Bachelor’s Thesis
Scope-based FCT-Handling in WS-BPEL 2.0

submitted by
David Spieler
on 2008-07-14

Supervisor
Prof. Dr.-Ing. Holger Hermanns

Advisor
Christian Eisentraut, M.Sc.

Reviewers
Prof. Dr.-Ing. Holger Hermanns
Prof. Bernd Finkbeiner, Ph.D.
Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 2008-07-14

Declaration of Consent

Hereby I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, 2008-07-14
Acknowledgements

I would like to thank Prof. Holger Hermanns for proposing such an interesting and challenging topic for my Bachelor’s thesis as well as for the insightful discussions.

Likewise, I would like to express my gratitude to my advisor Christian Eisentraut for his never-ending patience and for his helpful advices throughout the whole conception of this thesis.

Special thanks go to Prof. Bernd Finkbeiner who spontaneously accepted to review this work.
1 Introduction

Web Services provide a concept to standardize cooperation of multiple distributed applications mainly over the Internet. This goal is achieved by using a set of standard technologies at different layers of collaboration. At the bottom communication layer the Simple Object Access Protocol (SOAP)\(^1\) is used for information exchange. The interfaces and message formats used thereby are specified using the Web Service Description Language (WSDL)\(^2\). The process of publishing and finding those services via their WSDL description is done using the indexing service UDDI (Universal Description, Discovery and Integration)\(^{[22]}\), which provides a SOAP interface for accessing stored data. All those technologies are based on XML, which ensures architectural independence and a structured and self-descriptive way of data presentation.

**Example 1.** Figure 1 illustrates two Web Services, a flight agency and a hotel, which could e.g. be used to book a flight or respectively reserve a room. Both Web Services are described using WSDL, a technology based on the XML standard. Assuming the services are registered to an UDDI indexing service, they can be found by a user looking for services matching the respective WSDL interfaces and message formats. The user afterwards can communicate with those Web Services via the SOAP protocol, also based on standard technologies, mainly XML.

Web Services can be programmed using standard programming languages like Java or C++. But another, sometimes more convenient, way to describe new Web Services is to specify the temporal and causal invocation of existing Web Services, already providing parts of the functionality that shall be offered. This concept is called **Web Service Orchestration**, i.e. the controlled execution of a set of Web Services by a central authority. The predefined behavior of those controlled Web Services is then called **Web Service Choreography**.

\(^1\)http://www.w3.org/TR/soap/
\(^2\)http://www.w3.org/TR/wsd1
Example 2. Figure 2 illustrates the concept of Web Service Orchestration using the example of a travel agency. Like in the preceding example, the two Web Services – FlightAgency and Hotel – can be used to book a flight or respectively a room. But instead of a user again directly accessing and manually coordinating those two services, a travel agency Web Service could be created, offering that coordination as a new service. The travel agency is then said to orchestrate the original Web Services. The travel agency Web Service may be registered to an UDDI indexing service for potential customers being able to find the new service on looking for its functionality, i.e. booking a travel – including a flight and a hotel.

![Figure 2: Web Service Orchestration Example: Travel Agency](image)

For the purpose of defining Web Service Orchestrations several languages have been proposed. The most popular language of that category at the moment is WS-BPEL 2.0 (Web Services Business Process Execution Language 2.0)[18]. WS-BPEL 2.0 provides primitives to specify the flow of execution and communication between a described process and its partner links, i.e. communication partners. In the following we will mainly use the term BPEL instead of WS-BPEL 2.0.

In general the duration of transactions inside those orchestrations may become quite long (Long-Running-Transactions, LRT), e.g. because of the dependency on batch processes, human interaction, or even explicit timing constraints, like waiting for a certain date. Thus it is not feasible to use ACID (Atomicity, Consistency, Isolation, Durability) transaction schemes common to database systems, i.e. requiring to block resources in case of possible parallel access for such a long time. Instead, in BPEL a concept called compensation is used, based on the seminal idea of Sagas [9]. Compensation denotes the controlled reversion of a partial execution by a sequence of process-specific, predefined actions. For that purpose BPEL provides a special scope construct, which allows encapsulation of activities in respect to compensation. Inside such a scope, any activity may occur, like the invocation of partner Web Services, assigning values to variables as well as nested scopes. Scopes may be accompanied by a fully programmable compensation handler, which can be called in case of an error.
to undo the execution of the that the scope’s inner activity. A scope in BPEL
may also be equipped with individual fault, termination, and event handlers.
Thus, an important aspect when designing Web Services in BPEL is to ensure
the correct interplay of those handlers in respect to dependable behavior, espe-
cially in cases of internal or external faults. Consequently, formal investigations
of that scope based fault, compensation, and termination (FCT) handling in
BPEL seems to be necessary.

1.1 Related Work

Many approaches of modeling certain aspects of BPEL have been made. An
overview over those approaches is given in [8]. A general theoretical framework
for compensable flows is introduced by Bruni et al. in [3] in terms of the Sagas
Calculus. In [2] Hoare et al. present different possible policies of compens-
able flow composition concerning compensation. That is whether to interrupt
current flows in case of a faulted branch or not and to use distributed or
coordinated compensation. They also give encodings of all four possible com-
binations into two calculi, namely Compensating CSP (cCSP) and the Sagas
Calculus as well as a comparison of the two.

A two-way encoding between the predecessor of BPEL namely BPEL-4WS
and the ISO-standard LOTOS is given in [6]. Foster’s approach [7] deals in detail
with a mapping of BPEL-4WS to Finite State Processes (FSP). Another seman-
tics, i.e. a feature-complete petri net semantics, is introduced by Lohmann in
[13]. The underlying model of this approach are open work flow nets (oWFNs),
an extension of petri nets by special places that can be triggered at any moment
from outside the process in order to model asynchronous message passing. The
translation of BPEL processes to those oWFNs is based on a collection of petri
net patterns for the primitives of BPEL.

Approaches especially concentrating on certain key aspects of BPEL are
those of Koshkina [10] and Pu et al. [21] [20]. Koshkina’s approach deals with
the link semantics of the flow construct. The author also covers the basic and
structured activities of BPEL, excluding the scope construct and consequently
also fault, compensation, event and termination handlers. Those constructs are
covered by Pu et. al in the form of a big steps semantics for BPEL-4WS. An
extension to timed automata, also dealing with time constraints, is then intro-
duced in [19]. In [16], Mazzara and Lucchi reduce the three mechanisms fault-,
event-, and compensation-handling to a single mechanism, namely notification,
using a calculus similar to the π-calculus [17].

A direct timed extension of the asynchronous π-calculus to deal with web
transactions is the webπ-calculus [11]. A derivate, the webπ∞-calculus is used
in [12] to describe the fundamental behavior of the scope construct of BPEL. A
complete encoding of the BPEL scope construct into the webπ∞-calculus finally
is given in [14].

1.2 Contribution of this work

The main contribution of this work is a formal semantics for the scope construct
of BPEL, including fully programmable fault, compensation and termination
handlers. Therefore a new calculus – \textit{BPEL}_{fct} – is introduced, designed from
scratch, sharing basics with Milner’s Calculus of Communicating Systems (CCS)
The goal of our calculus and the respective encodings from BPEL is to be as close to the official BPEL 2.0 specification as possible. Keeping the main subject of consideration in mind, our calculus will model the behavior of a single BPEL process in isolation, i.e. excluding synchronization in communication with its partner Web Services. Consequently, also other abstractions, like from data and time dependencies, have been made in order to concentrate on the mechanisms mentioned before.

After an overview over the language BPEL itself, we will present a lightweight, i.e. XML-free, syntax of BPEL, namely \textit{BPEL$'$}. We then introduce the syntax of our new calculus \textit{BPEL$_fct$}, its design issues as well as comparisons to existing approaches to model BPEL. We then give an encoding of the simplified syntax of BPEL to our calculus. Next, the semantics of the basic and structured activities as well as the scope activity and process encapsulation of \textit{BPEL$_fct$} will be described. Finally, we will deal with advanced issues of BPEL and their implementation in \textit{BPEL$_fct$}, i.e. we will treat the concepts of compensation, default compensation order, and all-or-nothing of compensation handlers in more detail.
2 Overview of WS-BPEL 2.0

We will start with an exhaustive overview of the primitives provided by BPEL. Therefore, we will first describe the layout of a BPEL process and its handlers (event and fault handler) followed by a coverage of the basic and structured activities of BPEL, including scopes, compensation, and termination handlers. Within that overview we will use the following notational conventions:

- ? behind a XML attribute or node denotes that it is optional.
- + denotes that at least one or more elements of that category must be specified.
- * denotes that an arbitrary number of elements of that category may be specified (including none).
- | between two or more elements will be treated as an exclusive-or concerning the occurrence of the respective elements.
- ... will be used to signal that attributes or nodes have been left out and will not be covered by our treatment.

Concerning the Extensible Markup Language (XML), the term node denotes the complete syntactical construct enclosed between two matching, i.e. opening and closing, XML-tags (e.g. between <someNode> and </someNode>). An attribute is an assignment of the form name=value inside an opening tag of a node.

2.1 Process

![Figure 3: Process Structure](image-url)
BPEL as an Orchestration language allows the developer to specify the behavior of a Web Service via temporally and causally structured communication primitives referencing other Web Services. The whole orchestration is represented by a process structure (cf. figure 3) which is basically an encapsulation of a main activity, i.e. an ordinary basic or structured activity (including scopes) like described later on. A process is also equipped with a fault and optionally an event handler. The syntax of a process is described in figure 4.

The Web Services the process communicates with, have to be declared globally inside the `<partnerLinks>` node of the process or locally inside nested scopes (described later on) in the main activity. Those declarations internally refer to WSDL (Web Service Description Language) descriptions of the structure (port and available operations) of the partner Web Services. Also the role of the process in the communication behavior w.r.t. the partner Web Services has to be specified in the `<partnerLinks>` node. This information is then used to define the interface of the process being described. Global variables, i.e. variables accessible to the main activity and all nested scopes, can be declared inside the `<variables>` node.

Multiple instances of a process may exist simultaneously. Such an instance will be spawned by the first communication primitive inside the main activity, listening to a partner link. In order to coordinate the communication of those instances with the respective instances of its partner Web Services, correlation sets defined in the `<correlationSets>` node are used for tagging correlated message streams (cf. figure 5).

Two or more requests of the same operation can be sent from one process to another, resulting in multiple parallel receive/reply chains inside a single process instance and correlation set. In order to disambiguate the relationship between those inbound message activities (IMA), message exchange tags, defined in the `<messageExchanges>` node, are used within communication primitives (cf. figure 6).

Please note that a process is the encapsulation of an activity that shall be executed as a BPEL process and that consequently, a process itself is not an activity and thus can not be embedded in another activity.
2.1.1 Event Handler

```
<eventHandlers>?
   (<onEvent partnerLink="NCName" operation="NCName" ...) |
   <scope ...> ... </scope>
   </onEvent>
   |<onAlarm> ...
   <scope ...> ... </scope>
   </onAlarm>
   )+
</eventHandlers>
```

Figure 7: WS-BPEL 2.0 - event handler syntax

Events, i.e. messages from partner links (onEvent) or time outs (onAlarm), that may occur concurrently to the execution of the main activity are handled by event handlers, defined inside the <eventHandler> node (cf. figure 7). If an event occurs, the respective event handler is instantiated and executed concurrently to the main activity.
Figure 6: message exchange

2.1.2 Fault Handler

```
<faultHandlers>
  <catch faultName="QName"? faultVariable="BPELVariableName"? ...>
    activity
  </catch>* <!-- + if no catchAll node -->
</catchAll> activity </catchAll>
</faultHandlers>
```

Figure 8: WS-BPEL 2.0 - fault handler syntax

Inside the main activity faults may occur, either implicitly (e.g., because of an error in the communication with a partner) or explicitly via a throw statement. Those faults then will be handled by the fault handler defined inside the `<faultHandlers>` node. Such a fault handler (cf. figure 8) consists of several `<catch>` nodes, each of them covering a certain set of faults matching the name (`faultName`) or variable type structure (`faultVariable`). The `<catchAll>` node can be used to catch all other remaining faults that are not covered by the `<catch>` nodes.

2.2 Basic Activities

In the following we will give an overview of all BPEL activities.
Every activity, i.e. each basic or structured activity or scope described below, may have source and target containers as standard elements attached to it (cf. figure 9). Sources and targets form links between activities. Those links can be used to establish a partial order of execution inside flow structures. That behavior will be described later on in the description of the flow activity. Please note that in the following the availability of those standard elements will not be mentioned explicitly in the description of the individual BPEL activities, but will be hinted by dots (...) directly inside the activity node.

### 2.2.1 Invoke

```
<invoke partnerLink="NCName"
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"? ... >

   <correlations>? ... </correlations>
   ...
   <!-- e.g. here optional sources and targets could be defined -->
</invoke>
```

Figure 10: WS-BPEL 2.0 - invoke activity syntax

The invoke activity (cf. figure 10) is used to call Web Services specified in the `<partnerLinks>` node of an enclosing scope or process. Invoke can be used for asynchronous invocation (one-way, only the `inputVariable` attribute, i.e. the input to the called service, needs to be specified) and synchronous invocation (request-response, both variables, `inputVariable` and `outputVariable`, need to be specified). In the asynchronous case the execution continues right after the request is sent, while in the synchronous case the activity blocks execution after sending the request until the response has been received. In order to correlate
relevant parts of the message traffic with the respective process instance, the
<correlations> node can be specified.

2.2.2 Receive

<receive partnerLink="NCName" 
operation="NCName" 
variable="BPELVariableName"?
createInstance="yes\|no"?
messageExchange="NCName"? ...>

<correlations>?
  <correlation set="NCName" initiate="yes\|join\|no"? />+
</correlations>
...
</receive>

Figure 11: WS-BPEL 2.0 - receive activity syntax

The receive activity (cf. figure 11) blocks execution until a request from one
of the partner links of an enclosing scope/process, using a specific operation
tag, arrives. On arrival, the data of the message can be stored in an optionally
specified variable. As mentioned before, a receive activity can be used to in-
stantiate a process, which can be enforced by setting the createInstance flag
to "yes". As usual, correlation sets can be specified. In the case of a receive ac-
tivity that is used to instantiate a business process, correlation sets can also be
initiated on demand by setting the initiate attribute to "yes" (create a new
correlation set) or "join" (adapt existing correlation set from sender). Multi-
ple receive/reply chains can be supported by specifying the messageExchange
attribute.

2.2.3 Reply

<reply partnerLink="NCName" 
operation="NCName" 
variable="BPELVariableName"?
faultName="QName"?
messageExchange="NCName"? ...>

<correlations>? ... </correlations>
...
</reply>

Figure 12: WS-BPEL 2.0 - reply activity syntax

When receiving messages from a partner link using a certain operation, e.g. via
the receive activity, a corresponding response can be sent via the reply activity.
Optionally, data stored in a variable, specified by the variable attribute, can be sent along with the response. Again a message exchange attribute and a correlation set can be specified. The reply activity can also be used to inform the communication partner about the occurrence of a certain fault. In that case, the faultName attribute is set and the variable contains the fault data.

2.2.4 Assign

<assign ...>
    ...
    ( <copy ...> from-spec to-spec </copy>
    | <extensionAssignOperation> ... </extensionAssignOperation> )+
</assign>

Figure 13: WS-BPEL 2.0 - assign activity syntax

Using the assign statement, multiple variables can be assigned atomically via a set of <copy> nodes. Inside such <copy> nodes, the variables and the values that will be assigned to them are specified. Those values can either be values of other variables or of complex expressions (usually XPATH 1.0 expressions) referencing variables in the standard BPEL namespace. Using the <extensionAssignOperation> node, elements (i.e. expressions, variables, or functions) of other namespaces can be used. In the from-spec expression the source variable is referenced (XPATH binding) or an arithmetic expression is given. The data will finally be copied to the variable referenced by the to-spec (XPATH) expression.

2.2.5 Validate

Variables in BPEL are typed and may have a WSDL message type or XML Schema type. The validate activity is used to validate the values of variables against their type. In case of an invalid value, the BPEL standard fault bpel:invalidVariables is thrown.

<validate variables="BPELVariableNames" ...>
    ...
</validate>

Figure 14: WS-BPEL 2.0 - validate activity syntax
2.2.6 Wait

<wait standard-attributes>
  ...
  <for expressionLanguage="anyURI">duration-expr</for>
  | <until expressionLanguage="anyURI">deadline-expr</until>
</wait>

Figure 15: WS-BPEL 2.0 - wait activity syntax

The wait activity is used to delay execution <for> a certain amount of time or <until> a certain deadline is reached.

2.2.7 Empty

<empty ...>
  ...
</empty>

Figure 16: WS-BPEL 2.0 - empty activity syntax

Sometimes the developer is forced by the language specification to specify an activity even in situations where no execution of an activity is wanted, e.g. in fault handlers suppressing a fault. In those cases the empty action can be used.

2.2.8 Throw

<throw faultName="QName" faultVariable="BPELVariableName"? ...>
  ...
</throw>

Figure 17: WS-BPEL 2.0 - throw activity syntax

Implicit faults are thrown for instance, when variables are validated against their type or if a fault is signaled by a communication partner inside a receive statement. But sometimes faults shall be thrown explicitly, e.g. when reaching a certain inconsistent state in the execution of a process. This behavior can be accomplished by using the throw statement, specifying the fault name in the faultName attribute and optional fault data in the faultVariable attribute.
2.2.9 Re-throw

```xml
<rethrow ...>
    ...
</rethrow>
```

Figure 18: WS-BPEL 2.0 - re-throw activity syntax

The re-throw activity re-throws a fault caught by a fault handler to the fault handler of the enclosing scope or process. Consequently, that primitive can only be used inside fault handlers.

2.2.10 Exit

```xml
<exit ...>
    ...
</exit>
```

Figure 19: WS-BPEL 2.0 - exit activity syntax

The exit activity will terminate the current process instance instantaneously without calling any fault, termination or compensation handler (the latter two will be described below when dealing with the scope construct).

2.3 Structured Activities

By now we have seen the basic activities of BPEL. In order to compose more complex processes out of those basic activities, several structured activities can be used. These include sequential and parallel composition, conditionals, loops, and selective behavior dependent on incoming messages and timeouts. In the following descriptions the term activity denotes basic activities, structured activities as well as scopes (described below).

2.3.1 Sequence

```xml
<sequence ...>
    ...
    activity+
</sequence>
```

Figure 20: WS-BPEL 2.0 - exit activity syntax

Using the sequence activity, several activities can composed to be executed sequentially, i.e. the execution of an activity inside a `<sequence>` node may start when the preceding activity (in XML document order) has completed successfully.
2.3.2 If

```xml
<if ...
   ...
   <condition expressionLanguage="anyURI">bool-expr</condition>
   activity <!-- 1 -->
   <elseif>
     <condition expressionLanguage="anyURI">bool-expr</condition>
     activity
   </elseif>*
   <else>?
     activity
   </else>
</if>
```

Figure 21: WS-BPEL 2.0 - if activity syntax

The if statement is used to specify conditional behavior in BPEL. When executing the if activity, the BPEL engine will at first evaluate the expression of the top most `<condition>` node. In case of a positive result, i.e. the expression evaluates to true, the first activity (1) will be executed. In case of a negative result, the conditions of the `<elseif>` nodes will be checked in XML document order from top to bottom, resulting in the execution of the inner activity of the first `<elseif>` node whose condition evaluated to true (if any). If all `<condition>` nodes evaluate to false and an optional `<else>` node is specified, the activity of that node will be executed. Upon successful completion of the execution of an inner activity (or immediately in the case that all conditions do not hold and no `<else>` node is specified), the if statement completes.
2.3.3 Pick

```xml
<pick createInstance="yes|no"? ...>
  ...
  <onMessage partnerLink="NCName"
    portType="QName"?
    operation="NCName"
    variable="BPELVariableName"?
    messageExchange="NCName"/>
  <correlations>
    <correlation set="NCName" initiate="yes|join|no"? />+
  </correlations>?
  ...
  activity
</onMessage>+
<onAlarm>
  {
    <for expressionLanguage="anyURI"?&gt;duration-expr&lt;/for>
  |
    <until expressionLanguage="anyURI"?&gt;deadline-expr&lt;/until>
  }
  activity
</onAlarm>*
</pick>
```

Figure 22: WS-BPEL 2.0 - pick activity syntax

Using the pick activity, a selective event processing behavior can be specified, similar to that of an event handler. When the pick activity is executed, further execution will be blocked, until a message from a partner link arrives or a timer event occurs. Incoming messages are accepted by the `<onMessage>` nodes according to the same scheme as of the receive activity. When a message matching an `<onMessage>` node arrives or a time out occurs, the respective inner activity of that node, and only that activity, will be executed. Consequently, there is a race between arriving message and timeouts (specified by `<onAlarm>` nodes), waiting for a certain amount of time or until a deadline is reached. It shall be noted that all `<onMessage>` nodes of a pick statement are used as instantiating activities for the business process if the `createInstance` attribute is set to "yes".
2.3.4 Flow

```
<flow ...>
  ...
  <links>?
    <link name="NCName">+</n
</links>
  activity+
</flow>
```

Figure 23: WS-BPEL 2.0 - flow activity syntax

The flow construct allows for concurrent execution of several activities. Consequently, the flow activity completes when each of the concurrent execution branches has completed. Recall the standard elements that may be specified for each activity (cf. figure 9). On completion of an activity with an inner `<source>` node, the `transitionCondition` (a boolean expression) is evaluated. If the evaluation results in true, the link specified of the `<source>` node is activated. A `<target>` node inside an activity thus can be used to specify when the activity may be executed dependent on the activation status (expressed in the boolean `joinCondition` attribute) of the links. Links, i.e. link names, can be declared inside the `<links>` node of a flow.

2.3.5 While, Repeat Until, For

```
<while ...>
  ...
  <condition expressionLanguage="anyURI">bool-expr</condition>
  activity
</while>
```

Figure 24: WS-BPEL 2.0 - while activity syntax

```
<repeatUntil ...>
  ...
  activity
  <condition expressionLanguage="anURI">bool-expr</condition>
</repeatUntil>
```

Figure 25: WS-BPEL 2.0 - repeat until activity syntax
The while and repeat until loop constructs are standard, i.e. they loop their inner activity as long as or until a certain condition holds. The semantics of the for each activity differs to the usual sequential semantics. The inner scope activity can also be executed as a concurrent flow (parallel = "yes") and the loop may already terminate when only a certain subset of execution branches (defined by the <completionCondition> node) has completed.

**2.4 Scope**

By now we have presented the process structure with fault and event handlers as well as basic and structured activities that can be used as their activities. But before we proceed with the last primitive provided by BPEL, we will explain the behavior of a process in case of a fault.

**Example 3.** Reconsider the business process example of a travel agency. Again we will want our customer to be able to book travels online. The service will now include booking a flight, booking a hotel room and renting a car. The corresponding process is illustrated in figure 27. Concerning the partner links, we have a customer partner link (needed to be able to communicate with the user of the service) and a link to each Web Service, our new service will orchestrate, i.e. the flight agency, the car rental, and the hotel. A process instance will be spawned upon each invoke on the customer partner link, signaled to the receive activity which is the first activity inside the main activity’s sequence. So after invocation, our Web Service will now invoke its partner Web Services, resulting in the booking of a flight, renting a car and reserving a hotel room. Finally, the customer will be signaled, that his order has been processed via a reply activity.

But of course, during the interaction of our Web Service with its partners, faults might occur. For example the invocation of the hotel room reservation might fail because of no room being available. That could result e.g. in our invoke operation throwing a H:noRoomAvailable fault. Therefore, we will have to define a fault handler that will intercept those faults and react accordingly. So let us define our fault handler the following way:

- **intercept FA:noFlightAvailable:** We know that this fault will be signaled to our Web Service by the flight agency, i.e. during the first invoke activity
of our process. We also know that no other relevant activity has been executed by now. Consequently, the only activity that we will have to do, is to inform the customer that the reservation failed (e.g. by replying on the customer partner link, specifying the faultData attribute).

- **intercept CR:noCarAvailable**: We can infer that this fault will have been signaled to us by the second invoke activity communicating with the car rental Web Service. What we will have to do now is to cancel the flight (e.g. by replying on the flight agency partner link using some cancel operation) before informing the customer because we do not want our customer to have to deal with the cancellation.

- **intercept H:noRoomAvailable**: This fault will be signaled to our Web Service by the invocation of the hotel room reservation, which is the third invoke activity in the process’ main activity’s sequence. In case of that fault we will have to cancel both, the reservation of the flight as well as of the car before informing the customer.

![Figure 27: Travel Agency Business Process revisited](image)

The definition of the fault handler in example 3 shows that in an abstract sense we are trying to undo the execution up to the point of failure in case of a fault. This mechanism is called **compensation**. At first glance the ability of the fault handler to distinguish between the different fault cases triggering the respective counter activities might seem to suffice. But now consider for example a fault inside a branch of a flow activity. The fault handler would have no chance to know exactly which activities have been executed inside the remaining branches up to the point of failure, and especially not in which order. The infeasibility to do manual case analysis and the resulting non-compositionality is another reason why a different approach to that mechanism of reversing done work up to a certain point in time is needed.
Therefore, BPEL provides the scope construct in order to encapsulate an inner activity alongside with a counter activity that, in case of a fault, compensates that inner activity, i.e. that tries to undo its effects.

Figure 28: Scope Construct

```xml
<scope ...>
  ...
  <variables>? ... </variables>
  <partnerLinks>? ... </partnerLinks>
  <messageExchanges>? ... </messageExchanges>
  <correlationSets>? ... </correlationSets>
  <eventHandlers>? ... </eventHandlers>
  <faultHandlers>? ... </faultHandlers>
  <compensationHandler>? ... </compensationHandler>
  <terminationHandler>? ... </terminationHandler>

  activity <!-- inner activity -->
</scope>
```

Figure 29: WS-BPEL 2.0 - scope activity syntax

Figure 28 illustrates the scope construct, while figure 29 shows its syntax. The inner activity might be any activity like described in sections 2.2 and 2.3 as well as nested scopes. When executing a scope, its inner activity will be executed as long as no fault occurs or the scope is being terminated. Termination of scopes will be described later on. If the inner activity completes successfully, the scope’s compensation handler, i.e. the counter activity that tries to undo the effects of the inner activity, will be installed. The compensation handler will then be available to the fault handler of an outer process or scope. If the scope is enclosed by another scope, the outer scope’s compensation and termination
handler might refer to that scope’s compensation handler, which will be shown later on.

If any fault occurs inside the main activity, that is not handled by the fault handler of a nested scope (if any), the fault handler of the current scope will be executed. Please note that in this case the compensation handler will not be installed because the inner activity did not complete successfully. The fault handler of a scope might re-throw a fault to the fault handler of the outer process or scope. A termination handler can also be specified for each scope. That handler will be called, if some execution branch, concurrent to the scope’s branch, faults, in order to give the scope a chance to compensate. Both, fault and termination handler, cancel further execution of the inner activity. In addition to a fault, compensation and termination handler, an event handler can be specified like for any process. That event handler will also be installed before execution of the scope’s inner activity and its instances are running concurrently to it. A scope may also define its own local partner links, it will communicate with. Of course it may also use the partner links of all outer scopes or the global partner links, defined by the enclosing process.

2.4.1 Compensation Handler and Compensation Activities

```
<compensationHandler>
  activity
</compensationHandler>
```

Figure 30: WS-BPEL 2.0 - compensation handler syntax

A compensation handler is just an encapsulation of an activity (cf. figure 30).

```
<compensateScope target="NCName" ...
...
</compensateScope
```

Figure 31: WS-BPEL 2.0 - compensate scope activity syntax

```
<compensate ...
...
</compensate>
```

Figure 32: WS-BPEL 2.0 - compensate activity syntax

There are two types of compensate commands that will explicitly trigger compensation and thus may occur inside fault, compensation, and termination handlers. The compensateScope activity calls the installed compensation handler(s)\(^3\) of the scope named target. That scope must be directly enclosed by the

\(^3\)A scope inside a loop construct might be executed multiple times and may therefore install multiple compensation handler instances. All those instances are implicitly grouped and will be called by a compensation activity altogether.
scope containing the activity. The `<compensate>` activity calls all compensation handlers of directly enclosed scopes in default compensation order. Default compensation order denotes reverse execution order in case of sequential activities (cf. figure 33), parallel branches stay parallel but in case of parallel sequences those sequences will be also be reversed recursively.

Example 4. A fault in scope_{i3} will be handled by its fault handler fh_{i3} which might be defined to re-throw that fault to fault handler fh_{i1} of the outer (directly enclosing) scope_{1}. The compensate activity inside fault handler fh_{i1} will call compensation handler C_{i2} before C_{i1} which is in default compensation order. C_{i3} was not installed because of the fault in scope_{i3} and consequently will not be executed.

The command to compensate can only occur inside a fault handler, a termination handler, or a compensation handler, but can be nested in a child scope of such a handler.

In principle, a developer has the choice to program a compensation handler for a scope. If there is no compensation handler defined explicitly, the standard compensation handler will be installed. The standard compensation handler simply calls the compensation handlers of all directly enclosed (child) scopes. So if the developer decides to define a compensation handler manually, not only he/she must find an appropriate static sequence/flow of actions to undo the execution of the corresponding scope before runtime, but also has to take care of calling the appropriate compensation handlers of the child scopes which are only accessible to this direct parent scope (cf. figure 34).
2.4.2 Termination Handler

A scope may also specify a termination handler, i.e. an encapsulation of an activity that will be called upon termination of that scope. Termination occurs when a flow faults, i.e. if one parallel branch of a flow faults, the top-level scopes of all other branches will be terminated (cf. figure 36). Compensate activities may occur inside a termination handler, such that the termination handler can be used for compensating the affected scope.
To sum up, BPEL provides three strategies to deal with errors inside business processes at different levels. Fault handlers are used to directly undo the effects of the activity that caused the fault (including the sequence, flow, loop, etc. it resides in) and provide the possibility to call further compensations. In case of nested scopes inside such a faulted activity the respective compensation handlers of those scopes can be called in order to reverse their work. Scopes in execution branches concurrent to the faulted activity are terminated but are given the chance to compensate using their respective termination handlers.
3 Formalization of Scope-based FCT-Handling in WS-BPEL 2.0

3.1 BPEL′ - A lightweight Syntax of WS-BPEL 2.0

Our focus is on investigating the concepts of fault, compensation and termination (FCT) handling in BPEL in respect to the scope construct. Therefore, we will at first present a lightweight version the language called BPEL′ (cf. figure 37), in which we will abstract away from data and time dependencies and the XML-based syntax as well as unrelated attributes. Furthermore, we will consider only a subset of the functionality provided by BPEL. In detail, primitives that will not be encoded in our calculus are loops (while, repeat until, for each) and event handlers, in order to concentrate on the core concepts of FCT handling. Moreover, we will not model the link semantics of flows, nor the instantiation of processes and consequently, also not correlation sets and the binding of partner links to specific Web Services. Please note that in the following we will assume well-formed BPEL process descriptions as the subject to further examination, i.e. we will assume that the process description we deal with have passed the necessary static analysis steps of BPEL (like stated in Appendix B of [18]).

Recall that concerning the communication structure of BPEL processes, their communication partners (other Web Services) are referenced by partner links. Every Web Service provides a set of operations that serve as communication channels, i.e. those operations can be invoked, replied on, and received from. Message exchange tags are used to disambiguate receive/reply chains of a certain partner link and operation. In the following, let \( P \) denote the set of partner link names, \( O \) the set of operation names and \( M \) the set of message exchange tag names. In order to allow message exchange tags to be specified optionally, let \( std \in M \) denote the standard message exchange tag which may be used if no message exchange tag is specified explicitly. Faults may be signaled by communication partners or be thrown internally. Therefore, let \( F \) denote the set of fault names. Every scope in BPEL has a unique name, consequently let \( S \) denote the set of scope names.

When mapping a BPEL process description to a bpel.process of BPEL′, a basic activity encoded as a XML node in BPEL will be encoded by the corresponding basic activity of BPEL′ via extraction of possible partner link, operation, message exchange, fault or scope names from the attributes of the nodes. Standard structured activities like sequences and flows will be mapped to their respective counterparts by composing the recursive mappings of their constituents. The basic activity re-throw and the structured activities if, pick and scope as well as the process itself have to be considered in detail.

We will require the re-throw activity to get the current fault, that was handled by the enclosing fault handler, passed as the argument \( f \). Further, recall the syntax of the if activity (cf. figure 21). Inside an if statement a complete hierarchy of conditions can be checked in order to select an appropriate action. We are abstracting from data dependencies, consequently, we will just collect all activities inside an \(<\text{if}>\) node, i.e. including all activities inside \(<\text{else}>\) and \(<\text{elseif}>\) nodes, in a set \( S \), which will be the argument of the if statement in BPEL′. A similar procedure is necessary for the pick activity (cf. figure 22 for
The set $M$ will contain all occurring combinations of partner link, operation, and message exchange names and the respective activities occurring in the `<onMessage>` nodes of the pick activity. In the set $A$ we will collect all activities of `<onAlarm>` nodes. Those two sets will then be used as the arguments of the pick statement in BPEL'. The names of processes and scopes are passed as the first argument of the BPEL' counterparts, mappings of their inner activity as the second. Both constructs may also define a fault handler which will be encoded by the set $F$ as the third argument. If a fault handler is defined, the respective `<catch>` and `<catchAll>` nodes need to be mapped to this set $F$. As already mentioned a `<catch>` node may be sensitive to a subset of faults matching certain fault name/fault variable type structures. We will assume that all possibly occurring subsets are encoded as a unique name in our set of fault names $\mathbb{F}$. Thus when encoding `<catch>` nodes in our set $F$, we will add for each such a node a tuple $(f, A) \in \mathbb{F} \times \text{structured}$, with $f$ being the encoding of the sensitivity of the node to a unique name and $A$ being the mapping of the activity to BPEL'. The specification of BPEL does not forbid overlapping sensitivity ranges of `<catch>` nodes in fault handlers. In those cases the activity of the first `<catch>` node (in XML document order), sensitive to a certain combination of fault name and fault variable type, will be called. For simplicity we will assume non overlapping sensitivity ranges which can be achieved by strengthening the respective sensitivity ranges. If a `<catchAll>`
node is specified, all remaining fault names, i.e. fault names not covered by the `<catch>` nodes, will be mapped to the respective mapping of the activity of the `<catchAll>` node. If no fault handler is specified for a scope or process, the standard fault handler will be installed (cf. figure 38, respective mapping to BPEL': \( F = \mathbb{F} \rightarrow \{ \text{sequence}(\text{compensate},\text{rethrow}) \} \)).

```xml
<faultHandlers>
  <catchAll>
    <sequence>
      <compensate/>
      <rethrow/>
    </sequence>
  </catchAll>
</faultHandlers>
```

Figure 38: WS-BPEL 2.0 - standard fault handler

A scope may also specify a compensation handler (fourth argument) and a termination handler (fifth argument). Like in the case of the fault handler, standard handlers, i.e. default handlers which will just compensate all child scopes, will be installed, if a custom handler is not specified (cf. figures 39 and 40, respective mapping to BPEL': compensate).

```xml
<compensationHandlers>
  <compensate/>
</compensationHandlers>
```

Figure 39: WS-BPEL 2.0 - standard compensation handler

```xml
<terminationHandlers>
  <compensate/>
</terminationHandlers>
```

Figure 40: WS-BPEL 2.0 - standard termination handler

### 3.2 Existing Approaches and Design Issues

Before we will cover our calculus \( BPEL_{fct} \) in detail, we will begin with a description of observations that were made while examining existing approaches to model certain aspects of BPEL and, consequently, of issues that occurred in the design process of \( BPEL_{fct} \).

#### 3.2.1 The Sagas Calculus and cCSP

At first, two foundational approaches to compensation of web transactions have been evaluated. These are cCSP (compensable Communicating Sequential Processes) [5], an extension to CSP by transactions and compensations, and the
Sagas Calculus [3], a formalization of the seminal idea of Sagas [9]. The approach that is followed by cCSP as well as BPEL is to allow aggregations of activities w.r.t. compensation. In cCSP, a process $A$ can be equipped with a compensating process $A^\circ$, using a compensation pair $A \div A^\circ$, forming a compensable process. Whenever $A$ needs to be compensated, $A^\circ$ will be executed. cCSP forces such a compensable process to be enclosed by a transaction $[A \div A^\circ]$, which behaves atomically in respect to failure and compensation.

That means, in any case, may it be faulting or succeeding behavior, the installed compensations of a transaction’s inner process are lost when exiting the construct – which is the intended behavior in respect to cCSP. Thus an encoding of a BPEL scope as a transaction seems to be infeasible because its inner activity can not be reverted in the case of successful completion and a future fault of the outer process.

Another problem arises in the fact that only processes and not compensable processes may be equipped by a compensation process. Consequently cCSP does not allow for programmable compensations, i.e. compensations that over-ride the default compensation. Programmable compensations may execute additional activities to the inner accumulated compensations or may even decide to compensate only a certain fraction of the accumulated compensations. In BPEL, compensation handlers are programmable as it is also the case in our calculus introduced later on.

The Sagas Calculus also allows processes to be enclosed by transactional blocks called Sagas $\{.,\}$ in order to allow inner fault handling without affecting the outer process. In contrast to cCSP’s transactional block primitive, a Saga exports its (remaining) accumulated inner compensations to the outer process, such that this outer process may access those inner compensations in case itself has to be compensated. The basic Sagas calculus only allows primitive activities to have compensating activities associated with them. An extension to the nested Saga Calculus (also shown in [3]) by partially programmable compensation breaks that restriction by providing the possibility to equip Sagas with compensating processes. But again like in the case of cCSP, those compensations can not access the accumulated compensations of the saga they are associated with.

3.2.2 Lohmann’s petri net approach

One side-goal in mind while designing our approach to a formal semantics for BPEL was the ability to have a Labeled Transition System (LTS) representation of the behavior of a BPEL process. Such a LTS could then be used e.g. for further examination of logical properties using standard model checkers or of timed reachability extending [1]. Of course several approaches have been evaluated in respect to our goals.

Despite being based on generalized petri nets, Lohmann’s approach [13] was considered as a framework for the examination of BPEL in the beginning of this work because of its feature completeness. A translation of oWFN (open Workflow Nets) to LTS was thought to be possible in general, but the resulting overhead of that translation was then found to be too high. The reason is that the oWFN patterns, the primitives of BPEL are mapped to, become unmanageably larger with increasing complexity of the primitive. For example the scope construct’s oWFN pattern consists of more than 40 places and 10 transitions.
This makes it hard to do manual, i.e. non tool-supported, formal reasoning on the oWFN level. Another disadvantage of that approach was found to be the limited observability of the oWFN encodings. For example the effects of $wait$ are invisible to an observer and would consequently be completely lost after translation.

In our calculus we were also modeling that activity, using the internal action $\tau$, but changing the encoding in order to allow an observer to see the detailed effects of that activity, i.e. waiting for which duration / until which date, seems to be trivial in contrast to the oWFN approach. The reason is that in the oWFN format the only visible encodings, including the respective names, are communication channels.

3.2.3 Encoding of BPEL into Web$\pi$:

In the works of Laneve & Zavattaro [11] [12] and Lucchi & Mazzara [14], an extension to the asynchronous $\pi$-calculus with web transactions is given. In the latter a complete encoding of the BPEL scope construct into the web$\pi_\infty$-calculus, a derivate of the web$\pi$-calculus without time constraints, is described. Although the calculus’ reduction semantics could in principle have been transformed into an operational semantics in respect to LTS, two obvious disadvantages led to the decision not to use that proposal as a framework.

First, a lot of overhead at encoding and execution time would have been produced. The reason is that a huge amount of message passing has to be done in order to synchronize causally dependent executions in the encoding of a BPEL process into the asynchronous $\pi$-calculus. An example for that is the sequence activity. In the web$\pi_\infty$-encoding the first part of the sequence is encoded sending a message after successful completion on a channel restricted to the encoding of the sequence. The second part is guarded by a reception of that completion message of the first part of the sequence. Both parts are then composed in parallel (see [14], 4.4.1). As a result, whenever the first part of a sequence completes, an internal transition is necessary in order to proceed with the next part. Similar overheads can be observed in the encodings of the flow and especially of the scope construct (including its handlers), which makes formal investigations difficult.

Second, compensation handlers of scopes are encoded as separate sub processes concurrent to the scope process and consist of a web transaction receptive to a certain channel. If a scope completes successfully, a command is sent to its compensation handler using that channel. The compensation handler’s web transaction then transforms into the encoding of the compensation handler’s main activity guarded by a input on a channel named like the original scope. Thus in order to compensate, a message has to be sent to that channel with that name. Consequently, in order to model a full compensate command, i.e. compensating all child scopes in default compensation order, the names of all child scopes having been executed successfully and their execution order must be known prior to execution, which is not trivially possible.

3.2.4 Foster’s vs. Koshkina’s LTS-based approaches

Next, Foster’s approach [7] was compared to Koshkina’s approach [10]. An obvious advantage of those two approaches was that both describe mappings
from BPEL to Labeled Transition Systems (in Foster’s approach a mapping from BPEL to Finite State Processes (FSP) is described, for which a LTS-based semantics is given in [15]). A first inconvenience of the FSP-based approach of Foster was found while examining the terminate activity of BPEL4WS, the predecessor of BPEL, in which this activity has been renamed to exit. The effect of terminate is to cancel the execution of a process. In Foster’s approach this action is modeled by providing a terminate transition from a state where termination is possible to a deadlocked state uniquely generated for each such termination-enabled state (cf. [7], chapter 4.2.4, figure 4-5). This is unpleasant and leads to an artificially bloated state space. Whereas in Koshkina’s approach, from each state that allows termination, an end-transition leads to only one special end state (without successors). Consequently, concerning immediate process termination behavior, we followed Koshkina’s approach (cf. inference rules [EXIT *]).

The flow primitive of BPEL allows for concurrent composition of sub-activities. Like already mentioned, those sub-activities can be connected using links to form a partial order of execution. In detail, that means that for a sub-activity to be ready for execution, certain preconditions, i.e. successful completion of other activities, are necessary. The successful completion of that activity consequently enables the start of further sub-activities. A boolean joinCondition expression (e.g. \( \text{link}_1 \land \text{link}_2 \lor \text{link}_3 \)) inside a targets node of an activity determines, which link state is essential for the start of the respective activity. Furthermore, a transitionCondition expression inside a sources node of an activity is used to control the activation of the outgoing links, in order to allow execution of further activities in that flow. Despite the description of the flow construct in Foster’s work (cf. [7], p.74, figure 4-7) and of incoming and outgoing links (cf. [7], p.75, figure 4-8), the modeling of the boolean join and transition conditions themselves is missing. The BPEL calculus described by Koshkina however introduces two special primitives modeling those conditions (\( \uparrow \text{ltcactivity} \) and \( c \Rightarrow \text{activity} \) with transition condition \( tc \) and join condition \( c \)). Another issue concerning linked transitions of BPEL, which has to be considered is Dead Path Elimination, which is dealt with in Koshkina’s approach but not in Foster’s. During the execution of that partial ordered flow of activities, it could happen that the whole system deadlocks, i.e. a sub-activity of the flow is waiting for the fulfillment of its join condition which will never evaluate to true because all preceding activities have already completed and have set their transition conditions resulting in a false join condition for the current activity. For those cases, the informal semantics of BPEL demands that these activities will be skipped, like all activities depending on it (Dead Path Elimination, DPE). In Koshkina’s approach, those cases are handled by a special inference rule Join2 (see [10], section 3.9). Despite the importance of links in flows and DPE, we decided not to model those aspects of BPEL in our calculus because of its complexity and its minor role concerning pure compensation and fault handling. However Koshkina’s ideas are orthogonal to our system such that an extension of \( BPEL_{fct} \) should be straightforward.

While Koshkina abstracts away from data dependencies using nondeterminism, Foster models variables explicitly as separate sub-processes. Those processes consist of one state per possible value, self-loops for reading access, and write transitions to the respective states, representing the written values. However, it is not explained how those value ranges are determined, which is not
a trivial task, since BPEL allows arbitrary XPath 1.0 expressions for variable access. Furthermore, variables may possess structured types which makes the modeling of variables even more complex. Foster’s approach also lacks of the modeling of implicit (writing) variable access through communication activities, like receive and synchronous invoke. In order to concentrate on the basic behavior concerning scope based fault-, compensation-, and termination-handling, BPEL also abstracts away from data dependencies using nondeterminism.

Last but not least, it shall be noted that both approaches lack of a detailed consideration of the scope construct of BPEL. While BPE does not contain any scope related primitives, Foster’s FSP approach deals with fault- and compensation handling on a basic level. However, there is no hint about how a scope’s inner activity is shielded from outer activities, e.g. by the hiding or renaming operators in FSP. Although an example illustrating the modeling of a BPEL scope with its own fault-handler is given (cf. [7], p.86, figure 4-15), it is not obvious how a fault will be handled only by the fault handler of the scope it occurred in and not also by a fault handler of an outer scope or process also synchronizing on that fault. Another issue of Foster’s approach is the treatment of compensations and compensation handlers. Any compensation is modeled statically like in the webπ-approaches, i.e. the complete sequence of compensation handlers called in case of a compensate activity must be known beforehand and is not dynamically created while runtime.

3.2.5 Scopes and Compensation-Handling

On looking for existing approaches dealing with the scope construct of BPEL, a formal operational semantics to a simplified version of BPEL4WS was found in [21]. In that work Pu et al. describe how to manage the hierarchy of accumulated compensation handlers, i.e. installed compensation handlers of scopes that have completed successfully, inside a big-step operational semantics. Their idea is to extend the configuration of a process from a process term to a tuple containing the current process term and a list of compensation triples containing a scope’s name, its installed compensation handler, and the respective compensation context of that compensation handler. Thus, the second component of a configuration stores a complete compensation tree mirroring the scope structure of the executed process term.

Concerning scope-based compensation, that idea was followed in our calculus with some modifications. The underlying semantics of BPEL was decided to be a small-step semantics in order to be as close as possible to a real BPEL engine, executing a Business Process step by step. So the first problem that occurred was that, unlike to the big-step semantics, intermediate compensation contexts had to be stored. Storing those at the configuration level only, like in Pu’s approach, would not have worked without major modifications. The reason is that in BPEL, scopes may be nested to arbitrary depth, which is also the reason why installed compensation handlers form a compensation tree (compensation handlers might call compensation handlers of child scopes of the original scope, which also might call further compensation handler of child scopes, and so on . . .). So, for example a scope’s inner activity might be a sequence of scopes, like in example 4. When executing the outer scope containing the inner scopes, the compensation handlers of successfully completed inner scopes will have to be stored intermediately. But they will have to be deleted if the
outer scope fails. Of course, the second component of the configuration could be used for storing them and passed through in execution from the outer scope to the inner sequence and, consequently, the inner scopes. But complex tree operations, i.e. deletion of specific nodes, belonging to the failed scope, would then be needed, in case of a fault. So, in order to keep the calculus simple, we extended our scope construct by an accumulated compensation context $C_A$, which is used to store intermediate compensation handlers, accumulated while executing the inner activity.

An inconvenience of the approach of Pu et al. is the way the installed compensation handlers and their respective contexts are stored. When calling those handlers, their main activity and context have to be extracted from the compensation triple before execution. In our approach we use specially marked scopes as containers for compensation handlers which allows for direct execution using the inference rules without extraction.

Another problem of the approach of Pu et al. is the restriction to a single context. As will be described in the following sections, it is sometimes necessary to define a compensation handler $C$ to possess inner scopes, e.g. $S_1, \ldots, S_n$. Inside those inner scopes $S_i$, compensations might be invoked. I.e. compensation handlers $D_j$ that have been installed before the execution of the current compensation handler $C$ shall be executed, enabling a scope’s compensation handler to compensate child scopes. But those scopes $S_i$ inside that compensation handler $C$ might install their own compensation handlers $E_i$. Those handlers $E_i$ shall not be executed if compensation is called. Therefore, the currently installed compensation handlers $E_i$ and the previously installed handlers $D_j$ must not be mixed up. This forced the extension of the scope activity of $BPEL_{fct}$ with a fixed compensation context $C_F$. To sum up, the accumulated compensation context $C_A$ of a scope is used to intermediately store installed compensation handlers until they are finally transferred to the fixed compensation context $C_F$ from which they may be executed in case of a compensation call. Consequently, also the configuration was extended to a triple s.t. each context now was represented on the configuration level.

In the beginning of the conception of our calculus, also the work [4] of Butler and Ferreira was evaluated while searching for approaches on how to store the hierarchy of installed compensation handlers. In that approach the authors give an operational semantics to $StAC_i$, a Business Process language inspired by the BPBeans framework. At first, the language $StAC_i$ is introduced sharing general concepts, like compensation and exception handling, with BPEL on a low level, i.e. the language allows for direct manipulation (execution, deletion, extension and merging) of the compensation context. Finally, a mapping from $StAC_i$ to $StAC$ is given, restricting those manipulations of the compensation contexts of scopes to their respective hierarchical level. Thus, a function mapping those level indices to installed compensation handlers is part of the configuration of a $StAC$ process. That complexity of the direct manipulation of the compensation tree and having a function inside the configuration finally was the reason for choosing to store that hierarchy on the process term level.

### 3.3 The Calculus $BPEL_{fct}$

In the following section we will introduce the calculus $BPEL_{fct}$, which will later on be used to give a formal semantics to the core concepts of scope based FCT
3.3.1 Syntax

In order to formally investigate the semantics of FCT handling in BPEL, we are following a step-wise approach of abstraction. BPEL’s XML-based syntax has been mapped to a more lightweight syntax $BPEL'$ in the preceding section. We will now describe the syntax of our calculus $BPEL_{fct}$, followed by a mapping from $BPEL'$ to $BPEL_{fct}$. Finally, we will present the semantics of our calculus.

The context-free grammar of $BPEL_{fct}$ is given in figure 41. While $B$ contains all basic operations, $S$ consists of the structured activities. The process construct as well as its possible end states are given in $P$.

$$B ::= 0 \quad \text{(stop process)}$$
$$| \quad \text{rec pl op me} \quad \text{(receive)}$$
$$| \quad \text{rep pl op me} \quad \text{(reply)}$$
$$| \quad \text{inv pl op} \quad \text{(asynchronous invoke)}$$
$$| \quad \text{invs pl op} \quad \text{(asynchronous invoke - sending part)}$$
$$| \quad \text{invr pl op} \quad \text{(asynchronous invoke - receiving part)}$$
$$| \quad \tau \quad \text{(internal action)}$$
$$| \quad \text{throw f} \quad \text{(fault)}$$
$$| \quad ↑ \quad \text{(compensate)}$$
$$| \quad ↑ n \quad \text{(compensate scope)}$$
$$| \quad \text{exit} \quad \text{(terminate process)}$$

$$S ::= B \quad \text{(basic action)}$$
$$| \quad S ; S \quad \text{(sequence)}$$
$$| \quad S + S \quad \text{(nondeterministic choice)}$$
$$| \quad S \parallel S \quad \text{(parallel)}$$
$$| \quad [S] \quad \text{(terminate)}$$
$$| \quad \{ S \oplus S \oplus S : f_h : S : [ S ] \}_{m}^{n} \quad \text{(scope)}$$

$$P ::= \{ [ S \oplus S \oplus S : f_h ] \}_{n}^{m'} \quad \text{(process)}$$
$$| \quad \checkmark \quad \text{(completed process)}$$
$$| \quad \varnothing \quad \text{(faulted process)}$$
$$| \quad \triangledown \quad \text{(terminated process)}$$

where $f \in \mathbb{F}$, $n \in \mathbb{S}$, $m \in \{\Box, \Diamond, \Box, \Diamond\}$, $m' \in \{\Box, \Diamond\}$, $f_h \in \mathbb{F} \rightarrow S$. Further let $pl$ range over $\mathbb{P}$, $op$ over $\mathbb{O}$ and $me$ over $\mathbb{M}$.

Figure 41: Grammar of $BPEL_{fct}$

The core of $BPEL_{fct}$ is a modified and extended version of the Calculus of Communicating Systems (CCS) by R. Milner [17]. We excluded synchronization, recursion and restriction from the plain-vanilla CCS-calculus. The reason is, that our interests are to model the behavior of a single BPEL process in isolation, in order to ease the understanding of the execution of its scopes in normal, faulted, terminating, and compensating mode, i.e. the FCT handling behavior.

Therefore, the stop process ($0$), sequence ($S_1; S_2$), parallel composition ($S_1 \parallel$...
2) and nondeterministic choice \((S_1 + S_2)\) are standard like in CCS. Instead of allowing an arbitrary set of primitive actions to occur, we restrict ourselves to the communication primitives provided by BPEL. Accordingly, receive, reply, and asynchronous invoke are modeled by `rev`, `rep`, and `inv`, followed by the partner link `pl`, the operation `op`, and in case of receive and reply, the message exchange tag `me`. Synchronous invoke will later on be modeled by the sequence of the requesting (i.e. sending) part `inv` and receiving part `invr`. We are interested in all possible executions of a BPEL process. Thus, we will abstract away from data and time dependencies as well as synchronization in communication, i.e. we model communication as strictly non-blocking. We will use the internal action \(\tau\) in situations when certain activities of BPEL are not modeled explicitly, like assignments and data validation. Consequently, data dependent execution of branches will be modeled using nondeterminism. The throw primitive allows for the modeling of the explicit fault throwing activity of BPEL (throw and re-throw) as well as faults in communication (e.g. a possibly failing receive activity might be modeled as `rec pl op me + throw f`). Calling of installed compensation handlers can be achieved by using the two compensate activities. While \(\uparrow\) triggers full compensation in default compensation order, \(\uparrow n\) calls the specific compensation handler of the scope named \(n\). If a fault occurs in an execution branch \(P\), the branch has to be terminated. This behavior is modeled by putting that branch into a termination context \([P]\). The scope construct \(\{P \div C_A \div C_F : f_H : C : T\}_n\) consists of the following parts:

- **\(P\) (main activity):** This activity will be executed as long as no error occurs.
- **\(C_A\) (accumulated compensation context):** The compensation handlers of successfully completed scope activities inside the main activity \(P\) will be installed into \(C_A\).
- **\(C_F\) (fixed compensation context):** If a compensate or a compensate scope activity is executed, it will be replaced by the complete fixed compensation context, respectively by a part of it. The exact behavior of those two activities will be described later on in section 3.4.8. \(C_F\) is the place the accumulated compensation context of a scope will finally be copied to, in case of a fault or scope completion. In detail, that means that a fault handler will be able to retrieve compensation handlers, installed by the faulted scope up to the point of failure, from \(C_F\) and install its own compensation handlers (of inner scopes) into \(C_A\). A compensation handler might also retrieve installed compensation handlers of child scopes from \(C_F\) and install its own compensation handlers into \(C_A\). The latter enables **full compensation**, i.e. the compensation handler of an outer scope might trigger compensation of all inner scopes as well as of the inner scope’s inner scopes and so on.
- **\(f_H\) (fault handler):** The fault handler is represented by a total function, mapping fault names \((\in F)\) to handlers, i.e. usual structured activities \((S)\). If a fault \(f\) occurs inside \(P\) and the scope is in normal or compensating mode the handler \(f_b(f)\) will be executed.
- **\(C\) (compensation handler):** If the scope completes successfully, its compensation handler \(C\) will be installed into the accumulated compensation context of its root scope.

37
T (termination handler): If the current scope has to be terminated, e.g. because of a faulted parallel branch, its termination handler T will be executed in order to enable compensation of that scope.

Each scope is in a certain mode m of operation. Possible modes are:

• □ (normal mode): No fault occurred by now.
• □ (faulted mode): A fault occurred inside that scope. The fault handler of the scope is being executed and compensation is made possible.
• □ (compensating mode): The scope is part of a compensation context or is executed as the result of a compensate command.
• □ (terminating mode): The scope has been forced to terminate and executes its termination handler.

Finally, a BPELfct process $\{P \div C_A \div C_F : f_h\}^m_n$ consists of a main activity $P$, an accumulated compensation context $C_A$, a fixed compensation context $C_F$, a fault handler $f_h$, a name $n$, and a mode $m \in \{□, □\}$ analogous to a scope. A process might either complete successfully ($✓$), fail ($∅$), or terminate prematurely ($▽$).

3.3.2 Mapping from BPEL’ to BPELfct

In this section we will give a mapping $\llbracket \rrbracket : BPEL' \rightarrow BPEL_{fct}$ from our restricted lightweight syntax of BPEL to our calculus BPEL$_{fct}$.

Communication primitives: Basic communication primitives like receive, reply and asynchronous invoke (invokeA) will just be mapped onto their respective counterparts rec, rep and inv.

$\llbracket \text{receive}(pl, op, me)\rrbracket = \text{rec} pl op me$

$\llbracket \text{reply}(pl, op, me)\rrbracket = \text{rep} pl op me$

$\llbracket \text{invokeA}(pl, op)\rrbracket = \text{inv} pl op$

Synchronous invoke (invokeS) will be mapped to a sequence of the requesting part invs and receiving part invr, instead of a single construct. The reason is to allow arbitrary interleavings in between the two parts, in case of an enclosing parallel flow.

$\llbracket \text{invokeS}(pl, op)\rrbracket = \text{invs} pl op ; \text{inver} pl op$

In order to model failures in communication, mappings of single communication primitives to their BPEL$_{fct}$ counterparts can be modified to fail. For example a receive on a certain partner link $pl_1$, operation $op_1$, and message exchange tag $me_1$ might nondeterministically collide with incoming messages of the same partner link and operation, but different correlation sets, i.e. instances of the BPEL process. The BPEL specification states that in those situations the standard fault $bipel:ambiguousReceive$ shall be thrown. Subsequently, we could model that behavior the following way:

$\llbracket \text{receive}(pl_1, op_1, me_1)\rrbracket = \text{rec} pl_1 op_1 me_1 + \text{throw bipel:ambiguousReceive}$
Data and time dependent activities: We are abstracting away from data and time dependencies, consequently, we will model assignments and waits as internal actions just like the empty action.

\[
\begin{align*}
\llbracket \text{assign} \rrbracket &= \tau \\
\llbracket \text{wait} \rrbracket &= \tau \\
\llbracket \text{empty} \rrbracket &= \tau
\end{align*}
\]

Validation of variables having an incompatible value in respect to their type will throw a \textit{bpel:invalidVariables} fault. We will model that possible effect using nondeterminism, i.e. choice (+), between validating successfully (\(\tau; 0\)) and throwing the fault (\(\text{throw } \textit{bpel:invalidVariables}\)).

\[
\llbracket \text{validate} \rrbracket = \tau; 0 + \text{throw } \textit{bpel:invalidVariables}
\]

The data dependent if statement will be mapped to a nondeterministic choice between the cases it distinguishes.

\[
\llbracket \text{if}(S) \rrbracket = \sum_{s \in S} [s]
\]

\textit{Pick} waits until an external message (modeled by \textit{rec}) or a timeout (modeled by the internal action \(\tau\) because of time abstraction) occurs and then executes the respective (guarded) activity. All possible cases are composed using nondeterministic choice in order to model all possible execution paths.

\[
\llbracket \text{pick}(M,A) \rrbracket = \sum_{(pl,op,me,s) \in M \textit{ rec } pl \ op \ me} [s] + \sum_{a \in A} \tau; [a]
\]

Faults, Compensation, and early Termination: Explicit throwing of faults will be mapped straightforwardly to its \textit{BPEL}\textsubscript{fct} counterpart, just like \textit{compensate}, \textit{compensateScope} and forced termination \textit{exit}. The \textit{rethrow} activity is also mapped to \textit{throw} with the respective enclosing fault handler's handled fault passed as its argument.

\[
\begin{align*}
\llbracket \text{throw}(f) \rrbracket &= \text{throw } f \\
\llbracket \text{rethrow}(f) \rrbracket &= \text{throw } f \\
\llbracket \textit{compensate} \rrbracket &= \uparrow \\
\llbracket \textit{compensateScope}(n) \rrbracket &= \uparrow n \\
\llbracket \text{exit} \rrbracket &= \text{exit}
\end{align*}
\]

Sequence and Flow: The mappings of Sequence and Flow to \textit{BPEL}\textsubscript{fct} are straightforward.

\[
\begin{align*}
\llbracket \textit{sequence}(s_1, s_2) \rrbracket &= [s_1]; [s_2] \\
\llbracket \textit{flow}(s_1, s_2) \rrbracket &= [s_1] \parallel [s_2]
\end{align*}
\]
Scope and Process: In the mapping of scopes and processes, the respective inner activity and in the case of a scope the compensation- and termination-handler, are mapped recursively by $[\_]$. Accumulated and fixed compensation contexts are initialized with 0, i.e. no compensation handlers are installed in the beginning of execution. For the mapping of the fault handler, an auxiliary function $\eta$, defined as $\eta(F) : F \times \text{structures} \rightarrow F \times S$, $(f,s) \mapsto (f,[[s]])$, is used to map the respective individual $<\text{catch}>/<\text{catchAll}>$ nodes.

$$
[\text{scope}(n,A,F,C,T)] = ([A] \div 0 \div 0 : \eta(F) : [C] : [T])_{n}
$$
$$
[\text{process}(n,A,F)] = ([A] \div 0 \div 0 : \eta(F))_{n}
$$

3.4 The Calculus $BPEL_{fct}$ - Semantics

In the following section we will describe the operational semantics of our calculus $BPEL_{fct}$ in terms of Labeled Transition Systems (LTS). At first we need a conception of the configuration of a Business Process.

Definition 1 (Configuration). A configuration is a triple $\langle E, \alpha, \beta \rangle \in (P \cup S) \times S \times S$ with:

- $E$: the business process / the structured activity being executed
- $\alpha$: the current accumulated compensation context
- $\beta$: the current fixed compensation context

In the following sections we will give an operational semantics for Business Processes described by a restricted subset of BPEL, namely $BPEL'$, transformed to our calculus $BPEL_{fct}$. Therefore, we need a notion of well-formed initial process configurations, our semantics can be applied to.

Definition 2 (Well-formed Initial Processes Configuration). A well-formed initial process configuration $W$ is a configuration $([\text{process}(n,A,F)],0,0)$ where $\text{process}(n,A,F)$ is a valid $BPEL'$ expression, obtained by the translation of a valid BPEL process specification, restricted to the constructs we model. By valid BPEL process specification we mean a process description fully complying to the BPEL standard [18] (including successful static analysis described in appendix B[18]).

Definition 3 (Labeled Transition System). A Labeled Transition System (LTS) then is a tuple $(S,A,\rightarrow,s_0)$ with:

- $S$ (set of states): We will use a subset of the set of configurations for the set of states, i.e. $S \subseteq (P \cup S) \times S \times S$.
- $A$ (alphabet): The alphabet will include all observable action labels that occur in the LTS as well as the internal action $\tau$ and special forced termination signal "exit". Let $C$ denote the set of all observable actions, i.e.
\[ C = \{ \text{rec } pl \text{ op me,} \]
\[ \text{rep } pl \text{ op me,} \]
\[ \text{inv } pl \text{ op,} \]
\[ \text{invs } pl \text{ op,} \]
\[ \text{inver } pl \text{ op} \quad | \text{pl } \in \mathcal{P}, \text{op } \in \mathcal{O}, \text{me } \in \mathcal{M} \}, \]

then \( \mathcal{A} \subseteq C \cup \{ \tau, \text{exit} \} \).

• \( \rightarrow \in S \times \mathcal{A} \times S \) (transition relation): The transition relation is the smallest relation containing exactly those transitions that respect the rules described in the following sections.

• \( s_0 \in W \subset S \) (initial state): The initial state is the start configuration of a process, i.e. \( s_0 \) will be of the form \( \langle \mathcal{P}, 0, 0 \rangle \). Thus, possible end states are \( \langle \mathcal{X}, 0, 0 \rangle \) (successful completion), \( \langle \emptyset, 0, 0 \rangle \) (failure) and \( \langle \varnothing, 0, 0 \rangle \) (early termination).

### 3.4.1 Basic Structural Congruence Rules

Following the tradition of Milner, we will introduce a set of basic structural congruence rules for our calculus in figure 42. In \( BPEL_{fad} \), parallel composition is neither associative nor commutative in order to preserve the initial term structure such that the parallel branches of the accumulated compensation context can be extended individually. Rule CB1 can 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. For example, \( 0; \alpha \equiv 0 \parallel 0; \alpha \). Therefore, the accumulated compensation context of a configuration must always be congruent to a context \( 0; \alpha' \) for some \( \alpha' \). Indeed, this is an invariant throughout the execution of well-formed initial configurations and can easily be shown using induction. CB2 allows for the reduction of enclosed separating 0s. In order to reduce the number of semantic rules for choice, the usual properties are stated in CB3-CB5. The remaining structural congruence rules, covering termination, will be presented later on.

[CB1] \( 0 \equiv 0 \parallel 0 \)

[CB2] \( S; 0 \equiv S \)

\((S,+,0)\) is an Abelian Monoid:

[CB3] \( S + (S' + S'') \equiv (S + S') + S'' \) (Associativity)

[CB4] \( S + S' \equiv S' + S \) (Commutativity)

[CB5] \( S + 0 \equiv S \) (Neutral Element)

Figure 42: Basic Congruence Rules

In the following section, the semantics of \( BPEL_{fad} \) processes will be defined up to structural congruence described above and in section 3.4.7.
3.4.2 Basic Activities

The basic activities consist of all communication primitives (receive, reply, asynchronous invoke, sending, and receiving part of synchronous invoke), internal action, and throw. All those activities can be observed. In the case of an internal action the fact that an internal action occurred is visible to the observer. It is important to note that none of these activities changes the accumulated compensation context. This models an interesting fact of BPEL: there is no automated construction of a compensation handler. That means, if a certain action $a$ may change the global state in a way it should be considered in case of compensation, a compensating action $a^{-1}$ for $a$ has to be found manually. Another consequence is that this compensating action $a^{-1}$ must reside in a compensation handler (usually of the scope containing action $a$ in its main activity), which is the only activity that is being installed into the accumulated and fixed compensation context (see rule [SCOPE END]). This behavior models another property of BPEL: only predefined parts of a BPEL process, i.e. the scope’s compensation handlers, will be installed and are available as compensation activities.

\[
\begin{align*}
\text{REC} & \quad \langle \text{rec pl op me}, \alpha, \beta \rangle \xrightarrow{\text{rec pl op me}} \langle 0, \alpha, \beta \rangle \\
\text{REP} & \quad \langle \text{rep pl op me}, \alpha, \beta \rangle \xrightarrow{\text{rep pl op me}} \langle 0, \alpha, \beta \rangle \\
\text{INV} & \quad \langle \text{inv pl op}, \alpha, \beta \rangle \xrightarrow{\text{inv pl op}} \langle 0, \alpha, \beta \rangle \\
\text{INVS} & \quad \langle \text{invsp pl op}, \alpha, \beta \rangle \xrightarrow{\text{invsp pl op}} \langle 0, \alpha, \beta \rangle \\
\text{INVR} & \quad \langle \text{invrp pl op}, \alpha, \beta \rangle \xrightarrow{\text{invrp pl op}} \langle 0, \alpha, \beta \rangle \\
\text{THROW} & \quad \langle \text{throw f}, \alpha, \beta \rangle \xrightarrow{\text{throw f}} \langle 0, \alpha, \beta \rangle \\
\text{INT} & \quad \langle \tau, \alpha, \beta \rangle \xrightarrow{\tau} \langle 0, \alpha, \beta \rangle
\end{align*}
\]

3.4.3 Structured Activities

The underlying rules for the structured activities sequence, parallel composition, and nondeterministic choice, are standard like in CCS. One modification concerns the process configuration used. Basically, both the accumulated and the fixed compensation context are passed to the execution of one of the composed activities and may be changed by it. The modified contexts are then passed back, stored intermediate at a lower execution level, and, consequently, are available to further executions. For the structured activity inference rules, let $x$ denote any observation including faults and the internal action, i.e. $x \in C \cup \{\tau\} \cup \{\text{throw f} \mid f \in F\}$. Please note that in case of the structured activities, faults are just passed through.

Sequence: The sequential operator is described by two rules. Usually the case of rule [SEQT] would be handled by a structural congruence rule $0 ; A \equiv A$. But a leading 0 will be used for marking the end of the first part of a sequence.
The reason is to allow a possibly following parallel composition to build up its own part of the accumulated compensation context like described below.

\[
\text{SEQ: } \langle S_1, \alpha, \beta \rangle \xrightarrow{\xi} \langle S'_1, \alpha', \beta' \rangle \\
\langle S_1 \parallel S_2, \alpha, \beta \rangle \xrightarrow{\xi} \langle S'_1 \parallel S_2, \alpha', \beta' \rangle
\]

\[
\text{SEQT: } \langle S, 0 ; \alpha, \beta \rangle \xrightarrow{\xi} \langle S', \alpha', \beta' \rangle \\
\langle 0 ; S, \alpha, \beta \rangle \xrightarrow{\xi} \langle 0 ; S', \alpha', \beta' \rangle
\]

**Parallel:** Also the two rules for parallel composition could be expressed by a single rule and the usual structural congruence rules ensuring Abelian Monoid properties for \( (P, \parallel, 0) \). But associativity and commutativity of \( \parallel \) would destroy the structure of the term. In our calculus this structure has to be preserved in order to be able to access the right subprocesses when extending the accumulated compensation context. When executing one branch \( (S_1 \parallel S_2) \) of a parallel composition, the corresponding accumulated compensation context \( (\alpha_1 \parallel \alpha_2) \) is extracted from the parallel composition of all accumulated compensation contexts of the two branches \( (\alpha_1 \parallel \alpha_2) \). Like in the sequential case, the context may be changed by the execution of the selected branch and will be written back to its position in the top context.

Whenever a new parallel flow may start, the accumulated compensation context \( \alpha \) will start with a leading 0, like stated before, i.e. formally \( \alpha \equiv 0 ; \alpha' \) will hold for some \( \alpha' \). Using structural congruence rule [CB1] this 0 can always be expanded to the structure of the parallel composition, which allows the application of the rules for parallel composition.

For example, assume a parallel activity \( X \equiv (x_1 \parallel x_2) \parallel x_3 \) being executed after the sequential execution of another activity (having installed compensation handlers \( \alpha' \) into \( \alpha \)). We may assume that \( \alpha = 0 ; \alpha' \), i.e. we are executing

\[\langle (x_1 \parallel x_2) \parallel x_3, 0 ; \alpha', \beta \rangle\]

and at first glance no further execution seems to be allowed. But repeated application of rule [CB1] allows expansion of the leading 0 of the accumulated compensation context, i.e. \( 0 \equiv 0 \parallel 0 \equiv (0 \parallel 0) \parallel 0 \). Consequently we may now execute our initial configuration as

\[\langle (x_1 \parallel x_2) \parallel x_3, (0 \parallel 0) \parallel 0 ; \alpha', \beta \rangle\]

allowing the application of rules [PARL] and [PARR].

\[
\text{PARL: } \langle S_1, \alpha_1, \beta \rangle \xrightarrow{\xi} \langle S'_1, \alpha'_1, \beta' \rangle \\
\langle S_1 \parallel S_2, (\alpha_1 \parallel \alpha_2) ; \alpha, \beta \rangle \xrightarrow{\xi} \langle S'_1 \parallel S_2, (\alpha'_1 \parallel \alpha_2) ; \alpha, \beta' \rangle
\]

\[
\text{PARR: } \langle S_2, \alpha_2, \beta \rangle \xrightarrow{\xi} \langle S'_2, \alpha'_2, \beta' \rangle \\
\langle S_1 \parallel S_2, (\alpha_1 \parallel \alpha_2) ; \alpha, \beta \rangle \xrightarrow{\xi} \langle S'_1 \parallel S_2, (\alpha_1 \parallel \alpha_2') ; \alpha, \beta' \rangle
\]

**Choice:** The rule for choice and its structural congruence rules (CB3 to CB5) are straightforward.
3.4.4 Scope Activity

In order to model the different behaviors of BPEL scopes in respect to prior execution and their semantic context, we allow scopes to have a mode tag. Every scope of the initial process configuration starts in normal mode $\Box$. If an error occurs, its mode changes to faulted mode $\ Orleans$. A scope in compensating mode $\circ$ denotes an installed or executing compensation handler. A scope in normal mode that gets terminated will be in terminating mode $\triangledown$. Let $y$ denote any observation including the internal action, i.e. $y \in \mathcal{C} \cup \{\tau\}$.

**Normal Behavior**

A scope in normal mode may execute its main activity $P$ as long as no fault occurs. In this case, successfully terminated inner scopes will install their compensation handler into the scope’s accumulated compensation context $C_A$, i.e. changes in $C_A$ will be stored back. Usually, no compensate activity is used inside normal mode scopes. The only exception are compensation handler whose inner activity consists of nested normal mode scopes. In that case, the fixed compensation context has to be passed through from the outer most compensating scope to the currently executing normal mode scope. This passing through is made possible by the third component of the configuration. Consequently, normal mode scopes receive the fixed compensation context in $\beta$ and forward it into the next level of execution.

\[
\text{SCOPE} \quad \langle S, C_A, \beta \rangle \xrightarrow{y} \langle S', C_A', \beta' \rangle
\]

\[
\langle \{S \div C_A \div 0 : f_h : C : T\}_n^{\triangledown}, \alpha, \beta \rangle \xrightarrow{y} \langle \{S' \div C_A' \div 0 : f_h : C : T\}_n^{\triangledown}, \alpha, \beta' \rangle
\]

Scopes in compensating, fault handling, or terminating mode behave roughly the same. The difference to normal mode scopes is that they got their own fixed compensation context when their mode changed or they were installed (in case of compensation handlers). Accordingly, they’ll pass their own fixed compensation context $C_F$ through when executing their main activity.

\[
\text{SCOPE FCT} \quad \langle S, C_A, C_F \rangle \xrightarrow{m} \langle S', C_A', C_F' \rangle
\]

\[
\langle \{S \div C_A \div C_F : \emptyset : 0 : 0\}_n^m, \alpha, \beta \rangle \xrightarrow{m} \langle \{S' \div C_A' \div C_F' : \emptyset : 0 : 0\}_n^m, \alpha, \beta' \rangle
\]

**Scope Completion**

Like already mentioned, scopes in normal mode will install their compensation handler $C$ into the current accumulated compensation context $\alpha$ (i.e. the enclosing scope’s accumulated compensation context $C_A$ on the configuration level), when successfully terminated, i.e. their main activity $P \equiv 0$. The handler gets
enclosed by a compensating scope with the handler itself as the main activity and the accumulated compensation context of the original scope $C_A$ stored as the fixed compensation context of the installed scope. This allows the handler $C$ to call compensation handlers of child scopes of scope $n$ which were accumulated inside $C_A$ of the original scope and install their own compensation handlers into the accumulated compensation handler of the compensating mode scope. No fault handler, i.e. $f_H = \emptyset$, is required, because a special inference rule [SCOPE FAULT C] handles faults inside compensating mode scopes directly. Also no compensation and termination handlers are required, since compensating mode scopes don’t install further compensation handlers upon successful completion. Termination of those scopes is handled by a special congruence rule [CT16], described later on.

SCOPE END

$$\langle \{0 \div C_A \div C_F : f_h : C : T\}_{n}^{\sigma}, \alpha, \beta \rangle \xrightarrow{\tau} \langle 0, \{C \div 0 \div C_A : \emptyset : 0 : 0\}_{n}^{\sigma} : \alpha, \beta \rangle$$

Upon successful termination of faulted or terminating scopes, no compensation handler will be installed like stated in the BPEL specification. The specification also demands that temporarily installed compensation handlers of completed compensation handler shall be deleted. This behavior is modeled by our calculus by also not allowing compensation handlers to install any further compensation handlers upon completion, i.e. discarding them. That behavior shows an interesting property of compensation in BPEL: committed (i.e. successfully completed) compensations can not be undone.

SCOPE END FCT

$$m \in \{\square, \lleq, \lleq\}$$

$$\langle \{0 \div C_A \div 0 : 0\}_{n}^{m} : \alpha, \beta \rangle \xrightarrow{\tau} \langle 0, \alpha, \beta \rangle$$

Scope Fault Behavior

In BPEL, abnormal behavior is signaled by faults. Faults are raised automatically by activities, e.g. type validation or receive, but can also be thrown explicitly via the throw activity. Inside fault handlers the handled fault can be re-thrown to the fault handler of the parent scope or process. Recall that in our calculus, all those cases are modeled by a single throw primitive.

If a fault $f$ occurs inside a normal mode scope, its main activity $P'$ (after having signaled the fault) gets terminated ($[P']$) and the scope’s mode changes to faulted mode $\lleq$. When termination ends, the appropriate fault handler $f_H(f)$ gets executed, which may access the compensation handlers of the original scope (before the fault) from the accumulated compensation context $C_A$ having been installed into the fixed compensation $C_F$ of the now faulted mode scope.

SCOPE FAULT

$$\langle S, C_A, \beta \rangle \xrightarrow{\text{throw } f} \langle S', C_A', \beta \rangle$$

$$\langle \{S \div C_A \div 0 : f_h : C : T\}_{n}^{\square}, \alpha, \beta \rangle \xrightarrow{\tau} \langle \{[S'] : f_h(f) \div 0 \div C_A' : \emptyset : 0 : 0\}_{n}^{\square}, \alpha, \beta \rangle$$
Scopes in faulted mode, i.e. scopes executing their fault handler, will terminate their fault handling behavior in case of a further fault. This fault will finally be re-thrown, i.e. forwarded, to the parent scope.

SCOPE FAULT F
\[
\begin{align*}
(S, C_A, C_F) \xrightarrow{\text{throw}_f} (S', C'_A, C_F) \\
\langle\{S \div C_A \div C_F : \emptyset : 0 : 0\} \overrightarrow{\infty}_n, \alpha, \beta \rangle \xrightarrow{\tau} \langle [S'] ; \text{throw}_f, \alpha, \beta \rangle
\end{align*}
\]

A compensating mode scope that faults ("internal faults") will at first undo its partial work according to the fault handler of a possibly enclosed normal mode root scope. In case of a nested normal mode scope hierarchy, the fault will be handled at first by the fault handler of the scope that caused the fault. The fault handler of that scope should then re-throw that fault. This behavior is stated by the BPEL specification and must be enforced statically before applying our calculus. Thus, the fault should be propagated upwards in the scope hierarchy, until the enclosing compensating mode scope is reached. When this happens or in case of no or multiple explicitly enclosed root scopes, the (remaining) partial work will be compensated in default order, which is now enforced by our calculus (cf. rule [SCOPE FAULT C]). In any case, the error will be propagated to the caller of the compensation of the compensation handler by re-throwing the fault in (now) faulted mode. This will result in terminating no remaining activity ([CT1] \(0 \equiv 0\)), according to rule [SCOPE FAULT F], and re-throwing the fault up to the calling scope (\(\text{throw}_f\)).

SCOPE FAULT C
\[
\begin{align*}
(S, C_A, C_F) \xrightarrow{\text{throw}_f} (S', C'_A, C_F) \\
\langle\{S \div C_A \div C_F : \emptyset : 0 : 0\} \overrightarrow{\infty}_n, \alpha, \beta \rangle \xrightarrow{\tau} \langle [↑ ; \text{throw}_f \div 0 \div C'_A : \emptyset : 0 : 0] \overrightarrow{\infty}_n, \alpha, \beta \rangle
\end{align*}
\]

Faults inside terminating scopes will immediately terminate the execution of the termination handler.

SCOPE FAULT T
\[
\begin{align*}
(S, C_A, C_F) \xrightarrow{\text{throw}_f} (S', C'_A, C_F) \\
\langle\{S \div C_A \div C_F : \emptyset : 0 : 0\} \overrightarrow{\infty}_n, \alpha, \beta \rangle \xrightarrow{\tau} \langle [S'] , \alpha, \beta \rangle
\end{align*}
\]

3.4.5 Process

The operation of processes is analogous to the execution of scopes. The main difference is that processes have no compensating or terminating mode. Instead processes end up in a success state \(\checkmark\) (successful completion), failure state \(\emptyset\) (fault while fault handling), or early terminated state \(\nabla\) (an exit signal occurred). Again, let \(y\) denote any observation including the internal action, i.e. \(y \in C \cup \{\tau\}\).
PROCESS
\[(S, C_A, 0) \xrightarrow{y} (S', C_A, 0)\]
\[\langle\{\|S\div C_A \div 0: f_h\}_{n}^{\mathbb{N}}, 0, 0\rangle \xrightarrow{y} \langle\{\|S'\div C'_A \div 0: f_h\}_{n}^{\mathbb{N}}, 0, 0\rangle\]

PROCESS F
\[(S, C_A, C_F) \xrightarrow{y} (S', C_A', C_F')\]
\[\langle\{\|S\div C_A \div C_F : \emptyset\}_{n}^{\mathbb{N}}, 0, 0\rangle \xrightarrow{y} \langle\{\|S'\div C'_A \div C'_F : \emptyset\}_{n}^{\mathbb{N}}, 0, 0\rangle\]

PROCESS FAULT
\[(S, C_A, 0) \xrightarrow{\text{throw} f} (S', C_A', 0)\]
\[\langle\{\|S\div C_A \div 0: f_h\}_{n}^{\mathbb{N}}, 0, 0\rangle \xrightarrow{\text{throw} f} \langle\{\|S'\div C'_A : \emptyset\}_{n}^{\mathbb{N}}, 0, 0\rangle\]

PROCESS FAULT F
\[(S, C_A, C_F) \xrightarrow{\text{throw} f} (S', C_A', C_F')\]
\[\langle\{\|S\div C_A \div C_F : \emptyset\}_{n}^{\mathbb{N}}, 0, 0\rangle \xrightarrow{\text{throw} f} \langle\emptyset, 0, 0\rangle\]

PROCESS END
\[m \in \{\square, \square\}\]
\[\langle\{0 \div C_A \div C_F : f_h\}_{n}^{m}, 0, 0\rangle \xrightarrow{\text{throw} f} \langle\checkmark, 0, 0\rangle\]

3.4.6 Exit

For the exit statement we will use a special exit signal \textit{exit}, used to enforce immediate termination. The exit activity itself simply sends out the signal (rule \texttt{[EXIT]}) that will be passed through by the structured activities (rules \texttt{[EXIT SEQ], [EXIT SEQ T], ...}) until it is intercepted by the process. The process will terminate immediately and end up in terminated state \(\checkmark\) (see rules \texttt{[EXIT PROCESS] and [EXIT PROCESS F]}).
EXIT
\[
\langle \text{exit}, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT SEQ
\[
\langle S_1, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle \quad \langle S_1 ; S_2, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT SEQT
\[
\langle S, 0 ; \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, 0 ; \alpha, \beta \rangle \quad \langle 0 ; S, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, 0 ; \alpha, \beta \rangle
\]

EXIT PARL
\[
\langle S_1, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle \quad \langle S_1 \parallel S_2, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT PARR
\[
\langle S_2, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle \quad \langle S_1 \parallel S_2, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT CHOICE
\[
\langle \text{exit}, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle \quad \langle S_1 + S_2, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT SCOPE
\[
\langle S, C_A, \beta \rangle \xrightarrow{\text{exit}} \langle 0, C_A, \beta \rangle \quad \langle \{\| S \div C_A \div 0 \} : T \} \div n, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT SCOPE FCT
\[
\langle S, C_A, C_F \rangle \xrightarrow{\text{exit}} \langle 0, C_A, C_F \rangle \quad m \in \{\|, \|, \|\} \quad \langle \{\| S \div C_A \div C_F : f_h \} \div m, \alpha, \beta \rangle \xrightarrow{\text{exit}} \langle 0, \alpha, \beta \rangle
\]

EXIT PROCESS
\[
\langle S, C_A, 0 \rangle \xrightarrow{\text{exit}} \langle 0, C_A, 0 \rangle \quad \langle \| S \div C_A \div 0 \} \div n, 0, 0 \rangle \xrightarrow{\text{exit}} \langle \forall, 0, 0 \rangle \quad \langle S, C_A, 0 \rangle \xrightarrow{\text{exit}} \langle 0, C_A, 0 \rangle \quad \langle \| S \div C_A \div C_F : 0 \} \div n, 0, 0 \rangle \xrightarrow{\text{exit}} \langle \forall, 0, 0 \rangle
\]

3.4.7 Termination
If inside a parallel composition of several activities a parallel branch fails, the remaining branches have to be terminated, according to the BPEL specification. In our calculus the scope or processes that intercepts that fault, encloses its failed inner activity by a termination context \([S]\) (cf. e.g. rules \([\text{SCOPE FAULT}]\) and \([\text{PROCESS FAULT}]\)). The semantics of that termination construct is fully described by congruence rules \([\text{CT1}]\) to \([\text{CT18}]\) (cf. figure 43). Termination of basic activities is instantaneous (\([B] \equiv 0\), cf. rules \([\text{CT1}]\) to \([\text{CT10}]\)). If a throw activity is terminated, the BPEL specification states that the activity MAY be allowed to complete. We model that behavior via the nondeterministic
choice between completing that action (\textit{throw f}) and terminating it (\(\tau : 0\)), cf. rule [CT11]. When terminating parallel branches, each branch is terminated in parallel [CT14]. In a sequence the front most activity will be terminated [CT13]. Nondeterministic choice will be terminated immediately [CT12].

Termination of scopes depends on their mode. A scope in normal mode is terminated by changing its mode to \(\oplus\), terminating its inner activity and executing its termination handler [CT15]. This gives the scope the chance to compensate, i.e. calling installed compensation handler from within the termination handler \(T\). The accumulated compensation context \(C_A\) is therefore being installed as the fixed context \(C_F\). The fault handler of that now terminating mode scope is set to the empty set, because every fault inside a terminating scope will result in immediate termination via inference rule [SCOPE FAULT T]. Also the compensation handler \(C\) and inner termination handler \(T\) are replaced by 0, because there is no chance they will ever be installed respectively executed. A compensating mode scope is terminated by terminating its inner activity [CT16]. Termination does not affect faulted or terminating scopes ([CT17] and [CT18]). The reason is that terminating scopes are already terminating and faulted scopes try to handle a fault and undo the execution of the respective scope.

\[
\begin{align*}
[\text{CT1}] & \quad [0] \equiv 0 \\
[\text{CT2}] & \quad [\tau] \equiv 0 \\
[\text{CT3}] & \quad [recpl op me] \equiv 0 \\
[\text{CT4}] & \quad [reqpl op me] \equiv 0 \\
[\text{CT5}] & \quad [invpl op] \equiv 0 \\
[\text{CT6}] & \quad [inveus pl op] \equiv 0 \\
[\text{CT7}] & \quad [inveur pl op] \equiv 0 \\
[\text{CT8}] & \quad [!] \equiv 0 \\
[\text{CT9}] & \quad [↑ n] \equiv 0 \\
[\text{CT10}] & \quad [exit] \equiv 0 \\
[\text{CT11}] & \quad [\text{throw f}] \equiv \text{throw f} + \tau : 0 \\
[\text{CT12}] & \quad [S + S'] \equiv 0 \\
[\text{CT13}] & \quad [S : S'] \equiv [S] \\
[\text{CT14}] & \quad [S \parallel S'] \equiv [S] \parallel [S'] \\
[\text{CT15}] & \quad [(S \div C_A : 0 : f_h : C : T)_{\oplus}] \equiv [(S : T \div 0 \div C_A : \emptyset : 0 : 0)_{\oplus}] \\
[\text{CT16}] & \quad [(S \div C_A \div C_F : \emptyset : 0 : 0)_{\oplus}] \equiv [S] \\
[\text{CT17}] & \quad [(S \div C_A \div C_F : \emptyset : 0 : 0)_{\oplus}] \equiv \{S \div C_A \div C_F : \emptyset : 0 : 0\}_{\oplus} \\
[\text{CT18}] & \quad [(S \div C_A \div C_F : \emptyset : 0 : 0)_{\oplus}] \equiv \{S \div C_A \div C_F : \emptyset : 0 : 0\}_{\oplus}
\end{align*}
\]

Figure 43: Structural Congruence Rules for Termination

\subsection{3.4.8 Compensation}

There are two types of compensation commands in our calculus. The full compensate activity \(↑\) calls the complete chain of child compensation handlers in standard compensation order, i.e. reverse execution order in case of sequential execution (parallel branches stay parallel). The compensate scope activity \(↑ n\) calls the single compensation handler of a specific direct child scope \(n\).

In case of a compensate activity \(↑ n\), the activity replaces itself syntactically
by the compensation handler of $n$ looked up in the fixed compensation context $C_F$. The called compensation handler will be deleted from $C_F$ in order to prevent multiple calls. The full compensate action $\uparrow$ replaces itself by the complete fixed compensation context $C_F$.

\[
\begin{align*}
\text{COMP} & \quad \text{COMP SCOPE} \\
\langle \uparrow, \alpha, \beta \rangle & \to \langle \beta, \alpha, 0 \rangle \\
\langle \uparrow n, \alpha, \beta \rangle & \to \langle \text{lookup}(n, \beta), \alpha, \text{rest}(n, \beta) \rangle
\end{align*}
\]

\[
\text{lookup}(0, n) ::= 0 \\
\text{lookup}(\{S \div 0 \div C_F : \emptyset : 0 : 0\}_n, n) ::= \{S \div 0 \div C_F : \emptyset : 0 : 0\}_n \\
\text{lookup}(\{S \div 0 \div C_F : \emptyset : 0 : 0\}_m, n) ::= 0 \quad n \neq m \\
\text{lookup}(S_1 ; S_2, n) ::= \text{lookup}(S_1, n) ; \text{lookup}(S_2, n) \\
\text{lookup}(S_1 \parallel S_2, n) ::= \text{lookup}(S_1, n) \parallel \text{lookup}(S_2, n)
\]

\[
\begin{align*}
\text{rest}(0, n) ::= 0 \\
\text{rest}(\{S \div 0 \div C_F : \emptyset : 0 : 0\}_n, n) ::= 0 \\
\text{rest}(\{S \div 0 \div C_F : \emptyset : 0 : 0\}_m, n) ::= \{S \div 0 \div C_F : \emptyset : 0 : 0\}_m \quad n \neq m \\
\text{rest}(S_1 ; S_2, n) ::= \text{rest}(S_1, n) ; \text{rest}(S_2, n) \\
\text{rest}(S_1 \parallel S_2, n) ::= \text{rest}(S_1, n) \parallel \text{rest}(S_2, n)
\end{align*}
\]
4 Issues concerning Compensation in WS-BPEL 2.0

Concerning compensation there are several advanced techniques and issues in BPEL our semantics have to comply. Those are default compensation order and all or nothing semantics of compensation handlers.

4.1 Compensation in BPEL$_{fct}$

But at first we will illustrate how the general mechanism of compensation works in our calculus. That means, we will show how the interplay of compensation handlers, compensation contexts, and compensation mode scopes provides the ability to compensate, i.e. to undo the effects of execution up to the point of failure. Each scope and process is equipped with two contexts, the accumulated compensation context $C_A$ and the fixed compensation context $C_F$ playing different roles, depending on the mode of the scope / process. Consider a scope $\{ S \div C_A \div C_F : f_h : C : T \}^m$ with mode $m =$

☐: The scope is in normal mode and might install compensation handlers of completed inner scopes of its inner activity $S$ into $C_A$, cf. rule [SCOPE] in combination with rule [SCOPE END]. $C_F = 0$ always holds, because the only way, normal mode scopes might execute compensation calls, is if they are placed inside compensation handlers, i.e. compensating mode ☐ scopes. But even then, they will only be able to execute those compensation handlers, the enclosing handler might call. Consequently, the normal mode scope will have to get those handler passed trough, which is done using the $\beta$ context of the configuration, cf. rule [SCOPE]. Allowing compensation handlers to have inner normal mode scopes is vital for enabling all or nothing semantics of compensation handlers, as will be seen later on in this section.

☐: The scope is in faulted mode and might draw compensation handlers that have been installed up to the point of failure from $C_F$, cf. rule [SCOPE FCT]. Therefore, upon mode change from normal to faulted mode, the normal mode’s accumulated compensation context is copied to the fixed compensation context $C_F$ of the now faulted mode scope, cf. rule [SCOPE FAULT]. Context $C_A$ will still accumulate compensation handlers that might be installed because of successful completion of the fault handler’s inner scopes, but they will never be accessible because upon completion or failure of the fault handler they will be discarded, cf. rule [SCOPE END FCT].

☐: The scope is in compensating mode, i.e. the scope is a compensation handler in execution. If a scope completes successfully, its compensation handler will be installed as a compensating scope into its parent scope’s or process’ accumulated compensation context. That compensating scope’s fixed compensation handler $C_F$ will be initialized with the accumulated compensation context of the original scope, containing the compensation handlers of its completed inner scopes, cf. rule [SCOPE END]. That way the compensation handler might call further compensation handlers of the original scope’s children. 

51
The scope is in terminating mode, i.e. it has been forced to terminate but has been given the chance to compensate its inner activity. Therefore, upon mode switch from normal mode to terminating mode, the normal mode’s accumulated compensation context has been installed as the fixed compensation context of the terminating mode, such that the termination handler might access the compensation handlers that have been installed.

Processes behave analogously to scopes. Now we will illustrate the mechanism of compensation in \( BPEL_{fct} \) using an example.

**Example 5.** Consider a scope \( S \) with a sequence of nested scopes \( s_1 \) and \( s_2 \) as its inner activity. The inner scopes will just invoke partner links \( pl_1 \) and \( pl_2 \) using operations \( op_1 \) and \( op_2 \), respectively. Further assume that the operation of the scope is optional but it shall either succeed or no effects of execution shall be visible in case of a fault. In order to allow for possible faults in invocations, we will model these as the nondeterministic choice of the invocation itself and a fault throwing construct – like mentioned in section 3.3.2. The respective fault handlers of the inner scopes will re-throw the fault up to scope \( S \)’s fault handler which will compensate its inner activities (remember: no effects shall be visible).

The compensation handlers of the inner scopes shall try to undo the effect of the inner activity, i.e. of the invocation. A plausible compensation would be a reply on the respective partner link, using some cancel-operation, e.g. \( co_1 \) respectively \( co_2 \). This results in the following model:

\[
f_r \in F \rightarrow S, f \mapsto \text{throw } f
\]

\[
f_c \in F \rightarrow S, f \mapsto \uparrow
\]

\[
s_1 = \{\text{inv } pl_1 \circ op_1 + \text{throw } f_1 \div 0 \div 0 \div f_c : \text{rep } pl_1 \circ co_1 \circ std : 0\}\]

\[
s_2 = \{\text{inv } pl_2 \circ op_2 + \text{throw } f_2 \div 0 \div 0 \div f_r : \text{rep } pl_2 \circ co_2 \circ std : 0\}\]

\[
S = \{s_1 ; s_2 \div 0 \div 0 \div f_c : \uparrow\}\]

which we will execute using initial configuration \( C \) defined as \( C = (S, 0, \beta) \). A possible execution is (simplified using structural congruence rules when applicable):

1. \( (S, 0, \beta) \)
2. \( \text{inv } pl_1 \circ op_1 \langle\{0 \div 0 \div 0 \div f_r : \text{rep } pl_1 \circ co_1 \circ std : 0\}\rangle_{S_1} \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0, \beta \)
3. \( \tau \langle\{0 \div s_2 \div \text{rep } pl_1 \circ co_1 \circ std \div 0 \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0, \beta \}
4. \( \tau \langle\{f_r(f_2) \div 0 \div 0 \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0 \div \text{rep } pl_1 \circ co_1 \circ std \div 0 \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0, \beta \}
5. \( \tau \langle\{\text{throw } f_2 \div 0 \div \text{rep } pl_1 \circ co_1 \circ std \div 0 \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0, \beta \}
6. \( \tau \langle\{f_c(f_2) \div 0 \div 0 \div \text{rep } pl_1 \circ co_1 \circ std \div 0 \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0, \beta \}
7. \( \tau \langle\{0 \div \text{rep } pl_1 \circ co_1 \circ std \div 0 \div 0 \div 0 \div f_c : \uparrow\rangle_{S_2} \div 0, \beta \)
At first, scope \( s_1 \) inside scope \( S \) gets executed \((2)\), resulting in the observable action \( \text{invpl}_1 \text{op}_1 \). Next, the compensation handler of \( s_1 \) is installed into scope \( S \)’s accumulated compensation context \((3)\). Now scope \( s_2 \) executes the throwing of a fault resulting in the activation of scope \( s_2 \)’s fault handler \( f_r \) \((4)\) and a mode switch to faulted mode. The handler \( f_r \) will simply re-throw \((5)\) the fault up to scope \( S \) which is the intended behavior of a fault in faulted mode. That re-thrown fault will then be intercepted by scope \( S \)’s fault handler \( f_c \) resulting in a mode switch of scope \( S \) to faulted mode. During that switch also the accumulated compensation context (the installed compensation handler of scope \( s_1 \)) will be copied to the fixed compensation context \((6)\). The fault handler \( f_c \) will compensate using the \( \uparrow \) command which will replace itself syntactically by the fixed compensation context \((7)\), resulting in the execution of scope \( s_1 \)’s compensation handler \((8)\) \((\text{reppl}_1 \text{co}_1 \text{std})\). In \((9)\) and \((10)\) the compensation handler and the fault handler finish.

Obviously, we were executing a branch in which the second invocation failed, resulting in the compensation of the whole inner activity of scope \( S \) up to that point of failure. In detail, the behavior we can observe is the first invocation \( \text{invpl}_1 \text{op}_1 \) followed by some internal activities \( \tau \) and the compensation of the invocation \( \text{reppl}_1 \text{co}_1 \text{std} \) (followed again by a sequence of \( \tau s \)). That is, any effects of execution have been reversed.

### 4.2 Default Compensation Order

Like mentioned above, fault, compensation, and termination handlers are fully programmable in BPEL. All those handlers try to reverse the partial or complete work done by the scope or process they are attached to using compensation actions. Single compensation handlers of child scopes \( n \) can be called directly via the compensate scope command \( \uparrow n \). In the case of complex activities, like sequences or flows of scopes, compensation should be done in reverse execution order. A convenient construct that allows for automated compensation of those structured activities in default compensation order is the compensate command \( \uparrow \). We will now show how that mechanism works in our semantics, again, using an example.

**Example 6.** Consider for illustration purposes a scope \( S \) with a sequence of child scopes \( s_1 \) to \( s_n \) as its main activity, i.e.:

\[
S = \{s_1 : \ldots : s_n : 0 \div 0 : f_h : C : T\}^\uparrow_S
\]

\[
s_i = \{P_i : 0 \div 0 : f_i^h : C_i : T_i\}^\uparrow_{s_i}
\]

Now let compensation handler \( C \) be a full compensate command, i.e. \( C = \uparrow \), which should call all compensation handlers \( c_i \) of scope \( S \)’s child scopes in reverse execution order, i.e. \( c_n \) to \( c_1 \). Assume scope \( S \) will be executed in some configuration \((S, \alpha, \beta)\) and that no fault occurs inside \( S \). Consequently, the inner activity of scope \( S \), i.e. the sequence \( s_1; \ldots; s_n \), will be executed in configuration
\[(s_1; \ldots; s_n, 0, \beta)\] according to \([SCOPE]\). Thus, because of \([SEQ]\), the inner scope \(s_1\) will be executed in \(\langle s_1, 0, \beta \rangle\), i.e. \(\langle \{P_1 \div 0 \div 0 : f_{1_k}^i : C_1 : T_1 \} \sqcup s_1, 0, \beta \rangle\). Using \([SCOPE]\) again, scope \(s_1\)'s inner action \(P_1\) will be executed in \(\langle P_1, 0, \beta \rangle\). Now assume the execution of \(P_1\) finished, i.e. we will have the configuration \(\langle \{0 \div C_1^1 \div 0 : f_{1_k}^i : C_1 : T_1 \} \sqcup s_1, 0, \beta \rangle\) of inner scope \(s_1\) where \(C_1^1\) is the compensation context that was accumulated while executing \(P_1\). Using \([SCOPE END]\) we will get

\[
\langle \{0 \div C_1^1 \div 0 : f_{1_k}^i : C_1 : T_1 \} \sqcup s_1, 0, \beta \rangle \xrightarrow{T} \langle 0, \{C_1 \div 0 \div C_A^1 : \emptyset : 0 : 0\} \sqcup s_1, 0, \beta \rangle
\]

and \(\langle 0 ; s_2; \ldots; s_n, \{C_1 \div 0 \div C_A^1 : \emptyset : 0 : 0\} \sqcup s_1, 0, \beta \rangle\) for the inner sequence activity of scope \(S\) which will result in the accumulated compensation context \(\{C_1 \div 0 \div C_A^1 : \emptyset : 0 : 0\} \sqcup s_1, 0, \beta \rangle\) for scope \(S\). Further execution of scope \(S\) will result in the execution of configuration \(\langle 0 ; s_2; \ldots; s_n, \{C_1 \div 0 \div C_A^1 : \emptyset : 0 : 0\} \sqcup s_1, 0, \beta \rangle\) for its inner activity. Using \([SEQT]\) and \([SEQ]\) we will get

\[
\langle 0 ; s_2; \ldots; s_n, \{C_2 \div 0 \div C_A^2 : \emptyset : 0 : 0\} \sqcup s_2, 0; \{C_1 \div 0 \div C_A^1 : \emptyset : 0 : 0\} \sqcup s_1, 0, \beta \rangle
\]

as an intermediate configuration of the inner sequence of scope \(S\). The final accumulated compensation context of scope \(S\) will then be

\[
C_A = \{C_n \div 0 \div C_A^n : \emptyset : 0 : 0\} \sqcup s_n; \ldots; \{C_1 \div 0 \div C_A^1 : \emptyset : 0 : 0\} \sqcup s_1; 0 \quad (\ast)
\]

which will be installed as the fixed compensation context for scope \(S\)'s compensation handler, i.e.

\[
(S, \alpha, \beta) \rightarrow^* \langle 0, \{C \div 0 \div C_A : \emptyset : 0 : 0\} \sqcup S, \alpha, \beta \rangle
\]

Thus, when compensating scope \(S\), the compensation handlers of its child scopes will get called in reverse execution order \(c_n\) to \(c_1\). For example:

\[
\langle 1, 0, \{C \div 0 \div C_A : \emptyset : 0 : 0\} \sqcup S, 0, 0 \rangle \xrightarrow{T} \langle \{C \div 0 \div C_A : \emptyset : 0 : 0\} \sqcup S, 0, 0 \rangle
\]

with \(C = \ast\):

\[
\langle \{C \div 0 \div C_A : \emptyset : 0 : 0\} \sqcup S, 0, 0 \rangle \xrightarrow{T} \langle \{C_A \div 0 \div \emptyset : 0 : 0\} \sqcup S, 0, 0 \rangle
\]

and \(C_A\) being the sequence of all compensation handlers of \(S\)'s child scopes in reverse execution order \((\ast)\).

### 4.3 All-or-nothing Semantics for Compensation Handlers

Sometimes a developer of a Business Process might want to ensure that either compensation of a scope completes successfully or the effects of compensation are undone in case of faulted compensation in order to prevent partial compensation. This behavior is called *all or nothing* semantics for compensation handlers. In BPEL, that behavior can be modeled by enclosing the inner compensation activity of a compensation handler by a scope with a manually defined fault handler. That fault handler should intercept faults concerning the inner compensation activity and provide compensation actions for each possible point of failure inside that compensation. Finally, the handler should re-throw the fault such that the caller of the compensation handler will be informed.
Example 7. In order to explain that concept, we will assume the activity of the scope considered to be a sequence of BPEL activities $A_1, \ldots, A_n$ and $A_1^{-1}, \ldots, A_n^{-1}$ to be their respective (usually structured) counter activities that try to undo their effects. So in order to equip scope $S = \{A_1 ; \ldots ; A_n \div 0 \div 0 : f_h : C : T\}$ with a compensation handler that has the desired all or nothing property, we will encapsulate its activities by a scope, i.e. we will specify $C$ as $C = \{A_n^{-1} ; \ldots ; A_1^{-1} \div 0 \div 0 : f_h^C : 0 \div 0\}$. Now the inner scope’s fault handler $f_h^C$ needs to be specified. For each possible point of failure (fault $f_i$) in the sequence $A_n^{-1} ; \ldots ; A_1^{-1}$, $f_h^C(f_i)$ provides an activity that tries to undo the actions of scope $C$ up to that point. For example, if activities $A_i^{-1}$ all are basic then $f_h^C(f_i) = (A_i^{-1})^{-1} ; \ldots ; (A_{n-i-1}^{-1})^{-1}$, where $(A_i^{-1})^{-1}$ compensates$^4$ $A_i^{-1}$.

If activities $A_i^{-1}$ are scopes then $f_h^C$ could be defined as $f_h^C(f_i) = \uparrow$, enabling compensation of activities inside compensation handlers.

Enabling this possibility, i.e. to have normal mode $\Box$ scopes inside compensation handlers, finally forced us to separate the accumulated compensation context $C_A$ from the fixed compensation context $C_F$ of a scope, when designing the calculus. In the scenario of a normal mode scope $N$ inside a compensation handler $C$ (an all-or-nothing compensation handler), scope $N$ will draw compensation handlers that it intended to call from $\beta$, which is part of the configuration and is the passed trough fixed compensation context of the outer compensation handler. But it might still install compensation handlers of its child scopes $c_i$ into $C_A$. So if compensation fails at some point, the fault handler of $N$ will intercept that fault and the scope switches to faulted mode. Thus, the original compensation handlers, scope $N$ should have been able to call, are not accessible any more (because faulted mode scopes are executed by passing through their own fixed context $C_F$), but the “compensation of the compensation”, i.e. handlers $c_i$ residing in $C_F$, are accessible.

Please note, that BPEL disallows a compensation handler to be specified for a single root scope (a scope enclosed by a compensation handler), arguing that this compensation handler could never be accessed in further execution. For the sake of compositionality, our semantics allows single root scopes to have a compensation handler, ensuring that it will not be accessible. The reason is, that the active compensation contexts of completed compensation handlers will not be installed (cf. rule [SCOPE END FCT]). Correspondence with Mark Ford$^5$ resulted in the acknowledgment that in the case of multiple root scopes, compensation handlers are allowed to be specified for each of them. Although this is vital for the all or nothing semantics to work, the official specification of BPEL [18] omitted that detail.

Example 8. Consider the case of a compensation handler $C$ of a scope $S$ consisting of a sequence of inner scopes $c_1$ to $c_n$ in order to compensate the inner activity $P$ of scope $S$, i.e. $S = \{P \div 0 \div 0 : f_h : C : T\}$ with $C = \{P_1 \div 0 \div 0 : f_h^C : C_1 : T_1\} ; \ldots ; \{P_n \div 0 \div 0 : f_h^C : C_n : T_n\}$. Assume scope $S$ will be compensated, resulting in the execution of scopes $c_1$ to $c_n$ inside the compensating mode scope, representing the installed compensation handler $C$.

$^4$Please note that $(A_i^{-1})^{-1} = A_i$ does not necessarily hold in general. Consider the example of an invoke activity. A compensation activity for that invoke could be a reply stating the cancellation of that invoke but it is not obvious whether invoking again will cancel the cancellation.

$^5$Active Endpoints, Inc.; Member of Editor Team of BPEL specification
Now, if one of those scopes $c_i (1 \leq i \leq n)$ faults, not only the partial execution of that scope has to be undone, but also the complete execution of the scope $c_1$ up to $c_{i-1}$, i.e. calling the compensation handlers of those scopes in default compensation order.

In the semantics of BPEL$_{fct}$ that case basically is handled by the specially designed inference rule [SCOPE FAULT C]. The inner scope that faults $c_i$ will call its fault handler $f_i$, which will undo the partial execution of $P_i$ and finally will (if specified properly) re-throw the fault up to the compensation handler $C$ of scope $S$ currently being executed. Rule [SCOPE FAULT C] will then disrupt further execution, set the compensation handler’s mode to faulted (\textit{Fault}), and initiate compensation in default compensation order (↑), resulting in the execution of the inner scope’s compensation handlers ($c_{i-1}$ to $c_1$). After compensation, i.e. undoing the partial execution of the compensation handler of $S$, the fault will be re-thrown (\textit{throw (f)}). Because of the compensation handler $C$ now being in faulted mode, and no other activities remaining (assuming successful completion of activities $c_{i-1}$ to $c_1$), the fault will be lifted up to the caller of compensation handler $C$, applying inference rule [SCOPE FAULT F].
5 Conclusion

Web Service Orchestration allows for rapid, modular, and therefore usually less error-prone development of Business Processes out of existing services. Several languages, designed for that task, have been proposed with WS-BPEL 2.0 having become the de facto standard. BPEL distributes the different parts of error handling onto various specialized handlers, namely fault, compensation, and termination (FCT) handlers. The granularity of error handling is user-definable by enclosing behavior in its own scope. Because of the usually long running time of Business Processes, a concept called compensation is used, in order to deal with failed transactions and to undo work that has been done up to the point of failure. Compensation in BPEL relies heavily on the correct interplay of the aforementioned handlers and therefore should be taken into account when formally investigating dependability of Business Processes specified in BPEL. Unfortunately, the specification of BPEL is given in prose, which complicates formal reasoning about the behavior of the composed Business Processes.

In this work the calculus $BPEL_{fct}$ has been proposed, a framework to model scope-based fault, compensation, and termination (FCT) handling in BPEL as precisely as possible. A lightweight syntax $BPEL'$ of the language is used to abstract from data and time dependencies, in order to model each possible behavior. A mapping from BPEL to the calculus then can be achieved easily using that intermediate step. We also showed how the central mechanism of compensation, as well as advanced techniques, like ensuring atomicity w.r.t. compensation (all-or-nothing semantics of compensation handlers), is handled by our calculus.

Future Work $BPEL_{fct}$ mainly deals with the core concepts of scope-based FCT-handling and therefore covers only a subset of the primitives provided by BPEL. For future work, the remaining primitives, like event handlers, loops, and links in flows could be added. For the link semantic of flows, several existing approaches, like [10] or [21], could be followed. Concerning compensation, the calculus is prepared for an extension with loop primitives. Compensation handlers of scopes can be installed multiple times when executing a scope inside a loop. Those installed handlers will be available as a compensation group, like stated in the BPEL specification, accessible via the name of the scope. Consequently, adding loops to the calculus should be straightforward. For an extension with event handlers (in order to achieve feature completeness of the calculus) the scope construct would have to be modified extensively. Completion of its inner activity would have to be intercepted, s.t. an added event handler can then be shut down. Also, executed event handler instances must be able to be compensated. Therefore, the accumulated compensation context will have to be modified carefully to also include the compensation handlers of completed inner scopes of those handler instances.

Although being an important aspect, exhaustive proving of correctness of our calculus in respect to existing approaches would be far beyond the scope of this Bachelor’s thesis. Nevertheless, work in progress compares the basic mechanisms of our calculus with the nested Sagas calculus. First results show that a certain subset of Sagas can be simulated by our calculus in respect to weak traces.
References


