline{\boxtimes})$

Case $S = [\![P; Q]\!]_\Gamma = [\![P]\!]_\Gamma ; [\![Q]\!]_\Gamma$**:**
For all $p \in \mathcal{S}([\![P]\!]_\Gamma, \beta), q \in \mathcal{S}([\![Q]\!]_\Gamma, (C_p, E_P|_{C_P}))$:
- $\underline{\text{If } t_P = \boxtimes \vee t_P = \overline{\boxtimes}}$: p
- $\underline{\text{If } t_P = \boxdot}$: $(V_p \cup V_q, E_p \cup E_q \cup V_p; V_q, (F_p \cup F_q, I_q, T_q, C_q, f_q), D_p \cup D_q, t_q)$

Case $S = [\![P \parallel Q]\!]_\Gamma = [\![P]\!]_\Gamma \parallel [\![Q]\!]_\Gamma$**:** For all $p \in \mathcal{S}([\![P]\!]_\Gamma, 0), q \in \mathcal{S}([\![Q]\!]_\Gamma, 0)$:
- $\underline{\text{If } t_P = t_q = \boxdot}$: $(V_p \cup V_q, E_p \cup E_q \cup E_\beta \cup V_p; V_\beta \cup V_q; V_\beta, (F_p \cup F_q, I_p \cup I_q, T_p \cup T_q, C_p \cup C_q \cup V_\beta, f_p \star' f_q), D_p \cup D_q, t_p \star t_q)$
- $\underline{\text{Otherwise:}}$ $(V_p \cup V_q \cup V_\beta, E_p \cup E_q \cup E_\beta \cup V_p; V_\beta \cup V_q; V_\beta, (F_p \cup F_q, I_p \cup I_q, T_p \cup T_q, C_p \cup C_q \cup V_\beta, f_p \star' f_q), D_p \cup D_q, t_p \star t_q)$

Case $S = [\![\{\!| P |\!\}]\!]_\Gamma = (\{[\![P]\!]_\Gamma : \uparrow : \uparrow : \uparrow\}, 0, 0)^n$**:** For all $p \in \mathcal{S}([\![P]\!]_\Gamma, 0)$:
- $\underline{\text{If } t_p = \boxdot}$: $(V_p, E_p \cup E_\beta \cup V_p; V_\beta, (F_p, I_p, T_p, C_p \cup V_\beta, f_p), D_p, \boxdot)$ and $(V_p \cup C_p \cup E_\beta, E_p \cup E_\beta \cup V_p; V_\beta, (F_p, I_p, T_p, C_p \cup V_\beta, f_p), D_p, \overline{\boxtimes})$
- $\underline{\text{If } t_p = \boxtimes}$: $(V_p, E_p, (F_p \cup f_p, I_p \cup T_p \cup C_p, \emptyset, \emptyset, 0), D_p \cup \{F_p, I_p, T_p, C_p, f_p\}, \boxdot)$ and $(V_p \cup V_\beta, E_p \cup E_\beta \cup V_p; V_\beta, (F_p \cup f_p, I_p \cup T_p \cup C_p, \emptyset, V_\beta, 0), D_p \cup \{F_p, I_p, T_p, C_p, f_p\}, \overline{\boxtimes})$
- $\underline{\text{If } t_p = \overline{\boxtimes}}$: $(V_p \cup E_\beta, E_p \cup E_\beta \cup V_p; V_\beta, (F_p, I_p, T_p, C_p \cup V_\beta, f_p), D_p, \overline{\boxtimes})$

The operations $\star$ and $\star'$ are defined as follows:

$\star$	$\boxdot$	$\boxtimes$	$\overline{\boxtimes}$
$\boxdot$	$\boxdot$	$-$	$-$
$\boxtimes$	$-$	$-$	$\boxtimes$
$\overline{\boxtimes}$	$-$	$\boxtimes$	$\overline{\boxtimes}$

$\star'$	τ_A	0
τ_B	$-$	τ_B
0	τ_A	0

The following theorem relates the big-step semantics to the originally defined small-step semantics. Before, we need some additional definitions.

Definition 7 (Execution of a Trace). *When $a_0 a_1 \dots a_{n-1} = \alpha \in \overline{\mathsf{Act}}^*$ is a trace, we let $(\alpha, \beta) \vdash P \xrightarrow{\alpha} (\alpha', \beta') \vdash P'$ mean that there are sequences of P_i, α_i and β_i with $0 \leq i \leq n$, such that $(\alpha_i, \beta_i) \vdash P_i \xrightarrow{a_i} (\alpha_{i+1}, \beta_{i+1}) \vdash P_{i+1}$ for $0 \leq i < n$ and $P_0 = P$, $\alpha_0 = \alpha$, $\beta_0 = \beta$ and $P_n = P'$, $\alpha_n = \alpha'$, $\beta_n = \beta'$.*

Definition 8 (Induced Traces). *If $(V_A, E_A) \in \mathsf{POT}$ then $\mathsf{Ind}(\alpha) \subseteq \overline{\mathsf{Act}}^*$ denotes the set of induced traces, i.e. the set of linear orders (V_A, E) with $(a, b) \in E \implies (b, a) \notin E_A$.*

In the following we write $\overline{\alpha}$ to mean the result of removing all occurrences of (unnamed) τ in the trace $\alpha \in \overline{\mathsf{Act}}^*$.

Theorem 1 (Semantic Equivalence). *Let $S \in BPEL'_{fct}$, then*

1. *for every maximal execution trace α of a process with $(0,0) \vdash (\{\!|S : \uparrow|\!\}, 0, 0)^{n'} \xrightarrow{\alpha} (0,0) \vdash \checkmark$ there is a spot $p \in \mathcal{S}(S, 0)$ such that $\overline{\alpha} \in \mathsf{Ind}(\delta(p))$ and*
2. *for every $p \in \mathcal{S}(S, 0)$ and every trace $\alpha' \in \mathsf{Ind}(\delta(p))$ there is a trace α such that $\overline{\alpha} = \alpha'$ and $(0,0) \vdash (\{\!|S : \uparrow|\!\}, 0, 0)^{n'} \xrightarrow{\alpha} (0,0) \vdash \checkmark$*

The proof of this theorem is considerably large. In the following we only sketch some crucial observations for the proof. In the following, all terms are from $BPEL'_{fct}$ if not stated differently. We will now explain how all possible behaviors of a process (or a scope) are completely determined by the possible behaviors of its inner activity. The following statements apply to processes, but can be analogously transfered to scopes.

- By rule PROCESS we see that as long as α does not contain $!_A$ for some A and $(\beta, 0) \vdash S \xrightarrow{\alpha} (\beta', 0) \vdash S'$ then also $(0,0) \vdash (\{\!|S : \uparrow|\!\}, \beta, 0)^{n'} \xrightarrow{\alpha} (0,0) \vdash (\{\!|S' : \uparrow|\!\}, \beta', 0)^{n'}$. Furthermore PROCESS and PROCESS END are the only applicable rules. Exactly when $S' \equiv 0$ then together with PROCESS END as final step we can derive only the execution $(0,0) \vdash (\{\!|S : \uparrow|\!\}, \beta, 0)^{n'} \xrightarrow{\alpha} (0,0) \vdash \checkmark$.
- As soon as S throws a fault $!_A$, i.e. $(\beta', 0) \vdash S' \xrightarrow{!_A} (\beta', 0) \vdash S'_1$ after a faultless trace α, the process is bound to behave in the following way: $(0,0) \vdash (\{\!|\, [\![S]\!] : \uparrow|\!\}, 0, 0)^n \xrightarrow{\alpha; !_A; \alpha'; \tau; \tau; \alpha''} (0,0) \vdash \checkmark$, where α' is an execution of $[S'_1]$ and α'' is an execution of the compensation β'.

One can see this as follows: As above we infer that $(0,0) \vdash (\{\!|\, [\![S]\!] : \uparrow |\!\}, \beta, 0)^n \xrightarrow{\alpha} (0,0) \vdash (\{\!|S' : \uparrow |\!\}, \beta', 0)^n$. By assumption furthermore $(\beta',0) \vdash S' \xrightarrow{!_A} (\beta',0) \vdash S'_1$. But then by rule PROCESS FAULT we get that $(0,0) \vdash (\{\!|S' : \uparrow |\!\}, \beta', 0)^n \xrightarrow{\tau_A} (0,0) \vdash ([S'_1]; \uparrow, 0, \beta')^f$. By our assumption about α' and repeated application of PROCESS F we get $(0,0) \vdash ([P'_1]; \uparrow, 0, \beta')^f \xrightarrow{\alpha'} (0,0) \vdash (0; \uparrow, \gamma, \beta')^f$ for some γ. By application of rule PROCESS F and rule SEQT and then by application of PROCESS F and rule COMP we obtain that $(0,0) \vdash (0; \uparrow, 0, \beta')^f \xrightarrow{\tau\tau} (0,0) \vdash (\beta', \gamma, 0)^f$. By assumption and repeated use of PROCESS F we can infer that $(0,0) \vdash (\beta', \gamma, 0)^f \xrightarrow{\alpha''} (0,0) \vdash (0, \gamma, 0)^f$. Finally using PROCESS END we obtain the desired result. In order to establish this we need additional lemmas stating that accumulated compensations are never changed during termination and compensation phases.

With this in mind we can represent processes/scope behavior by only looking at the forward activities of its inner activity and the behavior of the inner activity in case of termination at arbitrary intermediate states plus the behavior of its so far accumulated compensations. The following lemma is hence the missing link between the two semantics. The lemma uses this definition:

Definition 9 (Cut Set of a Partial Order). *A cut set X of a partial order (V,E) is a subset of V such that whenever $a \in X$ and $(b,a) \in E$ then also $b \in X$.*

Lemma 1. *Let $(\beta,0) \vdash S \xrightarrow{\alpha} (\beta',0) \vdash S'$. Let α contain no $!_A$. We now find exactly one $p \in \mathcal{S}(S,\beta)$ for each of the following cases*

1. *if $S' \equiv 0$ then $f_p = 0$ and $t_p = \boxdot$*
2. *if $(\beta,0) \vdash S' \xrightarrow{\tau_A} (\beta,0) \vdash S''$ then $f_p = \tau_A$ and $t_p = \boxtimes$ and for ρ' such that $(\beta',0) \vdash [S''] \xrightarrow{\rho'} (\beta',0) \vdash 0$ we have $\overline{\rho'} \in \mathsf{Ind}(\delta(p)|_{(I_p - I'_p) \cup T_p})$ where I'_p is a cut set of $(I_p, E_p|_{I_p})$*
3. *$f_p = 0$, $t_p = \overline{\boxtimes}$ and for ρ' such that $(\beta',0) \vdash [S'] \xrightarrow{\rho'} (\beta',0) \vdash 0$ we have $\overline{\rho'} \in \mathsf{Ind}(\delta(p)|_{(I_p - I'_p) \cup T_p})$ where I'_p is a cut set of $(I_p, E_p|_{I_p})$.*

and in addition in each case $\overline{\alpha} \in \mathsf{Ind}(\delta(p)|_{F_p \cup I'_p})$ and for ρ such that
$(0,0) \vdash \beta' \xrightarrow{\rho} (0,0) \vdash 0$ we have $\overline{\rho} \in \mathsf{Ind}(\delta(p)|_{C_p})$

Proof (Sketch). This lemma can be proven by induction over the term structure of the inner activity S of the process. The proof idea is to consider all states S' reachable from S by a faultless trace toghether with the compensation context accumulated so far and a distinction whether S' can fault in the next step. All statements about reachable states, traces, contexts and possible faults can always be determined recursively by corresponding statements for the direct subterms of S. Taking parallel composition as an example, i.e. $S = P \parallel Q$ for some P and Q, by induction we can find for each subterm P and Q, exactly one $s$pot in $\mathcal{S}(P,0)$ and $\mathcal{S}(Q,0)$ that fits according to the lemma to all statements made. By the way $\mathcal{S}(P, \beta)$ is constructed out of the elements of $\mathcal{S}(P, 0)$ and $\mathcal{S}(Q, 0)$ we can conclude that there is again exactly one $s\mathsf{pot}$ in $\mathcal{S}(P, 0)$ that satisfies the lemma. Note that in the case of parallel composition we need to use the assumption that at most one fault can occur at the same level.

After we have related the two different semantics for $BPEL'_{fct}$, we are now ready to compare $Sagas$ and $BPEL_{fct}$ in a formal way:

Theorem 2 (Correspondence Theorem)
For all sagas S and environments Γ: Whenever $\Gamma \vdash \langle\{\!|\, S \,|\!\}, \beta\rangle \xrightarrow{\alpha} \langle t', \beta'\rangle$ *there is* $(V, E, P, D, t, f) \in \mathcal{S}(\llbracket S \rrbracket_{\Gamma}, \llbracket \beta \rrbracket_{cl})$ *(and vice versa) with* $t = t'$ *and* $\alpha = (V, E|_V)$.

Intuitively, this means that the two languages behave identical (on the subsets considered) up to the more constrained behavior inside faulted and aborted transactions (scopes and the process itself) in BPEL that are imposed due to the more restrictive compensation policy. It is worth noting that if no (sub)transaction faults, the behaviors are completely identical.

Proof The proof is by structural induction over S along the recursive definition of $\mathcal{S}(.,.)$.

4.2 Relating Other Features to BPEL

Other more advanced features conceptually exist in different form in both languages. However, we found them hardly comparable in a reasonable manner. The most important difference is the way faults during compensation are handled. Consider the example of a transaction in BPEL where the inner activity is a parallel construct $P \parallel Q$. Assume that P faults. Then the transaction –which is either a scope or a process– will execute the fault handler and hence switch to faulted mode (f). The default fault handler will then trigger the compensation of P and Q. If now again a fault is caused by the compensation of, say, P, this will immediately cancel the whole compensation by rule SCOPE FAULT F, including the compensations for Q! In Sagas the same situation would be handled differently. By rule F-PAR we see that even if the compensation of P fails, Q's compensations will be completely executed and not aborted prematurely as in BPEL.

Another interesting aspect is that the principles of all-or-nothing semantics and programmable compensations are supported within Sagas via a generalization of the syntax which enables composite compensations and by adding the inference rule REPEATED-COMP. In BPEL the occurrence of a successful compensation of a failed compensation is communicated to a possible enclosing scope or to the process (cf. SCOPE FAULT C), whereas in Sagas such an occurrence remains hidden to an enclosing saga. Although the two languages do not differ fundamentally in this point, both realize different perceptions of all-or-nothing semantics.

In our comparison we used the fault handling mechanism of BPEL only to trigger compensations. Since fault handlers are fully-programmable, this mechanism can be used like the standard exception handling mechanism found in most modern programming languages. BPEL does not come with an explicit exception handling construct aside of scopes. Exception handling can be added to Sagas in form of a sagas/exception-handler pair **try** S **with** P [3], such that when an exception occurs in S, the execution continues with P. The compensations that are accumulated during the execution of both S and P are stored. In BPEL however, a fault handler is not allowed to install compensations at the same level as the activity it has been activated by, hence the two approaches to general exception handling are incompatible.

Other interesting differences and similarities may be found by a comparison to other important formal approaches to model FCT-handling like StAC [4] and cCSP [5]. StAC provides powerful compensation mechanisms similar to those of Sagas, so we also expect $BPEL_{fct}$ and StAC to share a non-trivial semantically equivalent subsets. However, we did not undertake formal investigations in this question and leave it to further research. In [2] it is shown that sequential Sagas without nesting and parallelism and cCSP coincide. The authors also show that cCSP can be changed such that both calculi behave equivalent in the presence of a parallel construct. Different to Sagas and $BPEL_{fct}$, the collected compensations of a successfully terminated subtransaction are dismissed and not stored for a possible later compensation of an enclosing transaction. This aspect and the result from [2] led us to only consider Sagas for a thorough formal comparison.

5 Conclusion

We have provided a fine-grained small-step semantics for BPEL. To the best of our knowledge this is the first process algebraic BPEL semantics that covers automated compensation handling including *all-or-nothing* semantics and compensation execution in default compensation order in its entirety. This makes it a natural choice for a comparison with recent process algebraic approaches to FCT-handling like Sagas. In this paper, we showed that Sagas coincides with a useful subset of BPEL apart from different compensation policies in the presence of parallelism. Sagas uses the concept of *distributed interruption*, which allows parallel branches to compensate their respective activities independently of each other when a fault has occurred in one branch. BPEL uses the *coordinated interruption* policy, where a fault forces all branches to start their compensations as soon as possible and at the same time. In BPEL an additional termination phase precedes the compensation phase where subtransactions are terminated in a safe manner. This phase is not distinguished as such in Sagas. Most notably, faulted compensations lead to evidently different behavior in the two compared calculi.

Our paper shows that FCT-handling in BPEL is rooted in firm formal grounds, and can be used in safe ways. However, the comparable common subsets of BPEL and Sagas seems to be rather small, although similar constructs are provided in both worlds. So in order to make FCT-handling still safer to use, more foundational analysis and comparative work has to be carried out.

References

1. Web services business process execution language version 2.0 - OASIS standard (April 2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`
2. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
3. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. SIGPLAN Not. 40(1), 209–220 (2005)
4. Butler, M., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949. Springer, Heidelberg (2004)
5. Butler, M., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
6. Koshkina, M., van Breugel, F.: Models and verification of BPEL. Technical Report M3J 1P3, York University, Toronto, Canada (2006)
7. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. 16(3), 249–259 (1987)
8. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River (1985)
9. Khalaf, R.: Supporting Business Process Fragmentation While Maintaining Operational Semantics - A BPEL Perspective. Ph.D thesis, Universität Stuttgart (2008)
10. Koshkina, M.: Verification of business processes for web services. Master's thesis, York University, Toronto (2003)
11. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
12. Laneve, C., Zavattaro, G.: Web-Pi at work. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 182–194. Springer, Heidelberg (2005)
13. Lohmann, N.: A feature-complete petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
14. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
15. Mazzara, M., Lucchi, R.: A framework for generic error handling in business processes. Electr. Notes Theor. Comput. Sci. 105, 133–145 (2004)
16. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
17. Pu, G., Zhao, X., Wang, S., Qiu, Z.: Towards the semantics and verification of BPEL4WS. Electr. Notes Theor. Comput. Sci. 151(2), 33–52 (2006)
18. Pu, G., Zhu, H., Qiu, Z., Wang, S., Zhao, X., He, J.: Theoretical Foundations of Scope-Based Compensable Flow Language for Web Service. Springer, Heidelberg (2006)
19. Qiu, Z., Wang, S., Pu, G., Zhao, X.: Semantics of BPEL4WS-Like Fault and Compensation Handling. Springer, Heidelberg (2005)