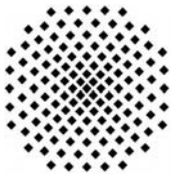

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Abstract Syntax of WS-BPEL 2.0

Oliver Kopp, Ralph Mietzner, Frank Leymann

Report 2008/06

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: D.3.1, H.4.1

Abstract

WS-BPEL 2.0 is the current version of the "Business Process Execution Language for Web Services". Until now, no formal definition of its syntax exists. We present a complete syntax of WS-BPEL 2.0 of both abstract and executable processes.

Contents

Chapter 1

Introduction

WS-BPEL 2.0 [6] is the current version of the "Business Process Execution Language for Web Services". Until now, no formal definition of the syntax exists. Ouyang et al. proposed in [7] a definition of the syntax of BPEL4WS 1.1, which is the predecessor of WS-BPEL 2.0. We took their formalization as basis and extended it to a formalization of the syntax of WS-BPEL 2.0 syntax including abstract BPEL processes. Note that we are not dealing with a formalization of the semantics of WS-BPEL 2.0 processes. The work presented here provides a formal basis for such a formalization and for other formal aspects of WS-BPEL 2.0. An overview of existing formalizations of the semantics of WS-BPEL 2.0 processes is presented in [2].

Chapters 3 to 5 introduce the general syntax of WS-BPEL 2.0 processes. Afterwards, Chapters 6 and 7 show additional constraints for executable and abstract processes. The definitions include references to the specification and the static validations rules. Static validation rules are listed in the WS-BPEL 2.0 specification in Appendix B and denoted as [SAxxx], where xxx denotes the number of the static validation rule. Chapter 8 lists the static validation rules, which are not covered by the formalization, because they are XML, extensibility, or runtime specific. To provide an application of the formalization, Chapter 9 presents the loan approval process expressed in the introduced formalization. Finally, Chapter 10 concludes and gives an outlook to future work.

1.1 General Considerations

WS-BPEL 2.0 is the current version of WS-BPEL. We use "WS-BPEL" as a shortcut for "WS-BPEL 2.0" in this report.

In order to facilitate reading, we do not formalize using names as glue between WS-BPEL artifacts. For example, we declare correlation sets directly at the scopes without declaring a name and then use the name as reference to a correlation set.

We use π_i to denote the projection to the i^{th} component of a tuple t. For example $\pi_2(a, b, c, d) = b$.

Chapter 2

Formalization of Necessary Parts of Related Standards

This section presents a formalization of necessary parts of related standards needed for a complete formalization of WS-BPEL. Note that only the relevant parts of these standards are formalized here and thus the formalization of these is not complete and is subject of further work.

Specifically, we do not formalize the syntax of XPath expressions. As a result, the WS-BPEL-defined XPath functions `doXslTransform` and `getVariableProperty` are not formally defined here.

2.1 Typing

The typing of WS-BPEL is based on XML-Schema [8]. We present the important concepts needed by WS-BPEL.

Definition 2.1.1 (Names)
The set $\mathcal{N}$ is the set of NCNames without `##opaque`*.*

Definition 2.1.2 (QNames)
The set QName *is the set of QNames.*

Definition 2.1.3 (Types defined or included in the process definition)
The set Struct *is the set of types, which are defined or included in the process definition.*

Definition 2.1.4 (Used types defined using XML-Schema)
The set XMLSTRUCT *is the infinite set of XML-Schema types that can be defined using XML-Schema and are included in the process definition.* $\text{XMLSTRUCT} \subset \text{STRUCT}$

Definition 2.1.5 (XSD simple types)
The set $\text{XMLSTRUCT}_{simple}$ *is the set of XSD simple types.*

Definition 2.1.6 (Element Information Item (EII))
EII *denotes the Element Information Item value type.*

Definition 2.1.7 (Elements defined by XML-Schema)
The set EL *is the set of elements (name, type pairs) as defined by XML-Schema. Every element in* EL *is of type* EII*.*

Definition 2.1.8 (Location Value)
$\mathbb{L}$ *denotes the lvalue type. In the context of XPath, an lvalue is a node-list containing a single node.*

Definition 2.1.9 (Text Information Item (TII))
TII *denotes the Text Information Item value type.*

Definition 2.1.10 (Character sequences)
The set TXT *denotes all character sequences that are allowed as text information items (TII) in XML.*

Definition 2.1.11 (Set of all types)
The set TYPES *denotes the infinite set of all types of all programming languages including all XML Schema types and the lvalue, EII, and TII type. The term "all programming languages" is used, since WS-BPEL supports types declared in past, present and future programming languages through its extensibility mechanism.*

Definition 2.1.12 (Set of queries)
The set $\mathcal{Q}$ *is the infinite set of queries.*

Definition 2.1.13 (Types of elements referred by an lvalue)
The function $\mathsf{type}_{\mathbb{L}} : \mathbb{L} \rightarrow \text{TYPES}$ *returns the type of the element referenced by the given lvalue.*

2.2 Expressions and Queries

Definition 2.2.1 (Set of all expressions)
The set EX *is the set of all expressions. (BPEL specification, page 57).*

Definition 2.2.2 (Type returned by an expression)
The function $\mathsf{type}_{\mathrm{Ex}} : \mathrm{Ex} \to \textsc{Struct}$ *returns the result type of an expression.*

Definition 2.2.3 (Boolean expressions)
The set of $\mathcal{C}$ *is the set of expressions returning a boolean value. (BPEL specification, page 57).*

$$\mathcal{C} \subseteq \mathrm{Ex} = \{ex \in \mathrm{Ex} \mid \mathsf{type}_{\mathrm{Ex}}(ex) = \texttt{xsd:boolean}\}$$

Definition 2.2.4 (Unsigned integer expressions)
The set of Ex_{int} *is the set of expressions returning an unsigned integer value. (BPEL specification, page 57).*

$$\mathrm{Ex}_{int} = \{ex \in \mathrm{Ex} \mid \mathsf{type}_{\mathrm{Ex}}(ex) = \texttt{xsd:unsignedInt}\}$$

Definition 2.2.5 (Duration and date expressions)
The set of $\mathrm{Ex}_{duration}$ *is the set of expressions returning a duration value. (BPEL specification, page 57).*

$$\mathrm{Ex}_{duration} = \{ex \in \mathrm{Ex} \mid \mathsf{type}_{\mathrm{Ex}}(ex) = \texttt{xsd:duration}\}$$

The set of Ex_{date} *is the set of expressions returning a date value. (BPEL specification, page 57).*

$$\mathrm{Ex}_{date} = \{ex \in \mathrm{Ex} \mid \mathsf{type}_{\mathrm{Ex}}(ex) = \texttt{xsd:date}\}$$

Definition 2.2.6 (Integer value of an expression)
The function $\mathsf{value}_{int} : \mathrm{Ex}_{int} \to \mathbb{N}$ *returns the integer value of an unsigned integer expression. We only specify the function for unsigned integers here since it is needed for static validation.*

Definition 2.2.7 (Expression languages)
The set ExLang *denotes the set of all expression languages. (BPEL specification, page 49).*

2.3 WSDL

This section presents a formalization of WSDL 1.1 artifacts [3] relevant to WS-BPEL.

Definition 2.3.1 (Set of WSDL messages)
The set MSG *is the infinite set of messages which can be declared using WSDL.*

Definition 2.3.2 (Name of a WSDL message)
The function $\mathsf{name}_{\mathrm{MSG}} : \mathrm{MSG} \to \mathcal{N}$ *assigns a name to a WSDL message.*

Definition 2.3.3 (Type of a named WSDL message)
The function $\mathsf{type}_{\mathrm{MSG}} : \mathrm{MSG} \rightarrow \textsc{Struct}$ *assigns a type to WSDL message.*

Definition 2.3.4 (Set of port types)
The set PT *is the set of port types, that can be declared using WSDL.*

Definition 2.3.5 (Names of operations)
The function $\mathsf{name}_{\mathrm{PT}} : \mathrm{PT} \rightarrow \mathcal{N}$ *assigns a name to a port type.*

Definition 2.3.6 (Set of operations)
The set $\mathcal{O}$ *is the set of operations, that can be declared using WSDL.*

Definition 2.3.7 (Names of operations)
The function $\mathsf{name}_{\mathcal{O}} : \mathcal{O} \rightarrow \mathcal{N}$ *assigns a name to an operation.*

Definition 2.3.8 (Assigning operations to port types)
The function $\mathsf{portType}_{operation} : \mathcal{O} \rightarrow \mathrm{PT}$ *assigns a port type to an operation.*

The names of operations contained in a port type must be unique. [SA00002]

$$\forall o, p \in \mathcal{O} :$$
$$\mathsf{portType}_{operation}(o) = \mathsf{portType}_{operation}(p) \wedge \mathsf{name}_{\mathcal{O}}(o) = \mathsf{name}_{\mathcal{O}}(p) \Rightarrow o = p$$

Definition 2.3.9 (Input, output, and fault messages)
The function $\mathsf{msg}_{input} : \mathcal{O} \rightarrow \mathrm{MSG} \cup \{\bot\}$ *assigns one input message to an operation.*

The function $\mathsf{msg}_{output} : \mathcal{O} \rightarrow \mathrm{MSG} \cup \{\bot\}$ *assigns one output message to an operation.*

The function $\mathsf{msg}_{fault} : \mathcal{O} \rightarrow 2^{\mathrm{MSG}} \cup \{\bot\}$ *assigns fault messages to an operation.*

Definition 2.3.10 (Types of WSDL operations)
The function $\mathsf{operType}: \mathcal{O} \rightarrow \{\texttt{one-way}, \texttt{request-response}, \texttt{solicit-response}, \texttt{notification}\}$ *assigns a type to an operation where* $o \in \mathcal{O}$ *is a WSDL operation:*

One way operations have only an input message.

$$\mathsf{operType}(o) = \texttt{one-way} \Rightarrow \mathsf{msg}_{input}(o) \neq \bot \ \wedge \ \mathsf{msg}_{output}(o) = \bot$$

Notification operations have only an output message.

$$\mathsf{operType}(o) = \texttt{notification} \Rightarrow \mathsf{msg}_{input}(o) = \bot \ \wedge \ \mathsf{msg}_{output}(o) \neq \bot$$

Request-Response and solicit-response operations have an input as well as an output message.

$$(\mathsf{operType}(o) = \texttt{request-response}) \ \vee \ (\mathsf{operType}(o) = \texttt{solicit-response}) \Rightarrow$$
$$(\mathsf{msg}_{input}(o) \neq \bot) \wedge (\mathsf{msg}_{output}(o) \neq \bot)$$

Definition 2.3.11 (Message parts)
The set MP *denotes the set of all message parts.*

Definition 2.3.12 (Type of a message part)
The function $\mathsf{type}_{\mathrm{MP}} : \mathrm{MP} \to \textsc{Struct}$ *is a function returning the type of the given message part.*

Definition 2.3.13 (Allowed names for message parts)
The function $\mathsf{name}_{\mathrm{MP}} : \mathrm{MP} \to \mathcal{N}$ *is a function returning the name of the given message part.*

Definition 2.3.14 (Message part)
The function $\mathsf{mpm} : \mathrm{MP} \to \mathrm{MSG}$ *assigns a WSDL message part to a message.*

Definition 2.3.15 (Message parts II)
The function $\mathsf{MP} : \mathrm{MSG} \to 2^{\mathrm{MP}},\ m \mapsto \{mp \in \mathrm{MP} \mid \mathsf{mpm}(mp) = m\}$ *returns the set of all message parts declared in message* m*.*

Definition 2.3.16 (Partner link types)
The set PLT *is the set of partner link types. (BPEL specification, page 36).*

Definition 2.3.17 (Roles)
A role is a tuple consisting of a name and a port type. (BPEL specification, page 36).

$$\mathrm{ROLES} \subseteq \mathcal{N} \times \mathrm{PT}$$

Definition 2.3.18 (Roles specified by a partner link type)
The function $\mathsf{roles} : \mathrm{PLT} \to \mathrm{ROLES} \times (\mathrm{ROLES} \cup \{\bot\})$ *assigns roles to a partner link type. There may be only one or two roles be assigned. (BPEL specification, pages 36,37).*

The names of the roles have to be unique.

$$\neg\exists((n, pt), (n, pt')) \in DOM(\mathsf{roles})$$

Definition 2.3.19 (Properties of WSDL messages)
The set $\mathrm{PROP} \subseteq \mathcal{N} \times (\mathrm{EL} \cup \textsc{Struct})$ *is a set of properties defined in the WSDL (BPEL specification, page 40). A property is defined by a unique name and an element or type [SA00019] (BPEL specification, page 41).*

Definition 2.3.20 (Properties of WSDL messages having a simple type)
The set PROP_{simple} *denotes the set of message properties having a simple type.*

$$\mathrm{PROP}_{simple} = \{m \in \mathrm{PROP} \mid \pi_2(m) \in \mathrm{XML}\textsc{Struct}_{simple}\}$$

Definition 2.3.21 (Accessing properties)
A property alias is used to extract a property out of a message part, element, or type using a query Q. [SA00020] (BPEL specification, page 43).

$$\text{PROPALIAS} \subseteq \text{PROP} \times \mathcal{Q} \times (\text{MP} \cup \text{EL} \cup \textsc{Struct})$$

There has to be at most one property alias extracting a property of a certain type. [SA00022]

$$\forall pa = (p, q, e), pb = (p', q', e') \in \text{PROPALIAS} : p = p' \wedge e = e' \Rightarrow pa = pb$$

Chapter 3

Basic Concepts of a WS-BPEL Process Model

3.1 The Process Model Itself

Definition 3.1.1 (WS-BPEL Process Model)
A WS-BPEL Process Model *is a tuple*
$M = (\mathcal{A}$, process, $\mathsf{type}_{\mathcal{A}}$, $\mathsf{name}_{\mathcal{A}}$, $\mathcal{E}$, $\mathsf{type}_{\mathcal{E}}$, HR, ExLang, QryLang, $\mathcal{V}$, $\mathsf{type}_{\mathcal{V}}$, $\mathsf{name}_{\mathcal{V}}$, declareVar, supjoinf, instance, PL, $\mathsf{type}_{\mathrm{PL}}$, myRole, partnerRole, initializePartnerRole, $\mathsf{partnerlink}_{\mathcal{CO}}$, declarePartnerLink, $\mathsf{name}_{\mathrm{PL}}$, $\mathsf{operation}_{\mathcal{CO}}$, $\mathsf{portType}_{\mathcal{CO}}$, inputVar, outputVar, FP, TP, $\mathsf{faultName}_{reply}$, MEX, declareMEX, $\mathsf{name}_{\mathrm{MEX}}$, mex, CORSET, declareCor, $\mathsf{name}_{\mathrm{CORSET}}$, CORRELATION, COR, Copy, assignCopy, $\mathsf{fromSpec}_{\mathcal{V}}$, $\textsc{From}_{var}$, $\textsc{From}_{pl}$, $\textsc{From}_{prop}$, $\textsc{From}_{ex}$, $\textsc{From}_{lit}$, $\textsc{From}_{empty}$, $\textsc{From}_{opaque}$, $\mathsf{type}_{\mathrm{FROM}}$, $\textsc{To}_{var}$, $\textsc{To}_{pl}$, $\textsc{To}_{prop}$, $\textsc{To}_{ex}$, $\mathsf{type}_{\mathrm{TO}}$, keepSrcElementName, ignoreMissingFromData, validate, $\mathsf{validate}_{\mathcal{A}_{assign}}$ valVar, $\mathsf{faultname}_{\mathcal{A}_{throw}}$, faultVariable, Ω_{if}, Ω_{seq}, for_{W}, for_{A}, until_{W}, until_{A}, $\mathsf{repeatEvery}_{A}$, $\mathcal{L}$, declareLink, $\mathsf{name}_{\mathcal{L}}$, SR, TR, tc, jc, startCounterValue, finalCounterValue, completionCondition, parallel, $\mathsf{counter}_{\mathcal{A}_{forEach}}$, exitOnStandardFault, isolated, $\mathsf{faultName}_{\mathcal{E}}$, $\mathsf{faultVariable}_{\mathcal{E}}$, catchAll, target, templateInstance, profile*)*,
where the elements are defined as presented in the following sections. All sets are disjoint.

The general idea of the formalization presented in [7] is to map the XML hierarchy structure into a hierarchy relation HR. We adopt this idea, to stay close to existing formalizations.

The general WS-BPEL Process Model describes both abstract and executable processes. Therefore, we describe the common base and use it as basis for both abstract and

executable processes. (BPEL specification, page 147) Thus,the opaque token is included here. WS-BPEL does not consider `##opaque` being part of declared sets, but it allows the usage of opaque attributes where such an element is referenced. For example, `##opaque` is not a partner link, but `partnerLink="##opaque"` is allowed in an invoke activity at abstract processes. Since we want to describe the generic process model from which both executable and abstract processes can be derived, we derive sets including `##opaque` for every set, where references to elements may be opaque.

Definition 3.1.2 (Opaque token)
`##opaque` denotes a token which is explicitly set to opaque.

Definition 3.1.3 (Process Profile)
WS-BPEL allows to specify a profile for an abstract process. Two profiles are declared in the WS-BPEL specification, the template profile, and the profile for observable behavior. $\mathsf{profile}$ *is the constant holding the profile for the WS-BPEL process model. If no profile is assigned the process ("$\perp$"), then the process is an executable process, otherwise an abstract process. (BPEL specification, page 147)*

$$\mathsf{profile} \in \{\perp, \mathit{template}, \mathit{observable}\}$$

Definition 3.1.4 (Opaque elements in arbitrary sets for template profiles)
If S is an arbitrary set, then $S^{\#_t}$ is the set S including the opaque value, if the current process model is abstract and of the template profile. The set S may not contain `##opaque`. (BPEL specification, page 148)

$$S^{\#_t} \leftarrow \begin{cases} S \cup \{\mathit{\#\#opaque}\} & \mathsf{profile} = \mathit{template} \\ S & \mathit{otherwise} \end{cases}$$

For example, the set $\mathbb{B}^{\#_t}$ is the set of boolean values plus the opaque value. $\mathbb{B}^{\#_t} = \{\texttt{true}, \texttt{false}, \texttt{\#\#opaque}\}$ For example, this opaque value may be used for the `createInstance` value for a receive activity as shown in Listing 1. The presented XML representation translates to $\mathsf{createInstance}(r) = \texttt{\#\#opaque}$ in our formalism. $\mathsf{createInstance}$ is defined in definition 4.4.3.

Listing 1 Receive with opaque create instance attribute in a WS-BPEL XML document

```
<receive createInstance="##opaque" ...>...</receive>
```

In the process profile for observable behavior, message related attributes are allowed to be opaque. All other attributes are not allowed to be opaque. Therefore, we define $S^{\#_{o,t}}$ as a variant of $S^{\#_t}$, where `##opaque` is also part of the set if the process is of the observable behavior.

Definition 3.1.5 (Opaque elements in arbitrary sets for abstract processes)
If S is a set, then $S^{\#_{o,t}}$ is the set S including the opaque value, if the current process model is abstract. The set S may not contain `##opaque`*. (BPEL specification, page 156)*

$$S^{\#_{o,t}} \leftarrow \begin{cases} S \cup \{\texttt{\#\#opaque}\} & \mathsf{profile} \in \{\texttt{template}, \texttt{observable}\} \\ S & \textit{otherwise} \end{cases}$$

Definition 3.1.6 (Possibly omitted elements in arbitrary sets)
If S is a set, then $S_{\underline{\perp}^{e,o,t}}$ is the set S including $\perp$. $\perp$ denotes an omitted element. The set S may not contain $\perp$.

$$S_{\underline{\perp}^{e,o,t}} \leftarrow S \cup \{\perp\}$$

For example, the set $\mathbb{B}_{\underline{\perp}^{e,o,t}}$ is the set of boolean values plus the undefined value.

$$\mathbb{B}_{\underline{\perp}^{e,o,t}} = \{\texttt{true}, \texttt{false}, \perp\}$$

For example, the value used for the `createInstance` at a receive activity is undefined, as shown in Listing 2. The presented XML representation translates to $\mathsf{createInstance}(r) = \perp$ in our formalism.

Listing 2 Receive with undefined create instance attribute in a WS-BPEL XML document

```
<receive />
```

It is important to note that createInstance returns `""` and not $\perp$ for the receive shown in Listing 3.

Listing 3 Receive with empty create instance attribute in a WS-BPEL XML document

```
<receive createInstance="" />
```

Attributes such as the `startCounterValue` may be omitted in the abstract common base, but must be specified in executable processes or the abstract process profile for templates. Therefore, we introduce $\perp^o$ to denote that a set may contain the omission shortcut in certain cases.

Definition 3.1.7 (Possibly omitted elements)
If S is a set, then $S_{\underline{\perp}^o}$ is the set S including $\perp$ if the process model follows the observable behavior. $\perp$ itself denotes an omitted element. The set S may not contain $\perp$.

$$S_{\underline{\perp}^o} \leftarrow \begin{cases} S \cup \{\perp\} & \mathsf{profile} = \texttt{observable} \\ S & \textit{otherwise} \end{cases}$$

The possibly opaque and omitted set specifications may be combined. For example, if S is a set, then $S^{\#_{o,t}}_{\underline{\perp}^{o,t}}$ is the set S including $\perp$ and `##opaque` for both abstract profiles.

$$S^{\#_{o,t}}_{\underline{\perp}^{o,t}} \leftarrow S_{\underline{\perp}^{o,t}} \cup S^{\#_{o,t}}$$

3.2 Activities

Definition 3.2.1 (Set of activity types)
The set $\mathcal{T}_{\mathcal{A}}$ is the set of all activity types defined in the WS-BPEL specification. (BPEL specification, page 24)

$$\begin{aligned}\mathcal{T}_{\mathcal{A}} = \{&sequence, flow, if, pick, while, repeatUntil, forEach, scope,\\ &invoke, receive, reply, wait, assign, empty, throw, rethrow,\\ &compensate, compensateScope, exit, validate,\\ &opaqueActivity, extensionActivity\}\end{aligned}$$

Definition 3.2.2 (Type of an activity)
The function $\mathsf{type}_{\mathcal{A}} : \mathcal{A} \rightarrow \mathcal{T}_{\mathcal{A}}$ is a function that assigns types to activities taken from the set of activity types. The set $\mathcal{A}$ is defined in definition3.2.6.

Definition 3.2.3 (Set of all activities of a given type)
The set $\mathcal{A}_t$ is the set of all activities of type t.

$$\forall t \in \mathcal{T}_{\mathcal{A}} : \mathcal{A}_t = \{a \in \mathcal{A} \mid \mathsf{type}_{\mathcal{A}}(a) = t\}$$

Definition 3.2.4 (Set of structured activities)
The set $\mathcal{A}_{structured}$ is the set of structured activities. (BPEL specification, page 98)

$$\mathcal{A}_{structured} = \mathcal{A}_{sequence} \cup \mathcal{A}_{flow} \cup \mathcal{A}_{if} \cup \mathcal{A}_{pick} \cup \mathcal{A}_{while} \cup \mathcal{A}_{repeatUntil} \cup \mathcal{A}_{forEach} \cup \mathcal{A}_{scope}$$

Definition 3.2.5 (Set of basic activities)
The set $\mathcal{A}_{basic}$ is the set of basic activities. (BPEL specification, page 84)

$$\begin{aligned}\mathcal{A}_{basic} =&\mathcal{A}_{invoke} \cup \mathcal{A}_{receive} \cup \mathcal{A}_{reply} \cup \mathcal{A}_{wait} \cup \mathcal{A}_{assign} \cup \mathcal{A}_{empty} \cup\\ &\mathcal{A}_{throw} \cup \mathcal{A}_{rethrow} \cup \mathcal{A}_{compensate} \cup \mathcal{A}_{compensateScope} \cup\\ &\mathcal{A}_{exit} \cup \mathcal{A}_{validate} \cup \mathcal{A}_{opaqueActivity}\end{aligned}$$

Definition 3.2.6 (Set of all activities)
$\mathcal{A}$ is a set of all WS-BPEL activities. These include the structured and basic activities, as defined in the WS-BPEL specification. Since an extension activity may be structured or non structured, it is neither included in the set of basic activities nor in the set of structured activities. (BPEL specification, page 96)

$$\mathcal{A} = \mathcal{A}_{structured} \cup \mathcal{A}_{basic} \cup \mathcal{A}_{extensionActivity}$$

Definition 3.2.7 (The "process" element)
The `process` $\in \mathcal{A}_{scope}$ *is the process activity. (BPEL specification, pages 19,21)*

3.3 Events

Events in this formalization are understood as general events in contrast to message events in the WS-BPEL specification. The events are used to trigger the respective handlers. For example, a fault event triggers a fault handler. The declaration of handlers at an activity is done by the hierarchy relation. The hierarchy relation connects the activity to the topmost child of a handler with the respective event. Details are described in Section 3.5.

Definition 3.3.1 (Set of event types)
The set $\mathcal{T}_{\mathcal{E}}$ *is the set of all event types. These types are (i) an* `onMessage` *event, (ii) an* `onAlarm` *event, (iii) a* `fault` *event, (iv) a* `compensation` *event, or (v) a* `termination` *event. (i),(ii) An* `onMessage` *event or* `onAlarm` *event triggers the respective branch in a pick activity or an event handler. (iii) A* `fault` *event triggers a fault handler. (iv) a* `compensation` *event triggers a compensation handler. (v) a* `termination` *event triggers a termination handler.*

$$\mathcal{T}_{\mathcal{E}} = \{message, alarm, fault, compensation, termination\}$$

It is important to note that the WS-BPEL specification distinguishes between `onMessage` branches in pick activities and `onEvent` branches in event handlers, but does not distinguish between the `onAlarm` branches in pick activities and event handlers. `onMessage` branches may not occur in event handlers and `onEvent` branches may not occur in pick activities. Nevertheless, both of them are used for receiving messages. We therefore identified `onMessage` and `onEvent` to `onMessage`.

Definition 3.3.2 (Type of an event)
The function $\mathsf{type}_{\mathcal{E}} : \mathcal{E} \rightarrow \mathcal{T}_{\mathcal{E}}$ *assigns types to events taken from the set of event types.*

Definition 3.3.3 (Set of all events of a given type)
The set $\mathcal{E}_t$ *is the set of all events of type* t*.*

$$\forall t \in \mathcal{T}_{\mathcal{E}} : \mathcal{E}_t = \{e \in \mathcal{E} \mid \mathsf{type}_{\mathcal{E}}(e) = t\}$$

Definition 3.3.4 (Set of normal events)
The set $\mathcal{E}_{normal}$ *provides a set of normal events.*

$$\mathcal{E}_{normal} = \mathcal{E}_{message} \cup \mathcal{E}_{alarm}$$

Definition 3.3.5 (Set of exceptional behavior events)
The set $\mathcal{E}_{exception}$ provides a set of exceptional behaviour events such as (fault, compensation and termination).

$$\mathcal{E}_{exception} = \mathcal{E}_{fault} \cup \mathcal{E}_{compensation} \cup \mathcal{E}_{termination}$$

Definition 3.3.6 (Set of all events)
$\mathcal{E}$ is the set of all events.

3.4 Conditions

Definition 3.4.1 (Set of conditions)
The set $\mathcal{C}$ is the set of Boolean conditions.

3.5 Hierarchy Relation

The hierarchy relation models the nesting of the activities. Three types of nesting are allowed: (i) Direct nesting, (ii) conditional nesting, (iii) nesting in an handler. (i) "Direct nesting" denotes that the enclosed activity is directly nested without any condition or event. (ii) The "conditional nesting" models the conditions of while, repeat-until, and if-activities. The nested activity may only be executed if the given condition evaluates to true. (iii) Handlers are connected to their parent scope using respective events.

Definition 3.5.1 (The set of connecting labels)
The set $\mathcal{B}$ is the set of labels used to connect parent and children activities in a WS-BPEL process. $\mathcal{B} = \mathcal{E} \cup_{\lfloor c,t}$ is a set of labels where "$\perp$" denotes direct nesting.

Definition 3.5.2 (The hierarchy relation)
The hierarchy relation $\mathsf{HR} \subseteq \mathcal{A} \times \mathcal{B} \times \mathcal{A}$ is a labeled tree which defines the relation between an activity and its direct sub-activities.

The hierarchy relation models the implicit includes relation of WS-BPEL. The child activities of an activity are modelled inside this activity in WS-BPEL. Activities modelled outside of an activity a can not be referenced as children from a. This implies, that an activity can only have one parent activity. In order to reflect this constraint, we restrict the relation HR so that at most one relation between two activities is allowed to exist.

$$\forall h, i \in \mathsf{HR} : \pi_1(h) = \pi_1(i) \wedge \pi_3(h) = \pi_3(i) \Rightarrow h = i$$

Listing 4 shows the syntax of a fault handler presented in the WS-BPEL specification. By definition 3.5.1 the `faultHandlers`, `catch`, and `catchAll` elements are implicitly included in the label representing a fault ($\mathcal{E}_{fault}$). If there exists a hierarchy relation connecting two activities via a fault event, then there exists either a `catch` or a `catchAll` inside a `faultHandler` element. [SA00080] (BPEL specification, page 128) Event handlers are treated the same way as fault handlers in the hierarchy relation. A hierarchy relation connecting two activities via an `onMessage` or `onAlarm` event implies an `eventHandler` XML element. [SA00083]

Listing 4 Syntax of a fault handler

```
<faultHandlers>?
  <!-- Note: There must be at least one faultHandler -->
  <catch faultName="QName"?
     faultVariable="BPELVariableName"?
     ( faultMessageType="QName" | faultElement="QName" )? >*
     activity
   </catch>
   <catchAll>?
     activity
   </catchAll>
</faultHandlers>
```

Definition 3.5.3 (Projection of HR on two activity sets)
The function $\mathsf{HR}_p : \mathsf{HR} \rightarrow \mathcal{A} \times \mathcal{A}$ *is the projection of* HR *on two activity sets.*

$$\mathsf{HR}_p = \pi_{1,3}(\mathsf{HR}(p))$$

Definition 3.5.4 (Children of an activity)
The function $\mathsf{children} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$ *returns the set of immediate descendants of an activity.*

$$\mathsf{children}(a) = \{a' \in \mathcal{A} \mid (a, a') \in \mathsf{HR}_p\}$$

Definition 3.5.5 (Descendants of an activity)
The function $\mathsf{descendants} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$ *returns the set of all descendants of an activity.*

$$\begin{aligned}\mathsf{descendants}(a) = \{a' \in \mathcal{A} \mid \\ \exists k : (a_1, a_2), \ldots, (a_j, a_{j+1}), \ldots, (a_{k-1}, a_k), (a_i, a_{i+1}) \in \mathsf{HR}_p, \\ 1 \leq i \leq k-1, a_1 = a, a_k = a'\}\end{aligned}$$

Definition 3.5.6 (Clan of an activity)
The function $\mathsf{clan} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$ *returns the set made up of a given activity and its descendants.*

$$\mathsf{clan}(a) = \{a\} \cup \mathsf{descendants}(a)$$

Chapter 4

Details of a WS-BPEL Process Model

4.1 Expression Languages

Definition 4.1.1 (Expression language used at an WS-BPEL construct)
The function $\mathsf{ExLang} : \{\texttt{process}\} \cup \mathrm{Ex} \to \textsc{ExLang}^{\#_t}_{\perp_e, o, t}$ *assigns the expression language to the process element and expressions. (BPEL specification, page 23,148)*

4.2 Query Languages

Definition 4.2.1 (Query languages used at an WS-BPEL construct)
The function $\mathsf{QryLang} : \{\texttt{process}\} \cup \mathcal{Q} \to \textsc{ExLang}^{\#_t}_{\perp_e, o, t}$ *assigns the query language to the process element. (BPEL specification, page 23)*

4.3 Variables

Definition 4.3.1 (The set of variables)
The set $\mathcal{V}$ *is the set of variables. (BPEL specification, page 45).*

Definition 4.3.2 (Type of a variable)
The function $\mathsf{type}_{\mathcal{V}} : \mathcal{V} \to \textsc{Struct}^{\#_t}$ *is a function returning the type of the given variable. [SA00025] (BPEL specification, page 9,45,46).*

Definition 4.3.3 (Allowed names for variables)
The set $\mathcal{N}_{wodot}$ is the set of NCNames not containing a dot (".").

$$\mathcal{N}_{wodot} \subset \mathcal{N}$$
$$\forall n \in \mathcal{N}_{wodot} : . \notin n$$

Definition 4.3.4 (Name of a variable)
The function $\mathsf{name}_{\mathcal{V}} : \mathcal{V} \to \mathcal{N}_{wodot}^{\#_t}$ assigns a name to a variable. A variable name is a NCName not containing a dot. [SA00024] (BPEL specification, pages 45,46).

Definition 4.3.5 (Declaration of variables)
The function $\mathsf{declareVar} : \mathcal{V} \to \mathcal{A}_{scope} \cup \{\bot\}$ is a function that declares a variable at a scope. (BPEL specification, pages 29,45). If a variable is implicitly declared, $\bot$ is returned.

Definition 4.3.6 (Uniqueness of variable naming)
The name of a variable must be unique among the names of all variables defined within the same scope. [SA00023] (BPEL specification, page 46).

$$\forall (v, w) \in \mathcal{V}, \mathsf{name}_{\mathcal{V}}(v) = \mathsf{name}_{\mathcal{V}}(w), \mathsf{declareVar}(v) = \mathsf{declareVar}(w) :$$
$$v = w \ \vee \ \mathsf{name}_{\mathcal{V}}(v) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathcal{V}}(w) \in \{\bot, \texttt{\#\#opaque}\}$$

Definition 4.3.7 (Variable visibility)
A variable can only be used in activities that are descendants of the scope the variable is declared in. The function $\mathsf{visible}_{\mathcal{V}} : \mathcal{A} \times \mathcal{V} \to \mathbb{B}$ returns true iff the given variable is visible at the given activity. That means, the variable is declared explicitly at an enclosing scope or implicitly declared at an enclosing event handler or enclosing for each activity. (BPEL specification, page 45,139).

$$\mathsf{visible}_{\mathcal{V}} : (a, v) \mapsto \begin{cases} \texttt{true} & \exists e \in \mathcal{E}_{message} : v = \mathsf{outputvar}(e) \wedge a \in \mathsf{clan}(e)) \\ \texttt{true} & \exists f \in \mathcal{A}_{forEach} : v = \mathsf{counter}(f) \wedge a \in \mathsf{clan}(f) \\ \texttt{true} & a \in \mathsf{clan}(\mathsf{declareVar}(v)) \\ \texttt{false} & \textit{otherwise} \end{cases}$$

$$\forall e \in \mathcal{E}_{message}, s \in \mathcal{A}_{scope}, (s, e, s') \in \mathsf{HR} : \mathsf{outputVar}(e) \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow$$
$$\neg \exists v \in \mathcal{V} : \mathsf{name}_{\mathcal{V}}(v) = \mathsf{name}_{\mathcal{V}}(\mathsf{outputVar}(e)) \ \wedge \ \mathsf{declarevar}(v) = s'$$

4.4 Standard Attributes for Activities

Activities may have a name assigned and may state whether they suppress a join failure. These are the standard attributes of all activities.

Definition 4.4.1 (Names of activities)
The function $\mathsf{name}_{\mathcal{A}} : \mathcal{A} \to \mathcal{N}^{\#_t}_{\perp^{e,o,t}}$ *assigns a name to an activity. (BPEL specification, pages 31,84).*

The WS-BPEL specification puts restrictions on the names of activities. These restrictions are covered by definition 5.9.21 and 5.9.22 on page 58.

Definition 4.4.2 (Suppressing join failures)
The function $\mathsf{supjoinf} : \mathcal{A} \to \mathcal{B}^{\#_t}_{\perp^{e,o,t}}$ *is a function assigning a boolean value to the* `suppressJoinFailure` *attribute of each activity. "*$\perp$*" is returned if the activity does not explicitly declare the* `suppressJoinFailure` *attribute. (BPEL specification, pages 23,31,84,103)*

Additionally, we define attributes which are valid at certain activities only.

Definition 4.4.3 (Instance creating activities)
The function $\mathsf{instance} : \mathcal{A}_{receive} \cup \mathcal{A}_{pick} \cup \mathcal{A}_{extensionActivity} \to \mathbb{B}^{\#_t}_{\perp^{e,o,t}}$ *is a function which assigns a boolean value to the attribute* `createInstance` *of a receive or a pick activity or an equivalent attribute of an extension activity. (BPEL specification, pages 34,90)*

Definition 4.4.4 (Start activities)
The set $\mathcal{A}_{start}$ *is the set of all instance creating activities.*

$$\mathcal{A}_{start} = \{a \mid \mathsf{instance}(a) = \mathtt{true}\}$$

4.5 Basic Activities

Definition 4.5.1 (Basic Activities)
Basic activities are atomic activities. They may not include child activities.

$$\forall s \in \mathcal{A}_{basic} : \mathsf{children}(s) = \emptyset$$

Regarding the hierarchy relation tree, basic activities are leaves of that tree.

4.6 Message Constructs

Definition 4.6.1 (Sending message activities)
The set $\mathcal{A}_{sending}$ is the set of activities, that can be used to send messages. It is called the sending message activities. It is made up of invoke and reply activities. (BPEL specification, pages 85,92)

$$\mathcal{A}_{sending} = \mathcal{A}_{invoke} \cup \mathcal{A}_{reply}$$

Definition 4.6.2 (Receiving message constructs)
The set $\mathcal{CO}_{receiving}$ is the set of activities and events, that can be used to receive messages. It is called the set of receiving message constructs. It is made up of invoke and receive activities as well as `onMessage` *events. (BPEL specification, pages 89,100)*

$$\mathcal{CO}_{receiving} = \mathcal{A}_{invoke_{twoWay}} \cup \mathcal{A}_{receive} \cup \mathcal{E}_{message}$$

The set $\mathcal{A}_{invoke_{twoWay}}$ corresponds to invokes calling a in-out operation and is formally defined in definition 4.9.4.

Definition 4.6.3 (Inbound message constructs)
The set of receive activities and `onMessage` *handlers $\mathcal{CO}_{IMA} = \mathcal{A}_{receive} \cup \mathcal{E}_{message}$ is the set of inbound message constructs. The WS-BPEL specification also refers to the set of inbound message constructs as the set of inbound message activities (IMA). (BPEL specification, page 89)*

Definition 4.6.4 (Message constructs)
The set of activities and events, that can be used to either send or receive messages from a process model, is called the set of message constructs.

$$\mathcal{CO}_{message} = \mathcal{A}_{sending} \cup \mathcal{CO}_{receiving}$$

Definition 4.6.5 (Partner links)
The set PL *is a set of partner links. (BPEL specification, page 37).*

Definition 4.6.6 (Type of a partner link)
The function $\mathsf{type}_{\mathrm{PL}} : \mathrm{PL} \rightarrow \mathrm{PLT}^{\#_t}$ *assigns a partner link type to a partner link. (BPEL specification, page 37).*

Definition 4.6.7 (MyRole of a partner link)
The function $\mathsf{myRole} : \mathrm{PL} \rightarrow \mathrm{ROLES}^{\#_t}_{\underline{|e|}, o, t}$ *assigns a role to a partner link, where the process uses the returned role as "my role". (BPEL specification, page 37).*

Definition 4.6.8 (PartnerRole of a partner link)
The function $\mathsf{partnerRole} : \mathrm{PL} \rightarrow \mathrm{ROLES}^{\#_t}_{\underline{|e|}, o, t}$ *assigns a role to a partner link, where the process uses the returned role as the role of the partner. (BPEL specification, page 37).*

Definition 4.6.9 (Validity of myRole and partnerRole assignments)
A partnerLink must specify the myRole, the partnerRole, or both. [SA00016] (BPEL specification, page 37)

$$\forall pl \in \mathrm{PL} : \mathsf{myRole}(pl) \neq \bot \vee \mathsf{partnerRole}(pl) \neq \bot$$

Definition 4.6.10 (Initialization of a partner role)
The function $\mathsf{initializePartnerRole} : \mathrm{PL} \to \mathbb{B}^{\#_t}_{\lfloor e,o,t}$ *assigns a Boolean value to the* ***initializePartnerRole*** *attribute of a partnerLink. An* ***initializePartnerRole*** *attribute can only be specified if a partnerRole is specified. [SA00017] (BPEL specification, page 38)*

$$\forall pl \in \mathrm{PL} : \mathsf{partnerRole}(pl) = \bot \Rightarrow \mathsf{initializePartnerRole}(pl) = \bot$$

Definition 4.6.11 (Assigning partner links to message constructs)
The function $\mathsf{partnerLink}_{\mathcal{CO}} : \mathcal{CO}_{message} \to \mathrm{PL}^{\#_t}$ *is a function assigning a partner link to a message construct. (BPEL specification, pages 86,89)*

Definition 4.6.12 (Declaration of partner links)
The function $\mathsf{declarePartnerLink} : \mathrm{PL} \to \mathcal{A}_{scope}$ *declares a partnerLink at a scope. (BPEL specification, page 29).*

Definition 4.6.13 (Partner link naming)
The function $\mathsf{name}_{\mathrm{PL}} : \mathrm{PL} \to \mathcal{N}^{\#_t}$ *assigns a name to a partner link. (BPEL specification, page 37).*

Definition 4.6.14 (Uniqueness of partner link naming)
The name of a partner link must be unique among the names of all partner links defined within the same immediately enclosing scope. [SA00018] (BPEL specification, page 38).

$$\begin{aligned}&\forall p, q \in \mathrm{PL} :\\ &(\mathsf{name}_{\mathrm{PL}}(p) = \mathsf{name}_{\mathrm{PL}}(q)) \wedge (\mathsf{declarePartnerLink}(p) = \mathsf{declarePartnerLink}(q)) \Rightarrow\\ &p = q \ \vee \ \mathsf{name}_{\mathrm{PL}}(p) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathrm{PL}}(q) \in \{\bot, \texttt{\#\#opaque}\}\end{aligned}$$

Definition 4.6.15 (Assigning operations to message constructs)
The function $\mathsf{operation}_{\mathcal{CO}} : \mathcal{CO}_{message} \to \mathcal{O}^{\#_t}$ *is a function assigning an operation to a message construct. (BPEL specification, pages 85,89). The operation has to be contained by the port type used by the message constructs. For sending activities, the operation has to be contained in the port type associated to the partnerRole of the partner link.*

$$\begin{aligned}&\forall a \in \mathcal{A}_{sending} :\\ &\mathsf{operation}_{\mathcal{CO}}(a) \neq \texttt{\#\#opaque} \wedge\\ &\mathsf{partnerLink}_{\mathcal{CO}}(a) \neq \texttt{\#\#opaque} \wedge\\ &\mathsf{partnerRole}(\mathsf{partnerLink}_{\mathcal{CO}}(a)) \neq \texttt{\#\#opaque} \Rightarrow\\ &\pi_2(\mathsf{partnerRole}(\mathsf{partnerLink}_{\mathcal{CO}}(a))) = \mathsf{operation}_{\mathcal{CO}}(a)\end{aligned}$$

Definition 4.6.16 (Direct declaration of port types)
The function $\mathsf{portType}_{\mathcal{CO}} : \mathcal{CO}_{message} \rightarrow \mathrm{PT}^{\#_t}_{\underline{\perp}^{e,o,t}}$ *declares a port type directly at a message construct. (BPEL specification, pages 24,25,28,85,89).*

Definition 4.6.17 (Validity of declarations of port types and operations)
If a port type is declared at a receiving activity it must correspond to the port type declared at the myRole of the corresponding partnerLink of the activity. (BPEL specification, page 24,28,89). [SA00005]

$$\begin{aligned}&\forall a \in (\mathcal{CO}_{receiving} \cup \mathcal{A}_{reply}) \setminus \mathcal{A}_{invoke_{twoWay}} :\\ &\mathsf{portType}_{\mathcal{CO}}(a) \notin \{\perp, \texttt{\#\#opaque}\} \Rightarrow\\ &\mathsf{portType}_{\mathcal{CO}}(a) = \pi_2(\mathsf{myRole}(\mathsf{partnerLink}_{\mathcal{CO}}(a))\end{aligned}$$

If a port type is declared at a sending activity, it must correspond to the port type declared at the partnerRole of the corresponding partnerLink of the activity. (BPEL specification, page 25). [SA00005]

$$\begin{aligned}&\forall a \in \mathcal{CO}_{sending} \setminus \mathcal{A}_{reply}\\ &\mathsf{portType}_{\mathcal{CO}}(a) \notin \{\perp, \texttt{\#\#opaque}\} \Rightarrow\\ &\mathsf{portType}_{\mathcal{CO}}(a) = \pi_2(\mathsf{partnerRole}(\mathsf{partnerLink}_{\mathcal{CO}}(a))\end{aligned}$$

Definition 4.6.18 (Input variables)
The function $\mathsf{inputVar} : \mathcal{A}_{sending} \rightarrow \mathcal{V}^{\#_{o,t}}_{\underline{\perp}^{e,o,t}}$ *assigns an input variable to a sending activity. (BPEL specification, page 85). The variable assigned to the sending activity must be visible at the sending activity.*

$$\forall a \in \mathcal{A}_{sending} : \mathsf{inputVar}(a) \notin \{\perp, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(a, \mathsf{inputVar}(a))$$

Definition 4.6.19 (Output variables)
The function $\mathsf{outputVar} : \mathcal{CO}_{receiving} \rightarrow \mathcal{V}^{\#_{o,t}}_{\underline{\perp}^{e,o,t}}$ *assigns an output variable to a receiving construct. (BPEL specification, page 85)*

The variable assigned to a receiving activity must be declared at an ancestor scope of this activity. This rule does not apply for on message handlers declared at a scope.

$$\begin{aligned}&\forall c \in \mathcal{CO}_{receiving} :\\ &\mathsf{outputVar}(c) \notin \{\perp, \texttt{\#\#opaque}\} \wedge\\ &\neg(c \in \mathcal{E}_{message} \ \wedge \ s \in \mathcal{A}_{Scope} \ \wedge \ \exists(s, c, a) \in \mathsf{LR}) \Rightarrow\\ &\mathsf{visible}_{\mathcal{V}}(c, \mathsf{outputVar}(c))\end{aligned}$$

On message handlers declared at a scope are treated differently. Their output variable is implicitly declared in the scope directly nested in the event handler element. The implicit declaration implies that in the directly nested scope of the handler, no variable with the same name as the output variable can be declared. [SA00095] (BPEL specification, page 139) See Definition 4.3.7.

Definition 4.6.20 (Validity of input variable declarations)
There are three possible declarations of the input variable of a sending activity: (i) The type is a messageType of the type of the input message of the operation. (ii) No input variable is defined. (iii) If the the only part of a message is declared as an element, then the variable can also be declared as an element. [SA00047], [SA00058], [SA00087] and [SA00090] (BPEL specification, pages 87,91,138,139)

$$
\begin{aligned}
&\forall s \in \mathcal{A}_{sending} : \\
&(\mathsf{type}_{\mathcal{V}}(\mathsf{inputVar}(s)) = \mathsf{type}_{\mathrm{MSG}}(\mathsf{msg}_{input}(\mathsf{operation}_{\mathcal{CO}}(s))) \vee \\
&\mathsf{inputVar}(s) \in \{\bot, \texttt{\#\#opaque}\} \vee \\
&(\left|\mathsf{MP}(\mathsf{msg}_{input}(\mathsf{operation}_{\mathcal{CO}}(s)))\right| = 1 \wedge \\
&\forall mp \in \mathsf{MP}(\mathsf{msg}_{input}(\mathsf{operation}_{\mathcal{CO}}(s))),\, t = \mathsf{type}_{\mathrm{MP}}(mp) : \\
&t \in \mathrm{EL} \wedge t = \mathsf{type}_{\mathcal{V}}(\mathsf{inputVar}(s)))
\end{aligned}
$$

Definition 4.6.21 (Validity of output variable declarations)
There are three possible declarations of the output variable of a receiving activity: (i) The type is a messageType of the type of the output message of the operation. (ii) No output variable is defined (iii) If the only part of a message is declared as an element, then the variable can also be declared as an element. [SA00048], [SA00058], [SA00087], and [SA00090] (BPEL specification, pages 87,91,138,139)

$$
\begin{aligned}
&\forall s \in \mathcal{CO}_{receiving} : \\
&(\mathsf{type}_{\mathcal{V}}(\mathsf{outputVar}(s)) = \mathsf{type}_{\mathrm{MSG}}(\mathsf{msg}_{output}(\mathsf{operation}_{\mathcal{CO}}(s))) \vee \\
&\mathsf{outputVar}(s) \in \{\bot, \texttt{\#\#opaque}\} \vee \\
&(\left|\mathsf{MP}(\mathsf{msg}_{output}(\mathsf{operation}_{\mathcal{CO}}(s)))\right| = 1 \wedge \\
&\forall mp \in \mathsf{MP}(\mathsf{msg}_{output}(\mathsf{operation}_{\mathcal{CO}}(s))),\, t = \mathsf{type}_{\mathrm{MP}}(mp) : \\
&t \in \mathrm{EL} \wedge t = \mathsf{type}_{\mathcal{V}}(\mathsf{outputVar}(s)))
\end{aligned}
$$

Definition 4.6.22 (Messaging constraints for start activities)
Start activities must not be part of notification or solicit-response message exchanges. [SA00001]

$$\forall a \in \mathcal{A}_{start} : \mathsf{operType}(\mathsf{operation}_{\mathcal{CO}}(a)) \notin \{\texttt{notification}, \texttt{solicit-response}\}$$

Definition 4.6.23 (From parts)
FP *is a relation assigning a receiving activity a set of from parts. A from part must be a part of the output message of the operation associated with this receiving activitiy. [SA00053] (BPEL specification, pages 85,91)*

$$\mathrm{FP} \subseteq \mathcal{CO}_{receiving} \times (\mathcal{V}^{\#_{o,t}}_{\bot^{e,\,o,\,t}}) \times (\mathrm{MP}^{\#_{o,t}}_{\bot^{e,\,o,\,t}})$$

$$\forall(a, v, mp) \in \mathrm{FP}, v, mp \neq \texttt{\#\#opaque} :$$
$$mp \in \mathsf{MP}(\mathsf{msg}_{output}(\mathsf{operation}_{\mathcal{CO}}(a))) \wedge \mathsf{type}_{\mathrm{MP}}(mp) = \mathsf{type}_{\mathcal{V}}(v)$$

The variable referenced in the from part must be visible at the construct the from part belongs to.

$$\forall(a, v, m) \in \mathrm{FP} : v \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(a, v)$$

Definition 4.6.24 (To parts)
TP *is a relation assigining a sending activity a set of to parts. A to part must be a part of the input message of the operation associated with this sending activitiy. (BPEL specification, page 85)*

$$\mathrm{TP} \subseteq \mathcal{A}_{sending} \times (\mathcal{V}^{\#_{o,t}}_{\perp^{e,o,t}}) \times (\mathrm{MP}^{\#_{o,t}}_{\perp^{e,o,t}})$$

$$\forall(a, v, mp) \in \mathrm{TP} : v, mp \neq \texttt{\#\#opaque} :$$
$$mp \in \mathsf{MP}(\mathsf{msg}_{input}(\mathsf{operation}_{\mathcal{CO}}(a))) \wedge \mathsf{type}_{\mathrm{MP}}(mp) = \mathsf{type}_{\mathcal{V}}(v)$$

The variable referenced in the to part must be visible at the activity the to part belongs to.

$$\forall(a, v, m) \in \mathrm{TP} : v \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(a, v)$$

Definition 4.6.25 (Validity of to part assignments)
For all message parts of a message corresponding to a non-opaque operation specified for a sending activity, there must be a corresponding to part if to parts are used in a sending activity. [SA00050] [SA00054] (BPEL specification, page 88)

$$\forall a \in \mathcal{A}_{sending} :$$
$$\mathsf{inputVar}(a) = \bot \Rightarrow$$
$$\forall mp \in \mathsf{MP}(\mathsf{msg}_{input}(\mathsf{operation}_{\mathcal{CO}}(a))) :$$
$$\exists(a, v, mp) \in \mathrm{TP} \ \vee \ \mathsf{operation}_{\mathcal{CO}}(a) = \texttt{\#\#opaque}$$

Definition 4.6.26 (Message types without parts for sending activities)
In case the WSDL definition of a message to be send by a sending activity does not contain any parts, no to part definitions are allowed for this activity (i.e., the size of the set of to parts for this activity must be zero).

$$\forall a \in \mathcal{A}_{sending} :$$
$$\mathsf{inputVar}(a) = \bot \wedge$$
$$\left|\mathsf{MP}(\mathsf{msg}_{input}(\mathsf{operation}_{\mathcal{CO}}(a)))\right| = 0 \ \wedge \ \mathsf{operation}_{\mathcal{CO}}(a) \neq \texttt{\#\#opaque} \ \Rightarrow$$
$$|\{f \in \mathrm{TP} \mid \pi_1(f) = a\}| = 0$$

Definition 4.6.27 (Message types without parts for receiving activities)
In case the WSDL definition of a message to be received by a receiving construct does not contain any parts, no from part definitions are allowed for this construct (i.e., the size of the set of from parts for this construct must be zero). [SA00047]. (BPEL specification, pages 89,92,100,138)

$$\forall a \in \mathcal{CO}_{receiving} : \mathsf{outputVar}(a)) = \bot \wedge$$
$$\left|\mathsf{MP}(\mathsf{msg}_{output}(\mathsf{operation}_{\mathcal{CO}}(a)))\right| = 0 \ \wedge \ \mathsf{operation}_{\mathcal{CO}}(a) \neq \texttt{\#\#opaque} \ \Rightarrow$$
$$|\{t \in \mathrm{FP} \mid \pi_1(t) = a\}| = 0$$

Definition 4.6.28 (Sending activities with to parts)
All sending activities that specify to parts must not specify an input variable seperately. This corresponds to [SA00051] and [SA00059]. (BPEL specification, pages 81,92).

$$\forall a \in \mathcal{A}_{sending} : (\exists p \in \mathrm{TP} \wedge \pi_1(p) = a) \Rightarrow \mathsf{inputVar}(a) = \bot$$

Definition 4.6.29 (Receiving constructs with from parts)
All receiving constructs that specify from parts must not specify an output variable seperately. This corresponds to [SA00052], [SA00055], [SA00063] and [SA00085]. (BPEL specification, pages 88,89,100,138)

$$\forall a \in \mathcal{CO}_{receiving} : (\exists p \in \mathrm{FP} \wedge \pi_1(p) = a) \Rightarrow \mathsf{outputVar}(a) = \bot$$

Definition 4.6.30 (Declaring a fault name at a reply activity)
The function $\mathsf{faultName}_{reply} : \mathcal{A}_{throw} \mapsto \mathrm{QNAME}^{\#_t}_{\bot_{e,\,o,\,t}}$ *assigns a fault name to a reply activity. (BPEL specification, page 93). The fault name is not required to be defined prior to the reply activity.*

Definition 4.6.31 (Validity of fault name declarations at a reply activity)
If a fault name is assigned to a reply activity, then it has to match a fault name of the operation. (BPEL specification, page 93)

$$\forall r \in \mathcal{A}_{reply} :$$
$$\mathsf{faultName}_{reply} \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{faultName}_{reply} \in \mathsf{msg}_{fault}(\mathsf{operation}_{\mathcal{CO}}(r))$$

Definition 4.6.32 (Validity of fault variable declarations at a reply activity)
If a fault name is assigned to a reply activity and a variable is declared, the type of the variable should match the type of the fault variable of the operation. (BPEL specification,

page 93)

$$\forall r \in \mathcal{A}_{reply} :$$
$$\mathsf{faultName}_{reply} \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow$$
$$\exists f \in \mathsf{msg}_{fault}(\mathsf{operation}_{\mathcal{CO}}(r)) : (\mathsf{type}_{\mathcal{V}}(\mathsf{outputVar}(r)) = \mathsf{type}_{\mathrm{MSG}}(f) \vee$$
$$\mathsf{outputVar}(r) \in \{\bot, \texttt{\#\#opaque}\} \vee$$
$$(|\mathsf{MP}(f)| = 1 \wedge$$
$$\forall mp \in \mathsf{MP}(f), t = \mathsf{type}_{\mathrm{MP}}(mp) : t \in \mathrm{EL} \wedge t = \mathsf{type}_{\mathcal{V}}(\mathsf{outputVar}(r)))$$

4.7 Message Exchange

Definition 4.7.1 (The set of message exchanges)
The set MEX *is the set of message exchanges. (BPEL specification, page 93)*

Definition 4.7.2 (Declaration of message exchanges)
The function declareMEX*:* $\mathrm{MEX} \rightarrow \mathcal{A}_{scope}$ *declares a message exchange at a scope. (BPEL specification, pages 29,117).*

Definition 4.7.3 (The name of a message exchange)
The function $\mathsf{name}_{\mathrm{MEX}} : \mathrm{MEX} \rightarrow \mathcal{N}^{\#_t}$ *assigns a name to a message exchange.*

Definition 4.7.4 (Uniqueness of message exchange naming)
The (non-opaque) name of a message exchange must be unique among the names of all message exchanges defined within the same immediately enclosing scope. [SA0000061] (BPEL specification, page 93)

$$\forall c, d \in \mathrm{MEX} :$$
$$\mathsf{name}_{\mathrm{MEX}}(c) = \mathsf{name}_{\mathrm{MEX}}(d) \wedge \mathsf{declareMEX}(c) = \mathsf{declareMex}(d) \Rightarrow$$
$$c = d \ \vee \ \mathsf{name}_{\mathrm{MEX}}(c) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathrm{MEX}}(d) \in \{\bot, \texttt{\#\#opaque}\}$$

Definition 4.7.5 (Assigning message exchanges to messaging constructs)
The function $\mathsf{mex} : \mathcal{CO}_{IMA} \cup \mathcal{A}_{reply} \rightarrow \mathrm{MEX}^{\#_t}_{\bot^{e,\,o,\,t}}$ *assigns a message exchange to an inbound message activity or a reply activity.*

The messaging construct, the message exchange is assigned to, must be in the descendants of the scope, where the message exchange is declared at. [SA00061] (BPEL specification, pages 93,100)

$$\forall a \in \mathcal{CO}_{IMA} \cup \mathcal{A}_{reply} : m \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow a \in \mathsf{descendants}(\mathsf{declareMEX}(m))$$

Definition 4.7.6 (Activities with the same associated message exchange)
The set $\mathcal{A}_{mx}$ is the set of activities with the same associated message exchange.

$$\mathcal{A}_{mx} = \{a \in \mathcal{CO}_{IMA} \cup \mathcal{A}_{reply} \mid \mathsf{mex}(a) = mx\}$$

Definition 4.7.7 (Parallel receive/reply pairs)
The explicit use of message exchange is needed only where the execution can result in multiple IMA-reply pairs on the same partnerLink and operation being executed simultaneously. [SA00060] (BPEL specification, pages 24,25,28,93). In these cases, the process definition MUST explicitly mark the pairing-up relationship.

$$\begin{aligned}
&\forall (i,r),(j,s) \in \mathcal{CO}_{IMA} \times \mathcal{A}_{reply}, (i,r) \neq (j,s), i \neq j, r \neq s : \\
&\mathsf{operation}_{\mathcal{CO}}(i) = \mathsf{operation}_{\mathcal{CO}}(j) \wedge \mathsf{partnerLink}_{\mathcal{CO}}(i) = \mathsf{partnerLink}_{\mathcal{CO}}(j) \wedge \\
&\mathsf{possibleRunInParallel}(r,s) \Rightarrow \\
&\mathsf{mex}(i) \neq \mathsf{mex}(j) \wedge \\
&((\mathsf{mex}(i) = \mathsf{mex}(r) \wedge \mathsf{mex}(j) = \mathsf{mex}(s)) \vee \\
&(\mathsf{mex}(i) = \mathsf{mex}(s) \wedge \mathsf{mex}(j) = \mathsf{mex}(r))) \wedge \\
&\mathsf{mex}(i) \neq \bot \wedge \mathsf{mex}(r) \neq \bot \wedge \mathsf{mex}(j) \neq \bot \wedge \mathsf{mex}(s) \neq \bot
\end{aligned}$$

possibleRunInParallel*(r,s) denotes that r and s are possibly executed in parallel. To determine possible simultaneous execution is part of further work and out of this scope for this document.*

4.8 Correlation

Definition 4.8.1 (The set of correlation sets)
The set CORSET *denotes the set of all correlation sets. A correlation set is a set of properties. [SA00045] (BPEL specification, pages 75,76).*

$$\mathrm{CORSET} \subseteq 2^{\mathrm{PROP}_{simple}},\ \mathrm{CORSET} \neq \emptyset$$

Definition 4.8.2 (Declaration of correlation sets)
The function declareCor*:* $\mathrm{CORSET} \rightarrow \mathcal{A}_{scope}$ *declares a correlation set at a scope. (BPEL specification, page 29). (BPEL specification, page 75).*

Definition 4.8.3 (The name of a correlation set)
The function $\mathsf{name}_{\mathrm{CORSET}} : \mathrm{CORSET} \rightarrow \mathcal{N}^{\#_t}$ *assigns a name to a correlation set. (BPEL specification, page 75).*

Definition 4.8.4 (Uniqueness of correlation set names)
The name of a correlation set must be unique among the names of all "correlation sets" defined within the same immediately enclosing scope. [SA00044] (BPEL specification, page 75).

$$\forall c, d \in \text{CORSET}:$$
$$\mathsf{name}_{\text{CORSET}}(c) = \mathsf{name}_{\text{CORSET}}(d) \wedge \mathsf{declareCor}(c) = \mathsf{declareCor}(d) \Rightarrow$$
$$c = d \ \vee \ \mathsf{name}_{\text{CORSET}}(c) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\text{CORSET}}(d) \in \{\bot, \texttt{\#\#opaque}\}$$

Definition 4.8.5 (Correlation patterns)
A correlation pattern denotes, where a correlation applies. The set PATTERN *contains the possible values.*

$$\text{PATTERN} = \{\texttt{request}, \texttt{response}, \texttt{request-response}\}$$

The specification of correlation patterns is required in two-way operations and not allowed in one-way operation. [SA00046] (BPEL specification, page 78).

$$\forall (c, p, a) \in \text{CORRELATION} : p \neq \bot \Leftrightarrow a \in \mathcal{A}_{invoke_{twoWay}}$$

Definition 4.8.6 (Possible correlation set initiations)
The set CorInit *contains the possible initiations of correlation sets. (BPEL specification, page 76).*

$$\text{CorInit} = \{\texttt{yes}, \texttt{no}, \texttt{join}\}$$

Definition 4.8.7 (Assigning correlation to a messaging construct)
The relation $\text{CORRELATION} \subseteq \text{CORSET} \times \text{PATTERN}^{\#_t}_{\bot_{e,\,o,\,t}} \times \text{CorInit}^{\#_t}_{\bot_{e,\,o,\,t}} \times \mathcal{CO}_{message}$ *assigns a correlation set to messaging constructs. The assignment states how the correlation set has to be initiated. (BPEL specification, pages 76,85).*

If a correlation set is assigned to a messaging construct, this messaging construct must be in the set of descendants of the scope, where the correlation set has been declared.

$$c = (s, p, i, a) \in \text{CORRELATION} \Rightarrow a \in \mathsf{descendants}(\mathsf{declareCor}(s))$$

Definition 4.8.8 (Correlation sets of an activity)
The function $\mathsf{COR} : \mathcal{CO}_{message} \rightarrow 2^{\text{CORSET}}$ *returns the "correlation sets" declared at an activity.*

$$\mathsf{COR} : a \mapsto \{s \,|\, (s, p, i, a) \in \text{CORRELATION}\}$$

Definition 4.8.9 (Correlation sets at start activities)
If a process has multiple start activities with correlation sets then all such activities must share at least one common correlationSet and all common correlationSets defined on all the activities must have the value of the initiate attribute be set to "`join`". [SA00057]. (BPEL specification, pages 34,90).

$$(|\mathcal{A}_{start}| > 1) \ \wedge \ \bigcup_{a \in \mathcal{A}_{start}} \mathsf{COR}(a) \neq \emptyset \Rightarrow$$
$$\bigcap_{a \in \mathcal{A}_{start}} \mathsf{COR}(a) = S \ \wedge \ S \neq \emptyset \ \wedge$$
$$\forall c = (s, p, i, a) \in \text{CORRELATION} : s \in S \ \wedge \ a \in \mathcal{A}_{start} \Rightarrow i = \texttt{join}$$

4.9 Invoke

"The *invoke* activity is used to call Web Services offered by service providers. The typical use is invoking an operation on a service, which is considered a basic activity. The *invoke* activity can enclose other activities, inlined in compensation handler and fault handlers, as detailed below. Operations can be request-response or one-way operations, corresponding to WSDL 1.1 operation definitions. WS-BPEL uses the same basic syntax for both, with some additional options for the request-response case." (BPEL specification, page 84)

Definition 4.9.1 (Fault and compensation handlers in invoke activities)
Fault and compensation handlers can be declared at invoke activities. Regarding the relation HR *introduced earlier, this means that invoke activities can have directly nested activities, but only those that are connected via either a fault or compensation event label. (BPEL specification, page 85)*

$$\mathsf{HR} \cap (\mathcal{A}_{invoke} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{invoke} \times (\mathcal{E}_{fault} \cup \mathcal{E}_{compensation}) \times \mathcal{A})$$

Definition 4.9.2 (Amount of fault handlers declared at an invoke)
Zero or more fault handlers can be declared at an invoke. (BPEL specification, pages 85,86).

$$\forall a \in \mathcal{A}_{invoke} : |\mathsf{HR} \cap (\{a\} \times \mathcal{E}_{fault} \times \mathcal{A})| \geq 0$$

Definition 4.9.3 (Amount of compensation handlers declared at an invoke)
At most one compensation handler can be declared at an invoke. (BPEL specification, pages 85,86).

$$\forall a \in \mathcal{A}_{invoke} : |\mathsf{HR} \cap (\{a\} \times \mathcal{E}_{compensation} \times \mathcal{A})| \leq 1$$

Definition 4.9.4 (Two-Way Invokes)
The set $\mathcal{A}_{invoke_{twoWay}}$ *denotes the set of invoke activities that have an operation associated with an input and output message that is of type* `request-response`*. (BPEL specification, page 86)*

$$\begin{aligned}\mathcal{A}_{invoke_{twoWay}} = \{i \mid i \in \mathcal{A}_{invoke},\\ \mathsf{msg}_{input}\mathsf{operation}_{\mathcal{CO}}(a) \neq \bot,\\ \mathsf{msg}_{output}\mathsf{operation}_{\mathcal{CO}}(a) \neq \bot,\\ \mathsf{operType}(\mathsf{operation}_{\mathcal{CO}}(a)) = \texttt{request-response}\end{aligned}$$

4.10 Receive and Reply

Receive and Reply activities do not need additional definitions over the definitions already presented for all activities and especially those for message constructs. Therefore no additional definitions are presented here.

4.11 Assign

An assign activity consists of a set of copy operations. (BPEL specification, page 59).

Definition 4.11.1 (Copy operations)
The set Copy *is the set of copy operations. A copy operation is a pair consisting of a from spec and a to spec. From and to specs are defined in Definition 4.11.5 and 4.11.14. (BPEL specification, page 26).* $\textsc{Copy} \subseteq \textsc{From} \times \textsc{To}$

Definition 4.11.2 (Extension assign operations)
The set ExtAssOp *is the set of all extension assign operations. (BPEL specification, page 26).*

Definition 4.11.3 (Assigning copy operations to assign activities)
The function $\mathsf{assignCopy} : \mathcal{A}_{assign} \rightarrow 2^{\textsc{Copy}} \cup 2^{\textsc{ExtAssOp}}$ *assigns copy operations to an assign activity. (BPEL specification, page 26,31).*

Definition 4.11.4 (Assigning from specs to variables declarations)
The function $\mathsf{fromSpec}_{\mathcal{V}} : \mathcal{V} \rightarrow \textsc{From}$ *assigns a from specs to a variable. (BPEL specification, page 45).*

Definition 4.11.5 (From specs)
The set From *is the set of from specs. The following types of from specs exist: [SA00032] (BPEL specification, page 60).*

1. *Variable variant. The set of from specs of the variable variant is denoted by* $\textsc{From}_{var}$.
2. *PartnerLink variant. The set of from specs of the partner link variant is denoted by* $\textsc{From}_{pl}$.
3. *Property variant. The set of from specs of the property variant is denoted by* $\textsc{From}_{prop}$.
4. *Expression variant. The set of from specs of the expression variant is denoted by* $\textsc{From}_{ex}$.
5. *Literal variant. The set of from specs of the literal variant is denoted by* $\textsc{From}_{lit}$.
6. *Empty variant. The set of from specs of the empty variant is denoted by* $\textsc{From}_{empty}$.
7. *Opaque variant. In abstract processes the set of from specs of the opaque variant is denoted by* $\textsc{From}_{opaque}$.

$$\textsc{From} = \textsc{From}_{var} \cup \textsc{From}_{pl} \cup \textsc{From}_{prop} \cup \textsc{From}_{ex} \cup \textsc{From}_{lit} \cup \textsc{From}_{empty} \cup \textsc{From}_{opaque}$$

Definition 4.11.6 (Type of a from spec)
The function $\mathsf{type}_{\textsc{From}} : \textsc{From} \rightarrow \textsc{types}$ *returns the type of the construct referenced by the from spec.*

Definition 4.11.7 (Variable variant from specs)
The variable variant from spec is a 3-tuple consisting of a variable, an optional part attribute (for variables of the WSDL message type), and an optional query attribute. All three attributes can be omitted or opaque in abstract processes. This omission is valid for all attributes in all types of from specs and we will therefore not explicitly mention this in the other descriptions but only include it in the formal definitions. (BPEL specification, page 60)

$$\textsc{From}_{var} \subseteq \mathcal{V}^{\#_t} \times \mathrm{MP}^{\#_t}_{\lfloor e,o,t} \times \mathcal{Q}^{\#_t}_{\lfloor e,o,t}$$

If the variable of a from spec is of a message type the part attribute might point to one of the parts of the message. [SA00034] (BPEL specification, page 60).

$$\forall (v,p,q) \in \textsc{From}_{var} : v \in \mathrm{MSG} \Rightarrow p \in \mathrm{MP}$$

If the type of the variable is not defined using XML the part attribute must be empty.

$$\forall (v,p,q) \in \textsc{From}_{var} : v \notin \textsc{XMLstruct} \Rightarrow p = \bot$$

The variable referenced in the variable variant from spec must be visible at the activity where the from spec is declared in a copy operation.

$$\forall f = (v, p, q) \in \textsc{From}_{var},\ c = (f, t) \in \textsc{Copy} :$$
$$v \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(\mathsf{assignCopy}(c), v)$$

Definition 4.11.8 (Partner link variant from specs)
The partner link variant from spec is a pair consisting of a partnerLink and a role (myRole or partnerRole). (BPEL specification, page 61).

$$\textsc{From}_{pl} \subseteq \mathrm{PL}^{\#_t} \times \{\texttt{myRole}, \texttt{partnerRole}\}^{\#_t}$$

If the value `myRole` *or* `partnerRole` *is used, the corresponding partner link declaration must specify the corresponding* `myRole` *or* `partnerRole`*. [SA00035] [SA00036]*

$$\forall f = (p, r) \in \textsc{From}_{pl} : r = \texttt{myrole} \Rightarrow \mathsf{myRole}(p) \neq \bot$$
$$\forall f = (p, r) \in \textsc{From}_{pl} : r = \texttt{partnerrole} \Rightarrow \mathsf{partnerRole}(p) \neq \bot$$

Definition 4.11.9 (Property variant from specs)
A from spec of the property variant is a pair consisting of a variable and a property. (BPEL specification, page 61).

$$\textsc{From}_{prop} \subseteq \mathcal{V}^{\#_t} \times \mathrm{PROP}^{\#_t}$$

The variable referenced in the property variant from spec must be visible at the activity where the from spec is declared in a copy operation.

$$\forall f = (v, m) \in \textsc{From}_{prop},\ c = (f, t) \in \textsc{Copy} :$$
$$v \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(\mathsf{assignCopy}(c), v)$$

Definition 4.11.10 (Expression variant from specs)
A from spec of the expression variant is an expression. (BPEL specification, page 61).

$$\textsc{From}_{ex} \subseteq \mathrm{Ex}^{\#_t}$$

Definition 4.11.11 (Literal variant from specs)
A from spec of the literal variant is an arbitrary text or XML-Element. [SA00038] (BPEL specification, page 62).

$$\textsc{From}_{lit} \subseteq \textsc{Txt} \cup \mathrm{EL}$$

Definition 4.11.12 (Empty variant from specs)
An empty variant from spec is used to declare custom from specs, we therefore do not define the syntax here. (BPEL specification, page 62)

Definition 4.11.13 (Opaque variant from specs)
An opaque from spec does not have any attributes, therefore there is no syntax to define here. (BPEL specification, page 151)

Definition 4.11.14 (To specs)
The set $\textsc{To}$ *is the set of to specs. The following types of to specs exist. [SA00032] (BPEL specification, page 60).*

1. *Variable variant. The set of to specs of the variable variant is denoted by* $\textsc{To}_{var}$.
2. *PartnerLink variant. The set of to specs of the partner link variant is denoted by* $\textsc{To}_{pl}$.
3. *Property variant. The set of to specs of the property variant is denoted by* $\textsc{To}_{prop}$.
4. *Expression variant. The set of to specs of the expression variant is denoted by* $\textsc{To}_{ex}$.

$$\textsc{To} = \textsc{To}_{var} \cup \textsc{To}_{pl} \cup \textsc{To}_{prop} \cup \textsc{To}_{ex}$$

A to spec must return an lvalue:

$$\forall t \in \textsc{To} : \mathsf{type}_{\textsc{To}}(t) = \mathbb{L}$$

In addition, a XPath query used in a to spec must begin with a XPath variable reference. [SA00033]

Note that we do not formalize the syntax of XPath and thus did not supply a formal definition of [SA00033].

Definition 4.11.15 (Type of a to spec)
The function $\mathsf{type}_{\textsc{To}} : \textsc{To} \to \textsc{types}$ *returns the type of the construct referenced by the to spec.*

Definition 4.11.16 (Variable variant to specs)
The variable variant to spec is a 3-tuple consisting of a variable, an optional part attribute (for variables of the WSDL message type), and an optional query attribute. All three attributes can be omitted or opaque in abstract processes. Again, as with the from specs this is true for all attributes in all types of to specs and we will therefore not explicitly mention this in the other descriptions but only include it in the formal definitions. (BPEL specification, page 60)

$$\textsc{To}_{var} \subseteq \mathcal{V}^{\#_t} \times \mathrm{MP}^{\#_t}_{\underline{\perp}^{e,\,o,\,t}} \times \mathcal{Q}^{\#_t}_{\underline{\perp}^{e,\,o,\,t}}$$

The variable referenced in the variable variant to spec must be visible at the activity where the to spec is declared in a copy operation.

$$\forall t = (v, m, q) \in \textsc{To}_{var},\ c = (f, t) \in \textsc{Copy}:$$
$$v \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(\mathsf{assignCopy}(c), v)$$

Definition 4.11.17 (Partner link variant to specs)
The partner link variant to spec specifies a partnerLink. (BPEL specification, page 61)

$$\textsc{To}_{pl} \subseteq \mathrm{PL}^{\#_t}$$

The partner link referenced in a to spec must specify a partner role. [SA00037]

$$\forall p \in \textsc{To}_{pl} : \mathsf{partnerRole}(p) \neq \bot$$

Definition 4.11.18 (Property variant to specs)
A to spec of the property variant is a pair consisting of a variable and a property. (BPEL specification, page 61)

$$\textsc{To}_{prop} \subseteq \mathcal{V}^{\#_t} \times \mathrm{PROP}^{\#_t}$$

The variable referenced in the property variant to spec must be visible at the activity where the to spec is declared in a copy operation.

$$\forall t = (v, m) \in \textsc{To}_{prop}, c = (f, t) \in \textsc{Copy}$$
$$v \notin \{\bot, \texttt{\#\#opaque}\} \Rightarrow \mathsf{visible}_{\mathcal{V}}(\mathsf{assignCopy}(c), v)$$

Definition 4.11.19 (Expression variant to specs)
A to spec of the expression variant is an expression. (BPEL specification, page 61)

$$\textsc{To}_{ex} \subseteq \textsc{Ex}^{\#_t}$$

Definition 4.11.20 (Keeping source element names)
The function keepSrcElementName $\textsc{Copy} \to \mathbb{B}^{\#_t}_{\underline{\{e,\, o,\, t}}$ *assigns a Boolean, empty, or opaque value to the* `keepSrcElementName` *attribute of a copy operation. (BPEL specification, page 63).*

The `keepSrcElementName` *attribute may only be set when the result of both from-spec and to-spec are EIIs (cf. Definition 2.1.6). [SA00042]*

$$\forall c = (f, t) \in \textsc{Copy}:$$
$$\mathsf{keepSrcElementName}(c) \neq \bot \Rightarrow$$
$$\mathsf{type}_{\textsc{From}}(f) \in \mathrm{EII} \wedge \mathsf{type}_{\textsc{To}}(t) \in \mathrm{EII}$$

Definition 4.11.21 (Ignoring missing from Data)
The function ignoreMissingFromData : COPY $\rightarrow \mathbb{B}^{\#_t}_{\underline{\perp}_{e,\,o,\,t}}$ *assigns a Boolean, empty, or opaque value to the* `ignoreMissingFromData` *attribute of a copy operation. (BPEL specification, page 63).*

Definition 4.11.22 (Variable validation)
The function $\mathsf{validate}_{\mathcal{A}_{assign}} : \mathcal{A}_{assign} \rightarrow \mathbb{B}^{\#_t}_{\underline{\perp}_{e,\,o,\,t}}$ *assign a Boolean, empty, or opaque value to the* `validate` *attribute of a copy operation. (BPEL specification, page 63).*

4.12 Validate

The `validate` activity is used for validation of variables. (BPEL specification, page 48).

Definition 4.12.1 (Variables to be validated)
The function valvar : $\mathcal{A}_{validate} \rightarrow (2^{\mathcal{V}})^{\#_t}$ *assigns the variables to be validated to a validate activity. (BPEL specification, page 26).*

4.13 Throw

"The `throw` activity is used when a business process needs to signal an internal fault explicitly. A fault MUST be identified with a QName. The `throw` activity provides the name for the fault, and can optionally provide data with further information about the fault. A fault handler can use such data to handle the fault and to populate any fault messages that need to be sent to other services." (BPEL specification, page 94)

Definition 4.13.1 (Declaring a fault name at a throw activity)
The function $\mathsf{faultName}_{\mathcal{A}_{throw}} : \mathcal{A}_{throw} \mapsto \text{QNAME}^{\#_t}$ *assigns a fault name to a throw activity. The fault name is not required to be defined prior to the throw activity. (BPEL specification, pages 26,94).*

Definition 4.13.2 (Declaring a fault variable at a throw activity)
The function faultVariable : $\mathcal{A}_{throw} \mapsto \mathcal{V}^{\#_t}_{\underline{\perp}_{e,\,o,\,t}}$ *assigns a fault variable to a throw activity. The throw activity must be a descendant of the scope the variable has been declared in. (BPEL specification, page 94)*

$$\forall a \in \mathcal{A}_{throw} :$$
$$\mathsf{faultVariable}(a) = v \ \wedge \ v \notin \{\perp, \texttt{\#\#opaque}\} \Rightarrow a \in \mathsf{descendants}(\mathsf{declareVar}(v))$$

4.14 Wait

The `wait` activity specifies a delay for a certain period of time or until a certain deadline is reached. (BPEL specification, page 95)

Definition 4.14.1 (Assigning a duration to a wait activity)
The function $\mathsf{for}_W : \mathcal{A}_{wait} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{duration,\underline{\perp}^{e,\,o,\,t}}$ *assigns a duration to a wait activity. (BPEL specification, pages 27,95).*

Definition 4.14.2 (Assigning a date to a wait activity)
The function $\mathsf{until}_W : \mathcal{A}_{wait} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{date,\underline{\perp}^{e,\,o,\,t}}$ *assigns a date to a wait activity. (BPEL specification, page 95)*

Definition 4.14.3 (Validity of date and duration assignments)
Either a duration or a date is assigned to a wait activity but not both. (BPEL specification, page 27).

$$\forall a \in \mathcal{A}_{wait} : (\mathsf{for}_W(a) = \perp \vee \mathsf{until}_W(a) = \perp)$$

4.15 Empty

"There is often a need to use an activity that does nothing, for example when a fault needs to be caught and suppressed. The `empty` activity is used for this purpose. Another use of `empty` is to provide a synchronization point in a `flow`." (BPEL specification, page 95)

Empty activities do not require any definition exceeding the standard definitions for all activities. We therefore do not introduce any definitions here. (BPEL specification, pages 27, 95).

4.16 Extension Activities

"A WS-BPEL process definition can include new activities, which are not defined by this specification, by placing them inside the `extensionActivity` element. These activities are known as extension activities. The contents of an `extensionActivity` element MUST be a single element qualified with a namespace different from WS-BPEL namespace."(BPEL specification, page 95)

The WS-BPEL specification defines the syntax of a extension activity as presented in Listing 5. (BPEL specification, pages 30,31,95). Note that the standard attributes and standard elements are not placed at the `extensionActivity` element, but in the nested `anyElementQName` element. The `extensionActivity` element does not have any attributes and `anyElementQName` as the only child. If an activity a has the type `extensionActivity`, the contents of the standard attributes and standard elements are directly assigned to a. Thus, we merge `extensionActivity` and `anyElementQName` together.

To include new activities in the formalization, subtypes of `extensionActivity` have to be introduced.

If an extension activity a is an instance creating activity ($\mathsf{createInstance}(a) = \texttt{true}$), it must exhibit the behavior of consuming an inbound message. (BPEL specification, page 96)

Listing 5 Extension Activity as defined in WS-BPEL

```
<extensionActivity>
  <anyElementQName standard-attributes>
    standard-elements
  </anyElementQName>
</extensionActivity>
```

4.17 Exit

"The `exit` activity is used to immediately end the business process instance. All currently running activities MUST be ended immediately without involving any termination handling, fault handling, or compensation behavior."(BPEL specification, page 96)

Exit activities do not require any definition exceeding the standard definitions for all activities. We therefore do not introduce any definitions here. (BPEL specification, pages 30,96).

4.18 Rethrow

"The `rethrow` activity is used in fault handlers to rethrow the fault they caught, i.e. the fault name and, where present, the fault data of the original fault. It can be used only within a fault handler" (BPEL specification, page 96)

Rethrow activities do not require any definition regarding their syntax exceeding the standard definitions for all activities. Only one new definition needs to be introduced.

Definition 4.18.1 (Occurence of rethrow activities)
Rethrow activities are only allowed to occur inside fault handlers. Regarding the hierarchy relation, each rethrow activity must be the descendant of an activity that contains a fault event label in the connection to it's father activity. [SA00006] (BPEL specification, pages 30,96).

$$\forall a \in \mathcal{A}_{rethrow}\ \exists (b, e, c) \in \mathsf{HR} : e \in \mathcal{E}_{fault} \wedge a \in \mathsf{descendants}(e)$$

Chapter 5

Structured Activities

Structured activities are activities that may contain child activities. In terms of the hierarchy relation, there can exist activities that are attached to a structured activity via an empty, a conditional or an event label. Some structured activities restrict one of those labels.

Definition 5.0.2 (Children of structured activities)
Structured activities always contain children.

$$\forall s \in \mathcal{A}_{structured} : \mathsf{children}(s) \neq \emptyset$$

5.1 Sequence

"A `sequence` activity contains one or more activities that are performed sequentially, in the lexical order in which they appear within the `sequence` element. The `sequence` activity completes when the last activity in the sequence has completed." (BPEL specification, page 98)

Definition 5.1.1 (Child activities of a sequence)
A sequence can have an arbitrary number of directly nested activities connected to the sequence through the empty label. (BPEL specification, page 98)

$$\mathsf{HR} \cap (\mathcal{A}_{sequence} \times B \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{sequence} \times \{\bot\} \times A)$$

Definition 5.1.2 (Set of all orders of children of sequence activities)
There exists a strict total order over the children of a sequence. (BPEL specification, pages 27,98). The set $\Omega_{seq} = \{<^s_{seq} \mid s \in \mathcal{A}_{sequence}\}$ *contains all strict total orders over the children of a sequence.*

Definition 5.1.3 (Strict total order of children of a sequence activity)
$<^s_{seq}$ defines a strict total order of all children of a sequence activity s.

$$\begin{aligned}\forall s \in \mathcal{A}_{sequence}: \quad & (a,x) \in <^s_{seq} \wedge (x,b) \in <^s_{seq} \Rightarrow (a,b) \in <^s_{seq} \\ & \forall (a,b) \in <^s_{seq}: a \in \mathsf{children}(s) \wedge b \in \mathsf{children}(s) \\ & (a,b) \in <^s_{seq} \wedge (a,x) \in <^s_{seq}: b = x \\ & (a,b) \in <^s_{seq} \wedge (x,b) \in <^s_{seq}: a = x \\ & \forall c \in \mathsf{children}(s): \exists (c,x) \in <^s_{seq} \vee (x,c) \in <^s_{seq} \\ & (a,b) \in <^s_{seq} \Rightarrow (b,a) \notin <^s_{seq}\end{aligned}$$

5.2 If Activity

"The `if` activity provides conditional behavior. The activity consists of an ordered list of one or more conditional branches defined by the `if` and optional `elseif` elements, followed by an optional `else` element. The `if` and `elseif` branches are considered in the order in which they appear. The first branch whose `condition` holds true is taken, and its contained activity is performed. If no branch with a condition is taken, then the `else` branch is taken if present."(BPEL specification, page 99)

Definition 5.2.1 (Branches of an if activity)
An if activity can have an arbitrary set of branches. The first branch is called the if branch, the last one may be an `else` branch. The other branches are `elseif` branches. Regarding the hierarchy relation, the conditions assigned to the respective branches are the labels of the relation HR *that connects the if activity with the topmost activity of the branch. In executable processes only the last branch might be connected to the if activity with an empty label, denoting that this is the else branch. We will introduce this restriction in definition 6.1.6 on page 62 in the section on executable processes. In the common base every condition can be empty or opaque. (BPEL specification, pages 27,99).*

$$\mathsf{HR} \cap (\mathcal{A}_{if} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{if} \times \mathcal{C}^{\#_{o,t}}_{\perp^{e,o,t}} \times \mathcal{A})$$

Definition 5.2.2 (Set of all orders of children of if activities)
There exists a strict total order over the children of an if activity. (BPEL specification, page 99) The set $\Omega_{if} = \{<^i_{if} \mid i \in \mathcal{A}_{if}\}$ contains all strict total orders over the children of an if activity.

Definition 5.2.3 (Strict total order of children of an if activity)
$<^i_{if}$ defines a strict total order of all children of an if activity i.

$$\begin{aligned}\forall i \in \mathcal{A}_{if} : \quad & (a,x) \in <^i_{if} \wedge (x,b) \in <^i_{if} \Rightarrow (a,b) \in <^i_{if} \\ & \forall (a,b) \in <^i_{if} : a \in \mathsf{children}(i) \wedge b \in \mathsf{children}(i) \\ & (a,b) \in <^i_{if} \wedge (a,x) \in <^i_{if} : b = x \\ & (a,b) \in <^i_{if} \wedge (x,b) \in <^i_{if} : a = x \\ & \forall c \in \mathsf{children}(i) : \exists (c,x) \in <^i_{if} \vee (x,c) \in <^i_{if} \\ & (a,b) \in <^i_{if} \Rightarrow (b,a) \notin <^i_{if}\end{aligned}$$

5.3 While

"The `while` activity provides for repeated execution of a contained activity. The contained activity is executed as long as the Boolean `condition` evaluates to true at the beginning of each iteration."(BPEL specification, page 99)

Definition 5.3.1 (Children of a while activity)
A while activity can have exactly one child activity. In terms of the hierarchy relation: There exists exactly one relation connecting the while activity to another (possibly structured) activity. The label for this relation can be a condition, empty or opaque for abstract processes. For executable processes the label is restricted to a condition, this restriction is defined in the section on executable processes. The condition denotes the condition that specifies whether the child activity is executed or not. (BPEL specification, pages 27,99).

$$\forall a \in \mathcal{A}_{while} : |\mathsf{children}(a)| \leq 1$$

$$\mathsf{HR} \cap (\mathcal{A}_{while} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{while} \times \mathcal{C}^{\#_{o,t}}_{\perp^o} \times \mathcal{A})$$

5.4 RepeatUntil

"The `repeatUntil` activity provides for repeated execution of a contained activity. The contained activity is executed until the given Boolean `condition` becomes true. The condition is tested after each execution of the body of the loop. In contrast to the `while` activity, the `repeatUntil` loop executes the contained activity at least once." (BPEL specification, page 100)

Definition 5.4.1 (Children of a repeat until activity)
A repeat until activity can have exactly one child activity. In terms of the hierarchy relation: There exists exactly one relation connecting the repeat until activity to another (possibly structured) activity. The label for this relation can be a condition, empty or opaque for abstract processes. For executable processes the label is restricted to a condition, this restriction is defined in the section on executable processes. The condition denotes whether the activity is executed another time or not. (BPEL specification, pages 28,99).

$$\forall a \in \mathcal{A}_{repeatUntil} : |\mathsf{children}(a)| \leq 1$$

$$\mathsf{HR} \cap (\mathcal{A}_{repeatUntil} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{repeatUntil} \times \mathcal{C}_{\underline{\perp}^o}^{\#_{o,t}} \times \mathcal{A})$$

5.5 On Alarm Events

"The `onAlarm` corresponds to a timer-based alarm. If the specified duration value in `for` is zero or negative, or a specified deadline in `until` has already been reached or passed, then the `onAlarm` event is executed immediately." (BPEL specification, page 100)

Definition 5.5.1 (Assigning a duration to an on alarm event)
The function $\mathsf{for}_A : \mathcal{E}_{alarm} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{duration,\perp}$ *assigns a duration to an on alarm event. (BPEL specification, page 100,141)*

Definition 5.5.2 (Assigning a date to an on alarm event)
The function $\mathsf{until}_A : \mathcal{E}_{alarm} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{date,\perp}$ *assigns a date to an on alarm event. (BPEL specification, page 100,141)*

Definition 5.5.3 (Validity of date and duration assignments)
Either a duration or a date is assigned to a wait activity but not both. At least one of those needs to be specified. (BPEL specification, page 100,141)

$$\forall e \in \mathcal{E}_{alarm} : (\mathsf{for}_A(e) = \perp \vee \mathsf{until}_A(e) = \perp) \wedge \neg(\mathsf{for}_A(e) = \perp \wedge \mathsf{until}_A(e) = \perp)$$

Definition 5.5.4 (Assigning a duration for repetition of an on alarm event)
The function $\mathsf{repeatEvery}_A : \mathcal{E}_{alarm} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{duration,\perp}$ *assigns a duration to an on alarm that denotes that the event should be repeated several times. (BPEL specification, page 141)*

5.6 Pick

"The `pick` activity waits for the occurrence of exactly one event from a set of events, then executes the activity associated with that event. After an event has been selected, the other events are no longer accepted by that `pick`. The `pick` activity is comprised of a set of branches, each containing an event-activity pair." (BPEL specification, page 100)

Definition 5.6.1 (Branches of a pick activity)
A pick activity can have an arbitrary number of branches. These branches are triggered by normal events (i.e., `onMessage` and `onAlarm` events). In terms of the hierarchy relation that means that the topmost activities of a branch are connected to a pick activity via a relation with a normal event label. (BPEL specification, pages 28,100).

$$\mathsf{HR} \cap (\mathcal{A}_{pick} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{pick} \times \mathcal{E}_{normal} \times \mathcal{A})$$

Definition 5.6.2 (Set of instance creating pick activities)
The set $\mathcal{A}_{pickstart}$ denotes the set of instance creating pick activities. An instance creating pick activities is a pick activity with a `createInstance` value of true.

$$\mathcal{A}_{pickstart} = \{s \in \mathcal{A}_{pick} \mid \mathsf{instance}(s) = \texttt{true}\}$$

Definition 5.6.3 (On alarm branches in instance creating pick activities)
All instance creating pick activities are not allowed to contain branches triggered by an alarm. In terms of the hierarchy relation that means that there must not be any relation that connects such pick activities with other activities having an alarm event as a label. [SA00062] (BPEL specification, page 100)

$$|\mathsf{HR} \cap (\mathcal{A}_{pickstart} \times \mathcal{E}_{Alarm} \times A)| = 0$$

Definition 5.6.4 (Repeat every constructs in pick on alarm events)
On alarm branches in pick activities must not specify a repeat every duration. (BPEL specification, page 101)

$$\forall e \in \mathcal{E}_{alarm} : \forall a \in \mathcal{A}_{pick} : \exists (a, e, b) \in \mathsf{LR} : \mathsf{repeatEvery}_A(e) = \bot$$

5.7 Flow

"A `flow` activity creates a set of concurrent activities directly nested within it. It enables synchronization dependencies between activities that are nested within it to any depth. The `link` construct is used to express these synchronization dependencies. Declaration of `link`s are enclosed by a `flow` activity."(BPEL specification, page 102)

Definition 5.7.1 (Child activities of a flow)
A flow can have an arbitrary number of directly nested activities. In terms of the hierarchy relation these activities are connected to the flow through the empty label. (BPEL specification, pages 29,102).

$$\mathsf{HR} \cap (\mathcal{A}_{flow} \times B \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{flow} \times \{\bot\} \times A)$$

Definition 5.7.2 (Set of links)
The set $\mathcal{L}$ is the set of links. (BPEL specification, page 102)

Definition 5.7.3 (Declaration of links at a flow activity)
The function $\mathsf{declareLink} : \mathcal{L} \rightarrow \mathcal{A}_{flow}$ *declares a link at a flow activity. (BPEL specification, pages 29,102).*

Definition 5.7.4 (Link naming)
The function $\mathsf{name}_{\mathcal{L}} : \mathcal{L} \rightarrow \mathcal{N}^{\#_t}$ *assigns a name to a link.*

The name of a link must be unique among all the names of links declared at the same directly enclosing flow or it must be either empty or opaque. [SA00064] (BPEL specification, page 102)

$$\begin{array}{l}\forall l, m \in \mathcal{L} : \\ \mathsf{name}_{\mathcal{L}}(l) = \mathsf{name}_{\mathcal{L}}(m)) \wedge \mathsf{declareLink}(l) = \mathsf{declareLink}(l) \Rightarrow \\ m = l \ \vee \ \mathsf{name}_{\mathcal{L}}(l) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathcal{L}}(m) \in \{\bot, \texttt{\#\#opaque}\}\end{array}$$

Definition 5.7.5 (Sources of links)
Every activity can be the source of a link. The set SR *is the set of relations assigning activities as sources to a link, the label denotes the transition condition. (BPEL specification, pages 31,84).*

$$\mathrm{SR} \subseteq \mathcal{A} \times \mathcal{C}^{\#_{o,t}}_{\bot^{e,\,o,\,t}} \times \mathcal{L}^{\#_t}$$

Each activity can use only one source element to attach itself as a source to a link. [SA00068] (BPEL specification, page 103)

$$\forall s, t \in \mathrm{SR} : \pi_1(s) = \pi_1(t) \ \wedge \ \pi_2(s) = \pi_2(t) \Rightarrow s = t$$

Definition 5.7.6 (Targets of links)
Every activity can be the target of a link. The set TR *is the set of relations assigning activities as targets to a link. (BPEL specification, pages 31,84).*

$$\mathsf{TR} \subseteq \mathcal{A} \times \mathcal{L}^{\#_t}$$

Each activity can use only one target element to attach itself as a target to a link. [SA00069] (BPEL specification, page 103)

$$\forall s, t \in \mathrm{TR} : \pi_1(s) = \pi_1(t) \ \wedge \ \pi_2(s) = \pi_2(t) \Rightarrow s = t$$

The link relation LR denotes the link relation between two activities including the transition condition. Since the source link relation SR and the target link relation TR are already defined, LR can be derived from them.

Definition 5.7.7 (Link relation)
The relation LR *denotes the link relation between two activities including the transition condition. (BPEL specification, pages 31,84,103) Note that* `##opaque` *and "*$\perp$*" are not part of* $\mathcal{L}$*. Thus, the set* LR *only contains link relations from and to activities, where the link name of source and target elements is specified. [SA00068], [SA00069].*

$$\mathsf{LR} \subset \mathcal{A} \times \mathcal{L} \times \mathcal{C}^{\#_{o,t}}_{\perp_{e,\,o,\,t}} \times \mathcal{A}$$

$$\mathsf{LR} = \{(s, l, c, t) \mid (s, c, l) \in \mathrm{SR}, (t, l) \in \mathrm{TR}\}$$

Only one link may be used to connect the same two activities. [SA00067] (BPEL specification, page 103)

$$\forall l, m \in \mathsf{LR} : \pi_1(l) = \pi_1(m) \wedge \pi_4(l) = \pi_4(m) \Rightarrow l = m$$

Each link must have exactly one source and exactly one target activity. [SA00066] (BPEL specification, page 103)

$$\forall l, m \in \mathsf{LR} : \pi_2(l) = \pi_2(m) \Rightarrow l = m$$

$G = (\mathcal{A}, \pi_{1,4}(\mathsf{LR}))$ *forms a labeled directed acyclic graph as presented in [5].*

Definition 5.7.8 (Source and target activities for links)
The set $\mathcal{A}_{source}$ *is the set of all activities that are the source activity for one or more links.*

$$\mathcal{A}_{source} = \{a \in \mathcal{A} \mid \exists l \in \mathcal{L} : (a, l) \in \pi_{1,2}(\mathsf{LR})\}$$

The set $\mathcal{A}_{target}$ *is the set of all activities that are the target for one or more links.*

$$\mathcal{A}_{target} = \{a \in \mathcal{A} \mid \exists l \in \mathcal{L} : (l, a) \in \pi_{3,4}(\mathsf{LR})\}$$

We explicitly do not take into account activities with opaque and omitted `linkName` *attributes here, since they are not contained in the link relation set* LR*.*

Definition 5.7.9 (Incoming and outgoing links at an activity)
The function $\mathcal{L}_{out} : \mathcal{A}[source] \to 2^{\mathcal{L}}$ *returns the set of all outgoing links of an activity.*

$$\mathcal{L}_{out} : a \mapsto \{l \in \mathcal{L} \mid \exists a' \in \mathcal{A} : (a, l, c, a') \in \mathsf{LR}\}$$

The function $\mathcal{L}_{in} : \mathcal{A}[target] \to 2^{\mathcal{L}}$ *returns the set of all incoming links of an activity.*

$$\mathcal{L}_{in} : a \mapsto \{l \in \mathcal{L} \mid \exists a' \in \mathcal{A} : (a', l, c, a) \in \mathsf{LR}\}$$

The transition condition on a link is included in LR. Thus, it the function tc assigning a transition condition on a link can be derived from LR as shown in Definition 5.7.10.

Definition 5.7.10 (Transition condition)
The function $\mathsf{tc} : \mathcal{L} \to \mathcal{C}^{\#_{o,t}}_{\perp^{e,\,o,\,t}}$ *assigns a transition condition to a link.*

$$\mathsf{tc} : l \mapsto c,\ (s, l, c, t) \in \mathsf{LR}$$

Definition 5.7.11 (Join condition)
The function $\mathsf{jc} : \mathcal{A} \to \mathcal{C}^{\#_t}_{\perp^{e,\,o,\,t}}$ *assigns a join condition to an activity. The join condition may only contain references to incoming links of the activity. [SA00073]. (BPEL specification, pages 31,84,103,105).*

$$\forall a \in \mathcal{A},\, \mathsf{jc}(a) = c,\, LI = \{i \,|\, (a, i) \in \mathsf{TR}\} : c \notin \{\perp, \texttt{\#\#opaque}\} \Rightarrow c \cap (\mathcal{L} \setminus LI) = \emptyset$$

Definition 5.7.12 (Successors of start activities)
$\mathcal{A}_{start\text{-}successors}$ *denotes the set of successors of start activities. A successor activity a is an activity that is either in the clan (cf. Definition 3.5.6) of an activity st that has a control dependency on a start activity s. An activity st has a control dependency on an activity s if it is situated behind s in a sequence or if it is connected via links to activity s.*

$$\begin{aligned}
\mathcal{A}_{start\text{-}successors} = \quad & \{a \mid \ a \in \mathsf{clan}(st), (sq, \perp, s), (sq, \perp, st) \in \mathsf{HR}, s \in \mathcal{A}_{start}, \\
& \quad sq \in \mathcal{A}_{sequence} \wedge s <^{sq}_{seq} st\} \\
\cup \quad & \{a \mid \ a \in \mathsf{clan}(st), (fl, \perp, s), (fl, \perp, st) \in \mathsf{HR}, s \in \mathcal{A}_{start}, \\
& \quad fl \in \mathcal{A}_{flow}, \exists lr_1, \ldots, lr_n : lr_i \in \mathsf{LR}, \\
& \quad lr_1 = (s, l_1, c_1, t_1), \\
& \quad lr_2 = (t_1, l_2, c_2, t_2), \ldots, \\
& \quad lr_n = (t_{n-1}, l_n, c_n, st), \\
& \quad lr_k = (t_{k-1}, l_k, c_k, t_k), 1 < k < n \vee n = 1\}
\end{aligned}$$

Definition 5.7.13 (Valid control dependencies)
All activities that are not start activities or not scope, flow, sequence, or extension activities must have a control dependency on a start activitiy or must be situated in a handler. [SA00056] (BPEL specification, page 90)

$$\begin{aligned}
\forall a \in \mathcal{A} : \quad & a \in \mathcal{A}_{start} \cup \mathcal{A}_{scope} \cup \mathcal{A}_{flow} \cup \mathcal{A}_{sequence} \cup \mathcal{A}_{extension} \vee \\
& a \in \mathcal{A}_{start\text{-}successors} \vee \\
& a \in \mathcal{A}^t_{\mathcal{H}}(s) \ \wedge \ t \in \mathcal{T}_{\mathcal{E}} \ \wedge \ s \in \mathcal{A}_{scope}
\end{aligned}$$

Definition 5.7.14 (Link scoping)
A link can only be used in activities that are descendants of the flow the link is declared in. [SA00065] (BPEL specification, page 103)

$$\forall (a, l, c, b) \in \mathsf{LR} : a, b \in \mathsf{descendants}(\mathsf{declareLink}(l))$$

The following boundary crossing restrictions for links apply (also known as "cross-boundary links"): Note that the boundary crossing restrictions refering to scopes and handlers are declared in the subsection about scopes (subsection 5.9), because definitions not introduced so far are needed.

Definition 5.7.15 (Connecting activities to ancestors)
A link cannot connect an activity to any of its ancestors. [SA00072] (BPEL specification, page 105)

$$\forall (a, l, c, b) \in \mathsf{LR} : a \notin \mathsf{clan}(b)$$

Definition 5.7.16 (Links in sequences)
A link between two activities of a sequence cannot connect a sub-activity or any of its descendants to any preceding sub-activity or any of its descendants.

$$\forall s \in \mathcal{A}_{sequence}, \forall \{x, x'\} \subseteq \mathsf{children}(s), \forall a \in \mathsf{clan}(x), \forall a' \in \mathsf{clan}(x') :$$
$$x <^{s}_{seq} x' \Rightarrow \neg \exists (a', l, c, a) \in \mathsf{LR}$$

Definition 5.7.17 (Links in repeatable constructs)
Links are not allowed to cross the boundary of a repeatable construct. For example, a link might not have an activity inside a while loop as it's source and another activity outside of this loop as a target or vice versa. [SA00070] (BPEL specification, page 104)

$$\forall r \in \mathcal{A}_{while} \cup \mathcal{A}_{repeatUntil} \cup \mathcal{A}_{forEach}, \forall a \in \mathsf{descendants}(r), \forall a' \in \mathcal{A} \setminus \mathsf{descendants}(r) :$$
$$\neg \exists l \in \mathcal{L} : (a, l, c, a') \in \mathsf{LR} \vee (a', l, c, a) \in \mathsf{LR}$$

Links used in a repeatable construct must be declared in a flow that itself is a descendant of the repeatable construct.

$$\forall l \in \mathcal{L}, \forall r \in \mathcal{A}_{while} \cup \mathcal{A}_{repeatUntil} \cup \mathcal{A}_{forEach}, \forall a, a' \in \mathsf{descendants}(r) :$$
$$(a, l, c, a') \in \mathsf{LR} \Rightarrow \mathsf{declareLink}(l) \in \mathsf{descendants}(r)$$

Definition 5.7.18 (Peer-scope dependencies)
A peer-scope dependency must not include cycles, i.e., if a link connects two activities in two different scopes, a link that connects the same two scopes in the opposite direction is not allowed. [SA00082] (BPEL specification, page 134)

$$\forall s, t \in \mathcal{A}_{scope} :$$
$$s \neq t \wedge (a, l, c, a') \in \mathsf{LR} \wedge a \in \mathsf{descendants}(s) \wedge a' \in \mathsf{descendants}(t)$$
$$\Rightarrow \neg \exists (a'', l, a''') \in \mathsf{LR} \wedge a''' \in \mathsf{descendants}(s) \wedge a'' \in \mathsf{descendants}(t)$$

5.8 ForEach Activity

"The `forEach` activity will execute its contained `scope` activity exactly N+1 times where N equals the `finalCounterValue` minus the `startCounterValue`." (BPEL specification, page 112)

Definition 5.8.1 (Child activities of a forEach activity)
A forEach activity must have at most one child activity which must be a scope activity or an opaque activity. (BPEL specification, page 28,112).

$$\forall a \in \mathcal{A}_{forEach} : |\mathsf{children}(a)| \leq 1$$

$$\mathsf{HR} \cap (\mathcal{A}_{forEach} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{forEach} \times \{\bot\} \times (\mathcal{A}_{scope} \cup \mathcal{A}_{opaqueActivity}))$$

Definition 5.8.2 (Start counter value)
The function $\mathsf{startCounterValue} : \mathcal{A}_{forEach} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{int,\underline{\bot}^o}$ *assigns an unsigned int expression to the start counter value of a forEach activity. (BPEL specification, pages 28,113).*

Definition 5.8.3 (Final counter value)
The function $\mathsf{finalCounterValue} : \mathcal{A}_{forEach} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{int,\underline{\bot}^o}$ *assigns an unsigned int expression to the final counter value of a forEach activity. (BPEL specification, pages 28,113).*

Definition 5.8.4 (Completion condition)
The function $\mathsf{completionCondition} : \mathcal{A}_{forEach} \rightarrow \mathrm{Ex}^{\#_{o,t}}_{int,\underline{\bot}^o} \times \mathcal{B}^{\#_{o,t}}_{\underline{\bot}^{e,\,o,\,t}}$ *returns the "completion" condition and "successful branches only" condition. The "completion" condition is an unsigned integer expression. The "successful branches only" condition is a Boolean condition stating whether the completion condition is valid for successful branches only. [SA00074] (BPEL specification, pages 28,113,114).*

Definition 5.8.5 (Constant counter and completion values)
In case the start counter, the final counter and the completion condition are all constant unsigned integer values, then we can perform static validation on these expression so that it is guaranteed that the amount of branches to be completed in the completionCondition is in the range of the difference between final and start counter. [SA00075] (BPEL specification, page 113)

$$\mathsf{value}_{int}(\mathsf{completionCondition}(a)) \leq$$
$$\mathsf{value}_{int}(\mathsf{finalCounterValue}(a)) - \mathsf{value}_{int}(\mathsf{startCounterValue}(a)) + 1$$

Definition 5.8.6 (Parallel forEach)
The function $\mathsf{parallel} : \mathcal{A}_{forEach} \rightarrow \mathbb{B}^{\#_{o,t}}_{\underline{\bot}^{e,\,o,\,t}}$ *assigns a Boolean value to the* ***parallel*** *attribute of forEach, denoting if the forEach is a sequential or parallel forEach. (BPEL specification, pages 28,113).*

Definition 5.8.7 (Counter)
The function $\mathsf{counter}_{\mathcal{A}_{forEach}} \rightarrow \mathcal{V}^{\#_t}$ *assigns a counter to a forEach. The type of a counter is always an unsigned integer. (BPEL specification, page 113)*

$$\forall f \in \mathcal{A}_{forEach} :$$
$$\mathsf{type}_{\mathcal{V}}(\mathsf{counter}_{\mathcal{A}_{forEach}}(f)) = \texttt{xsd:unsingedInt} \ \vee \ \mathsf{counter}(f) \in \{\bot, \texttt{\#\#opaque}\}$$

Definition 5.8.8 (Rules for counter names)
The counter name of a forEach activity must not have the same name as a variable declared in the child scope of a forEach activity. [SA00076] (BPEL specification, page 113)

$$\forall a \in \mathcal{A}_{forEach}, \forall v \in \mathcal{V} :$$
$$\mathsf{declareVar}(v) \in \mathsf{children}(a) \Rightarrow$$
$$\mathsf{name}_{\mathcal{V}}(\mathsf{counter}(a)) \neq \mathsf{name}_{\mathcal{V}}(v) \vee$$
$$\mathsf{counter}(a) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathcal{V}}(v) \in \{\bot, \texttt{\#\#opaque}\}$$

5.9 Scopes

"A `scope` provides the context which influences the execution behavior of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler. Contexts provided by `scope` activities can be nested hierarchically, while the root context is provided by the `process` construct". (BPEL specification, page 115)

Definition 5.9.1 (Child activities of a scope)
A scope can have exactly one child activity. However, in terms of the hierarchy relation, an arbitrary number of relations connecting activities to a scope may exist. The reason for this is, that a scope can contain event, fault, compensation and termination handlers. In terms of the hierarchy relation a scope is valid if there is exactly one activity that is connected to the scope with an empty label (denoting the activity that is directly nested in the scope). Activities in the handlers are connected via relations using the respective event label. An arbitrary amount of event and fault handlers can exist while there can be at most one compensation or termination handler. (BPEL specification, pages 24,115)

Activities can be connected to a scope via a relation containing either an empty or an event label. (BPEL specification, page 116)

$$\mathsf{HR} \cap (\mathcal{A}_{scope} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{scope} \times (\mathcal{E} \cup \{\bot\}) \times \mathcal{A})$$

For the following definitions we assume that s is an arbitrary scope out of the set of scopes $\mathcal{A}_{scope}$:

$$\forall s \in \mathcal{A}_{scope}$$

Definition 5.9.2 (Directly nested main activity of a scope)
At most one activity can be directly nested in the scope. This activity is called the main activity. (BPEL specification, page 115)

$$|\mathsf{HR} \cap (\{s\} \times \{\bot\} \times \mathcal{A})| \leq 1$$

The set $\mathcal{A}_{mainset}$ is the set constituting of the main activity of a scope and all descendants of this activity.

$$\forall s \in \mathcal{A}_{scope} : \mathcal{A}_{mainset}(s) = \{a \in \mathcal{A} \mid \exists x \in \mathcal{A}, \mathsf{HR}(s, \bot, x) \wedge a \in \mathsf{clan}(x)\}$$

"Each scope, including the process scope, can have a set of event handlers. These event handlers can run concurrently and are invoked when the corresponding event occurs. The child activity within an event handler MUST be a `scope` activity. There are two types of events. First, events can be inbound messages that correspond to a WSDL operation. Second, events can be alarms, that go off after user-set times." (BPEL specification, page 137)

Definition 5.9.3 (Event handlers in scopes)
A scope can have zero or more event (`onMessage` und `onAlarm`) handlers. These handlers must contain a directly nested scope. (BPEL specification, pages 29,137,138).

$$\forall s \in \mathcal{A}_{scope} : |\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{normal} \times \mathcal{A}_{scope})| \geq 0$$

"Fault handling in a business process can be thought of as a mode switch from the normal processing in a scope. Fault handling in WS-BPEL is designed to be treated as 'reverse work', in that its aim is to undo the partial and unsuccessful work of a scope in which a fault has occurred. The completion of the activity of a fault handler, even when it does not rethrow the handled fault, is not considered successful completion of the attached scope. Compensation is not enabled for a scope that has had an associated fault handler invoked."(BPEL specification, page 127)

Definition 5.9.4 (Fault handlers in scopes)
A scope can have zero or more fault handlers. (BPEL specification, pages 29,127).

$$\forall s \in \mathcal{A}_{scope} : |\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{fault} \times \mathcal{A})| \geq 0$$

"The ability to declare compensation logic alongside forward-working logic is the underpinning of the application-controlled error-handling framework of WS-BPEL. WS-BPEL allows scopes to delineate that part of the behavior that is meant to be reversible in an application-defined way by specifying a compensation handler. Scopes with compensation and fault handlers can be nested without constraint to arbitrary depth." (BPEL specification, page 118)

Definition 5.9.5 (Compensation handlers in scopes)
A scope can have at most one compensation handler. (BPEL specification, pages 29,118).

$$\forall s \in \mathcal{A}_{scope} : |\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{compensation} \times \mathcal{A})| \leq 1$$

A process definition (which is a special scope element) is not allowed to have any compensation handler.

$$a = \texttt{process} \Rightarrow |\mathsf{HR} \cap (\{a\} \times \mathcal{E}_{compensation} \times \mathcal{A})| = 0$$

"The behavior of a fault handler for a scope C begins by disabling the scope's event handlers and implicitly terminating all activities enclosed within C that are currently active (including all running event handler instances)."(BPEL specification, page 135)

Definition 5.9.6 (Termination handlers in scopes)
A scope can have at most one termination handler. (BPEL specification, pages 29,135)

$$\forall s \in \mathcal{A}_{scope} : |\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{termination} \times \mathcal{A})| \leq 1$$

Definition 5.9.7 (Handlers in scopes)
The function $\mathcal{A}^t_{\mathcal{H}} : \mathcal{A}_{scope} \rightarrow 2^{\mathcal{A}}$ is the function returning all activities used for handling events of type t at scope s. The result consists of the top level activities (represented by x) used for handling all events of type t for scope s and all descendants of these activities.

$$\forall t \in \mathcal{T}_{\mathcal{E}} : \mathcal{A}^t_{\mathcal{H}}(s) = \{a \in \mathcal{A} \mid \exists (s, e, x) \in \mathsf{HR} : a \in \mathsf{clan}(x)\}$$

Definition 5.9.8 (Variable declaration of event handlers)
Variables used in the `onEvent` element of an event handlers are implicitly declared in the direct enclosing scope. Thus, they may not have the same name as variables declared in that scope. [SA00086] (BPEL specification, page 139)

$$\forall (s, e, a) \in \mathsf{HR}, s \in \mathcal{A}_{scope}, e \in \mathcal{E}_{message} :$$
$$\mathsf{name}_V(\mathsf{outputVar}(e)) \notin \{n \mid n = \mathsf{name}_V(v), v \in \mathcal{V}, \mathsf{declareVar}(v) = s\} \wedge$$
$$\forall fp \in \mathrm{FP}(e), fp = (c, v, p) :$$
$$\mathsf{name}_V(v) \notin \{n \mid n = \mathsf{name}_V(v'), v' \in \mathcal{V}, \mathsf{declareVar}(v') = s\}$$

Definition 5.9.9 (Links and event handlers)
The set $\mathcal{A}^{event}_h(s, e)$ is the set of activities in an event handler enclosed in scope s triggered by event e.

Links cannot cross the boundary of an event handler, i.e., there exists no link from one of the activities contained in the event handler to another activity outside the event handler and vice versa.

$$\forall a \in \mathcal{A}^{event}_h(s, e), \forall a' \in \mathcal{A} \setminus \mathcal{A}^{event}_h(s, e) :$$
$$\neg \exists l \in \mathcal{L} : (a, l, c, a') \in \mathsf{LR} \vee (a', l, c, a) \in \mathsf{LR}$$

Definition 5.9.10 (Links in compensation handlers)
Links are not allowed to cross the boundary of a compensation handler. [SA00070]

$$\forall s \in \mathcal{A}_{scope}, \forall a \in \mathcal{A}_{\mathcal{H}}^{compensation}(s), \forall a' \in \mathcal{A} \setminus \mathcal{A}_{\mathcal{H}}^{compensation}(s) :$$
$$\neg\exists l \in \mathcal{L} : (a, l, c, a') \in \mathsf{LR} \vee (a', l, c, a) \in \mathsf{LR}$$

Definition 5.9.11 (Links in fault and termination handlers)
A link crossing a fault or termination handler must be outbound. [SA00071] (BPEL specification, page 105)

$$\forall s \in \mathcal{A}_{scope}, A_{FT} = A[H][fault](s) \cup \mathcal{A}_{\mathcal{H}}^{termination}(s), a \in A_{FT}, b \in \mathcal{A} \setminus A_{FT} :$$
$$\neg\exists l \in \mathcal{L} : (a', l, c, a) \in \mathsf{LR}$$

Definition 5.9.12 (Event behavior of a scope)
The function $\mathcal{A}_{\mathcal{H}}^{normal} : \mathcal{A}_{scope} \rightarrow 2^{\mathcal{A}}$ returns the set of all activities used by all event handlers of a scope.

$$\mathcal{A}_{\mathcal{H}}^{normal}(s) = \mathcal{A}_{\mathcal{H}}^{message}(s) \cup \mathcal{A}_{\mathcal{H}}^{alarm}(s)$$

Definition 5.9.13 (Normal behavior of a scope)
The function $\mathcal{A}_{normal} : \mathcal{A}_{scope} \rightarrow 2^{\mathcal{A}}$ returns the set of all activities that define the normal behavior of scope s. [SA00083] (BPEL specification, page 137)

$$\mathcal{A}_{normal}(s) = \mathcal{A}_{mainset}(s) \cup \mathcal{A}_{\mathcal{H}}^{normal}(s)$$

Definition 5.9.14 (Exit on standard fault)
The function exitOnStandardFault $: \mathcal{A}_{scope} \rightarrow \mathcal{B}_{\perp^{e,o,t}}^{\#_{o,t}}$ *is a function assigning a Boolean value to the* `exitOnStandardFault` *attribute of a scope. (BPEL specification, page 23)*

Definition 5.9.15 (BPEL standard faults)
$\mathcal{E}_{fault_{standardbpel}}$ *is the set of all WS-BPEL standard faults as defined in Appendix A of the WS-BPEL specification. (BPEL specification, pages 192,193)*

$$\begin{aligned}\mathcal{E}_{fault_{standardbpel}} = \{&\texttt{ambiguousReceive}, \texttt{completionConditionFailure},\\ &\texttt{conflictingReceive}, \texttt{conflictingRequest},\\ &\texttt{correlationViolation}, \texttt{invalidBranchCondition},\\ &\texttt{invalidExpressionvalue}, \texttt{invalidVariables},\\ &\texttt{joinFailure}, \texttt{mismatchedAssignmentFailure},\\ &\texttt{missingReply}, \texttt{missingRequest},\\ &\texttt{scopeInitializationFailure}, \texttt{selectionFailure},\\ &\texttt{subLanguageExecutionFault}, \texttt{uninitializedPartnerRole},\\ &\texttt{uninitializedVariable}, \texttt{unsupportedReference},\\ &\texttt{xsltInvalidSource}, \texttt{xsltStylesheetNotFound}\}\end{aligned}$$

Definition 5.9.16 (Rules for standard faults)
If the attribute `exitOnStandardFault` *of a scope is set to true, no fault handler handling one of the standard faults excluding the join failure fault must be specified.[SA00003]. (BPEL specification, page 23)*

$$\forall s \in \mathcal{A}_{scope} :$$
$$\mathsf{exitOnStandard}(s) = \texttt{true} \Rightarrow |\mathsf{HR} \cap (s, \mathcal{E}_{fault_{standardbpel}} \setminus \{\texttt{joinFailure}\}, \mathcal{A})| = 0$$

"The isolated attribute of a scope, when set to "yes", provides control of concurrent access to shared resources: variables, partner links, and control dependency links. Such a scope is called an isolated scope. The default value of the isolated attribute is "no"." (BPEL specification, page 143)

Definition 5.9.17 (Isolated scopes)
The function $\mathsf{isolated} : \mathcal{A}_{scope} \rightarrow \mathcal{B}^{\#_{o,t}}_{\perp^{e,o,t}}$ *is a function assigning a boolean, opaque, or empty value to the* `isolated` *attribute of a scope. (BPEL specification, page 143)*

Definition 5.9.18 (Nesting of isolated scopes)
Isolated scopes cannot be nested. That means, it is not allowed to include an isolated scope in an isolated scope. [SA00091] (BPEL specification, page 144)

$$\forall s \in \mathcal{A}_{scope}, \forall a \in \mathsf{descendants}(s) :$$
$$\mathsf{isolated}(s) = \texttt{true} \ \wedge \ a \in \mathcal{A}_{scope}) \Rightarrow$$
$$\mathsf{isolated}(a) = \texttt{false}$$

Definition 5.9.19 (Isolation of the process element)
The process element itself cannot be an isolated scope, furthermore it does not have an isolated attribute.

$$\mathsf{isolated}(\texttt{process}) = \perp$$

Definition 5.9.20 (Directly enclosed scopes of a scope)
The function $\mathcal{A}_{scope_{de}} : \mathcal{A}_{scope} \rightarrow 2^{\mathcal{A}_{scope}}$ *returns the set of scopes directly nested in a scope. That means, the returned set is the set of scopes where no scope exists that is a child of the given scope and a father of the scope in question.*

$$\begin{aligned}\mathcal{A}_{scope_{de}}(s) = \{v \in \mathcal{A}_{scope} \mid & \\ & v \in \mathsf{descendant}(s) \wedge \\ & \neg\exists w \in \mathcal{A}_{scope}, w \in \mathsf{descendant}(s) \wedge v \in \mathsf{descendant}(w)\}\end{aligned}$$

Definition 5.9.21 (Uniqueness of scope naming)
The names of scopes enclosed in the same father scope must be unique, this is required for named compensation. [SA00092] (BPEL specification, page 123)

$$\forall s \in \mathcal{A}_{scope} : \forall a, a' \in \mathcal{A}_{scope_{de}}(s) :$$
$$\mathsf{name}_{\mathcal{A}}(a) = \mathsf{name}_{\mathcal{A}}(a') \Rightarrow$$
$$a = a' \ \vee \ \mathsf{name}_{\mathcal{A}}(a) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathcal{A}}(a') \in \{\bot, \texttt{\#\#opaque}\}$$

Definition 5.9.22 (Uniqueness of activity naming)
The names of named activities enclosed in the same father scope must be unique. [SA00078] (BPEL specification, page 123)

$$\forall s \in \mathcal{A}_{scope} : \forall a, a' \in \mathsf{children}(s) :$$
$$\mathsf{name}_{\mathcal{A}}(a) = \mathsf{name}_{\mathcal{A}}(a') \Rightarrow$$
$$a = a' \ \vee \ \mathsf{name}_{\mathcal{A}}(a) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{name}_{\mathcal{A}}(a') \in \{\bot, \texttt{\#\#opaque}\}$$

5.10 Fault Handling

Definition 5.10.1 (Assigning a fault name to a fault handler)
The function $\mathsf{faultName}_{\mathcal{E}} : \mathcal{E}_{fault} \mapsto \mathrm{QNAME}^{\#_t}_{\underline{\bot}_{e,\,o,\,t}}$ *assigns a fault name to a fault handler. (BPEL specification, page 127)*

Definition 5.10.2 (Assigning a fault variable to a fault handler)
The function $\mathsf{faultVariable}_{\mathcal{E}} : \mathcal{E}_{fault} \rightarrow V^{\#_t}_{\underline{\bot}_{e,\,o,\,t}} \times (\mathrm{MSG}^{\#_t}_{\underline{\bot}_{e,\,o,\,t}} \cup \mathrm{EL}^{\#_t}_{\underline{\bot}_{e,\,o,\,t}})$ *assigns a fault variable and a fault message type or a fault element to a fault event. This definition ensures that if a fault variable is assigned to a fault handler, either a fault message type or a fault element must be specified but not both. [SA00081] (BPEL specification, page 128)*

If no fault variable is specified, no fault message type and fault element is allowed to be specified.

$$\forall e \in \mathcal{E}_{fault}, \mathsf{faultVariable}_{\mathcal{E}}(e) = (v, me) : \ v = \bot \Rightarrow me = \bot$$

The function catchAll is introduced in definition 5.10.3 to distinguish a `catch` element in an abstract process where no `faultName` and `faultVariable` is defined from a `catchAll` element.

Definition 5.10.3 (Catch all fault handler)
The function catchAll *assigns a boolean value to a fault handler, denoting that this fault handler is the default catch all fault handler. (BPEL specification, page 127)*

$$\mathsf{catchAll} : \mathcal{E}_{fault} \rightarrow \mathbb{B}$$

A catch all fault handler cannot specify a fault name and a fault variable.

$$\forall e \in \mathcal{E}_{fault} :$$
$$\mathsf{catchAll}(e) = \texttt{true} \Rightarrow \mathsf{faultName}_{\mathcal{E}}(e) = \bot \ \wedge \ \mathsf{faultVariable}_{\mathcal{E}}(e) = (\bot, \bot)$$

Definition 5.10.4 (Identical fault handlers)
Identical catch constructs are not allowed at the same scope. Identical catch constructs are catch constructs with the same fault element, fault message type and fault name. [SA00093] (BPEL specification, page 128)

$$\forall i, h \in \mathsf{HR}, h = (a, e, a'), i = (b, f, b'),$$
$$\mathsf{faultVariable}_{\mathcal{E}}(e) = (v, ml), \mathsf{faultVariable}_{\mathcal{E}}(f) = (w, no)$$
$$\mathsf{faultName}_{\mathcal{E}}(e) = \mathsf{faultName}_{\mathcal{E}}(f) \ \wedge \ ml = no \ \wedge \ a = b \Rightarrow$$
$$h = i \ \vee \ \mathsf{faultName}_{\mathcal{E}}(e) \in \{\bot, \texttt{\#\#opaque}\} \ \vee \ \mathsf{faultName}_{\mathcal{E}}(f) \in \{\bot, \texttt{\#\#opaque}\} \vee$$
$$ml = \texttt{\#\#opaque} \ \vee \ no = \texttt{\#\#opaque}$$

Definition 5.10.5 (Declaration of catch all fault handlers)
Only one catch all fault handler can be declared in a scope, i.e., there can only exist one hierarchy relation connecting an activity to a scope with a catch all fault handler label. (BPEL specification, page 128)

$$\forall h, i \in \mathsf{HR} :$$
$$\mathsf{catchAll}(\pi_2(h)) = true \ \wedge \ \mathsf{catchAll}(\pi_2(i)) = true \ \wedge \ \pi_1(h) = \pi_1(i)' \Rightarrow h = i$$

5.11 Compensation

Definition 5.11.1 (Compensation activities)
Compensate and compensateScope activities may be only used in fault, termination and compensation handlers, not in the normal behavior of a scope, i.e. each compensation activity (compensate or compensateScope) must be contained within a fault, termination or compensation handler. This corresponds to the static validation rules [SA00007] and [SA00008]. (BPEL specification, pages 29,112).

$$\forall c \in \mathcal{A}_{compensate} \cup \mathcal{A}_{compensateScope}$$
$$\exists \mathcal{A}^t_{\mathcal{H}}(s), s \in \mathcal{A}_{scope}, t \in (\mathcal{E}_{fault} \cup \mathcal{E}_{termination} \cup \mathcal{E}_{compensation}), c \in \mathcal{A}^t_{\mathcal{H}}(s)$$

Definition 5.11.2 (Assigning a target to a compensateScope activity)
The function $\mathsf{target} : \mathcal{A}_{compensateScope} \rightarrow {\mathcal{A}_{scope}}^{\#_t}$ *assigns a scope to a compensateScope activity.*

The scope assigned to the target attribute of a compensateScope activity must be one of the directly nested scopes that fulfill the following requirement: Let s be the scope of the handler the compensateScope activity is in. Then the target must be a directly nested scope of the normal behavior of the scope. [SA00077] (BPEL specification, page 123)

$$\forall c \in \mathcal{A}_{compensateScope}, c \in \mathsf{descendants}(s)$$
$$\neg\exists s' : s' \in \mathcal{A}_{scope} \wedge \mathsf{descendants}(s) : \mathsf{target}(c) \in \mathcal{A}_{scope_{de}}(s)$$

Definition 5.11.3 (Non-normal behavior of scopes)
The function $\mathcal{A}^{fct}_{\mathcal{H}} : \mathcal{A}_{scope} \rightarrow 2^{\mathcal{A}}$ returns all activities contained in fault, compensation and termination handlers of the given scope. That means, the returned activities are activities of the scope employed for handling non-normal behavior of the scope. (BPEL specification, page 123)

$$\mathcal{A}^{fct}_{\mathcal{H}}(s) = \mathcal{A}^{fault}_{\mathcal{H}}(s) \cup \mathcal{A}^{compensation}_{\mathcal{H}}(s) \cup \mathcal{A}^{termination}_{\mathcal{H}}(s)$$

Definition 5.11.4 (Compensation handlers in FCT handlers)
All root scopes (i.e., the topmost scopes) of fault, compensation and termination handlers (FCT handlers) must not include a compensation handler. [SA00079] (BPEL specification, page 125)

$$\forall s \in \mathcal{A}_{scope} \cap (\mathcal{A}^{fct}_{\mathcal{H}}), \forall t \in \mathcal{A}_{scope} \cap (\mathcal{A}^{fct}_{\mathcal{H}}), s \neq \mathsf{descendants}(t) \Rightarrow$$
$$|\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{compensation} \times \mathcal{A})\} = 0$$

Chapter 6

Executable Processes

An executable process is a process, where $\mathsf{profile} = \perp$. The definition of the set S^* (definition 3.1.4) ensures that opaque tokens are not present in executable processes. Nevertheless, executable processes have to follow additional constraints.

6.1 General rules for executable processes

Definition 6.1.1 (Opaque activities)
Opaque activities are not allowed in executable processes. (BPEL specification, page 148)

$$\forall a \in \mathcal{A} : type(a) \neq opaqueActivity$$

Definition 6.1.2 (Opaque from specs)
From specs may not be opaque in executable processes. (BPEL specification, page 148)

$$|\textsc{From}_{opaque}| = 0$$

Definition 4.11.14 ensures that there are no opaque to specs.

Definition 6.1.3 (Instance creating activities)
All executable processes must have at least one instance creating pick or receive activity. [SA00015]

$$|\{a \in \mathcal{A}_{receive} \cup \mathcal{A}_{pick} \mid \mathsf{instance}(a) = \mathtt{true}\}| > 0$$

Definition 6.1.4 (Pick activities)
Each pick in an executable process must have one or more message events.

$$|\mathsf{HR} \cap (\mathcal{A}_{pick} \times \mathcal{E}_{message} \times \mathcal{A})| \geq 1$$

Definition 6.1.5 (Last branch of an if activity)
last *is the sub activity of the last branch in an if activity.*

$$\forall s \in \mathcal{A}_{if} : \mathsf{last}(s) \in \mathsf{children}(s) \neg\exists a \in \mathsf{children}(s) : \mathsf{last}(s) <^{s}_{if} a$$

Definition 6.1.6 (Else branch in if activities)
The last branch of an if activity may be connected to the if activity with an empty label. That connection denotes the else branch. An else branch is only allowed if at least one other branch exists.

$$\forall h = (i, c, a) \in \mathsf{HR}, i \in \mathcal{A}_{if}, c = \bot \Rightarrow |\mathsf{children}(i)| > 1 \ \wedge \ \mathsf{last}(i) = a$$

Definition 6.1.7 (Receiving constructs)
Receiving constructs must either specify an output variable or to parts. It is important to note that definition 4.6.29 ensures that not both are specified.

$$\forall r \in \mathcal{CO}_{receiving} : \mathsf{outputVar}(r) \neq \bot \ \vee \ \exists (r, v, mp) \in \mathrm{FP}$$

Definition 6.1.8 (Sending activities)
Sending activities must either specify an input variable or from parts. It is important to note that definition 4.6.28 ensures that not both are specified.

$$\forall s \in \mathcal{A}_{sending} : \mathsf{inputVar}(s) \neq \bot \ \vee \ \exists (s, v, mp) \in \mathrm{TP}$$

Definition 6.1.9 (Wait activities)
Either a until or for element must be specified for a wait activity.

$$\forall a \in \mathcal{A}_{wait} : \mathsf{for}_W(a) \neq \bot \vee \mathsf{until}_W(a) \neq \bot$$

6.2 Explicit Sub Activities

The WS-BPEL specification allows the sub activities to omitted by the omission shortcut in certain abstract profiles. There may be no omitted activities in executable processes.

Definition 6.2.1 (Sub activity of a scope)
A scope must have exactly one direct sub activity.

$$|\mathsf{HR} \cap (\mathcal{A}_{scope} \times \{\bot\} \times \mathcal{A})| = 1$$

Definition 6.2.2 (Loops)
Each while, repeatUntil, and forEach activity must have a defined sub-activity.

$$|\mathsf{HR} \cap ((\mathcal{A}_{while} \cup \mathcal{A}_{repeatUntil} \cup \mathcal{A}_{forEach}) \times \mathcal{C} \times \mathcal{A})| = 1$$

Definition 6.2.3 (Sub activity of forEach)
The sub activity of a forEach activity must be present and must be a scope activity.

$$|\mathsf{HR} \cap (\mathcal{A}_{forEach} \times \mathcal{C} \times \mathcal{A}_{scope})| = 1$$

Chapter 7

Abstract Processes

The WS-BPEL specification defines two profiles for abstract processes: The profile for observable behavior and the profile for templates. (BPEL specification, page 147)

7.1 Profile for Observable Behavior

The definitions 3.1.4 to 3.1.7 ensure that the opaque attributes are only allowed for attributes for messaging constructs. (BPEL specification, pages 155,156)

Definition 7.1.1 (Opaqueness of join conditions)
The join condition is not allowed to be opaque. (BPEL specification, page 156)

$$\forall a \in \mathcal{A} : \mathsf{jc}(a) \neq \texttt{\#\#opaque}$$

Definition 7.1.2 (Exit activities)
Exit activities are not allowed. (BPEL specification, page 156)

$$|\mathcal{A}_{exit}| = 0$$

Opaque attributes are only allowed for message related constructs. This is ensured by Definitions 3.1.4 to 3.1.7.

7.2 Profile for Templates

The restrictions are covered by the definitions 3.1.4 to 3.1.7.

Definition 7.2.1 (Opaque start activities)
templateInstance : $\mathcal{A}_{opaqueActivity} \rightarrow \mathbb{B} \cup \{$`##opaque`$\}$ *is a function which assigns a boolean value to the attribute* `createInstance` *of a receive or a pick activity or the* `template:createInstance` *attribute of an opaque activity. (BPEL specification, page 159)*

Chapter 8

Static Validation Rules and Details not Covered

This work showed the formalization of most WS-BPEL static validation rules. Due to the exclusion of runtime behavior and detailed formalization of related specifications, such as XPath, we did not cover following static validation rules and specification details:

XPath related [SA00026], [SA00027], [SA00028], [SA00029], [SA00030], [SA00031], [SA00033], [SA00039], [SA00040], [SA00094].

XSLT related [SA00041].

Type related [SA00043].

Document Linking [SA00010], [SA00011], [SA00012], [SA00013], [SA00014], [SA00053], [SA00054].

Extensibility, runtime and engine behavior [SA00004] [SA00009], [SA00084], [SA00088], [SA00089], (BPEL specification, page 24)

Documentation `<documentation>` construct. (BPEL specification, page 24)

Chapter 9

Sample

This section presents the formal definition of the loan approval example given on page 179ff. in the WS-BPEL specification.

9.1 WSDL

Following port types are used:

$$\mathrm{PT} \supset \{loanServicePT, riskAssessmentPT, loanApprovalPT, \ldots\}$$

The port types have names:

$$\begin{array}{lcl} \mathsf{name}_{\mathrm{PT}}(loanServicePT) & = & loanServicePT \\ \mathsf{name}_{\mathrm{PT}}(riskAssessmentPT) & = & riskAssessmentPT \\ \mathsf{name}_{\mathrm{PT}}(loanApprovalPT) & = & loanApprovalPT \end{array}$$

Following operations are used there:

$$\mathcal{O} \supset \{request, check, approve\}$$

The operations have names:

$$\begin{array}{lcl} \mathsf{name}_{\mathcal{O}}(request) & = & request \\ \mathsf{name}_{\mathcal{O}}(check) & = & check \\ \mathsf{name}_{\mathcal{O}}(approve) & = & approve \end{array}$$

These operations are declared at the port types:

$$\begin{aligned}
\mathsf{portType}_{operation}(request) &= loanServicePT \\
\mathsf{portType}_{operation}(check) &= riskAssessmentPT \\
\mathsf{portType}_{operation}(approve) &= loanApprovalPT
\end{aligned}$$

Following messages are sent:

$$\begin{aligned}
\mathrm{MSG} \supset \{&creditInformationMessage, approvalMessage, \\
&riskAssessmentMessage, errorMessage\}
\end{aligned}$$

These messages consist of parts:

$$\begin{aligned}
\mathrm{MP} \supset \{&firstName, name, amount, \\
&accept, level, errorCode\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{mpm}(firstName) &= creditInformationMessage \\
\mathsf{mpm}(name) &= creditInformationMessage \\
\mathsf{mpm}(amount) &= creditInformationMessage \\
\mathsf{mpm}(accept) &= approvalMessage \\
\mathsf{mpm}(level) &= riskAssessmentMessage \\
\mathsf{mpm}(errorCode) &= errorMessage
\end{aligned}$$

The message parts have types:

$$\begin{aligned}
\mathsf{type}_{\mathrm{MP}}(firstName) &= \texttt{xsd:string} \\
\mathsf{type}_{\mathrm{MP}}(name) &= \texttt{xsd:string} \\
\mathsf{type}_{\mathrm{MP}}(amount) &= \texttt{xsd:integer} \\
\mathsf{type}_{\mathrm{MP}}(accept) &= \texttt{xsd:string} \\
\mathsf{type}_{\mathrm{MP}}(level) &= \texttt{xsd:string} \\
\mathsf{type}_{\mathrm{MP}}(errorCode) &= \texttt{ens:integer}
\end{aligned}$$

The operations have input, output, and fault messages:

$$
\begin{array}{lcl}
\mathsf{msg}_{input}(request) & = & creditInformationMessage \\
\mathsf{msg}_{output}(request) & = & approvalMessage \\
\mathsf{msg}_{fault}(request) & = & errorMessage_1 \\
\mathsf{name}_{\mathrm{MSG}}(errorMessage_1) & = & unableToHandleRequest \\
\mathsf{msg}_{input}(check) & = & creditInformationMessage \\
\mathsf{msg}_{output}(check) & = & riskAssessmentMessage \\
\mathsf{msg}_{fault}(check) & = & errorMessage_2 \\
\mathsf{name}_{\mathrm{MSG}}(errorMessage_2) & = & loanProcessFault \\
\mathsf{msg}_{input}(approve) & = & creditInformationMessage \\
\mathsf{msg}_{output}(approve) & = & approvalMessage \\
\mathsf{msg}_{fault}(approve) & = & errorMessage_3 \\
\mathsf{name}_{\mathrm{MSG}}(errorMessage_3) & = & loanProcessFault
\end{array}
$$

Following partner link types are declared:

$$\mathrm{PLT} = \{loanPartnerLT, loanApprovalLT, riskAssessmentLT\}$$

Following roles are declared:

$$
\begin{array}{l}
\mathrm{ROLES} = \{(loanService, loanServicePT), \\
\qquad\qquad (approver, loanApprovalPT), \\
\qquad\qquad (assessor, riskAssessmentPT)\}
\end{array}
$$

The roles belong to partner link types:

$$
\begin{array}{lcl}
\mathsf{roles}(loanPartnerLT) & = & ((loanService, loanServicePT), \bot) \\
\mathsf{roles}(loanApprovalLT) & = & ((approver, loanApprovalPT), \bot) \\
\mathsf{roles}(riskAssessmentLT) & = & ((assessor, riskAssessmentPT), \bot)
\end{array}
$$

9.2 BPEL Process

The following sections present the elements of the tuple presented in Definition 3.1.1. If an element is not explicitly defined below, sets default to $\emptyset$. There is no need to define functions taking empty sets as input.

9.2.1 The Process Element

`process` is the process element of loan approval process. The name of the process is "loanApprovalProcess":

$$\mathsf{name}_{\mathcal{A}}(\texttt{process}) = \mathit{loanApprovalProcess}$$

Join failures are suppressed.

$$\mathsf{supjoinf}(\texttt{process}) = \texttt{true}$$

9.2.2 Global Declarations

Following partner links are declared:

$$\begin{array}{lcl} \mathrm{PL} & = & \{\mathit{customer}, \mathit{approver}, \mathit{assessor}\} \\ \mathsf{declarePartnerLink}(\mathit{customer}) & = & \texttt{process} \\ \mathsf{declarePartnerLink}(\mathit{approver}) & = & \texttt{process} \\ \mathsf{declarePartnerLink}(\mathit{assessor}) & = & \texttt{process} \end{array}$$

The partner links have following types:

$$\begin{array}{lcl} \mathsf{type}_{\mathrm{PL}}(\mathit{customer}) & = & \mathit{loanPartnerLT} \\ \mathsf{type}_{\mathrm{PL}}(\mathit{approver}) & = & \mathit{loanApprovalLT} \\ \mathsf{type}_{\mathrm{PL}}(\mathit{assessor}) & = & \mathit{riskAssessmentLT} \end{array}$$

The roles are defined as follows:

$$\begin{array}{lcl} \mathsf{myRole}(\mathit{customer}) & = & \mathit{loanService} \\ \mathsf{partnerRole}(\mathit{customer}) & = & \bot \\ \mathsf{myRole}(\mathit{approver}) & = & \bot \\ \mathsf{partnerRole}(\mathit{approver}) & = & \mathit{approver} \\ \mathsf{myRole}(\mathit{assessor}) & = & \bot \\ \mathsf{partnerRole}(\mathit{assessor}) & = & \mathit{assessor} \end{array}$$

Following variables are declared. Note that *error* is implicitly declared at the fault handler.

$$
\begin{array}{lcl}
\mathcal{V} & = & \{request, risk, approval, error\} \\
\mathsf{name}_{\mathcal{V}}(request) & = & \texttt{request} \\
\mathsf{name}_{\mathcal{V}}(risk) & = & \texttt{risk} \\
\mathsf{name}_{\mathcal{V}}(approval) & = & \texttt{approval} \\
\mathsf{type}_{\mathcal{V}}(request) & = & creditInformationMessage \\
\mathsf{type}_{\mathcal{V}}(risk) & = & riskAssessmentMessage \\
\mathsf{type}_{\mathcal{V}}(approval) & = & approvalMessage \\
\mathsf{declarevar}(request) & = & \texttt{process} \\
\mathsf{declarevar}(risk) & = & \texttt{process} \\
\mathsf{declarevar}(approval) & = & \texttt{process}
\end{array}
$$

9.2.3 Events in the Process

The fault event is the only event defined in the process.

$$
\begin{array}{lcl}
\mathcal{E} & = & \{loanProcessFault\} \\
\mathsf{faultName}_{\mathcal{E}}(loanProcessFault) & = & loanProcessFault \\
\mathsf{faultVariable}_{\mathcal{E}}(loanProcessFault) & = & (error, errorMessage)
\end{array}
$$

9.2.4 Activities in the Process

Following activities are contained in the process:

$$
\begin{aligned}
\mathcal{A} = \{&flow, receive, invoke_1, invoke_2, assign, \\
&reply, faultReply\}
\end{aligned}
$$

The Receive Activity

$$
\begin{array}{lcl}
\mathsf{type}_{\mathcal{A}}(receive) & = & receive \\
\mathsf{partnerLink}_{\mathcal{CO}}(receive) & = & customer \\
\mathsf{portType}_{\mathcal{CO}}(request) & = & loanServicePT \\
\mathsf{operation}_{\mathcal{CO}}(receive) & = & request \\
\mathsf{outputVar}(receive) & = & request \\
\mathsf{createInstance}(receive) & = & \texttt{true}
\end{array}
$$

The First Invoke Activity

$$
\begin{array}{lcl}
\mathsf{type}_{\mathcal{A}}(invoke_1) & = & invoke \\
\mathsf{partnerLink}_{\mathcal{CO}}(invoke_1) & = & assessor \\
\mathsf{portType}_{\mathcal{CO}}(invoke_1) & = & riskAssessmentPT \\
\mathsf{operation}_{\mathcal{CO}}(invoke_1) & = & check \\
\mathsf{inputVar}(invoke_1) & = & request \\
\mathsf{outputVar}(invoke_1) & = & risk
\end{array}
$$

The Assign Activity

$$
\begin{array}{lcl}
\mathsf{type}_{\mathcal{A}}(assign) & = & assign \\
\textsc{From}_{lit} & = & \{\texttt{yes}\} \\
\textsc{To}_{var} & = & \{(approval, accept)\} \\
\textsc{Copy} & = & \{(\texttt{yes}, (approval, accept)\} \\
\mathsf{assignCopy}(assign) & = & \{\{(\texttt{yes}, (approval, accept)\}\}
\end{array}
$$

The Second Invoke Activity

$$
\begin{array}{lcl}
\mathsf{type}_{\mathcal{A}}(invoke_2) & = & invoke \\
\mathsf{partnerLink}_{\mathcal{CO}}(invoke_2) & = & approver \\
\mathsf{portType}_{\mathcal{CO}}(invoke_2) & = & loanApprovalPT \\
\mathsf{operation}_{\mathcal{CO}}(invoke_2) & = & approve \\
\mathsf{inputVar}(invoke_2) & = & request \\
\mathsf{outputVar}(invoke_2) & = & approval
\end{array}
$$

The Reply Activity in the Flow

$$
\begin{array}{lcl}
\mathsf{type}_{\mathcal{A}}(reply) & = & reply \\
\mathsf{partnerLink}_{\mathcal{CO}}(reply) & = & customer \\
\mathsf{portType}_{\mathcal{CO}}(reply) & = & loanServicePT \\
\mathsf{operation}_{\mathcal{CO}}(reply) & = & request \\
\mathsf{inputVar}(reply) & = & approval
\end{array}
$$

The Reply Activity in the Fault Handler

$$
\begin{array}{lcl}
\mathsf{type}_{\mathcal{A}}(faultReply) & = & reply \\
\mathsf{partnerLink}_{\mathcal{CO}}(fautReply) & = & customer \\
\mathsf{portType}_{\mathcal{CO}}(faultReply) & = & loanServicePT \\
\mathsf{operation}_{\mathcal{CO}}(faultReply) & = & request \\
\mathsf{inputVar}(faultReply) & = & error \\
\mathsf{faultName}_{reply} & = & unableToHandleRequest
\end{array}
$$

9.2.5 The Hierarchy Relation

$$
\begin{aligned}
\mathsf{HR} = \{&(\texttt{process}, \bot, flow), \\
&(\texttt{process}, loanProcessFault, faultReply) \\
&(flow, \bot, receive), \\
&(flow, \bot, invoke_1), \\
&(flow, \bot, invoke_2), \\
&(flow, \bot, assign), \\
&(flow, \bot, reply)\}
\end{aligned}
$$

9.2.6 The Links

$$
\begin{array}{lcl}
\mathcal{L} & = & \{l_1, l_2, l_3, l_4, l_5, l_6\} \\
\mathsf{name}_{\mathcal{L}}(l_1) & = & \textit{receive-to-assess} \\
\mathsf{name}_{\mathcal{L}}(l_2) & = & \textit{receive-to-approval} \\
\mathsf{name}_{\mathcal{L}}(l_3) & = & \textit{approval-to-reply} \\
\mathsf{name}_{\mathcal{L}}(l_4) & = & \textit{assess-to-setMessage} \\
\mathsf{name}_{\mathcal{L}}(l_5) & = & \textit{setMessage-to-reply} \\
\mathsf{name}_{\mathcal{L}}(l_6) & = & \textit{assess-to-approval} \\
\mathsf{declareLink}(l_1) & = & \textit{flow} \\
\mathsf{declareLink}(l_2) & = & \textit{flow} \\
\mathsf{declareLink}(l_3) & = & \textit{flow} \\
\mathsf{declareLink}(l_4) & = & \textit{flow} \\
\mathsf{declareLink}(l_5) & = & \textit{flow} \\
\mathsf{declareLink}(l_6) & = & \textit{flow} \\
\mathrm{SR} & = \{ & (\textit{receive}, \texttt{\$request.amount\<10000}, l_1), \\
 & & (\textit{receive}, \texttt{\$request.amount >= 10000}, l_2), \\
 & & (\textit{invoke}_1, \texttt{\$risk.level = 'low'}, l_4), \\
 & & (\textit{invoke}_1, \texttt{\$risk.level != 'low'}, l_6), \\
 & & (\textit{assign}, \bot, l_5), \\
 & & (\textit{invoke}_2, \bot, l_3)\ \} \\
\mathrm{TR} & = \{ & (\textit{invoke}_2, l_1), \\
 & & (\textit{assign}, l_5), \\
 & & (\textit{inovke}_2, l_2), \\
 & & (\textit{invoke}_2, l_6), \\
 & & (\textit{reply}, l_5), \\
 & & (\textit{reply}, l_3)\ \}
\end{array}
$$

Chapter 10

Conclusion and Outlook

This work gives a complete formalization of the syntax WS-BPEL 2.0. It covers all syntactical details such as activities, event handlers, fault handlers, compensation handlers, and termination handlers. Furthermore, we cover parts of WSDL to describe the most important aspects related to WS-BPEL 2.0. The WS-BPEL 2.0 specification lists a set of static validation rules a process model has to comply to. We expressed most of the static validation rules in a formal way to provide a better understanding of them. The provided example shows the application of the syntax to an existing WS-BPEL process.

The static validation rules need the function possibleRunInParallel which denotes that two activities may be executed in parallel. In future work, we will provide an approximation of that function.

Several extensions of WS-BPEL have been proposed. Our future work includes the formalization some extensions, such as BPEL4Chor [4] or BPEL4People [1].

Acknowledgements

Oliver Kopp is funded by the German Federal Ministry of Education and Research (project Tools4BPEL, project number 01ISE08). We thank Michael Reiter for his feedback on an earlier version of this report.

Bibliography

[1] WS-BPEL extension for people (BPEL4People), version 1.0. `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf`.

[2] F. v. Breugel and M. Koshkina. Models and Verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`, 2006.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[4] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *ICWS 2007*. IEEE Computer Society, July 2007.

[5] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[6] Web Services Business Process Execution Language Version 2.0 – OASIS Standard. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), Mar 2007. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`.

[7] C. Ouyang, H. M. W. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL (revised version), Oct. 01 2005.

[8] XML schema part 1: Structures, W3C recommendation, 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`.

All links were last followed on July 4$^{\text{th}}$, 2008.