Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

A Generic Transformation of Existing Service Composition Models to a Unified Model

Katharina Görlach

Report 2013/01

(Version 2.0)

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: D.2.11, D.3.1, D.3.2, F.3.2, F.4.2, H.4.1

Abstract

This report presents a generic transformation from existing service composition models to a unified model based on formal grammars. The presented unified model is especially designed to be suitable for different kinds of modeling paradigms, e.g. imperative and declarative models.

At first, the formal grammars that are used for service compositions are defined. Afterwards, grammar-based representations for modeling constructs provided by the existing service composition specification languages WS-BPEL, Scufl, and ConDec are presented. However, the transformation to the grammar-based representations is discussed by use of general modeling constructs, e.g. looping control flow. The transformation of concrete modeling constructs, e.g. while and for loops can be implemented in the same way.

1 Introduction

High modeling languages for service compositions provide simple modeling constructs mostly hiding complex operational semantics. A modeler that uses a modeling construct needs to be aware of the operational semantics but is not supposed to implement the corresponding logic. Instead, an engine supporting a specific modeling language provides the implementation of the operational semantics, i.e. is suitable to interpret and execute a model. However, various modeling languages exist and even multiple engines exist corresponding to one single modeling language. Engine providers often implement their own view on the modeling language, e.g. specialize fuzzy issues in the meta-model, modify the predefined operational semantics, or introduce additional modeling constructs. Furthermore, each engine provide implements an own internal model that is used for the execution of service compositions. In summary, multiple modeling languages with multiple engines each implementing an own internal processing model is the state of the art.

The approach at hand introduces a unified model that is intended to be used as internal processing model enabling a unified execution of service compositions and avoiding the need for multiple engines in case multiple specification languages need to be supported. The unified model is intended to be suitable for different kinds of modeling paradigms. In particular, imperative, i.e. control-flow-based languages and data-flow-based languages as well as declarative, i.e. constraint-based languages need to be supported. This report presents a generic transformation from the existing service composition specification languages BPEL [2], Scufl [3], and ConDec [4] to the unified model. The unified model is based on formal grammars and is intended to be derived by transformation. That means, no human is assumed to directly define grammar-based models.

In the grammar-based unified model production rules specify single steps in the execution of a service composition similar to assembler code that specifies single steps in the execution of an application that also can be implemented by using a high specification language. A formal automaton implementing the operational semantics of the unified model is suitable to execute service composition by interpreting and processing the grammar-based model. However, a single automaton is considered to be an instance-specific engine. That means, an automaton is aware about the model by the given grammar but doesn't implement instance management. Instead, for each service composition instance a new instance of the automaton is created. However, automata are related to the same grammar iff the service composition instances realized by the automata are related to same service composition model.

For an appropriate usage of formal grammars for the modeling of service compositions formal grammars need to be modified. In particular, non-terminals in formal grammars are extended by types in order to enable an association with services. Furthermore, the structure of production rules is restricted and the rules are interpreted in a special way. Special attention is payed to the processing of grammars, i.e. the application of production rules instead of focusing on the created words. In detail, a sequence of applied production rules is considered to represent a specific run of a service composition. Different sequences of applied rules represent different runs (showing different runtime behaviour) but do not necessarily produce different words.

2 Formal Grammars for Service Composition Models

A formal grammar $G = (N, \Sigma, P, S)$ is a special rewriting system $(N \cup \Sigma, P)$. In detail, the alphabet of rewriting systems is separated into a set of non-terminal and terminal symbols. Furthermore, a specific non-terminal is explicitly specified as the start symbol. However, rewriting systems do not provide an algorithm for substituting terms with each other but provide a set of possible rule applications. The substitution algorithm defines the operational semantics of a rewrite system. Considering formal grammars the corresponding automata implement the substitution algorithm, i.e the operational semantics of formal grammars.

The approach at hand uses special formal grammars and special automata for the representation and the processing of service composition models. Regarding the formal grammars that are used for service compositions further separation of symbols is required. In particular, the non-terminals are typed and the non-terminal types can be associated with a service. By use of the non-terminal types the non-terminals can be classified. Additionally, the production rules that are used for service composition are considered to specify an unordered set of non-terminals and terminal on the left-hand-side (*lhs*) as well as on the right-hand-side (*rhs*). This means, in contrast to formal grammars where the production rules specify an ordered set of symbols the approach at hand ignores the order of symbols but focusses on the (concurrent) existence of symbols.

Furthermore, the approach at hand neglects the language that is related to the used grammars but pays special attention to the grammar itself. That means, the processing of a grammar is of high importance whereas the result is of less importance. Regarding the correlation to service compositions that means the processing of a service composition is of high importance whereas the trace is of less importance.

In the following preliminary definitions are presented. Afterwards, the characteristics of formal grammars that are used for service compositions, i.e. service grammars are discussed and service grammars are defined. Additionally, a substitution algorithm for service grammars is presented for determining the operational semantics of grammar-based service compositions.

2.1 Preliminary Definitions

Definition 1 (Formal Grammar)
A formal grammar is a 4-Tupel $G = (V, \Sigma, P, S)$ *with*

- V *as set of non-terminal symbols;*
- Σ *as set of terminal symbols;*

- $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ *as a set of production rules;*
- $S \in V$ *as start symbol;*

where $^+$ *denotes the Kleene Plus,* * *denotes the Kleene star, and* $\cup$ *denotes the set union.*

⌟

Definition 2 (Word)
A word w over an alphabet set Σ is a finite sequence $(x_1, x_2, x_3, \ldots, x_n)$ with $x_i \in \Sigma$ and $n \geq 0$. For $n = 0$ the empty word is denoted ε. ⌟

Definition 3 (Word Concatenation $\oplus$)
Two words $w = (x_1, x_2, ..., x_n)$ and $v = (y_1, y_2, ..., y_m)$ can be concatenated by the operator $\oplus$ that is defined by:

$$w \oplus v = (x_1, x_2, ..., x_n) \oplus (y_1, y_2, ..., y_m) := (x_1, x_2, ..., x_n, y_1, y_2, ..., y_m)$$

⌟

Definition 4 (Word Subtraction $\ominus$)
A word w over an alphabet A minus a word v can be calculated if the word v is included in the word w and is defined by:

$$w \ominus v := u \Leftrightarrow \exists m, n \in A^* : w = m \oplus v \oplus n \ \wedge \ u = m \oplus n$$

⌟

Definition 5 (Operation word $\cap$ set)
Assuming a finite word $\omega = (x_1, x_2, ...x_n)$ and a set of symbols Φ.
Then the operation $\omega \cap \Phi$ calculates the partial word ω_Φ containing all symbols of ω that are part of Φ:

$$\omega \cap \Phi = \omega_\Phi := \begin{cases} \omega, & \forall x_i \in \omega : x_i \in \Phi; \\ (\omega \ominus x_i) \cap \Phi, & \exists x_i \in \omega : x_i \notin \Phi. \end{cases}$$

⌟

Definition 6 (Word Union $\sqcup$)
Assuming a word $\alpha = (\alpha_1, \alpha_2, ..., \alpha_m)$ and a word $\beta = (\beta_1, \beta_2, ..., \beta_n)$.
Then the word union $\alpha \sqcup \beta$ is defined by:

$$\alpha \sqcup \beta = (\alpha_1, \alpha_2, ..., \alpha_m) \sqcup (\beta_1, \beta_2, ..., \beta_n) := \alpha \oplus \beta \ominus (\alpha_i, \alpha_j, ..., \alpha_k) \ \wedge$$

$(\alpha_i \in \alpha \wedge \alpha_i \in \beta) \wedge (\alpha_j \in \alpha \wedge \alpha_i \in \beta) \wedge ... \wedge (\alpha_k \in \alpha \wedge \alpha_k \in \beta)\ , 1 \leq i \leq j \leq k \leq n$

⌟

Definition 7 (Multiset)
*A multiset $\mathfrak{M}$ over a set A is a function $\mathfrak{M} : A \rightarrow \mathbb{N}_0$ represented by a finite collection $\mathfrak{M} = [x_1, x_2, x_3, \ldots, x_n]$ with $x_i \in \Sigma$. That means, a multiset is denoted using square brackets, e.g. $[a, a, b, c]$, where $[a, a, b, c] = [a, b, a, c] = [a, b, c, a] = [b, a, a, c]$ etc.
The emtpy multiset is denoted ε.* ⌟

Definition 8 (Multiset Union $\oplus$)
The union of two multisets $\mathfrak{W} = [x_1, x_2, ..., x_n]$ and $\mathfrak{V} = [y_1, y_2, ..., y_m]$ is defined by:

$$\mathfrak{W} \oplus \mathfrak{V} = [x_1, x_2, ..., x_n] \oplus [y_1, y_2, ..., y_m] := [x_1, x_2, ..., x_n, y_1, y_2, ..., y_m]$$

⌟

Definition 9 (Multiset Subtraction $\ominus$)
A multiset $\mathfrak{W}$ over a set A minus a multiset $\mathfrak{V}$ can be calculated if the multiset $\mathfrak{V}$ is included in the multiset $\mathfrak{W}$ and is defined by:

$$\mathfrak{W} \ominus \mathfrak{V} := \mathfrak{U} \Leftrightarrow \exists \mathfrak{U} : B \rightarrow \mathbb{N}_0 \wedge \mathfrak{W} = \mathfrak{V} \oplus \mathfrak{U}$$

⌟

Definition 10 (Operation multiset $\cap$ set)
*Assuming a multiset $\mathfrak{M} = [x_1, x_2, ...x_n]$ and a set of symbols Φ.
Then the operation $\mathfrak{M} \cap \Phi$ calculates the partial multiset $\mathfrak{M}_\Phi : \Phi \rightarrow \mathbb{N}_0$ containing all symbols of $\mathfrak{M}$ that are part of Φ:*

$$\mathfrak{M} \cap \Phi = \mathfrak{M}_\Phi(x_i) := \begin{cases} 0, & x_i \in \Phi \wedge x_i \notin \mathfrak{M}; \\ \mathfrak{M}(x_i), & x_i \in \Phi \wedge x_i \in \mathfrak{M}. \end{cases}$$

⌟

Definition 11 (Multiset Union $\sqcup$)
*Assuming a multiset $\mathfrak{M} : A \rightarrow \mathbb{N}_0$ and a multiset $\mathfrak{N} : B \rightarrow \mathbb{N}_0$.
Then the union $\mathfrak{M} \sqcup \mathfrak{N}$ calculates the multiset $\mathfrak{U} : A \cup B \rightarrow \mathbb{N}_0$ that is defined by:*

$$\mathfrak{U}(x_i) := \begin{cases} \mathfrak{M}(x_i), & x_i \in A \wedge ((x_i \in B \wedge \mathfrak{M}(x_i) \geq \mathfrak{N}(x_i)) \vee x_i \notin B); \\ \mathfrak{N}(x_i), & x_i \in B \wedge ((x_i \in A \wedge \mathfrak{M}(x_i) < \mathfrak{N}(x_i)) \vee x_i \notin A). \end{cases}$$

⌟

2.2 Service Grammars

For enabling service-oriented computing in formal grammars non-terminals are associated with service operations. The additional information, e.g. associated service operations increase the complexity of non-terminals.

Definition 12 (Complexity of Non-Terminals)
Assuming a grammar $G = (V, \Sigma, P, S)$ *each non-terminal* $N \in V$ *is represented by a*

- *1-tuple* (d_1) *if the non-terminal is 1-dimensional;*
- *2-tuple* (d_1, d_2) *if the non-terminal is 2-dimensional;*
- *3-tuple* (d_1, d_2, d_3) *if the non-terminal is 3-dimensional;*

where:

- d_1 *specifies the identifier* N *of the non-terminal;*
- $d_2 = (Type, (Input, Ouput))$ *specifies the associated service operation (Type) including input and output parameters where* $Input$ *and* $Output$ *may be empty,*
- $d_3 : String \rightarrow V$ *specifies the mapping of service operation results and 1-dimensional non-terminals.*

⌟

Conventional Chomsky non-terminals are 1-dimensional non-terminals exclusively specifying the identifier of the symbol. In the unified model 1-dimensional non-terminals are helper symbols that are required for ensuring the correct order of service calls. However, 2-dimensional non-terminals represent service calls whereas the associated service operation is specified by use of a non-terminal type. The invocation parameters of the service call that is represented by a 2-dimensional non-terminal are specified by constant values or data references in the second dimension of the non-terminal next to the non-terminal type. 3-dimensional non-terminals also represent service calls but the return value of a 3-dimensional is not stored by use of a data reference, e.g. a variable. In contrast, the return value of a service call that is represented by a 3-dimensional non-terminal is mapped to a grammar-internal representation, i.e. a non-terminal corresponding to the third dimension.

The production rules of formal grammars that are used for service composition are c-interpreted. The c-interpretation of production rules covers the concurrent existence of symbols but ignore the order of symbols. That means, c-interpreted production rules handle multisets instead of words.

Definition 13 (Multiset-Interpretation of Words)
The multiset-interpretation $\mathfrak{m}(w)$ *of a word* $w = (x_1, x_2, ..., x_n)$ *over an alphabet* Σ *is defined by the multiset* $\mathfrak{M} : \Sigma \rightarrow \mathbb{N}_0$ *with:*

$$\mathfrak{m}(w) = [x_1, x_2, ..., x_n]$$

┘

Definition 14 (c-Interpretation of Production Rules)
The c-interpretation i_c *of a production rule* $p = (\alpha, \beta)$ *specifying the words* α, β *results in a c-interpreted production rule* $p_c = (\mathfrak{m}(\alpha), \mathfrak{m}(\beta))$ *with the multiset-interpretation* $\mathfrak{m}$.

$i_c : Words \times Words \rightarrow Multisets \times Multisets$

$$i_c((\alpha, \beta)) = (\mathfrak{m}(\alpha), \mathfrak{m}(\beta))$$

┘

The structure of production rules in formal grammars that are used for service grammars are restricted for ensuring the simplest automaton class for processing. In particular, the production rules are restricted to specify variant, terminal-based context. The variant context allows to change the context that is specified in production rules. In particular, the production rules may substitute multiple symbols on the *lhs* by a single or multiple symbols on the *rhs*. However, the terminal-based context allows to exclusively specify a single non-terminal on the *lhs* whereas multiple terminals may be specified on the *lhs*. Terminal-based context enables to uniquely determine the processing symbol of a production rule. The processing symbol is defined by the non-terminal on the *lhs* that is intended to be substituted by the *rhs* while rule application. For more information about context types please see [1].

Definition 15 (Variant, Terminal-based Context)
Let $G = (\Sigma, V, P, S)$ *be a formal grammar. Then the grammar provides variant, terminal-based context if* $P \subseteq \Sigma^* V \Sigma^* \times (\Sigma \cup V)^*$ ┘

For the executability of formal grammars that are used for service compositions the grammars need to provide exclusive and complete context as well as dynamic determinism. Exclusive context ensures that at most a single context-sensitive or unrestricted production rule can be applied in each reachable automaton configuration. Complete context ensures that all possible context alternatives that can occur at runtime are specified in context-sensitive or unrestricted production rules. Finally, dynamic determinism ensures the deterministic selection of production rules at runtime.

Definition 16 (Exclusive Context)
A formal grammar $G = (V, \Sigma, P, S)$ *provides exclusive context if for each pair of*

context-sensitive or unrestricted production rules x, y with the same processing symbol N but different context symbols of N the entire entire set of context symbols of rule x cannot occur at the same time as the entire set of context symbols of rule y. That means, the context of a fixed processing symbol in production rules is mutually exclusive at runtime.

$\forall x, y \in P \wedge N \in V \wedge N \in LHS(x) \wedge N \in LHS(y) \wedge LHS(x) \setminus \{N\} \neq LHS(y) \setminus \{N\} \neq \emptyset$
$\nexists w_i \in \sigma = S \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_n \wedge LHS(x) \in w_i \wedge LHS(y) \in w_i$

⌟

Exclusive context is achieved by context symbols that are mutually exclusive at runtime. By use of the mutual exclusive context symbols the deterministic selection of context-sensitive or unrestricted production rules with different *lhs* is ensured by enabling at most a single context-sensitive or unrestricted production rule for application in each reachable automaton configuration.

Definition 17 (Complete Context)
A formal grammar $G = (V, \Sigma, P, S)$ provides complete context if for each non-terminal N representing the processing symbol in a context sensitive or unrestricted production rule

(1) *there exist exclusively context-sensitive rules specifying the non-terminal N as processing symbol*
$p \in P \wedge N \in LHS(p) \wedge LHS(p) \setminus \{N\} \neq \emptyset \Rightarrow \forall q \in P \exists \alpha \in (\Sigma \cup V)^+ \wedge N \in LHS(q) \wedge \alpha \in LHS(q)$

(2) *the set of all rules specifying N as processing symbol covers all possible context alternatives that can occur at runtime.*
$\forall w_i \in \sigma = S \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_n \wedge N \in w_i$
$\exists p \in P \; \exists \alpha \in (\Sigma \cup V)^+ \wedge N \in LHS(p) \wedge \alpha = LHS(p) \setminus \{N\} \wedge \alpha \in w_i$

⌟

Property (1) in definition 17 ensures the deterministic selection of production rules with different *lhs* at runtime by enabling either context-free or context-sensitive (and unrestricted) production rules for application in each reachable automaton configuration. Furthermore, property (2) ensures the termination of the processing of a context-sensitive or unrestricted grammar, i.e. at least one production rule can be applied in case a processing symbol needs to be processed in context of other symbols.

Dynamic determinism ensures the deterministic selection of rules with the same *lhs* at runtime. The use of complex non-terminals enables the inclusion of dynamic information at runtime. Based on the dynamic information at runtime a deterministic selection of rules at runtime can be realized.

Definition 18 (Dynamic Determinism)
A formal grammar is dynamic deterministic, i.e. deterministic at runtime iff a single rule can be deterministically selected out of production rules specifying the same lhs *based on dynamic information at runtime.*

A formal grammar $G = (V, \Sigma, P, S)$ using complex non-terminals is dynamic deterministic iff each production rule with a lhs *that is also specified by another rule specifies a 3-dimensional non-terminal N on the* lhs *and exclusively a single non-terminal M on the* rhs *which need to be part of the third dimension of N:*
$\forall r \in \{p \in P \mid \exists q \in P \wedge q \neq p \wedge N \in V \wedge LHS(q) = LHS(p) \wedge N \in LHS(p) \wedge N \in LHS(q)\}$:

1. $r \in \Sigma^* V \Sigma^* \times V$
2. $(V \cap LHS(r)).d_3 \neq \emptyset$
3. $\exists x : RHS(r) = (V \cap LHS(r)).d_3(x)$

⌟

By use of the previous definitions the formal grammars that are used for service composition, i.e service grammars can be defined as presented in definition 19. Furthermore, definition 20 presents a substitution algorithm for service grammars determining the operational semantics. The presented substitution algorithm requires a normal form $V^*\Sigma^*$ for stored symbols at runtime. For the processing of a service grammar the approach at hand assumes a queued automaton that naturally provides the normal form required by the substitution algorithm. In particular, the queue of the queued automaton is assumed to store non-terminals whereas the tape exclusively stores terminals.

Definition 19 (Service Grammar)
A service grammar is a formal grammar $G = (V, \Sigma, P, S)$ providing dynamic determinism and exclusive, complete context with:

V - *Non-empty and finite set of complex non-terminals*

Σ - *Non-empty and finite set of terminals*

P - *Non-empty and finite set of c-interpreted production rules with variant, terminal-based context*

S - *Start symbol $S \in V$*

⌟

Definition 20 (Substitution Algorithm for Service Grammars)
Assuming a service grammar $G = (V, \Sigma, P, S)$ *and a sequence* $\Gamma = \Omega\alpha$ *of symbols with* $\Omega \in V^*$ *and* $\alpha \in \Sigma^*$*. Then, the substitution of symbols in* Γ *following the rules in* P *is defined by the function* $substitute_P : V^*\Sigma^* \rightarrow V^*\Sigma^*$ *with:*

$$substitute(\Gamma) = \begin{cases} f_P(\Gamma), & f_P(\Gamma) \cap V = \varepsilon; \\ substitute(f_P(\Gamma)), & f_P(\Gamma) \cap V \neq \varepsilon. \end{cases}$$

with:

$$f_P(\Gamma) = \begin{cases} \Gamma, & \Omega = \varepsilon; \\ \\ \Delta\ f_P(\beta\delta'\Omega'), & \Omega = N\Omega',\ N \in V,\ r = ruleSelection_P(\alpha, N), \\ & \delta = LHS(r) \cap \Sigma,\ \beta = \alpha \ominus \delta, \\ & \delta' = RHS(r) \cap \Sigma, \Delta \in RHS(r) \cap V. \end{cases}$$

⌟

Definition 21 (Rule Selection)
Assuming the processing of a service grammar $G = (V, \Sigma, P, S)$ *at a specific point in time the selection of a rule is defined by the function* $ruleSelection_P : \Sigma^* \times V \rightarrow (\Sigma^* V \times (\Sigma \cup V)^*)$

$ruleSelection(\alpha, N) = p \in P :$

$$\begin{cases} N \in LHS(p) \wedge (LHS(p) \cap \Sigma) \in \alpha, & \textit{1-dimensional } N; \\ N \in LHS(p) \wedge (LHS(p) \cap \Sigma) \in \alpha \wedge invoke(dim_2(N)), & \textit{2-dimensional } N; \\ N \in LHS(p) \wedge (LHS(p) \cap \Sigma) \in \alpha \wedge M \in RHS(p) & \\ \qquad \wedge o = invoke(dim_2(N)) \wedge\ dim_3(N, o) = M, & \textit{3-dimensional } N. \end{cases}$$

⌟

When processing a service grammar the set of applicable rules needs to be determined by the processing automaton. Definition 22 defines the set of applicable rules at runtime.

Definition 22 (Set of Applicable Rules)
Assuming a service grammar and a specific point in processing time providing a current processing symbol, i.e. non-terminal as well as a current context.

Then the set of applicable rules is determined by the set of production rules in the grammar that specify the current processing symbol on the lhs *and all other symbols on the* lhs *are part of the current context.* ⌟

While rule application, i.e. after reading the *lhs* and before writing the *rhs*, service invocation might be required if the automaton's current processing symbol is a two- or three-dimensional non-terminal. In detail:

Definition 23 (Need for Service Invocation while Rule Application)
A service invocation is required while rule application if

- *A two-dimensional non-terminal X is the processing symbol and*
  - *the set of applicable rules exclusively contains regular or context-free production rules;*
  - *the set of applicable rules A exclusively contains context-sensitive or unrestricted production rules and all rules in A specify significant modifications as effects to the processing of X:*
    $\forall r \in A : RHS \neq LHS \ominus X$
- *A three-dimensional non-terminal is processed and*
  - *the set of applicable rules exclusively contains regular or context-free production rules;*
  - *the set of applicable rules A exclusively contains context-sensitive or unrestricted production rules and all rules in A specify exclusively non-terminals on the* rhs*.*

⌟

3 Unified Model

This chapter introduces grammar-based representations for modeling constructs in existing service composition specification languages. At first, the representation of service calls in the unified model are discussed in general as service calls create the common basis for service composition specification language. Afterwards, the representations of modeling constructs in imperative, i.e. flow-based specification languages (e.g. BPEL [2], Scufl [3]) are discussed in detail. Finally, representations of modeling constructs in declarative, i.e. constraint-based specification languages (e.g. ConDec [4]) are discussed and an algorithm for the combination of grammar-based models realizing constraints is presented.

The unified model presented in this section use non-terminal types that are identified by studying BPEL and corresponding engines. In summary, information related to the engine's navigator is assumed to be implemented by service grammars, i.e. no external service and no non-terminal type is used for implementing navigator functionality. However, other engine functionality is assumed to be provided by external services. That means, corresponding services and non-terminal types are used in the unified model presented in this section. Table 1 summarizes the non-terminal types that are used in this section.

Table 1: Non-terminal types in the unified model.

Type of Non-Terminals	**Complexity of Associated Non-Terminals**	**Description**
Helpers	1-dimensional	Non-Terminals representing navigator-related information without association to an external service.
ExpressionEvaluator	2- or 3-dimensional	A (synchronous) web service operation that is responsible for expression evaluation, e.g. for expression-based data assignments or conditional control flow.
InsertData	2-dimensional	A (synchronous) web service operation of a reference resolution system that allows the creation of a data reference (cf. [8]).
GetData	2-dimensional	A (synchronous) web service operation of a reference resolution system that returns data related to a reference (cf. [8]).

Type of Non-Terminals	**Complexity of Associated Non-Terminals**	**Description**
UpdateData	2-dimensional	A (synchronous) web service operation of a reference resolution system that allows to change the data related to a reference (cf. [8]).
DeleteData	2-dimensional	A (synchronous) web service operation of a reference resolution system that allows the deletion of a data reference (cf. [8]).
AlarmService_For	2.dimensional	An (asynchronous) operation of a web service providing alarm events that are determined by `for` expressions (cf. section 3.2.6).
Event_X	3-dimensional	A (synchronous) web service operation that provides information about the occurrence of event X to the service composition (cf. section 3.2.6).
GetEvent_X	2-dimensional	A (synchronous) web service operation that provides an occurring event X to the service composition (cf. section 3.2.6).
Alarm_X	3-dimensional	A (synchronous) web service operation that provides information about the occurrence of an alarm X to the service composition (cf. section 3.2.6).
Fault_X	3-dimensional	A (synchronous) web service operation that provides information about the occurrence of a fault X to the service composition (cf. section 3.2.6).
GetFault_X	2-dimensional	A (synchronous) web service operation that provides an occurring fault X to the service composition (cf. section 3.2.6).
(Services)	(2-dimensional)	(An abstract type representing associated services that are composed by a service composition.)
Calculator	2-dimensional	A synchronous web service operation for calculations with numbers, e.g. operation add is provided for adding two numbers.
Calculator_{in}	2-dimensional	An asynchronous web service operation for calculating with numbers.
Calculator_{out}	2-dimensional	Callback of the asynchronous web service operation for calculating with numbers.

3.1 Service Calls

This section introduces the unified model for service calls. The called service is specified by use of WSDL [6]. The original models are specified by use of BPEL [2]. However, specifications by use of other service composition languages can be transformed in the same way as introduced in this section.

In general, a service call in the unified model requires a 2-dimensional non-terminal representing the service call. The definition of the 2-dimensional non-terminal requires information from the original service composition model, i.e. the invoked service operation as well as the invocation parameters. The service operation is specified in a non-terminal by use of a non-terminal type. A non-terminal type needs to be introduced for each operation that is invoked by the respective service composition. The non-terminal type specifies the service address and the associated operation. Assuming a web service that is defined by WSDL the definition of a non-terminal type requires information from the particular WSDL description (i.e. the location of the WSDL file, the service name, and the port) as well as information from the original service composition model (i.e. the called operation) where the information needs to be correlated by means of the called operation and the selected port.

Typically, service composition distinguishes between synchronous and asynchronous calls of services. For instance, BPEL provides the invoke activity for synchronous calls and for asynchronous calls. The unified model reflects these differences by use of non-terminal types but not by use of production rules. In particular, separated types exist for synchronous and asynchronous calls analogous to the different kinds of service operations. For the discussion of the unified model for service calls a sample web service, i.e. the calculator service is used in the following. The complete WSDL description of the calculator is presented in appendix A. In summary, the calculator service provides two operation, i.e. a synchronous and a asynchronous operation add.

Figure 1 shows a BPEL specification of a synchronous service call by use of the invoke activity. Figure 2 presents the unified model for a synchronous service call corresponding to the original model in figure 1. In the unified model the invoke activity in the original model is represented by a non-terminal S_1 of type *Calculator*. Rule (2) in the unified model realizes the execution of the service call. That means, the service call is processed between reading the *lhs* and writing the *rhs* of the production rule. After the service call a produced terminal s_1 indicates the successful finishing of the service call.

Figure 2(a) shows the xml-based representation of the non-terminal S_1. Figure 2(c) shows the xml-based representation of the non-terminal type `Calculator`. The service name, the partnerLink, the service operation, and the name of the WSDL port in the non-terminal type are adopted from the original BPEL specification of the service call or the WSDL file of the calculator service. Additionally, the non-terminal type explicitly specifies the location of the service operation implementation by a WS-Addressing

`EndpointReference` [7] that is adopted from the utilized WSDL port. The port is determined by use of the WSDL description of the calculator service. However, the selected port needs to specify the service operation that is specified for the service call in the original the BPEL model.

A service call does not necessarily finish successfully, i.e. can finish with returning a fault. The invoke activity allows to specify the faults that can be returned from the web service as well as fault handling logic. However, in BPEL an invoke activity that specifies fault handling is equivalent to a scope activity containing the invoke activity but specifying the fault handling logic by use of fault handlers. In the unified model a service call including fault handling is also specified by use of the equivalent model specifying scopes that are responsible for the fault handling. That means, the original model is transformed to the equivalent model specifying scopes at first. Afterwards, the equivalent model is transformed to the unified model. The unified model for scopes including fault handling is presented in the following. However, for reasons of simplicity exclusively service calls without fault handling are considered in the following.

The BPEL invoke activity also can be used for specifying an *asynchronous call of a web service*. In contrast to the synchronous call the asynchronous call only provides a variable for the input parameter but does not provide a variable for the output parameter. That means, the corresponding non-terminal also provides only an input parameter but no output parameter. However, a following receive activity needs to be specified in order to catch the response of the asynchronous call. Figure 3(b) shows the BPEL specification of an asynchronous call of the calculator service by use of the `invoke` activity with a following receive activity for the receiving of the response. For the asynchronous call an asynchronous operation `add2` is provided by the calculator service.

Figure 4 presents the unified model for an asynchronous service call corresponding to the original model in figure 3. The non-terminal S_1 represents the call of the calculator service that is originally specified by the `invoke` activity. The non-terminal S_2 represents the callback that is originally specified by the `receive` activity. Activities that are assumed to be executed in between the `invoke` activity and the `receive` activity are represented by the 1-dimensional helper non-terminal H.

Note that the operation specified in the receive activity is provided by the service composition instance but not by the calculator service. That means, the corresponding non-terminal type does not specify the endpoint for the calculator service. In particular, the non-terminal type $\texttt{Calculator}_{\texttt{out}}$ information that is adopted from a utility service that is related to the service composition instance, i.e. processing automaton. The utility service is responsible for receiving messages and providing received messages to the service composition instance. That means, for each incoming message the utility service provides two operations. One operation is called by other services for sending

```
<wsdl:definitions targetNamespace="http://example"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
                  xmlns:cal="http://calculator" >
    ...
    <plnk:partnerLinkType name="CalculatorPLT">
        <plnk:role name="CalculatorService" portType="cal:CalculatorPT" />
        </plnk:partnerLinkType>
</wsdl:definitions>
```

(a) WSDL description of the service composition that synchronously calls the calculator service (Composition.wsdl).

```
<process name="sampleProcess"
         targetNamespace="http://example"
         xmlns:tns="http://example"
       xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
         xmlns:cal="http://calculator" />
    <bpel:import importType="http://schemas.xmlsoap.org/wsdl/"
                 location="Calculator.wsdl"
                 namespace="http://calculator" />
    <bpel:import importType="http://schemas.xmlsoap.org/wsdl/"
                 location="Composition.wsdl"
                 namespace="http://example" />
    ...
    <bpel:partnerLinks>
        <bpel:partnerLink name="partnerLink1"
                          partnerLinkType="tns:CalculatorPLT"
                          partnerRole="CalculatorService" />
    </bpel:partnerLinks>
    ...
    <bpel:variables>
        <bpel:variable name="in" messageType="cal:addRequest" />
        <bpel:variable name="out" messageType="cal:addResponse" />
    </bpel:variables>
    ...
    <bpel:invoke partnerLink="partnerLink1"
                 operation="add"
                 inputVariable="in"
                 outputVariable="out" />
</bpel:process>
```

(b) BPEL-based service composition specifying a synchronous call of the operation `add` provided by calculator service.

Figure 1: Original service composition model specifying a synchronous call of the calculator service.

```
<nonTerminal>
    <name> S_1 </name>
  <type> Calculator </type>
  <parameters>
        <input>
            <reference> in </reference>
        </input>
        <output>
            <reference> out </reference>
        </output>
    </parameters>
</nonTerminal>
```

(a) Non-Terminal S_1 representing a synchronous service call.

$$\begin{array}{lrcl} (1) & Start & \longrightarrow & S_1 \\ (2) & S_1 & \longrightarrow & s_1 \\ \\ with: & S_1 & \hookleftarrow & Calculator \\ & S_1 & \in & V \\ & s_1 & \in & \Sigma \end{array}$$

(b) Production rules for the processing of a synchronous service call.

```
<nonTerminalType name=''Calculator''>
    <service>Calculator</service>
    <operation>   add  </operation>
    <port>CalculatorHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
            http://localhost:9763/services/Calculator.CalculatorHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
    <partnerLink> partnerLink1 </partnerLink>
</nonTerminalType>
```

(c) Non-terminal type `Calculator`

Figure 2: Unified service composition model specifying a synchronous call of the calculator service.

```
<wsdl:portType name="CalculatorCallbackPT">
    <wsdl:operation name="addResponse">
        ...
    </wsdl:operation>
    ...
</wsdl:portType>
...
<plnk:partnerLinkType name="CalculatorPLT">
    <plnk:role name="CalculatorService" portType="cal:CalculatorPT" />
    <plnk:role name="CalculatorRequester" portType="CalculatorCallbackPT" />
</plnk:partnerLinkType>
```

(a) Extension to the WSDL description of the service composition presented in figure 1 for enabling a asynchronously call of the calculator service (Composition.wsdl).

```
<partnerLinks>
    <partnerLink name="PartnerLink3"
                 partnerLinkType="CalculatorCallbackPLT"
                 myRole="CalculatorRequester"
                 partnerRole="CalculatorService"/>
</partnerLinks>
...
<invoke partnerLink="PartnerLink1"
        operation="add2"
        inputVariable="in" />
...
<receive partnerLink="PartnerLink3"
         operation="addResponse"
         variable="out" />
```

(b) Adaptations to the BPEL-based service composition in figure 1 for specifying an asynchronous call of the operation `add2` provided by calculator service.

Figure 3: Original service composition model specifying an asynchronous call of the calculator service.

```
<nonTerminal>
    <name> S_1 </name>
    <type> Calculator_in </type>
    <parameters>
        <input>
            <reference> in </reference>
        </input>
    </parameters>
</nonTerminal>
```

```
<nonTerminal>
    <name> S_2 </name>
    <type> Calculator_out </type>
    <parameters>
        <output>
            <reference> out </reference>
        </output>
    </parameters>
</nonTerminal>
```

(a) Non-Terminal S_1 representing an asynchronous service call and non-terminal S_2 representing the receiving of the response.

$$
\begin{array}{llll@{\qquad}rcl}
(1) & Start & \longrightarrow S_1 & with: & S_1 & \leftarrowtail & Calculator_{in} \\
(2) & S_1 & \longrightarrow s_1 H & & S_2 & \leftarrowtail & Calculator_{out} \\
(3) & H & \longrightarrow h_1 S_2 & & H & \leftarrowtail & Helpers \\
(4) & S_2 & \longrightarrow s_2 & & S_i, H & \in & V \\
 & & & & s_i & \in & \Sigma
\end{array}
$$

(b) Production rules for the processing of an asynchronous service call and the receiving of the response.

```
<nonTerminalType name="Calculator_in">
    <service>Calculator</service>
    <operation>   add2  </operation>
    <port>CalculatorHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
            http://localhost:9763/services/Calculator.CalculatorHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
     <partnerLink> PartnerLink1 </partnerLink>
</nonTerminalType>
<nonTerminalType name=‘‘\emph{Calculator_out}’’>
    <service>CompositionUtils</service>
    <operation>   addResponse  </operation>
    <port>CompositionUtilsHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
        http://localhost:9763/services/CompositionUtils.CompositionUtilsHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
    <partnerLink> PartnerLink3 </partnerLink>
</nonTerminalType>
```

(c) Non-terminal types $Calculator_{in}$ and $Calculator_{out}$.

Figure 4: Unified service composition model specifying an asynchronous service call of the calculator service.

a message to the service composition instance. The second operation is called by the service composition for fetching a received message.

3.2 Imperative Languages

This section introduces grammar-based representations for the most common control and data flow constructs. The transformation to the unified model is conceptually described. That means, the grammar-based representation is introduced for a single representative of the modeling constructs instead of discussing all forms of the modeling constructs. For instance, looping control flow is discussed by use of a while loop whereas other loops (e.g. repeat-until, for) can be transformed in the same way.

3.2.1 Sequential Control Flow

Figure 5 presents the unified model for sequential control flow by use of a sequence of service calls S_1, S_2, and S_3. Sequential control flow is realized by production rules specifying the sequential dependencies between the activation and finishing of activities, e.g. service calls. In particular, rule (2) in figure 5 realizes the execution of service S_1 and the following activation of the service call S_2.

```
<sequence>
    <invoke name="S_1" ... />
    <invoke name="S_2" ... />
    <invoke name="S_3" ... />
</sequence>
```

(a) BPEL-based `sequence` activity specifying the successive service calls S_1, S_2, and S_3.

$$
\begin{array}{lrcl}
(1) & Start & \longrightarrow & S_1 \\
(2) & S_1 & \longrightarrow & s_1 S_2 \\
(3) & S_2 & \longrightarrow & s_2 S_3 \\
(4) & S_3 & \longrightarrow & s_3 \\
 & & & \\
with: & S_i & \leftarrowtail & Services \\
 & S_i & \in & V \\
 & s_i & \in & \Sigma
\end{array}
$$

(b) Production rules for sequential control flow, i.e. the sequence S_1, S_2, S_3 is executed.

Figure 5: Unified model for sequential control flow.

3.2.2 Alternative Control Flow

Control flow alternatives are realized by alternative production rules that are activated at the same time in the unified model. For instance, the following production rules concurrently activate the rules (2) and (3) realizing the execution of the service call S_1 but specifying different alternatives for further processing. However, both rules specify alternatives for the further processing. The decision about selecting rule (2) or rule (3) is non-deterministically taken in general.

$$\begin{array}{lrcl}
(1) & Start & \longrightarrow & S_1 \\
(2) & S_1 & \longrightarrow & s_1 S_2 \\
(3) & S_1 & \longrightarrow & s_1 S_3 \\
(4) & S_2 & \longrightarrow & s_2 \\
(5) & S_3 & \longrightarrow & s_3
\end{array}
\qquad
\begin{array}{lrcl}
with: & S_i & \in & Services \\
 & s_i & \in & \Sigma
\end{array}$$

Conditional Control Flow Alternative control flow usually becomes deterministic by including conditions. The conditions decide for each alternative control flow path whether it is executed. Figure 6(c) presents the unified model for conditional control flow. The production rules in figure 6(b) are static non-deterministic as the rules (3) and (4) are concurrently activated. However, the production rules are assumed to be dynamic deterministic, i.e. a single rule can be deterministically selected at runtime. In detail, the 3-dimensional non-terminal C_1 represents the call of an expression evaluator supporting the particular condition language, e.g. an XPath-solver for XPath expressions. After the execution of the call the return value is mapped to a grammar-based representation, i.e. a non-terminal corresponding to the third dimension of C_1 (cf. figure 6(c)). In dependence of the mapped non-terminal the production rule (2) or (3) in figure 6(b) is deterministically selected for further processing.

3.2.3 Looping Control Flow

Figure 7 presents the unified model for looping control flow. The non-terminal L_1 represents the entry point of the loop that is repeatedly activated. The loop condition is specified in the 3-dimensional non-terminal C_1 representing a call of the expression evaluator. In dependence of the evaluation result the loop body is executed, i.e. the sequential services S_1 and S_2 are executed.

3.2.4 Parallel Control Flow

Figure 8 presents the unified model for parallel control flow. Rule (1) simultaneously activates the parallel service calls S_1, S_2, and S_3.[1] Additionally, a 1-dimensional non-terminal H_2 representing the need for synchronization is activated. Rule (5) realizes

[1]The order of symbols is specified w.l.o.g. in the c-interpreted production rules.

```
<if>
    <condition>  \$account/type="Gold"  </condition>
        <invoke name="S_2" .../>
    <else>
        <invoke name="S_3" .../>
    </else>
</if>
```

(a) BPEL-based `if` activity specifying condition control flow concerning the alternative service calls S_2 and S_3.

$$
\begin{array}{lrcl}
(1) & Start & \longrightarrow & C_1 \\
(2) & C_1 & \longrightarrow & T_1 \\
(3) & C_1 & \longrightarrow & F_1 \\
(4) & T_1 & \longrightarrow & S_2 \\
(5) & F_1 & \longrightarrow & S_3 \\
(6) & S_2 & \longrightarrow & s_2 \\
(7) & S_3 & \longrightarrow & s_3
\end{array}
\qquad
with: \quad
\begin{array}{rcl}
S_i & \longleftarrow & Services \\
C_1 & \longleftarrow & ExpressionEvaluator \\
T_1, F_1 & \longleftarrow & Helpers \\
S_i, C_1, T_1, F_1 & \in & V \\
s_i & \in & \Sigma
\end{array}
$$

(b) Production rules for conditional control flow concerning the alternative service calls S_2 and S_3.

```
<nonTerminal>
    <name> C_1 </name>
    <type> ExpressionEvaluator </type>
    <parameters>
        <input>
            <reference position="1"> account </reference>
            <value position="2"> account/type="Gold" </value>
        </input>
    </parameters>
    <relations>
        <relation>
            <outputValue> True </outputValue>
            <nonTerminalRef> T_1 </nonTerminalRef>
        </relation>
        <relation>
            <outputValue> False </outputValue>
            <nonTerminalRef> F_1 </nonTerminalRef>
        </relation>
    </relations>
</nonTerminal>
```

(c) Non-terminal C_1 representing condition evaluation.

Figure 6: Unified model for conditional control flow.

```
<sequence>
    <while>
        <condition>  x > 5  </condition>
            <sequence>
                <invoke name="S_1" .../>
                <invoke name="S_2" .../>
            </sequence>
    </while>
    <invoke name="S_3" .../>
</sequence>
```

(a) BPEL-based while activity specifying the successive service calls S_1 and S_2 in the loop body and a successive service call S_3.

$$
\begin{array}{rrcl}
(1) & Start & \longrightarrow & L_1 \\
(2) & L_1 & \longrightarrow & C_2 \\
(3) & C_2 & \longrightarrow & T_1 \\
(4) & C_2 & \longrightarrow & F_1 \\
(5) & T_1 & \longrightarrow & S_1 \\
(6) & F_1 & \longrightarrow & S_3 \\
(7) & S_1 & \longrightarrow & s_1 S_2 \\
(8) & S_2 & \longrightarrow & s_2 L_1 \\
(9) & S_3 & \longrightarrow & s_3
\end{array}
\qquad
\begin{array}{lrcl}
with: & S_i & \hookleftarrow & Services \\
 & C_2 & \hookleftarrow & ExpressionEvaluator \\
 & L_1, T_1, F_1 & \hookleftarrow & Helpers \\
 & S_i, C_2, L_1, T_1, F_1 & \in & V \\
 & s_i & \in & \Sigma
\end{array}
$$

(b) Production rules for looping control flow with the successive service calls S_1 and S_2 in the loop body and a successive service call S_3.

Figure 7: Unified model for looping control flow.

the execution of the synchronization in case the execution of the parallel control flow paths is finished.

```
<sequence>
    <flow>
        <invoke name="S_1" .../>
        <invoke name="S_2" .../>
        <invoke name="S_3" .../>
    </flow>
    <invoke name="S_4" .../>
</sequence>
```

(a) BPEL-based `flow` activity specifying the parallel service calls S_1, S_2 and S_3.

$$\begin{array}{llcl}
(1) & Start & \longrightarrow & S_1S_2S_3H_2 \\
(2) & S_1 & \longrightarrow & s_1 \\
(3) & S_2 & \longrightarrow & s_2 \\
(4) & S_3 & \longrightarrow & s_3 \\
(5) & s_1s_2s_3H_2 & \longrightarrow & s_1s_2s_3S_4 \\
(6) & S_4 & \longrightarrow & s_4 \\
 & & & \\
with: & S_i & \longleftarrow\!\!\prec & Services \\
 & H_2 & \longleftarrow\!\!\prec & Helpers \\
 & S_i, H_2 & \in & V \\
 & s_i & \in & \Sigma
\end{array}$$

(b) Production rules for parallel control flow, i.e. the service calls S_1, S_2 and S_3 are executed in parallel.

Figure 8: Unified model for parallel control flow.

Links In BPEL a flow activity allows to specify links between activities that are contained in the flow activity. The links specify sequential dependencies are represented by sequential control flow in the unified model. Assuming an additional service call S_x in figure 8(a) and a link between the service call S_1 and the newly introduced service call S_x requires the substitution of the rules (2) and (5) by the following rules (2') and (5') as well as the introduction of the following rule (7):

$$\begin{array}{lrl}
(2') & S_1 & \longrightarrow s_1S_x \\
(5') & s_1s_2s_3s_x & \longrightarrow s_2s_3s_xS4 \\
(7) & S_x & \longrightarrow s_x
\end{array}$$

Additionally, transition conditions and join conditions can be specified in case links are specified in a BPEL flow activity. Transition conditions are evaluated by use of the external expression evaluation service. In contrast, join conditions can be internally evaluated as join conditions in BPEL are allowed to exclusively specify link statuses and operations on boolean values. However, an explicit representation of link statuses by terminal symbols is needed in the unified model for enabling the internal evaluation of join conditions. That means, the result of a transition condition evaluation must be mapped to a grammar-internal representation instead of storing the result in a variable.

Figure 9 shows a BPEL-based original model with an explicit transition condition and an explicit join condition in linked control flow. Figure 10 presents the unified model corresponding to the original model in figure 9. The evaluation of the transition condition of the link from S_1 to S_3 is represented by a 3-dimensional non-terminal C_4 in the unified model. The evaluation result is mapped to a non-terminal T_1 or F_1 corresponding to the third dimension of the non-terminal C_4. Afterwards, the non-terminals T_1 and F_1 are substituted by the terminals t_1 and f_1 as the represented information is context information for the production rules realizing the join condition evaluation and production rules in service grammars are assumed to provide terminal-based context. The default transition condition of the link from S_2 to S_3 is not evaluated by use of the external evaluation service as the default transition condition is evaluated to `true` by definition. In contrast, the evaluation result of the default transition condition of the link from S_2 to S_3 is immediately represented by a terminal t_2 in the unified model.

The join condition is internally evaluated in the unified model presented in figure 10, i.e. no external service is used for the evaluation. In particular, the production rules realize the evaluation of the join condition by implementing the lookup table for the join condition on the *lhs*. For instance, the rules (8–10) in figure 10 represent the rows in the lookup table that are evaluated to true. In contrast, rule (11) represents the single row in the lookup table that is evaluated to false.

Dead Path Elimination In case a join condition is evaluated to false the corresponding activity is not executed and the status of all outgoing links of this activity needs to be set to false. That means, the link status false is propagated along successive links until a join condition is evaluated to true. In BPEL this approach is called Dead-Path Elimination. Figure 11 illustrates the Dead-Path Elimination in the unified model. The service call S_6 specifies a single incoming link, the default join condition, and a single outgoing link. Because of the default join condition the service call S_6 is not allowed to be executed if the status of the incoming link is false and the link status false needs to be propagated to the outgoing link of S_6. Rule (10) in figure 11 realizes the propagation of the link status `false` in case the default join condition of the service call S_6 is evaluated to false. In particular, the terminal f_5 representing the value false for the status of the link from S_5 to S_6 is substituted by the terminal f_6 representing the value false for the status of the link from S_6 to S_7.

3.2.5 Data Handling

In the unified model data values are explicitly represented only if the particular data is needed for the service composition logic. In detail, the handling of data by value is only supported for data specifying decisions about the processing of control flow alternatives or for data specifying states of scopes. Other data is handled by reference as it is

```
<sequence>
    <flow>
        <links>
            <link name="1to3" />
            <link name="2to3" />
        </links>
        <invoke name="S_1" ... >
            <sources>
                <source linkName="1to3">
                    <transitionCondition>contains($var1, $var2)</transitionCondition>
                </source>
            </sources>
        </invoke>
        <invoke name="S_2" ...>
            <sources>
                <source linkName="1to3" />
            </sources>
        </invoke>
        <invoke name="S_3" ...>
            <targets>
                <joinCondition>$1to3 or $2to3</joinCondition>
                <target linkName="1to3">
                <target linkName="2to3">
            </targets>
        </invoke>
    </flow>
    <invoke name="S_4" .../>
</sequence>
```

Figure 9: BPEL-based `flow` activity specifying links with explicit transition and join condition.

$$
\begin{array}{rrl}
(1) & Start & \longrightarrow S_1 S_2 J_1 H_3 \\
(2) & S_1 & \longrightarrow s_1 C_4 \\
(3) & S_2 & \longrightarrow s_2 t_2 \\
(4) & C_4 & \longrightarrow T_1 \\
(5) & C_4 & \longrightarrow F_1 \\
(6) & T_1 & \longrightarrow t_1 \\
(7) & F_1 & \longrightarrow f_1 \\
(8) & t_1 t_2 J_1 & \longrightarrow S_3 \\
(9) & t_1 f_2 J_1 & \longrightarrow S_3 \\
(10) & f_1 t_2 J_1 & \longrightarrow S_3 \\
(11) & f_1 f_2 J_1 & \longrightarrow j_3 \\
(12) & S_3 & \longrightarrow s_3 j_3 \\
(13) & j_3 H_3 & \longrightarrow S_4 \\
(14) & S_4 & \longrightarrow s_4
\end{array}
\qquad
with:
\qquad
\begin{array}{rcl}
S_i & \leftarrowtail & Services \\
C_4 & \leftarrowtail & ExpressionEvaluator \\
F_1, T_1, J_1 & \leftarrowtail & Helpers \\
S_i, C_4, F_1, T_1, J_1 & \in & V \\
s_i, j_3, t_1, f_1 & \in & \Sigma
\end{array}
$$

(a) Production rules realizing an internal evaluation of a join condition and an external evaluation of a transition condition.

```
<nonTerminal>
    <name> C_4 </name>
    <type> ExpressionEvaluator </type>
    <parameters>
        <input>
            <reference position="1"> var1 </reference>
            <reference position="1"> var2 </reference>
            <value position="2"> contains(var1, var2) </value>
        </input>
    </parameters>
    <relations>
        <relation>
            <outputValue> True <outputValue>
            <nonTerminalRef> T_1 </nonTerminalRef>
        </relation>
        <relation>
            <outputValue> False <outputValue>
            <nonTerminalRef> F_1 </nonTerminalRef>
        </relation>
    </relations>
</nonTerminal>
```

(b) Non-Terminal C_4 representing the evaluation of the transition condition of the link from S_1 to S_3.

Figure 10: Unified model for the evaluation of transition conditions and join conditions corresponding to the original model in figure 9.

```
<flow>
    <links>
        <link name="5to6" />
        <link name="6to7" />
    </links>
    <invoke name="S_5" ... >
        <sources>
            <source linkName="5to6" >
                <transitionCondition> contains($var1, $var2) </transitionCondition>
            </source>
        </sources>
    </invoke>
    <invoke name="S_6" ...>
        <targets> <target linkName="5to6" />  </targets>
        <sources>
            <source linkName="6to7" />
                <transitionCondition> contains($var3, $var4) </transitionCondition>
            </source>
        </sources>
    </invoke>
    <invoke name="S_7" ...>
        <targets> <target linkName="6to7" />  </targets>
    </invoke>
</flow>
```

(a) BPEL-based `flow` activity specifying a link-based sequence of service calls S_5, S_6, and S_7 as well as explicit transition conditions.

$$
\begin{array}{rrcl}
(1) & Start & \longrightarrow & S_5 J_6 H_7 \\
(2) & S_5 & \longrightarrow & s_5 C_5 \\
(5) & C_5 & \longrightarrow & T_5 \\
(6) & C_5 & \longrightarrow & F_5 \\
(7) & T_5 & \longrightarrow & t_5 \\
(8) & F_5 & \longrightarrow & f_5 \\
(9) & t_5 J_6 & \longrightarrow & S_6 J_7 \\
(10) & f_5 J_6 & \longrightarrow & f_6 J_7 \\
(11) & S_6 & \longrightarrow & s_6 C_6 \\
(16) & C_6 & \longrightarrow & T_6 \\
(17) & C_6 & \longrightarrow & T_6 \\
(18) & T_6 & \longrightarrow & t_6 \\
(19) & F_6 & \longrightarrow & f_6 \\
(20) & t_6 J_7 & \longrightarrow & S_7 \\
(21) & f_6 J_7 & \longrightarrow & j_7 \\
(22) & S_7 & \longrightarrow & s_7 j_7 \\
(23) & j_7 H_7 & \longrightarrow & ...
\end{array}
\qquad
with: \quad
\begin{array}{rcl}
S_i & \hookleftarrow & Services \\
C_i & \hookleftarrow & ExpressionEvaluator \\
J_i, T_i, F_i, H_7 & \hookleftarrow & Helpers \\
S_i, C_i, H_7, J_i, T_i, F_i & \in & V \\
s_i, d_i, j_i, t_i, f_i & \in & \Sigma
\end{array}
$$

(b) Production Rules illustrating Dead-Path Elimination in a link-based sequence of service calls S_5, S_6, and S_7 in combination to the internal evaluation of join conditions.

Figure 11: Unified model illustrating Dead-Path Elimination in combination with the internal evaluation of join conditions.

simply transmitted from service to service, i.e. the service composition does not need to be aware of concrete values. Data references are specified in the second dimension of non-terminals.

For the handling of data by reference the unified model uses an external data management service implementing a reference resolution system [8]. In detail, the reference resolution system is responsible for storing data in a database as well as for assigning and resolve reference identifiers. The non-terminal types *InsertData*, *GetData*, *UpdateData*, and *DeleteData* are associated with the data management service but address different operations. However, the output of non-terminals of type *InsertData* does not specify the storage location of the operation return value. Instead, the output specifies the variable name that needs to be mapped to the reference identifier that is returned by the service operation. Furthermore, the input of non-terminals of type *DeleteData* does not specify the storage location of the operation input value. Instead, the input specifies the variable name that needs to be mapped to the reference identifier that is the concrete input parameter for the service operation. The processing automaton is responsible for the mapping of variable names and reference identifiers.

Data Containers Figure 12 presents the unified model for variable creation, variable initialization with constants, and variable deletion. For each variable a non-terminal of type *InsertData* representing the storage allocation is introduced. In case the variable is initialized by a constant the initial value is specified as input parameter. Otherwise the value `null` is specified as input parameter. As mentioned before, the output parameter needs to be mapped to the reference identifier that is returned by the data management service. Furthermore, a non-terminal of type *DeleteData* representing the deallocation of the storage needs to be introduced for each variable. The non-terminals of type *DeleteData* are processed at the end of the lifetime of the corresponding variable. In general, the creation of variables needs to be processed sequentially as variables can be initialized by use of previously created variables. In contrast, the deletion of variables can be processed simultaneously.

A variable that is initialized by use of expressions but not by constants requires an additional non-terminal of type *ExpressionEvaluator* that is processed immediately after the storage allocation, i.e the non-terminal of type *insertData*. However, the initialization by use of expressions is equivalent to data assignments using expressions.

Data Assignment The unified model for assignments of constants is presented in figure 13. For each assignment of a constant a single non-terminal (e.g. D_1) of type *UpdateData* is introduced. The first input parameter of the associated service operation specifies the data reference whereas the second input parameter specifies the new value of the referenced data. That means, the first input parameter of the introduced non-

```
<process>
    <variables>
        <variable name="varX"
                  messageType="sResponse"/>
        <variable name="varY"
                  type="xsd:integer">
            <from> 1 </from>
        <\variable>
    </variables>
    ...
</process>
```

(a) BPEL-based variable declaration and initialization with constants.

$$
\begin{array}{lrcl}
(1) & Start & \longrightarrow & D_1 \\
(2) & D_1 & \longrightarrow & D_2 \\
(3) & D_2 & \longrightarrow & ... \\
(4) & ... & \longrightarrow & D_3 D_4 \\
(5) & D_3 & \longrightarrow & \varepsilon \\
(6) & D_4 & \longrightarrow & \varepsilon \\
\\
with: & D_1, D_2 & \leftarrowtail & InsertData \\
 & D_3, D_4 & \leftarrowtail & DeleteData \\
 & D_i & \in & V
\end{array}
$$

(b) Production rules for variable creation, initialization with constants, and variable deletion.

```
<nonTerminal>
    <name> D_1 </name>
    <type> InsertData </type>
    <parameters>
        <input>
            <value> null </value>
        </input>
        <output>
            <reference> varX </reference>
        </output>
    </parameters>
</nonTerminal>
```

```
<nonTerminal>
    <name> D_2 </name>
    <type> InsertData </type>
    <parameters>
        <input>
            <value> 1 </value>
        </input>
        <output>
            <reference> varY </reference>
        </output>
    </parameters>
</nonTerminal>
```

(c) Non-Terminals D_1 and D_2 for variable creation and initialization with constants.

```
<nonTerminal>
    <name> D_3 </name>
    <type> DeleteData </type>
    <parameters>
        <input>
            <reference>
                varX
            </reference>
        </input>
    </parameters>
</nonTerminal>
```

```
<nonTerminal>
    <name> D_4 </name>
    <type> DeleteData </type>
    <parameters>
        <input>
            <reference>
                varY
            </reference>
        </input>
    </parameters>
</nonTerminal>
```

(d) Non-Terminals D_3 and D_4 for variable deletion.

Figure 12: Unified model for data variables.

terminal (e.g. D_1) is given by the content of the assignment target whereas the second input parameter is given by the assignment source.

```
<assign>
    <copy>
        <from> 24 </from>
        <to variable="x" />
    </copy>
</assign>
...
```

(a) BPEL specification of an assignment of constants.

$$
\begin{array}{lrl}
(1) & Start & \longrightarrow D_1 \\
(2) & D_1 & \longrightarrow .. \\
 & & \\
with: & D_1 & \hookleftarrow UpdateData \\
 & D_1 & \in V
\end{array}
$$

(b) Production rules for the assignment of constants.

```
<nonTerminal>
    <name> D_1 </name>
    <type> UpdateData  </type>
    <parameters>
        <input>
            <reference position="1">
                x
            </reference>
            <value position="2">
                24
            </value>
        </input>
    </parameters>
</nonTerminal>
```

(c) Non-terminal for the assignment of constants.

Figure 13: Unified model for assignment of constants.

The unified model for assignments using expressions is presented in figure 14. Similar to conditions the expression that is used by an assignment needs to be evaluated. However, the return value of the evaluator need to be stored in a data container in contrast to condition evaluation where the return value needs to be mapped to a non-terminal. That means, for data assignment the handling of the return value is specified in the second dimension of the corresponding non-terminal.

In the unified model for each assignment using an expression a non-terminal (e.g. E_3 in figure 14) of type *ExpressionEvaluator* is introduced. The non-terminal represents the evaluation of the expression and the storage of the evaluation result in the assignment target. That means, the non-terminal specifies the assignment source, i.e. the expression as well as variables that are used in the expression as input parameters.

Additionally, the non-terminal specifies the assignment target as output parameter for enabling the storing of the assigned data, i.e. the expression evaluation result.

```
<assign>
    <copy>
        <from>
        $bookstore/book[price>25]/title
        </from>
        <to variable="titles" />
    </copy>
</assign>
...
```

(a) BPEL specification for assignments using expressions.

$$
\begin{array}{llll}
(1) & Start & \longrightarrow E_3 \\
(2) & E_3 & \longrightarrow ... \\
 & & \\
with: & E_3 & \longleftarrow ExpressionEvaluator \\
 & E_3 & \in V
\end{array}
$$

(b) Production rules for assignments using expressions.

```
<nonTerminal>
    <name> E_3 </name>
    <type> ExpressionEvaluator </type>
    <parameters>
        <input>
            <reference position="1"> bookstore </reference>
            <value position="2"> bookstore/book[price>25]/title </value>
        </input>
        <output>
            <reference> titles </reference>
        </output>
    </parameters>
</nonTerminal>
```

(c) NonTerminals for assignmnents using expressions.

Figure 14: Unified model for assignments using expressions.

In case the assignment source is a child element of a variable, e.g. a BPEL variable part, the input parameter of the non-terminal realizing the data assignment needs to be prepared. In particular, the child element needs to be prefixed to the expression. For instance, figure 15 assumes a variable `myAddRequest` storing an input message `addRequest` for the calculator service. The WSDL file (cf. section A) specifies the message `addRequest` that contains a part `parameters` referencing to an element `add`. That means, the variable `myAddRequest` is required to contain a child element `add`. Consequently, the expression that is used to determine assignment source needs to be prefixed by the `add` element (cf. figure 15(c)).

In case the assignment target is a child element of a variable (e.g. a BPEL variable part) the storage of the assigned data requires further preparation. At first, the variable enclosing the assignment target needs to be fetched by use of a non-terminal of type *GetData*. Afterwards, XSL Transformation (XSLT) [5] can to be applied for changing

```
<assign>
    <copy>
        <from variable="myAddRequest"
              part="parameters"/>
            <query>
              /x
            </query>
        <to variable="y" />
    </copy>
</assign>
...
```

(a) BPEL specification of an assignment of a variable part.

$$
\begin{array}{llrl}
(1) & & Start & \longrightarrow E_6 \\
(2) & & E_6 & \longrightarrow \ldots \\
 & & & \\
with: & & E_6 & \longleftarrow ExpressionEvaluator \\
 & & E_6 & \in V
\end{array}
$$

(b) Production rules for the assignment of a variable part.

```
<nonTerminal>
    <name> E_6 </name>
    <type> ExpressionEvaluator </type>
    <parameters>
        <input>
            <reference position="1">
                myAddRequest
            </reference>
            <value position="2">
                /add/x
            </value>
        </input>
        <output>
            <reference> y </reference>
        </output>
    </parameters>
</nonTerminal>
```

(c) Non-terminals for the assignment of a variable part.

Figure 15: Unified model for assignments of variable parts.

the data of the variable's child element. However, the XSLT stylesheet needs to be parameterized with the assignment source. Finally, the changed variable needs to be stored by use of a non-terminal of type *UpdateData*.

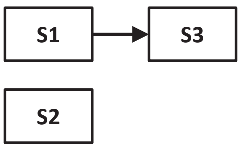
Data Flow Data flow in service compositions specifies the transfer of data between service calls or other tasks. Data flow always implies control flow, i.e. a data dependency from A to B implies a sequential control dependency from A to B. Figure 16 presents the unified model for data flow from a service call S_1 to a service call S_3. Figure 16(a) shows a graphical representation of the original model where the SCUFL specification language is used as SCUFL allows an explicit specification of data flow. The xml-based specification of the SCUFL workflow in figure 16(a) is presented in section A.2.

Rule (2) in the unified model realizes the sequential control dependency corresponding to the data flow between S_1 and S_3. The data aspect of the data flow is realized by the non-terminals S_1 and S_3 where the output parameter `x` of S_1 is us as input parameter of S_3. The independent service call S_2 is activated at the very beginning of the service composition. In particular, S_2 is activated in rule (1) simultaneously to S_1 as S_2 has no data dependency to S_1 and S_3.

3.2.6 Scopes

Scopes isolate the lifetime of data containers, i.e. variables and restrict their visibility allowing the reuse of variable names. The life time of data variables is correlated with the life time of scopes, i.e. variables are created, accessible and deleted in the context of a specific scope. Figure 17 presents the unified model for a scope containing a variable as well as a service call. The activation of the scope is represented by non-terminal R_1 of the type *Helpers*. Rule (1) activates the scope by producing the non-terminal R_1 simultaneously to the creation of the data variable represented by D_1. After the variable creation the first task in the scope, i.e. the service call S_1 is activated. The last task in the scope needs to create a terminal b_1 indicating the ability to complete the scope. Rule (4) completes the scope and activates the deletion of the contained variable by producing the non-terminal D_2.

In BPEL a scope additionally provides event handlers, fault handlers, termination handlers, and compensation handlers. The realization of these handlers in the unified model is discussed in the following. Typically, events and faults are produced in the environment and need to be integrated at runtime. The approach at hand integrates information from the environment by calling the utility service corresponding to a service composition instance, i.e. processing automaton. The utility service is responsible for receiving events and faults and providing them to the service composition instance.



(a) Graphical representation of a scufl workflow with data flow from service call S_1 to service call S_3.

$$
\begin{array}{llcl}
(1) & Start & \longrightarrow & S_1 S_2 \\
(2) & S_1 & \longrightarrow & s_1 S_3 \\
(3) & S_2 & \longrightarrow & s_2 \\
(4) & S_3 & \longrightarrow & s_3
\end{array}
$$

$$
\begin{array}{ll}
with & S_i \leftarrowtail Services \\
 & S_i \in V \\
 & s_i \in \Sigma
\end{array}
$$

(b) Production rules specifying the control dependency between the service calls S_1 and S_3 resulting from the data flow.

```
<nonTerminal>
    <name> S_1 </name>
    <type> Services </type>
    <parameters>
        <input> ... </input>
        <output>
            <reference>
                x
            </reference>
        </output>
    </parameters>
</nonTerminal>
```

```
<nonTerminal>
    <name> S_3 </name>
    <type> Services </type>
    <parameters>
        <input>
            <reference>
                x
            </reference>
        </input>
        <output> ...  </output>
    </parameters>
</nonTerminal>
```

(c) Non-Terminals S_1 and S_3 specifying the data aspect of the data flow by parameters.

Figure 16: Unified model for data flow.

```
<scope name="R_1">
    <variables>
        <variable name="myVarX"
                  type="xs:integer"/>
    </variables>
    <invoke name="S_1" ... />
</scope>
```

(a) BPEL-based scope activity defining a variable and containing a service call.

$$
\begin{array}{lrcl}
(1) & Start & \longrightarrow & D_1 R_1 \\
(2) & D_1 & \longrightarrow & S_1 \\
(3) & S_1 & \longrightarrow & s_1 b_1 \\
(4) & b_1 R_1 & \longrightarrow & D_2 \\
(5) & D_2 & \longrightarrow & \varepsilon \\
 & & & \\
with: & R_1 & \longleftarrow & Helpers \\
 & D_1 & \longleftarrow & InsertData \\
 & S_1 & \longleftarrow & Services \\
 & D_2 & \longleftarrow & DeleteData \\
 & D_i, R_1, S_1 & \in & V \\
 & s_1, b_1 & \in & \Sigma
\end{array}
$$

(b) Production rules for a scope defining a variable and containing a service call.

Figure 17: Unified model for scopes

Event Handlers Event handlers are executed concurrently to the regular logic of the corresponding scope. Figure 18 shows a BPEL scope R_1 specifying a message event handler and an alarm event handler by an `onEvent` element and an `onAlarm` element, respectively. The unified model for event handlers is presented in figure 19 corresponding to the original model in figure 18. The detailed specifications of the 2- and 3-dimensional non-terminals in the unified model for event handlers are presented in section B. The meaning of non-terminals and terminals in the unified model for event handlers is summarized in the following:

R_i**:** Represents the activated scope R_i

D_i**:** Represents the insertion or the deletion of a data variable at the beginning or end of a scope

E_i**:** Represents the evaluation of an alarm expression

A_i**:** Represents a call of the alarm service

H_i**:** Represents a helper for synchronization

S_i**:** Represents a call of a composed service

M_i**:** Represents the check for the occurrence of an event by use of the utility service

G_i: Represents a call of the utility service for getting an occurred event and storing the event in a local variable

U_i: Represents a call of the utility service for informing the service about the completion of the scope, i.e. following events need to be rejected

T_i: Represents a helper symbolizing the boolean value true

F_i: Represents a helper symbolizing the boolean value true

x_i: Indicates the execution of the regular logic of scope R_i

y_i: Indicates a running event handler and separates the regular processing of events from the processing of waiting events

b_i: Indicates the finishing of the processing of the regular logic in scope R_i

u_i: Indicates that the utility service is aware of the completion of the scope's regular logic

m_i: Indicates that no waiting event for a specific event handler is remaining

For each alarm handler some preparation is needed at the very beginning of the associated scope. In particular, the alarm expression needs to be evaluated (cf. E_1) and the alarm service needs to be called for starting the clock and ordering alarms (cf. A_1). Afterwards, the regular logic of the scope can be executed and the event handlers are enabled to process events. Rule (4) in figure 19 produces a terminal x_1 indicating the execution of the regular logic of scope R_1 and activates the first task, i.e. service call S_1. Furthermore, rule (4) activates the checking for events by a non-terminal M_i for each event handler as well as a non-terminal H_1 enabling the synchronization after the event handling when the regular logic of scope R_1 finishes.

The 3-dimensional non-terminals M_i are repetitively activated as long as the regular logic of scope R_1 is executed (cf. rule (10), (18), (23), (26)). That means, the instances of an event handler are executed successively in the unified model as the number of required handler instances cannot be statically determined. However, in case the check for an event was true the corresponding event handler is activated. For instance, the message event handler, i.e. scope R_2 is activated in rule (9). At the very beginning of the scope R_2 two variables are created by use of the non-terminals D_2 and D_3. Afterwards, the message event is stored in a previously created variable by use of the non-terminal G_1. Finally, the regular logic of the event handler scope is executed and the scope is completed. The event handler is allowed to be executed again in case the regular logic of scope R_1 is still be executed. Rule (18) and (19) realize the reactivation or the stopping of reactivating the check for events in dependence of the existence of the mutual existing terminals x_1 and b_1.

```
<scope name="R_1" >
    <variables>
        <variable name="duration" type="xs:integer">
            <from> 4 </from>
        </variable>
    </variables>
    <eventHandlers>
        <onEvent partnerLink="event1PL" operation="event1op" variable="myEvent">
            <scope name="R_2">
                <variables>
                    <variable name="b" type="xs:string" />
                </variables>
                <invoke name=" S_3" />
            </scope>
        </onEvent>
        <onAlarm>
            <for>$duration</for>
            <scope name = "R_3">
                <invoke name=" S_4" />
            </scope>
        </onAlarm>
    </eventHandlers>
    <sequence>
        <invoke name=" S_1" />
        <invoke name=" S_2" />
    </sequence>
</scope>
```

Figure 18: BPEL specification of a scope with event handlers.

(1) $Start \longrightarrow D_1 R_1$
(2) $D_1 \longrightarrow E_1$
(3) $E_1 \longrightarrow A_1$
(4) $A_1 \longrightarrow x_1 M_1 M_2 H_1 S_1$
(5) $S_1 \longrightarrow s_1 S_2$
(6) $x_1 S_2 \longrightarrow s_2 b_1$
// Message event handler
(7) $M_1 \longrightarrow T_1$
(8) $M_1 \longrightarrow F_1$
(9) $T_1 \longrightarrow y_1 D_3 R_2$
(10) $x_1 F_1 \longrightarrow x_1 M_1$
(11) $b_1 F_1 \longrightarrow b_1 m_1$
(12) $D_3 \longrightarrow D_4$
(13) $D_4 \longrightarrow G_3$
(14) $G_3 \longrightarrow S_3$
(15) $S_3 \longrightarrow s_3 b_2$
(16) $b_2 R_2 \longrightarrow D_5 D_6$
(17) $D_5 \longrightarrow \varepsilon$
(18) $x_1 y_1 D_6 \longrightarrow x_1 M_1$
(19) $b_1 y_1 D_6 \longrightarrow b_1 m_1$
// Alarm event handler
(20) $M_2 \longrightarrow T_2$
(21) $M_2 \longrightarrow F_2$
(22) $T_2 \longrightarrow y_2 S_4 R_3$
(23) $x_1 F_2 \longrightarrow x_1 M_2$
(24) $b_1 F_2 \longrightarrow b_1 m_2$
(25) $S_4 \longrightarrow s_4 b_3$
(26) $x_1 y_2 b_3 R_3 \longrightarrow x_1 M_2$
(27) $b_1 y_2 b_3 R_3 \longrightarrow b_1 m_2$
// Prepare the completion of scope R_1
(28) $b_1 m_1 m_2 H_1 \longrightarrow U_1 U_2 H_2$
(29) $U_1 \longrightarrow u_1$
(30) $U_2 \longrightarrow u_2$
(31) $u_1 u_2 H_2 \longrightarrow M_3 M_4 H_3$
// Process waiting events
(32) $M_3 \longrightarrow T_3$
(33) $M_3 \longrightarrow F_3$
(34) $T_3 \longrightarrow y_3 D_3 R_2$
(35) $F_3 \longrightarrow m_1$
(36) $y_3 D_6 \longrightarrow M_3$
(37) $M_4 \longrightarrow T_4$
(38) $A_4 \longrightarrow F_4$
(39) $T_4 \longrightarrow y_4 S_4 R_3$
(40) $F_4 \longrightarrow m_2$
(41) $y_4 b_3 R_3 \longrightarrow M_4$
// Complete scope R_1
(42) $m_1 m_2 R_1 \longrightarrow D_2$
(43) $D_2 \longrightarrow ...$

$with:$

$$
\begin{array}{rcl}
D_1, D_3, D_4 & \leftarrowtail & InsertData \\
T_i, F_i, H_i, R_i & \leftarrowtail & Helpers \\
E_1 & \leftarrowtail & ExpressionEvaluator \\
A_1 & \leftarrowtail & AlarmService\_For \\
M_1, M_3 & \leftarrowtail & Event_1 \\
M_2, M_4 & \leftarrowtail & Alarm_1 \\
S_i & \leftarrowtail & Services \\
G_1 & \leftarrowtail & GetEvent_1 \\
U_1 & \leftarrowtail & CancelEvent_1 \\
U_2 & \leftarrowtail & CancelAlarm_1 \\
D_2, D_5, D_6 & \leftarrowtail & DeleteData
\end{array}
$$

$$T_i, F_i, G_i, H_i, R_i, S_i, M_i, A_i, U_i, D_i \in V$$
$$s_i, b_i, x_i, y_i, m_i, u_i \in \Sigma$$

Figure 19: Unified model for event handlers corresponding to the original model in figure 18.

After the finishing of the regular logic of scope R_1 the handling of events is not allowed any longer. The non-terminals U_i represent a call of the utility service for informing the service about the point in time when following events need to be rejected. However, waiting events possibly exist as events are sequentially processed in the unified model. For the processing of waiting events additional non-terminals M_i need to be introduced. For instance, the non-terminal M_3 is introduced for the check for waiting message events additionally to the non-terminal M_1 for the check for message events while running the regular scope's logic.

Termination Handlers The termination of a scope requires to delete all activated tasks without further effects. Figure 20 presents the unified model for termination assuming a scope R_1 as defined figure 18 but without regard to the event handlers. For enabling the termination of a scope R_1 mutual exclusive context symbols, i.e. terminals t_1 and f_1 need to be introduced. Furthermore, the *lhs* of production rules realizing the regular logic of the scope need to be extended by a context symbol t_1. Additionally, production rules specifying a context symbol f_1 and realizing the deletion of non-terminals without further effects need to be introduced. For instance, rule (5) and rule (8) realize the deletion of the non-terminals S_1 and S_2 without service invocation (cf. definition 23).

$$
\begin{array}{rrll}
(1) & \mathit{Start} & \longrightarrow D_1 t_1 x_1 R_1 \\
(2) & D_1 & \longrightarrow S_1 \\
(3) & ...t_1 x_1... & \longrightarrow ...f_1... \\
(4) & t_1 S_1 & \longrightarrow t_1 s_1 S_2 \\
(5) & f_1 S_1 & \longrightarrow f_1 \\
(6) & k_1 S_1 & \longrightarrow k_1 \\
(7) & t_1 x_1 S_2 & \longrightarrow t_1 s_2 b_1 \\
(8) & f_1 S_2 & \longrightarrow f_1 \\
(9) & k_1 S_2 & \longrightarrow k_1 \\
(10) & t_1 b_1 R_1 & \longrightarrow r_1 D_2 \\
(11) & f_1 R_1 & \longrightarrow k_1 D_2 \\
(12) & D_2 & \longrightarrow \varepsilon
\end{array}
\qquad
\mathit{with}:
\qquad
\begin{array}{rcl}
D_1 & \leftarrowtail & \mathit{InsertData} \\
R_1 & \leftarrowtail & \mathit{Helpers} \\
S_i & \leftarrowtail & \mathit{Services} \\
D_2 & \leftarrowtail & \mathit{DeleteData} \\
 & & \\
D_i, R_1, S_i & \in & V \\
t_1, f_1, s_i, b_1, r_i, k_i & \in & \Sigma
\end{array}
$$

Figure 20: Unified model for termination in the scope R_1 of figure 18 without regard to the event handlers.

Termination handlers allows to control the termination behavior to some degree. In particular, a user-defined termination handler allows to specify extra activities that are executed after the termination of a scope. Figure 21 presents the unified model for a user-defined termination handler by use of a scope R_0 containing a child scope R_1

specifying an user-defined termination handler. The termination of a scope R_0 requires the termination of a child scope R_1. Therefore, the terminal f_0 indicating the need for termination of scope R_0 also indicates the need for termination of the enclosing scope R_1 (cf. rule (8), rule (12), and rule (16)). The activation of the user defined termination handler of R_1 is represented by the 1-dimensional non-terminal H_1. In figure 21 exclusively rule (16) activates the user-defined termination handler of R_1 as a termination handler is only enabled for scopes that are in a normal mode.

The default termination handler in BPEL executes the compensation activity after terminating all activated activities. That means, the default termination handler behaves like the default fault handler. Therefore, the production rules for the default fault handler that are introduced in the following are also valid for the default termination handler.

Fault Handlers In general, the occurrence of a fault requires termination. Afterwards, the logic given by a fault handler is executed. Fault handlers are enabled as long as the scope is not completed. The unified model for fault handlers is similar to the unified model for event handlers. In particular, fault handlers are also repeatedly activated as long as the regular logic of the scope is not completed. However, the execution of the fault handler requires the termination of the associated scope in contrast to the execution of an event handler.

Figure 22 presents the unified model for a user-defined fault handler. The meaning of used non-terminals and terminals is summarized in the following in addition or in substitution to the already presented meaning of non-terminals and terminals for event handlers:

N_i**:** Represents the check for the occurrence of a fault

T_i**:** Indicates that the check for a fault was true

F_i**:** Indicates that the check for a fault was false

t_i**:** Indicates the regular processing mode of scope R_i

f_i**:** Indicates the termination mode of scope R_i

h_i**:** Indicates the finished handling of a fault that was checked by use of N_i or N_{i+1}

n_i**:** Indicates the completion of the check for a fault with N_i

r_i**:** Indicates the successfully finished and completed scope R_i

q_i**:** Indicates the unsuccessfully finished and completed scope R_i because of a fault

```
<scope name="R_0" >
    <sequence>
    ...
    <scope name="R_1" >
        <terminationHandler>
            <invoke name=" S_5" />
        </terminationHandler>
        <sequence>
            <invoke name=" S_1" />
            <invoke name=" S_2" />
        </sequence>
    </scope>
    </sequence>
</scope>
```

(a) BPEL-based scope R_1 defining a user-defined termination handler.

$$
\begin{array}{lrl}
(1) & Start & \longrightarrow t_0 x_0 Y_0 R_0 \\
(2) & t_0 Y_0 & \longrightarrow t_0 ... \\
(...) & ...t_0 x_0 ... & \longrightarrow ...f_0 ... \\
(3) & ... & \longrightarrow X_1 \\
(4) & t_0 X_1 & \longrightarrow t_0 t_1 x_1 S_1 R_1 \\
(5) & f_0 X_1 & \longrightarrow f_0 k_1 \\
(6) & k_0 X_1 & \longrightarrow k_0 k_1 \\
(7) & t_0 t_1 S_1 & \longrightarrow t_0 t_1 s_1 S_2 \\
(8) & f_0 t_1 S_1 & \longrightarrow f_0 t_1 \\
(9) & f_1 S_1 & \longrightarrow f_1 \\
(10) & k_1 S_1 & \longrightarrow k_1 \\
(11) & t_0 t_1 S_2 & \longrightarrow t_0 t_1 s_2 b_1 \\
(12) & f_0 t_1 S_2 & \longrightarrow f_0 t_1 \\
(13) & f_1 S_2 & \longrightarrow f_1 \\
(14) & k_1 S_2 & \longrightarrow k_1 \\
(15) & t_0 x_0 t_1 b_1 R_1 & \longrightarrow t_0 r_1 y_1 b_0 \\
(16) & f_0 t_1 x_1 R_1 & \longrightarrow f_0 H_1 \\
(17) & f_1 R_1 & \longrightarrow k_1 y_1 b_0 \\
(18) & H_1 & \longrightarrow S_5 \\
(19) & S_5 & \longrightarrow s_5 k_1 y_1 \\
(20) & t_0 b_0 y_1 R_0 & \longrightarrow r_0 \\
(21) & f_0 y_1 R_0 & \longrightarrow k_0 \\
 & & \\
with: & R_1, H_1 & \longleftarrow Helpers \\
 & D_1 & \longleftarrow InsertData \\
 & S_i & \longleftarrow Services \\
 & D_2 & \longleftarrow DeleteData \\
 & D_i, R_1, S_i & \in V \\
 & t_1, f_1, s_i, b_1, r_i, k_i & \in \Sigma
\end{array}
$$

(b) Production rules for a scope R_1 defining a user-defined termination handler.

Figure 21: Unified model for user-defined termination handlers.

```
<scope name="R_1" >
    <faultHandlers>
        <catch faultName="F_2" variable="myFault">
            <invoke name="S_4" />
        </catch>
        <catchAll>
            <invoke name="S_5" />
        </catch>
    </faultHandlers>
    <sequence>
        <invoke name="S_1" />
        <invoke name="S_2" />
        <invoke name="S_3" />
    </sequence>
</scope>
```

(a) BPEL specification of a scope with user-defined fault handlers.

(1) $Start \longrightarrow t_1 x_1 N_2 N_4 S_1 H_1 R_1$
(2) $t_1 S_1 \longrightarrow t_1 s_1 S_2$
(3) $f_1 S_1 \longrightarrow f_1$
(4) $q_1 S_1 \longrightarrow q_1$
(5) $t_1 S_2 \longrightarrow t_1 s_2 S_3$
(6) $f_1 S_2 \longrightarrow f_1$
(7) $q_1 S_2 \longrightarrow q_1$
(8) $t_1 x_1 S_3 \longrightarrow t_1 s_3 b_1$
(9) $f_1 S_3 \longrightarrow f_1$
(10) $q_1 S_3 \longrightarrow q_1$
// Check for fault F_2
(11) $N_2 \longrightarrow T_2$
(12) $N_2 \longrightarrow F_2$
(13) $t_1 x_1 F_2 \longrightarrow t_1 x_1 N_2$
(14) $t_1 b_1 F_2 \longrightarrow t_1 b_1 n_2$
(15) $f_1 F_2 \longrightarrow f_1 n_2$
(16) $t_1 x_1 T_2 \longrightarrow f_1 D_1 n_2$
(17) $t_1 b_1 T_2 \longrightarrow t_1 b_1 n_2$
(18) $f_1 T_2 \longrightarrow f_1 n_2$
// Fault handler for F_2
(19) $D_1 \longrightarrow G_1$
(20) $G_1 \longrightarrow S_4$
(21) $S_4 \longrightarrow s_4 D_2$
(22) $D_2 \longrightarrow h_2$
// Check for other faults
(23) $N_4 \longrightarrow T_4$
(24) $N_4 \longrightarrow F_4$
(25) $t_1 x_1 F_4 \longrightarrow t_1 x_1 N_4$
(26) $t_1 b_1 F_4 \longrightarrow t_1 b_1 n_4$
(27) $f_1 F_4 \longrightarrow f_1 n_4$
(28) $t_1 x_1 T_4 \longrightarrow f_1 S_5 n_4$
(29) $t_1 b_1 T_4 \longrightarrow t_1 b_1 n_4$
(30) $f_1 T_4 \longrightarrow f_1 n_4$
// Fault handler for other faults
(31) $S_5 \longrightarrow s_5 h_4$
// Prepare completion of the scope
(32) $b_1 n_2 n_4 H_1 \longrightarrow N_3 N_5$
(33) $h_2 n_2 n_4 H_1 \longrightarrow h_2 n_3 n_5$
(34) $h_4 n_2 n_4 H_1 \longrightarrow h_4 n_3 n_5$
// Final fault handling
(35) $N_3 \longrightarrow T_3$
(36) $N_3 \longrightarrow F_3$
(37) $t_1 F_3 \longrightarrow t_1 n_3$
(38) $f_1 F_3 \longrightarrow f_1 n_3$
(39) $t_1 T_3 \longrightarrow f_1 D_1 n_3$
(40) $f_1 T_3 \longrightarrow f_1 n_3$
(41) $N_5 \longrightarrow T_5$
(42) $N_5 \longrightarrow F_5$
(43) $t_1 F_5 \longrightarrow t_1 n_5$
(44) $f_1 F_5 \longrightarrow f_1 n_5$
(45) $t_1 T_5 \longrightarrow f_1 S_3 n_5$
(46) $f_1 T_5 \longrightarrow f_1 n_5$
// Complete the scope
(47) $t_1 n_3 n_5 R_1 \longrightarrow r_1$
(48) $f_1 h_2 n_3 n_5 R_1 \longrightarrow q_1$
(49) $f_1 h_4 n_3 n_5 R_1 \longrightarrow q_1$

with : $H_1, R_1, T_i, F_i \longleftarrow Helpers$
$N_2, N_4 \longleftarrow Fault_2$
$N_3, N_5 \longleftarrow Fault_3$
$S_i \longleftarrow Services$
$D_1 \longleftarrow InsertData$
$D_2 \longleftarrow DeleteData$
$G_1 \longleftarrow GetFault_2$
$H_1, R_1, T_i, F_i, N_i, S_i, D_i, G_i \in V$
$b_i, t_i, f_i, x_i, n_i, r_i, q_i \in \Sigma$

(b) Production rules for user-defined fault handlers.

Figure 22: Unified model for user-defined fault handlers.

For each user-defined fault handler a 3-dimensional non-terminal N_j representing the check for the occurrence of a fault needs to be introduced similar to the check for events. The result of the check is only considered if the associated scope R_i is in the regular processing mode (indicated by t_i) and the scope didn't finish the regular logic (indicated by x_i).The check for faults are repeatedly activated as long as the regular logic of the corresponding scope is processed. For instance, rule (13) reactivates the non-terminal N_2 in case the the check for a fault was false and the regular logic of the scope is processed. In contrast, rule (16) starts the termination of the scope R_1 by producing the terminal f_1 and starts the user defined fault handling procedure by activating the non-terminal D_1. Furthermore, a terminal n_2 is produced indicating that the check for the fault is completed, i.e. is not required to be reactivated. The rules (19–22) realize the user-defined fault handling procedure including the creation of the variable `myFault` (cf. D_1), the request for the fault and storing the fault in the variable (cf. G_1), the service call S_4, and the variable deletion (cf. D_2). Finally, a terminal h_2 is produced indicating the finished processing of the fault.

As the checks for faults are successively processed a final check for each fault need to be processed after the finishing of the regular scope logic and before scope completion. The final check is also true for faults that are previously ignored because of the finishing of the scope's regular logic in order to ensure the handling of these faults. However, the final check is only activated in case no fault was previously handled (cf. rule (32)). In all other cases the final check is not activated but the immediate completion of the scope is enabled (cf. rule (33) and rule (34)). In case no fault occurred at all rule (47) completes the scope by producing a terminal r_1 indicating the successful completion of the scope R_1. In case a fault was handled the rule (48) or rule (49) completes the scope R_1 by producing a terminal q_1 indicating the unsuccessful completion of the scope R_1.

The user-defined fault handler is allowed to rethrow the original fault to the parent scope. The realization of rethrowing a fault is presented in the unified model for the default fault handler in the following. The default fault handler is executed if no user-defined fault handler is specified for an occurring fault. That means, the production rules realizing the user-defined fault handler need to be combined with the production rules realizing the default fault handler if the user-defined fault handler does not specify fault handling logic for each fault that can occur at runtime.

The BPEL default fault handler terminates the associated scope R_1 and compensates the child scopes of R_1 in the reverse order afterwards. After the compensation the fault is rethrowed to the parent scope of R_1. In the following the unified model for the default fault handler is presented covering the compensation of a child scope as well as the rethrowing of faults to the parent scope. However, the compensation of child scopes in the reverse order is delayed to the unified model for the default compensation handler.

Figure 23 shows a BPEL specification of scope R_1 containing the default fault handler and a single child scope R_2. Figure 24 presents the unified model for the default fault handler corresponding to the original model in figure 23. The meaning of used non-terminals and terminals is summarized in the following in addition to the already presented meaning of non-terminals and terminals for user-defined fault handlers and event handlers:

Z_i: Represents the check for the need for compensation of the scope R_i

W_i: Represents the rethrowing of a fault to the parent scope of scope R_i

z_i: Indicates the finished check for the need for compensation

w_i: Indicates the finished rethrowing of a fault

k_i: Indicates the terminated and completed scope R_i without fault handling

c_i: Indicates the compensated scope R_i

```
<scope name="R_1" >
    <sequence>
        <scope name="R_2">
            <compensationHandler>
                <invoke name=" $S_C$" />
            </compensationHandler>
            <sequence>
                <invoke name="S_1" />
                <invoke name="S_2" />
            </sequence>
        </scope>
        <invoke name="S_3" />
    </sequence>
</scope>
```

Figure 23: BPEL specification of a scope with the default fault handler.

Similar to the unified model for user-defined fault handlers the unified model for the default fault handler realizes the checks for faults, preparation for the completion of scopes as well as the completion of scopes. In contrast to the user-defined fault handler the unified model for the default fault handler activates a non-terminal Z_2 in rule (15) representing the check for the need for the compensation of the scope R_2. The compensation of the scope R_2 is exclusively allowed if the scope is successfully completed (cf. rule (67)). In all other cases the scope is not allowed to be compensated and the non-terminal Z_i is deleted without further effects (cf. rules (68–70)). The rule (71) realizes the execution of the user-defined compensation handler of R_2 by executing the

// Scope R_1

(1) $Start \longrightarrow t_1x_1N_3H_1X_2R_1$

(2) $t_1X_2 \longrightarrow t_1t_2x_2N_5H_2S_1R_2$

(3) $f_1X_2 \longrightarrow f_1k_2y_2$

(4) $q_1X_2 \longrightarrow q_1k_2y_2$

(5) $t_1x_1S_3 \longrightarrow t_1s_3b_1$

(6) $f_1S_3 \longrightarrow f_1$

(7) $q_1S_3 \longrightarrow q_1$

// Check for faults in R_1

(8) $N_3 \longrightarrow T_3$

(9) $N_3 \longrightarrow F_3$

(10) $t_1x_1F_3 \longrightarrow t_1x_1N_3$

(11) $t_1b_1F_3 \longrightarrow t_1b_1n_3$

(12) $f_1F_3 \longrightarrow f_1n_3$

(13) $t_1b_1T_3 \longrightarrow t_1b_1n_3$

(14) $f_1T_3 \longrightarrow f_1n_3$

// Run default fault hander in R_1:

(15) $t_1x_1T_3 \longrightarrow f_1Z_2H_{12}n_3$

(16) $z_2H_{12} \longrightarrow h_3$

//Prepare completion of scope R_1

(17) $b_1n_3y_2H_1 \longrightarrow N_4$

(18) $h_3n_3y_2H_1 \longrightarrow h_3n_4$

(19) $N_4 \longrightarrow T_4$

(20) $N_4 \longrightarrow F_4$

(21) $t_1F_4 \longrightarrow t_1n_4$

(22) $f_1F_4 \longrightarrow f_1n_4$

(23) $t_1T_4 \longrightarrow f_1Z_2H_{12}n_4$

(24) $f_1T_4 \longrightarrow f_1n_4$

// Complete R_1

(25) $t_1n_4R_2 \longrightarrow r_1$

(26) $f_1h_3n_4R_2 \longrightarrow q_1$

$$
\begin{array}{lrcl}
with: & H_i, X_2, R_i, T_i, F_i & \longleftarrow & Helpers \\
 & N_3, N_5 & \longleftarrow & Fault_{R1} \\
 & N_4, N_7 & \longleftarrow & Fault_{R2} \\
 & D_1 & \longleftarrow & InsertData \\
 & G_2 & \longleftarrow & GetFault_{R2} \\
 & W_2 & \longleftarrow & SetFault_{R1} \\
 & D_2 & \longleftarrow & DeleteData \\
 & S_i & \longleftarrow & Services \\
 & H_i, X_i, R_i, T_i, F_i, N_i, D_i, G_i, W_i, S_i & \in & V \\
 & b_i, d_i, t_i, f_i, x_i, y_i, n_i, r_i, q_i, k_i, c_i, z_i & \in & \Sigma
\end{array}
$$

// Scope R_2

(27) $t_1t_2S_1 \longrightarrow t_1t_2s_1S_2$

(28) $f_1t_2S_1 \longrightarrow f_1t_2$

(29) $f_2S_1 \longrightarrow f_2$

(30) $k_2S_1 \longrightarrow k_2$

(31) $q_2S_1 \longrightarrow q_2$

(32) $t_1t_2x_2S_2 \longrightarrow t_1t_2s_2b_2$

(33) $f_1t_2S_2 \longrightarrow f_1t_2$

(34) $f_2S_2 \longrightarrow f_2$

(35) $k_2S_2 \longrightarrow k_2$

(36) $q_2S_2 \longrightarrow q_2$

// Check for faults in R_2

(37) $N_5 \longrightarrow T_5$

(38) $N_5 \longrightarrow F_5$

(39) $t_1t_2x_2F_5 \longrightarrow t_1t_2x_2N_5$

(40) $t_1t_2b_2F_5 \longrightarrow t_1t_2b_2n_5$

(41) $f_1t_2F_5 \longrightarrow f_1t_2n_5$

(42) $f_2F_5 \longrightarrow f_2n_5$

(43) $t_1t_2x_2T_5 \longrightarrow t_1f_2D_1H_{21}n_5$

(44) $t_1t_2b_2T_5 \longrightarrow t_1t_2b_2n_5$

(45) $f_1t_2T_5 \longrightarrow f_1t_2n_5$

(46) $f_2T_5 \longrightarrow f_2n_5$

// Default fault hander in R_2:

(47) $D_1 \longrightarrow G_2$

(48) $G_2 \longrightarrow W_2$

(49) $W_2 \longrightarrow D_2$

(50) $D_2 \longrightarrow d_2$

(51) $d_2H_{21} \longrightarrow h_5$

// Prepare completion of scope R_2

(52) $b_2n_5H_2 \longrightarrow N_6$

(53) $h_5n_5H_2 \longrightarrow h_5n_6$

(54) $f_1t_2x_2n_5H_2 \longrightarrow f_1t_2n_6$

(55) $N_6 \longrightarrow T_6$

(56) $N_6 \longrightarrow F_6$

(57) $t_1t_2F_6 \longrightarrow t_1t_2n_6$

(58) $f_1t_2F_6 \longrightarrow f_1t_2n_6$

(59) $f_2F_6 \longrightarrow f_2n_6$

(60) $t_1t_2T_6 \longrightarrow t_1f_2W_2H_{21}n_6$

(61) $f_1t_2T_6 \longrightarrow f_1t_2n_6$

(62) $f_2T_6 \longrightarrow f_2n_6$

// Complete R_2

(63) $t_1t_2n_6R_2 \longrightarrow t_1r_2y_2S_3$

(64) $f_1t_2n_6R_2 \longrightarrow f_1k_2y_2$

(65) $t_1f_2h_5n_6R_2 \longrightarrow t_1q_2y_2S_3$

(66) $f_1f_2h_5n_6R_2 \longrightarrow f_1q_2y_2$

//Compensation handler of R_2

(67) $r_2Z_2 \longrightarrow S_C$

(68) $q_2Z_2 \longrightarrow q_2z_2$

(69) $k_2Z_2 \longrightarrow k_2z_2$

(70) $c_2Z_2 \longrightarrow c_2z_2$

(71) $S_C \longrightarrow s_cc_2z_2$

Figure 24: Unified model for the default fault handler corresponding to the original model in figure 23.

service call S_C. The eventually produced terminal z_2 indicates the finishing of the compensation and the ability to proceed the default fault handler.

In general, the default fault handler of a scope R_x proceeds with rethrowing the handled fault to the parent scope of R_x. The default fault handler of the scope R_1 in figure 24 does not rethrow the handled fault as the scope R_1 has no parent scope. However, scope R_2 also specifies the default fault handler requiring to rethrow a handled fault to the parent scope R_1. For the rethrowing the default fault handler of scope R_2 starts with processing the non-terminal D_1 representing the creation of a variable that is used to store the fault. Afterwards, the 2-dimensional non-terminal G_2 representing a call of the utility service for getting the particular fault data is processed. The rethrowing is realized by processing the non-terminal W_2 in the following whereas the non-terminal W_2 represents a call of the utility service for registering a fault for scope R_1. Finally, the non-terminal D_2 representing the deletion of the variable storing the fault is processed.

The terminals k_i and c_i represent the completion states of scopes that are introduced by the default fault handling. The terminal k_i indicates the terminated and completed scope R_i without fault handling in R_i. The terminal c_i indicates the compensated scope R_i and disallows the repeated compensation of the scope R_i in further processing. Regarding the production rules realizing the deletion of non-terminals without further effects after the (termination and) unsuccessful completion of a scope R_i rules specifying the context symbol k_i need to be introduced in addition to the rules specifying the context symbol q_i (cf. rules (30–31) and (35–36) in figure 24). However, rules specifying the context symbol c_i and realizing the deletion without further effects do not need to be introduced as the compensation requires the successful completion of the scope's regular logic previously to the compensation. That means, no non-terminals need to be deleted without further effects in this case.

Compensation Handlers The representation of a user-defined compensation handler in the unified model was already introduced in figure 24 (cf. rules (67–71)). Figure 25 shows an original BPEL model specifying a scope R_1 with the child scope R_2 containing the default compensation handler. The default compensation handler of scope R_2 needs to compensate the child scopes R_3, R_4, and R_5 in the reverse order. Figure 26 presents the unified model for the default compensation handler corresponding to the original model in figure 25.

As mentioned before, the default compensation handler of a scope needs to compensate the child scopes in the reverse order. In general, the reverse order of child scopes needs to be statically determined for the representation of the default compensation handler in the unified model. For instance, the scope R_2 executes the child scope R_3 at first. Afterwards, the child scopes R_4 and R_5 are executed simultaneously. Therefore, default compensation handler of R_2 needs to simultaneously compensate the child scopes R_4 and R_5 at first. Afterwards, the child scope R_3 needs to be compensated.

```
<scope name="R_1" >
    <scope name="R_2" >
        <sequence>
            <scope name="R_3">
                <compensationHandler>
                    <invoke name=" S_6" />
                </compensationHandler>
                <invoke name=" S_3" />
            </scope>
            <flow>
                <scope name="R_4">
                    <compensationHandler>
                        <invoke name=" S_7" />
                    </compensationHandler>
                    <invoke name=" S_4" />
                </scope>
                <scope name="R_5">
                    <compensationHandler>
                        <invoke name=" S_8" />
                    </compensationHandler>
                    <invoke name=" S_5" />
                </scope>
            </flow>
        </sequence>
    </scope>
</scope>
```

Figure 25: BPEL specification of a scope R_1 with the default fault handler invoking the default compensation handler of the child scope R_2.

As mentioned before, the non-terminals Z_i represent the checks for the need of the compensation of the scopes R_i in the unified model. In figure 28 the default compensation handlers of R_2 starts with activating the non-terminals Z_4 and Z_5. Similar to the parallel execution of the scopes R_4 and R_5 the parallel compensation of the scopes requires a following synchronization that is realized by use of the helper non-terminal H_6. After the synchronization the non-terminal Z_3 is activated for enabling the compensation of the scope R_3.

3.3 Declarative Languages

This section introduces grammatical production rules for constraints provided in ConDec [4]. The discussed constraints exemplarily illustrate grammatical representations for dependencies between activities specified in declarative workflow models. In declarative languages for service compositions a model specifies requirements on the order of tasks but a human or an external software component needs to select an enabled task for further processing. In the unified model the selection of an enabled task for further

// Scope R_1

(1) See figure 24

(2) $t_1X_2 \longrightarrow t_1t_2x_2N_5H_2X_3R_2$

(3 – 26) See figure 24

// Scope R_2

(27) $t_1t_2X_3 \longrightarrow t_1t_2t_3x_3N_{31}H_3S_3R_3$

(28) $f_1t_2X_3 \longrightarrow f_1t_2k_3y_3y_4y_5$

(29) $f_2X_3 \longrightarrow f_2k_3y_3y_4y_5$

(30) $k_2X_3 \longrightarrow k_2k_3$

(31) $q_2X_3 \longrightarrow q_2k_3$

(32) $t_1t_2X_4 \longrightarrow t_1t_2t_4x_4N_{41}H_4S_4R_4$

(33) $f_1t_2X_4 \longrightarrow f_1t_2k_4y_4$

(34) $f_2X_4 \longrightarrow f_2k_4y_4$

(35) $k_2X_4 \longrightarrow k_2k_4$

(36) $q_2X_4 \longrightarrow q_2k_4$

(37) $t_1t_2X_5 \longrightarrow t_1t_2t_5x_5N_{51}H_5S_5R_5$

(38) $f_1t_2X_5 \longrightarrow f_1t_2k_5y_5$

(39) $f_2X_5 \longrightarrow f_2k_5y_5$

(40) $k_2X_5 \longrightarrow k_2k_5$

(41) $q_2X_5 \longrightarrow q_2k_5$

(42) $t_1t_2x_2y_4y_5H_{45} \longrightarrow t_1t_2b_2y_4y_5$

(43) $f_1t_2H_{45} \longrightarrow f_1t_2$

(44) $f_2H_{45} \longrightarrow f_2$

(45) $k_2H_{45} \longrightarrow k_2$

(46) $q_2H_{45} \longrightarrow q_2$

// Check for faults in R_2

(47) $N_5 \longrightarrow T_5$

(48) $N_5 \longrightarrow F_5$

(49) $t_1t_2x_2F_5 \longrightarrow t_1t_2x_2N_5$

(50) $t_1t_2b_2F_5 \longrightarrow t_1t_2b_2n_5$

(51) $f_1t_2F_5 \longrightarrow f_1t_2n_5$

(52) $f_2F_5 \longrightarrow f_2n_5$

(53) $t_1t_2x_2T_5 \longrightarrow t_1f_2\mathbf{Z_2}D_1H_{21}n_5$

(54) $t_1t_2b_2T_5 \longrightarrow t_1t_2b_2n_5$

(55) $f_1t_2T_5 \longrightarrow f_1t_2n_5$

(56) $f_2T_5 \longrightarrow f_2n_5$

// Default fault hander in R_2:

(57) $\mathbf{z_2}D_1 \longrightarrow G_2$

(58) $G_2 \longrightarrow W_2$

(59) $W_2 \longrightarrow D_2$

(60) $D_2 \longrightarrow d_2$

(61) $d_2H_{21} \longrightarrow h_5$

// Prepare completion of scope R_2

(62) $b_2n_5y_3y_4y_5H_2 \longrightarrow N_6$

(63) $h_5n_5y_3y_4y_5H_2 \longrightarrow h_5n_6$

(64) $f_1t_2x_2n_5y_3y_4y_5H_2 \longrightarrow f_1t_2n_6$

(65) $N_6 \longrightarrow T_6$

(66) $N_6 \longrightarrow F_6$

(67) $t_1t_2F_6 \longrightarrow t_1t_2n_6$

(68) $f_1t_2F_6 \longrightarrow f_1t_2n_6$

(69) $f_2F_6 \longrightarrow f_2n_6$

(70) $t_1t_2T_6 \longrightarrow t_1f_2W_2H_{21}n_6$

(71) $f_1t_2T_6 \longrightarrow f_1t_2n_6$

(72) $f_2T_6 \longrightarrow f_2n_6$

// Complete R_2

(73) $t_1t_2n_6R_2 \longrightarrow t_1r_2\mathbf{b_1}$

(74) $f_1t_2n_6R_2 \longrightarrow f_1k_2$

(75) $t_1f_2h_5n_6R_2 \longrightarrow t_1q_2\mathbf{b_1}$

(76) $f_1f_2h_5n_6R_2 \longrightarrow f_1q_2$

// Default Compensation Handler R_2

(77) $r_2Z_2 \longrightarrow \mathbf{Z_4Z_5H_6}$

(78) $q_2Z_2 \longrightarrow q_2z_2$

(79) $k_2Z_2 \longrightarrow k_2z_2$

(80) $c_2Z_2 \longrightarrow c_2z_2$

(81) $\mathbf{z_4z_5H_6} \longrightarrow \mathbf{Z_3H_3}$

(82) $\mathbf{z_3H_3} \longrightarrow \mathbf{c_2z_2}$

with :

$H_i, X_i, R_i, T_i, F_i, W_i \longmapsfrom \textit{Helpers}$

$N_3, N_5 \longmapsfrom \textit{Fault}_{R1}$

$N_4, N_7 \longmapsfrom \textit{Fault}_{R2}$

$N_{31}, N_{32} \longmapsfrom \textit{Fault}_{R3}$

$N_{41}, N_{42} \longmapsfrom \textit{Fault}_{R4}$

$N_{51}, N_{52} \longmapsfrom \textit{Fault}_{R5}$

$D_1, D_3, D_5, D_7 \longmapsfrom \textit{InsertData}$

$G_i \longmapsfrom \textit{GetFault}_{Ri}$

$W_2 \longmapsfrom \textit{SetFault}_{R1}$

$W_3, W_4, W_5 \longmapsfrom \textit{SetFault}_{R2}$

$D_2, D_4, D_6, D_8 \longmapsfrom \textit{DeleteData}$

$S_i \longmapsfrom \textit{Services}$

$H_i, X_i, R_i, T_i, F_i, N_i, D_i, G_i, W_i, S_i \in V$

$b_i, d_i, t_i, f_i, x_i, y_i, n_i, r_i, q_i, k_i, c_i, z_i \in \Sigma$

Figure 26: Unified model for the default compensation handler corresponding to the original model in figure 25. Differences to figure 24 regarding the compensation handler are highlighted. Continuation of the model is presented in figure 27 and figure 28.

// Scope R_3

(83) $t_1t_2t_3x_3S_3 \longrightarrow t_1t_2t_3s_3b_3$

(84) $f_1t_2t_3S_3 \longrightarrow f_1t_2t_3$

(85) $f_2t_3S_3 \longrightarrow f_2t_3$

(86) $f_3S_3 \longrightarrow f_3$

(87) $k_3S_3 \longrightarrow k_3$

(88) $q_3S_3 \longrightarrow q_3$

// Check for faults in R_3

(89) $N_{31} \longrightarrow T_{31}$

(90) $N_{31} \longrightarrow F_{31}$

(91) $t_1t_2t_3x_3F_{31} \longrightarrow t_1t_2t_3x_3N_{31}$

(92) $t_1t_2t_3b_3F_{31} \longrightarrow t_1t_2t_3b_3n_{31}$

(93) $f_1t_2t_3F_{31} \longrightarrow f_1t_2t_3n_{31}$

(94) $f_2t_3F_{31} \longrightarrow f_2t_3n_{31}$

(95) $f_3F_{31} \longrightarrow f_3n_{31}$

(96) $t_1t_2t_3x_3T_{31} \longrightarrow t_1t_2f_3D_3H_{31}n_{31}$

(97) $t_1t_2t_3b_3T_{31} \longrightarrow t_1t_2t_3b_3n_{31}$

(98) $f_1t_2t_3T_{31} \longrightarrow f_1t_2t_3n_{31}$

(99) $f_2t_3T_{31} \longrightarrow f_2t_3n_{31}$

(100) $f_3T_{31} \longrightarrow f_3n_{31}$

// Default fault hander in R_3:

(101) $D_3 \longrightarrow G_3$

(102) $G_3 \longrightarrow W_3$

(103) $W_3 \longrightarrow D_4$

(104) $D_4 \longrightarrow d_4$

(105) $d_4H_{31} \longrightarrow h_{31}$

//Prepare completion of scope R_3

(106) $b_3n_{31}H_3 \longrightarrow N_{32}$

(107) $h_{31}n_{31}H_3 \longrightarrow h_{31}n_{32}$

(108) $f_1t_2t_3x_3n_{31}H_3 \longrightarrow f_1t_2t_3n_{32}$

(109) $f_2t_3x_3n_{31}H_3 \longrightarrow f_2t_3n_{32}$

(110) $N_{32} \longrightarrow T_{32}$

(111) $N_{32} \longrightarrow F_{32}$

(112) $t_1t_2t_3F_{32} \longrightarrow t_1t_2t_3n_{32}$

(113) $f_1t_2t_3F_{32} \longrightarrow f_1t_2t_3n_{32}$

(114) $f_2t_3F_{32} \longrightarrow f_2t_3n_{32}$

(115) $f_3F_{32} \longrightarrow f_3n_{32}$

(116) $t_1t_2t_3T_{32} \longrightarrow t_1t_2f_3W_3H_{31}n_{32}$

(117) $f_1t_2t_3T_{32} \longrightarrow f_1t_2t_3n_{32}$

(118) $f_2t_3T_{32} \longrightarrow f_2t_3n_{32}$

(119) $f_3T_{32} \longrightarrow f_3n_{32}$

// Complete R_3

(120) $t_1t_2t_3n_{32}R_3 \longrightarrow t_1t_2r_3y_3X_4X_5H_{45}$

(121) $f_1t_2t_3n_{32}R_3 \longrightarrow f_1t_2k_3y_3y_4y_5$

(122) $f_2t_3n_{32}R_3 \longrightarrow f_2k_3y_3y_4y_5$

(123) $t_2f_3h_{31}n_{32}R_3 \longrightarrow t_2q_3y_3X_4X_5H_{45}$

(124) $f_2f_3h_{31}n_{32}R_3 \longrightarrow f_2q_3y_3y_4y_5$

//Compensation handler of R_3

(125) $r_3Z_3 \longrightarrow S_6$

(126) $q_3Z_3 \longrightarrow q_3z_3$

(127) $k_3Z_3 \longrightarrow k_3z_3$

(128) $c_3Z_3 \longrightarrow c_3z_3$

(129) $S_6 \longrightarrow s_6c_3z_3$

// Scope R_4

(130) $t_1t_2t_4x_4S_4 \longrightarrow t_1t_2t_4s_6b_4$

(131) $f_1t_2t_4S_4 \longrightarrow f_1t_2t_4$

(132) $f_2t_4S_4 \longrightarrow f_2t_4$

(133) $f_4S_4 \longrightarrow f_4$

(134) $k_4S_4 \longrightarrow k_4$

(135) $q_4S_4 \longrightarrow q_4$

// Check for faults in R_4

(136) $N_{41} \longrightarrow T_{41}$

(137) $N_{41} \longrightarrow F_{41}$

(138) $t_1t_2t_4x_4F_{41} \longrightarrow t_1t_2t_4x_4N_{41}$

(139) $t_1t_2t_4b_4F_{41} \longrightarrow t_1t_2t_4b_4n_{41}$

(140) $f_1t_2t_4F_{41} \longrightarrow f_1t_2t_4n_{41}$

(141) $f_2t_4F_{41} \longrightarrow f_2t_4n_{41}$

(142) $f_4F_{41} \longrightarrow f_4n_{41}$

(143) $t_1t_2t_4x_4T_{41} \longrightarrow t_1t_2f_4D_5H_{41}n_{41}$

(144) $t_1t_2t_4b_4T_{41} \longrightarrow t_1t_2t_4b_4n_{41}$

(145) $f_1t_2t_4T_{41} \longrightarrow f_1t_2t_4n_{41}$

(146) $f_2t_4T_{41} \longrightarrow f_2t_4n_{41}$

(147) $f_4T_{41} \longrightarrow f_4n_{41}$

// Default fault hander in R_4:

(148) $D_5 \longrightarrow G_4$

(149) $G_4 \longrightarrow W_4$

(150) $W_4 \longrightarrow D_6$

(151) $D_6 \longrightarrow d_6$

(152) $d_6H_{41} \longrightarrow h_{41}$

//Prepare completion of scope R_4

(153) $b_4n_{41}H_4 \longrightarrow N_{42}$

(154) $h_{41}n_{41}H_4 \longrightarrow h_{41}n_{42}$

(155) $f_1t_2t_4x_4n_{41}H_4 \longrightarrow f_1t_2t_4n_{42}$

(156) $f_2t_4x_4n_{41}H_4 \longrightarrow f_2t_4n_{42}$

(157) $N_{42} \longrightarrow T_{42}$

(158) $N_{42} \longrightarrow F_{42}$

(159) $t_1t_2t_4F_{42} \longrightarrow t_1t_2t_4n_{42}$

(160) $f_1t_2t_4F_{42} \longrightarrow f_1t_2t_4n_{42}$

(161) $f_2t_4F_{42} \longrightarrow f_2t_4n_{42}$

(162) $f_4F_{42} \longrightarrow f_4n_{42}$

(163) $t_1t_2t_4T_{42} \longrightarrow t_1t_2f_4W_4H_{41}n_{42}$

(164) $f_1t_2t_4T_{42} \longrightarrow f_1t_2t_4n_{42}$

(165) $f_2t_4T_{42} \longrightarrow f_2t_4n_{42}$

(166) $f_4T_{42} \longrightarrow f_4n_{42}$

// Complete R_4

(167) $t_1t_2t_4n_{42}R_4 \longrightarrow t_1t_2r_4y_4$

(168) $f_1t_2t_4n_{42}R_4 \longrightarrow f_1t_2k_4y_4$

(169) $f_2t_4n_{42}R_4 \longrightarrow f_2k_4y_4$

(170) $t_2f_4h_{41}n_{42}R_4 \longrightarrow t_2q_4y_4$

(171) $f_2f_4h_{41}n_{42}R_4 \longrightarrow f_2q_4y_4$

//Compensation handler of R_4

(172) $r_4Z_4 \longrightarrow S_7$

(173) $q_4Z_4 \longrightarrow q_4z_4$

(174) $k_4Z_4 \longrightarrow k_4z_4$

(175) $c_4Z_4 \longrightarrow c_4z_4$

(176) $S_7 \longrightarrow s_7c_4z_4$

Figure 27: Continuation of the unified model in figure 26.

// Scope R_5

$$\begin{array}{lrl}
(177) & t_1t_2t_5x_5S_5 & \longrightarrow t_1t_2t_5s_6b_5 \\
(178) & f_1t_2t_5S_5 & \longrightarrow f_1t_2t_5 \\
(179) & f_2t_5S_5 & \longrightarrow f_2t_5 \\
(180) & f_5S_5 & \longrightarrow f_5 \\
(181) & k_5S_5 & \longrightarrow k_5 \\
(182) & q_5S_5 & \longrightarrow q_5
\end{array}$$

// Check for faults in R_5

$$\begin{array}{lrl}
(183) & N_{51} & \longrightarrow T_{51} \\
(184) & N_{51} & \longrightarrow F_{51} \\
(185) & t_1t_2t_5x_5F_{51} & \longrightarrow t_1t_2t_5x_5N_{51} \\
(186) & t_1t_2t_5b_5F_{51} & \longrightarrow t_1t_2t_5b_5n_{51} \\
(187) & f_1t_2t_5F_{51} & \longrightarrow f_1t_2t_5n_{51} \\
(188) & f_2t_5F_{51} & \longrightarrow f_2t_5n_{51} \\
(189) & f_5F_{51} & \longrightarrow f_5n_{51} \\
(190) & t_1t_2t_5x_5T_{51} & \longrightarrow t_1t_2f_5D_7H_{51}n_{51} \\
(191) & t_1t_2t_5b_5T_{51} & \longrightarrow t_1t_2t_5b_5n_{51} \\
(192) & f_1t_2t_5T_{51} & \longrightarrow f_1t_2t_5n_{51} \\
(193) & f_2t_5T_{51} & \longrightarrow f_2t_5n_{51} \\
(194) & f_5T_{51} & \longrightarrow f_5n_{51}
\end{array}$$

// Default fault hander in R_5:

$$\begin{array}{lrl}
(195) & D_7 & \longrightarrow G_5 \\
(196) & G_5 & \longrightarrow W_5 \\
(197) & W_5 & \longrightarrow D_8 \\
(198) & D_8 & \longrightarrow d_8 \\
(199) & d_8H_{51} & \longrightarrow h_{51}
\end{array}$$

//Prepare completion of scope R_5

$$\begin{array}{lrl}
(200) & b_5n_{51}H_5 & \longrightarrow N_{52} \\
(201) & h_{51}n_{51}H_5 & \longrightarrow h_{51}n_{52} \\
(202) & f_1t_2t_5x_5n_{51}H_5 & \longrightarrow f_1t_2t_5n_{52} \\
(203) & f_2t_5x_5n_{51}H_5 & \longrightarrow f_2t_5n_{52} \\
(204) & N_{52} & \longrightarrow T_{52} \\
(205) & N_{52} & \longrightarrow F_{52} \\
(206) & t_1t_2t_5F_{52} & \longrightarrow t_1t_2t_5n_{52} \\
(207) & f_1t_2t_5F_{52} & \longrightarrow f_1t_2t_5n_{52} \\
(208) & f_2t_5F_{52} & \longrightarrow f_2t_5n_{52} \\
(209) & f_5F_{52} & \longrightarrow f_5n_{52} \\
(210) & t_1t_2t_5T_{52} & \longrightarrow t_1t_2f_5W_5H_{51}n_{52} \\
(211) & f_1t_2t_5T_{52} & \longrightarrow f_1t_2t_5n_{52} \\
(212) & f_2t_5T_{52} & \longrightarrow f_2t_5n_{52} \\
(213) & f_5T_{52} & \longrightarrow f_5n_{52}
\end{array}$$

// Complete R_5

$$\begin{array}{lrl}
(214) & t_1t_2t_5n_{52}R_5 & \longrightarrow t_1t_2r_5y_5 \\
(215) & f_1t_2t_5n_{52}R_5 & \longrightarrow f_1t_2k_5y_5 \\
(216) & f_2t_5n_{52}R_5 & \longrightarrow f_2k_5y_5 \\
(217) & t_2f_5h_{51}n_{52}R_5 & \longrightarrow t_2q_5y_5 \\
(218) & f_2f_5h_{51}n_{52}R_5 & \longrightarrow f_2q_5y_5
\end{array}$$

//Compensation handler of R_5

$$\begin{array}{lrl}
(219) & r_5Z_5 & \longrightarrow S_8 \\
(220) & q_5Z_5 & \longrightarrow q_5z_5 \\
(221) & k_5Z_5 & \longrightarrow k_5z_5 \\
(222) & c_5Z_5 & \longrightarrow c_5z_5 \\
(223) & S_8 & \longrightarrow s_8c_5z_5
\end{array}$$

Figure 28: Continuation of the unified model in figure 26 and figure 27.

processing is realized by use of 3-dimensional non-terminals. Similar to conditional control flow introduced in section 3.2.2 the selected task leads to a production rule that is used for further processing.

In general the rules for declarative service compositions intensively use the start symbol, i.e. a start symbol S is provided in different versions S_1, S_2, S_3 etc.. Furthermore, multiple grammatical production rules with a start symbol version on the *lhs* exist. Production rules with the same *lhs* specify different alternative tasks on the *rhs* that are enabled at the same point in runtime. Similar to conditional control flow introduced in the section 3.2 a component in the environment needs to select an enabled task for further processing leading to the selected production rule that is used for further processing. The selection of the symbol ε that is possibly specified on the *rhs* of a production rule represents the finishing of the execution of the service composition.

The unified model presented in this section assumes tasks that are exclusively implemented by service calls. That means, this section assumes original models that exclusively compose service calls. Data is handled by reference, i.e. in the same way as in the unified model for imperative service compositions.

The grammatical production rules introduced in this section assume a complete set of tasks in the considered service composition (e.g. $\{A, B, C, D\}$). Furthermore, rules that basically specify a constraint concerning two tasks A and B also needs to cover side effects to other tasks in the service composition. For example, the constraint *response(A,B)* specifies that B must be executed in future when A is executed at least once. However, in between the execution of other tasks (e.g. C, D) is allowed. Therefore, the rules specifying the constraint *response(A,B)* needs to ensure the particular relation between A and B but also needs to cover the relation to other tasks, e.g. C, D.

This section presents grammar-based representations for single constraints. For the transformation of a complete ConDec model containing multiple constraints the constraints are assumed to be transformed independently at first. Afterwards, the resulting grammars are combined by use of a combination algorithm that is presented at the end of this section.

In the following let

- S_1 be the start symbol of the presented grammars,
- $\mathit{Tasks} = \{A, B, C\}$ be the complete set of tasks.

3.3.1 Existence Templates

In the following production rules for the *existence templates* provided by ConDec are introduced. The existence templates cover unary constraints. That means, introduced rules basically restrict the execution of the single task A. The execution of the tasks

B and C is not restricted by the constraints but covered in the presented production rules.

Figure 29 presents the unified model for the constraint *existence(A)*. This constraint is parameterized with a natural number indicating the minimum number of occurrences of the task A. For example, figure 29 shows the rules for the number 2.[2] In the beginning all possible tasks are allowed to be activated and executed. Therefore, the rules (1–3) specify the start symbol S_1 on the *lhs* but provide alternative *rhs* for each possible task A, B, and C. However, if task A is executed the occurrence of this task needs to be counted in order to evaluate the constraint. The approach at hand implements the required counting by switching to another version of the start symbol, i.e. to switch to S_2 (cf. rule (4)). The execution of tasks that can occur in between of the executions of task A do not switch the version of the start symbol as they are not effected by this constraint (cf. rule (5),(6),(11),(12)). For finishing the service composition rule (16) specifies the ε on the *rhs*.

$$
\begin{array}{clll}
(1-3) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 & \\
(4) & A_1 & \longrightarrow aS_2 & \\
(5) & B_1 & \longrightarrow bS_1 & \\
(6) & C_1 & \longrightarrow cS_1 & \\
(7-9) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 & \\
(10) & A_2 & \longrightarrow aS_3 & \\
(11) & B_2 & \longrightarrow bS_2 & \\
(12) & C_2 & \longrightarrow cS_2 & \\
(13-16) & S_3 & \longrightarrow A_3 \mid B_3 \mid C_3 \mid \varepsilon & \\
(17) & A_3 & \longrightarrow aS_3 & \\
(18) & B_3 & \longrightarrow bS_3 & \\
(19) & C_3 & \longrightarrow cS_3 &
\end{array}
\qquad
\begin{array}{rrl}
\mathit{with:} & A_i, B_i, C_i & \in \mathit{Services} \\
 & S_i & \in \mathit{Helpers} \\
 & a, b, c & \in \Sigma
\end{array}
$$

Figure 29: Unified model for the constraint *existence*$_2$*(A)*, i.e. activity A occurs at least 2 times.

Similar to the existence of a task execution the absence of task executions can be specified by constraints. Figure 30 presents the unified model for the constraint *absence(A)*, i.e. the rules are exemplarily shown for the parameter number 2.[3] The constraint is parameterized with the maximum number of occurrences of the particular task A analogous to the existence constraint. However, in contrast to the existence constraint the execution of the task A is limited to a maximum number in the absence constraint.

[2] Rules for the generic constraint existence$_N$(A) are analogous to the rules in figure 29 specifying the concrete constraint existence$_2$(A).

[3] Rules for the generic constraint absence$_N$(A) are analogous to the rules in figure 30 specifying the concrete constraint absence$_2$(A).

In particular, the rules in figure 30 allows to execute the task A twice in maximum. The start symbol version S_3 indicates that task A was executed twice and cannot be activated again.

$$
\begin{array}{clll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & \quad \qquad\qquad S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_1 & \quad \qquad\qquad a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & \\
(8-11) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 \mid \varepsilon & \\
(12) & A_2 & \longrightarrow aS_3 & \\
(13) & B_2 & \longrightarrow bS_2 & \\
(14) & C_2 & \longrightarrow cS_2 & \\
(15-17) & S_3 & \longrightarrow B_3 \mid C_3 \mid \varepsilon & \\
(18) & B_3 & \longrightarrow bS_3 & \\
(19) & C_3 & \longrightarrow cS_3 &
\end{array}
$$

Figure 30: Unified model for the constraint *absence*$_2$*(A)*, i.e. activity A occurs at most 2 times.

The constraint *exactly*$_N$*(A)* is used to specify that a task A should be executed exactly N times. Similar to previously discussed constraint this constraint is parameterized with the natural number N indicating the exact number of occurrences of the task A. Figure 31 presents the unified model for the constraint exactly$_2$(A).[4] Characteristically, the rules do not allow the finishing of the composition until task A is executed twice. In particular, the symbol ε is only allowed to be selected if S_3 is activated (cf. rule(16)).

The constraint *init(A)* is used to specify that activity A must be the first executed in the service composition. Figure 32 presents the unified model for the init constraint. Rule (1) exclusively allows the task A at the beginning. If task A is executed once rule (3)[5] switches to the start symbol version S_1. Afterwards, the rules (4–7) allow to activate all tasks in *Tasks*. The execution of these tasks have no further effect and require no further switch of the start symbol version (cf. rule (8–10)).

A service composition can specify multiple init constraints for different tasks. Therefore, rule (1) is required to specify a helper non-terminal H_0 for the synchronization after the execution of all initial tasks. For example, assuming two tasks A and B to be initial tasks in one single service composition the following rules need to be provided:

[4]Rules for the generic constraint exactly$_N$(A) are analogous to the rules in figure 31 specifying the concrete constraint exactly$_2$(A).

[5]The order of symbols is presented w.l.o.g. However, each order of specified symbols is covered by the presented production rule.

$$\begin{array}{rlll} (1-3) & S_1 & \longrightarrow A_1 \;\mid\; B_1 \;\mid\; C_1 & \quad with: \quad A_i, B_i, C_i \;\in Services \\ (4) & A_1 & \longrightarrow aS_2 & \qquad\qquad\qquad S_i \;\in Helpers \\ (5) & B_1 & \longrightarrow bS_1 & \qquad\qquad\qquad a, b, c \;\in \Sigma \\ (6) & C_1 & \longrightarrow cS_1 & \\ (7-10) & S_2 & \longrightarrow A_2 \;\mid\; B_2 \;\mid\; C_2 & \\ (11) & A_2 & \longrightarrow aS_3 & \\ (12) & B_2 & \longrightarrow bS_2 & \\ (13) & C_2 & \longrightarrow cS_2 & \\ (14-16) & S_3 & \longrightarrow B_3 \;\mid\; C_3 \;\mid\; \varepsilon & \\ (17) & B_3 & \longrightarrow bS_3 & \\ (18) & C_3 & \longrightarrow cS_3 & \end{array}$$

Figure 31: Unified model for the constraint *exactly*$_2$*(A)*, i.e. activity A occurs exactly 2 times.

$$\begin{array}{rlll} (1) & S_0 & \longrightarrow A_0H_0 & \quad with: \quad A_i, B_i, C_i \;\in Services \\ (2) & A_0 & \longrightarrow a & \qquad\qquad\qquad S_i, H_i \;\in Helpers \\ (3) & aH_0 & \longrightarrow aS_1 & \qquad\qquad\qquad a, b, c \;\in \Sigma \\ (4-7) & S_1 & \longrightarrow A_1 \;\mid\; B_1 \;\mid\; C_1 \;\mid\; \varepsilon & \\ (8) & A_1 & \longrightarrow aS_1 & \\ (9) & B_1 & \longrightarrow bS_1 & \\ (10) & C_1 & \longrightarrow cS_1 & \end{array}$$

Figure 32: Unified model for the constraint *init(A)*, i.e. activity A must be the first executed activity in the service composition.

$$
\begin{array}{rrll}
(1) & S_0 & \longrightarrow A_0B_0H_0 & with: \quad A_i, B_i \in Services \\
(2) & A_0 & \longrightarrow a & \qquad\quad S_i, H_i \in Helpers \\
(2') & B_0 & \longrightarrow b & \qquad\quad a, b, c \in \Sigma \\
(3) & abH_0 & \longrightarrow abS_1 &
\end{array}
$$

Note that the init constraint has special semantics that need to be handled different to other constraints in ConDec. In detail, the combination of init constraints requires an union operator whereas the combination of other constraints in ConDec require an intersection operator (cf. section 3.3.6). In order to simplify the algorithm in section 3.3.6 production rules covering the init constraint are required to provide synchronization after the execution of the initial tasks. Note that this is also required for production rules covering exactly one initial task.

In order to allow a special handling of production rules covering the init constraints in the algorithm of section 3.3.6 such production rules are related to the start symbol version S_0, i.e. the symbol S_0 creates the start symbol of the grammar. Instead, production rules for other constraints are related to the start symbol version S_1 or higher, i.e. the symbol S_1 creates the start symbol of the grammar.

3.3.2 Relation Templates

In contrast to existence templates covering unary constraints the relation templates provide binary constraints. This section discusses these constraints specifying dependencies between two tasks are discussed. In general, the execution of other tasks is allowed in between.

The constraint *responded existence(A,B)* is used to specify that task B has to be executed if task A is executed. However, task B can be executed before or after task A. Figure 33 presents the unified model for the constraint responded existence. If task B is not executed the first execution of task A requires to switch to the start symbol version S_2 in rule (5). In the following, all tasks in *Tasks* are allowed to be activated but the finishing of the service composition is not allowed as the constraint requires to execute task B (cf. rule (8–10)). However, if task B is executed in further processing rule (12) is applied causing a switch to the start symbol version S_4. The version S_4 indicates the fulfillment of the constraint allowing all tasks as well as the finishing without restrictions in further processing (cf. rule (21–24)).

In contrast to the constraint responded existence(A,B) specifying a directed dependency between A and B the constraint *co-existence(A,B)* specify the same dependency in both directions, i.e. A needs to be executed if B is executed and vice versa. Figure 34 presents the unified model for the co-existence constraint. The rules are equal to the rules in figure 33 for the responded existence constraint excepts rule (17) in figure 33. The rule

$$
\begin{array}{cllll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & \quad with: & A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_3 & & a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & \\
(8-10) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 & & \\
(11) & A_2 & \longrightarrow aS_2 & & \\
(12) & B_2 & \longrightarrow bS_4 & & \\
(13) & C_2 & \longrightarrow cS_2 & & \\
(14-17) & S_3 & \longrightarrow A_3 \mid B_3 \mid C_3 \mid \varepsilon & & \\
(18) & A_3 & \longrightarrow aS_4 & & \\
(19) & B_3 & \longrightarrow bS_3 & & \\
(20) & C_3 & \longrightarrow cS_3 & & \\
(21-24) & S_4 & \longrightarrow A_4 \mid B_4 \mid C_4 \mid \varepsilon & & \\
(25) & A_4 & \longrightarrow aS_4 & & \\
(26) & B_4 & \longrightarrow bS_4 & & \\
(27) & C_4 & \longrightarrow cS_4 & &
\end{array}
$$

Figure 33: Unified model for the constraint *responded existence(A,B)*, i.e. if A is executed B also has to be executed at any time (either before or after A).

(17) allows the finishing if task B is executed but task A is not executed. However, the co-existence constraint does not allow the finishing in this case.

The constraint *response(A,B)* is used to specify that task B must be executed in future when task A is executed at least once. Figure 35(a) presents the unified model for the constraint *response*. In the beginning, the activation of all tasks as well as the finishing of the execution of the service composition is allowed (cf. rule (1–4)). However, if task A is executed the rule (5) causes the switch to S_2 disabling the finishing of the service composition. If task B is finally executed rule (12) switches back to S_1 allowing to finish the service execution as the constraint is fulfilled.

For the response constraint and other constraints supplements exists that strengthen the restrictions to the order of participating tasks. In particular, the *alternate* supplement requires that participating tasks alternate whereas the *chain* supplement requires that executions of participating tasks are next to each other.

Figure 35(b) presents the unified model for the constraint *alternate response(A,B)*. The constraint alternate response(A,B) is used to specify that task B must be executed after the execution of task A and A can be executed again only after activity B is executed. The rules for the alternate response constraint are similar to the rules for the response constraint shown in figure 35(a). However, rule (8) in figure 35(a) is not allowed to be

$$
\begin{array}{clll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & \textit{with}: \; A_i, B_i, C_i \in \textit{Services} \\
(5) & A_1 & \longrightarrow aS_2 & \qquad\qquad S_i \in \textit{Helpers} \\
(6) & B_1 & \longrightarrow bS_3 & \qquad\qquad a,b,c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & \\
(8-10) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 & \\
(11) & A_2 & \longrightarrow aS_2 & \\
(12) & B_2 & \longrightarrow bS_4 & \\
(13) & C_2 & \longrightarrow cS_2 & \\
(14-16) & S_3 & \longrightarrow A_3 \mid B_3 \mid C_3 & \\
(17) & A_3 & \longrightarrow aS_4 & \\
(18) & B_3 & \longrightarrow bS_3 & \\
(19) & C_3 & \longrightarrow cS_3 & \\
(20-23) & S_4 & \longrightarrow A_4 \mid B_4 \mid C_4 \mid \varepsilon & \\
(24) & A_4 & \longrightarrow aS_4 & \\
(25) & B_4 & \longrightarrow bS_4 & \\
(26) & C_4 & \longrightarrow cS_4 &
\end{array}
$$

Figure 34: Unified model for the constraint *co-existence(A,B)*, i.e. if one of the activities A or B is executed the other one has to be executed as well.

part of the rules for the alternate response constraint as task A is not allowed to be executed after another execution of A as long as task B is executed.

The constraint *chain response(A,B)* is even more restrictive than the alternate response constraint. In detail, the constraint requires that task B has to be executed directly after an execution of task A. Figure 35(c) presents the unified model for the chain response. The rules are similar to the rules for the (alternate) response constraint. However, after the execution of task A exclusively task B can be executed, i.e. the rules (8–9) in figure 35(a), i.e. the rule (9) in figure 35(b) are not allowed to be part of the rules for the chain response constraint.

Similar but not equal to the response constraint the constraint *precedence(A,B)* is used to specify that A needs to have been executed when B is activated. Figure 36(a) presents the unified model for the precedence constraint. In the beginning only task A and other tasks in *Tasks* excepts B are allowed to be executed (cf. rule (1–3)). Once task A is executed rule (4) switches to the start symbol version S_2 indicating that task B is allowed to be executed. In the following the execution of all tasks in *Tasks* as well as the finishing of the service composition is allowed.

Figure 36(b) presents the unified model for the constraint *alternate precedence(A,B)*. This constraint is used to specify that A needs to have been executed when B is activated and task B cannot be repeatedly executed before A is also executed again. That means, the rules for the alternate precedence constraint are similar to the rules for the

$$
\begin{array}{lllll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & with: & A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_1 & & a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & \\
(8-10) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 & & \\
(11) & A_2 & \longrightarrow aS_2 & & \\
(12) & B_2 & \longrightarrow bS_1 & & \\
(13) & C_2 & \longrightarrow cS_2 & &
\end{array}
$$

(a) Constraint *response(A,B)*, i.e. activity B must be executed in future when A is executed at least once.

$$
\begin{array}{lllll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & with: & A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_1 & & a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & \\
(8-9) & S_2 & \longrightarrow B_2 \mid C_2 & & \\
(10) & B_2 & \longrightarrow bS_1 & & \\
(11) & C_2 & \longrightarrow cS_2 & &
\end{array}
$$

(b) Constraint *alternate response(A,B)*, i.e. after the execution of A activity B has to be executed and the activity A can be executed again only after activity B is executed.

$$
\begin{array}{lllll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & with: & A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_1 & & a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & \\
(8) & S_2 & \longrightarrow B_2 & & \\
(9) & B_2 & \longrightarrow bS_1 & &
\end{array}
$$

(c) Constraint *chain response(A,B)*, i.e. activity B has to be executed directly after A.

Figure 35: Unified model for the response constraints.

$$
\begin{array}{clll}
(1-3) & S_1 & \longrightarrow A_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(4) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(5) & C_1 & \longrightarrow cS_1 & \quad a, b, c \in \Sigma \\
(6-9) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 \mid \varepsilon & \\
(10) & A_2 & \longrightarrow aS_2 & \\
(11) & B_2 & \longrightarrow bS_2 & \\
(12) & C_2 & \longrightarrow cS_2 &
\end{array}
$$

(a) Constraint *precedence(A,B)*, i.e. activity A needs to be executed when B begins to execute.

$$
\begin{array}{clll}
(1-3) & S_1 & \longrightarrow A_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(4) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(5) & C_1 & \longrightarrow cS_1 & \quad a, b, c \in \Sigma \\
(6-8) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 & \\
(9) & A_2 & \longrightarrow aS_2 & \\
(10) & B_2 & \longrightarrow bS_1 & \\
(11) & C_2 & \longrightarrow cS_2 &
\end{array}
$$

(b) Constraint *alternate precedence(A,B)*, i.e. activity B has to be executed after A and B cannot be executed again before the activity A is also executed again.

$$
\begin{array}{clll}
(1-3) & S_1 & \longrightarrow A_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(4) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(5) & C_1 & \longrightarrow cS_1 & \quad a, b, c \in \Sigma \\
(6-9) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 \mid \varepsilon & \\
(10) & A_2 & \longrightarrow aS_2 & \\
(11) & B_2 & \longrightarrow bS_1 & \\
(12) & C_2 & \longrightarrow cS_1 &
\end{array}
$$

(c) Constraint *chain precedence(A,B)*, i.e. each B is directly preceded by an A.

Figure 36: Unified model for the precedence constraints.

precedence constraint but rule (10) is not allowed to hold the start symbol version S_2 as shown in figure 36(a). Instead, rule (10) needs to switch to the start symbol version S_1 disabling the execution of task B until A is executed once again.

Figure 36(c) presents the unified model for the constraint *chain precedence(A,B)*. This constraint is used to specify that A needs to have been executed immediately before B is activated and executed. This additional restriction restricts the rules in figure 36(a). Therefore, rule (11) in figure 36(a) needs to be adapted analogous to adaptations required by the alternate precedence constraint. Furthermore, rule (12) is not allowed to hold the start symbol version S_2 as shown in figure 36(a). Instead rule (12) needs to switch to the start symbol version S_1 disabling the execution of task B as task C is executed immediately after A, i.e. B is not executed immediately after A.

Figure 37 presents the unified model for the constraints succession, alternate succession, and chain succession. The constraint *succession(A,B)* specifies the combination of the response and precedence constraints, i.e. both constrained need to be fulfilled for participating tasks. Therefore, production rules for both constraints need to be merged. The merging algorithm introduced at the end of this section can be used. Similar to the response and precedence constraints the supplements alternate and chain also exist for the succession constraint.

3.3.3 Negation Templates

Negation templates provide constraints representing the negated versions of some relation templates. However, the negation should not be interpreted as the "logical negation" [4]. At first, figure 38 presents the unified model for the constraints not responded existence and not co-existence. The constraint *not responded existence(A,B)* is used to specify that task B is not allowed to be executed at all if task A is executed. In particular, task B is not allowed to be executed either before or after the execution of task A. The constraint *not co-existence(A,B)* specifies the same dependency in both directions, i.e. A is not allowed to be executed if B is executed and vice versa. As both constraint are equivalent (cf. the ConDec specification in [4]) the production rules presented in figure 38 are valid for both constraints.

The constraints not response, not precedence, and not succession create an equivalence class similar to the equivalence class created by the constraints not responded existence and not co-existence. The unified model for the constraints not response, not precedence, and not succession is represented in figure 39. In detail, the constraint *not response(A,B)* is used to specify that the task B cannot be executed after the execution of task A. Similar, the constraint *not precedence(A,B)* is used to specify that task B cannot be preceded by task A. As both constraint are equivalent the combination of these constraint, i.e. the constraint *not succession(A,B)* is also equivalent to the not response and the not precedence constraint.

$$
\begin{array}{rlll}
(1-3) & S_1 & \longrightarrow A_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(4) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(5) & C_1 & \longrightarrow cS_1 & \quad a, b, c \in \Sigma \\
(6-8) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 & \\
(9) & A_2 & \longrightarrow aS_2 & \\
(10) & B_2 & \longrightarrow bS_3 & \\
(11) & C_2 & \longrightarrow cS_2 & \\
(12-15) & S_3 & \longrightarrow A_3 \mid B_3 \mid C_3 \mid \varepsilon & \\
(16) & A_3 & \longrightarrow aS_2 & \\
(17) & B_3 & \longrightarrow bS_3 & \\
(18) & C_3 & \longrightarrow cS_3 &
\end{array}
$$

(a) Constraint *succession(A,B)*, i.e. response(A,B) AND precedence(A,B).

$$
\begin{array}{rlll}
(1-3) & S_1 & \longrightarrow A_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(4) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(5) & C_1 & \longrightarrow cS_1 & \quad a, b, c \in \Sigma \\
(6-7) & S_2 & \longrightarrow B_2 \mid C_2 & \\
(8) & B_2 & \longrightarrow bS_1 & \\
(9) & C_2 & \longrightarrow cS_2 &
\end{array}
$$

(b) Constraint *alternate succession(A,B)*, i.e. alternate response(A,B) AND alternate precedence(A,B).

$$
\begin{array}{rlll}
(1-3) & S_1 & \longrightarrow A_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(4) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(5) & C_1 & \longrightarrow cS_1 & \quad a, b, c \in \Sigma \\
(6) & S_2 & \longrightarrow B_2 & \\
(7) & B_2 & \longrightarrow bS_1 &
\end{array}
$$

(c) Constraint *chain succession(A,B)*, i.e. chain response(A,B) AND chain precedence(A,B).

Figure 37: Unified model for the succession constraints.

$$\begin{array}{clcll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & with: & A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_3 & & a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & \\
(8-10) & S_2 & \longrightarrow A_2 \mid C_2 \mid \varepsilon & & \\
(11) & A_2 & \longrightarrow aS_2 & & \\
(12) & C_2 & \longrightarrow cS_2 & & \\
(13-15) & S_3 & \longrightarrow B_3 \mid C_3 \mid \varepsilon & & \\
(16) & B_3 & \longrightarrow bS_3 & & \\
(17) & C_3 & \longrightarrow cS_3 & &
\end{array}$$

Figure 38: Unified model for the constraint *not responded existence(A,B)*, i.e. if A is executed B must never be executed (before and after A) and the constraint *not co-existence(A,B)*, i.e. not responded existence(A,B) AND not responded existence(B,A).

$$\begin{array}{clcll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & with: & A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_1 & & a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & \\
(8-10) & S_2 & \longrightarrow A_2 \mid C_2 \mid \varepsilon & & \\
(11) & A_2 & \longrightarrow aS_2 & & \\
(12) & C_2 & \longrightarrow cS_2 & &
\end{array}$$

Figure 39: Unified model for the constraint *not response(A,B)*, i.e. if A is executed B cannot be executed any more, the constraint *not precedence(A,B)*, i.e. B cannot be preceded by A, and the constraint *not succession(A,B)*, i.e. not response(A,B) AND not precedence(A,B).

Similarly, the constraints chain response, chain precedence, and chain succession create an equivalence class. Figure 40 presents the unified model for this class. The constraint *not chain response* is used to specify that task A is not allowed to be directly followed by the task B. Equivalently, the constraint *not chain precedence* is used to specify that task B is not allowed to be directly preceded by the task A. The constraint *not chain succession* the constraints not chain response and not chain precedence.

$$
\begin{array}{clllll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon & with: & A_i, B_i, C_i & \in Services \\
(5) & A_1 & \longrightarrow aS_2 & & S_i & \in Helpers \\
(6) & B_1 & \longrightarrow bS_1 & & a, b, c & \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & & & \\
(8-10) & S_2 & \longrightarrow A_2 \mid C_2 \mid \varepsilon & & & \\
(11) & A_2 & \longrightarrow aS_2 & & & \\
(12) & C_2 & \longrightarrow cS_1 & & &
\end{array}
$$

Figure 40: Unified model for the constraint *not chain response(A,B)*, i.e. A should never be followed directly by B, the constraint *not chain precedence(A,B)*, i.e. B should never be preceded directly by A, and the constraint *not chain succession(A,B)*, i.e. not chain response(A,B) AND not chain precedence(A,B).

3.3.4 Choice Templates

In contrast to previous templates covering unary and binary constraints this section introduces n-ary constraints for the first time. The constraints covered by the choice templates specify the need for choosing between several activities. In general, the execution of other tasks in between is allowed. Consistently to previous sections the other tasks are exemplarily represented by the task C in introduced grammatical production rules. However, the set of all possible tasks need to be extended in order to allow further tasks next to A, B, and C. In the following the utilized sets of all possible tasks are individually specified as needed.

Let $Tasks = \{A, B, C, D\}$ be the set of all possible tasks. Then, the constraint *1of3(A,B,D)* can be used to specify that at least one of the three tasks A, B, and D has to be executed. However, all three tasks can be executed and each of them can be executed multiple times. Figure 41 presents the unified model for the constraint 1of3. Actually the constraint 1of3 is an instance of the generic constraint *NofM*. Additionally other instances can be specified, e.g. 1of2, 1of3, 1of8, 2of21. However, the creation of rules implements the same procedure for each instance. Figure 42 shows the algorithm implementing the creation of rules for the generic constraint $NofM(X_1, X_2, ..., X_M)$ assuming a concrete value for N and M. That means, the algorithm can be used for generating the production rules for all instances of the constraint NofM. In detail, the

$$
\begin{array}{llll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid D_1 & \\
(5) & A_1 & \longrightarrow aS_2 & \\
(6) & B_1 & \longrightarrow bS_2 & \\
(7) & C_1 & \longrightarrow cS_1 & \\
(8) & D_1 & \longrightarrow dS_2 & \\
(9-13) & S_2 & \longrightarrow A_2 \mid B_2 \mid C_2 \mid D_2 \mid \varepsilon & \\
(14) & A_2 & \longrightarrow aS_2 & \\
(15) & B_2 & \longrightarrow bS_2 & \\
(16) & C_2 & \longrightarrow cS_2 & \\
(17) & D_2 & \longrightarrow dS_2 &
\end{array}
\qquad
with: \begin{array}{rl}
A_i, B_i, C_i & \in Services \\
S_i & \in Helpers \\
a, b, c & \in \Sigma
\end{array}
$$

Figure 41: Unified model for the constraint 1of3(A,B,D), i.e. at least one of the three activities A, B, and D has to be executed.

algorithm creates a grammar storing the set of non-terminals V, the set of terminals Σ, and the start symbol S next to the production rules P. In this case the set of all possible tasks is $Tasks = \{X_1, X_2, ..., X_N, C\}$ where C represents another task that is not covered by the constraint.

In created production rules the index indicating the version of the start symbol version is more complex than in previously introduced rules. In detail, the index i is generic as the constraint and is represented by an M-tuple. For each task covered by the constraint the index indicates whether the task was executed once or not. In particular,

$$i \subseteq \{0,1\} \times ... \times \{0,1\} = \{0,1\}^M$$

The concrete value $i(k) = 1$ indicates that the task $X_k \in \{X_1, X_2, ..., X_M\}$ is executed one or more times. In contrast the concrete value $i(k) = 0$ indicates that the task X_k was not executed so far. For example, the index $i = (0, 0, ..., 0)$ was introduced for the start symbol version $S_{(0,0,...,0)}$ indicating the very beginning of the execution. The complex index, i.e. the m-tuple needs to be transferred to a simple index before the grammar is allowed to be merged by the corresponding algorithm introduced at the end of this chapter. In particular, the m-tuple needs to be transferred to a single number. For example, the index $i = (0, 0, ..., 0)$ should be transferred to the index $i = 0$ and the index $i = (0, 1, 0, ..., 0)$ should be transferred to the index $i = 0$ and so on.

Exemplarily, the production rules for the constraint $2of4(X_1, X_2, X_3, X_4)$ created by the algorithm in figure 42 are shown in figure 43. Obviously, the execution of the task X_2 causes the switch to the start symbol version $S_{(0,1,0,0)}$ in rule (7). This start symbol version with $index = (0, 1, 0, 0)$ indicates that the task X_2 was executed at least once as $index(2) = 2$. If the task X_2 is executed once again in the next step rule (27) causes to stay in this start symbol version as no important information for the constraint was added by the repeated execution of the task X_2. However, if another task, e.g. X_1

Require: Constraint NofM$(X_1, X_2, ..., X_M)$
Ensure: $G = (V, \Sigma, P, S)$

$\mathcal{S} = \{S_{(0,0,...,0)}\}$
$S = S_{(0,0,...,0)}$
while $\mathcal{S} \neq \emptyset$ **do**
  $V = V \cup \mathcal{S}$
  for all $q : S_q \in \mathcal{S}$ **do**
    $\exists X_{j_q} \in V \ \forall j \in \{1, 2, ..., M\}$
    $\exists (S_q, X_{j_q}) \in P \ \forall j \in \{1, 2, ..., M\}$
    $\exists C_q \in V$
    $\exists (S_q, C_q) \in P$
    $\exists (C_q, cS_q) \in P$
  end for
  $\mathcal{T} = \{X_{j_q} \mid \forall q : S_q \in \mathcal{S}, \ \forall j \in \{1, 2, ..., M\}\}$
  $\mathcal{S} = \emptyset$
  for all $X_{j_i} \in \mathcal{T}$ **do**
    $i = (r_1, r_2, ..., r_M)$
    if $card(1 \in i) = N - 1$ **then**
      $S_k = S_F$
    else if $card(1 \in i) < N - 1$ **then**

$$S_k = S_q \text{ with } q = \begin{cases} (1, r_2, ..., r_M) & \text{j=1} \\ (r_1, 1, ..., r_M) & \text{j=2} \\ ... & \\ (r_1, r_2, ..., 1) & \text{j=M} \end{cases}$$

    end if
    $\exists (X_{j_i}, x_j S_k) \in P$
    if $S_k \neq S_F \ \wedge S_k \notin V$ **then**
      $\mathcal{S} = \mathcal{S} \cup \{S_k\}$
    end if
  end for
end while
$\exists S_F \in V$
$\exists X_{j_F} \in V \ \forall j \in \{1, 2, ..., M\}$
$\exists (S_F, X_{j_F}) \in P \ \forall j \in \{1, 2, ..., M\}$
$\exists (S_F, \varepsilon) \in P$
$\exists (X_{j_F}, x_j S_F) \in P \ \forall j \in \{1, 2, ..., M\}$
$\exists (S_F, C_F) \in P$
$\exists (C_F, cS_F) \in P$

Figure 42: Creating production rules for the constraint NofM$(X_1, X_2, ..., X_M)$, i.e. at least N of the M tasks $X_1, ..., X_M$ have to be executed.

would be executed in the next step rule (26) would allow to switch to the final version S_F of the start symbol as the constraint 2of4 would be fulfilled.

$$
\begin{array}{crll}
(1-5) & S_{(0,0,0,0)} & \longrightarrow & X1_{(0,0,0,0)} \mid X2_{(0,0,0,0)} \mid X3_{(0,0,0,0)} \mid X4_{(0,0,0,0)} \mid C_{(0,0,0,0)} \\
(6) & X1_{(0,0,0,0)} & \longrightarrow & x_1 S_{(1,0,0,0)} \\
(7) & X2_{(0,0,0,0)} & \longrightarrow & x_2 S_{(0,1,0,0)} \\
(8) & X3_{(0,0,0,0)} & \longrightarrow & x_3 S_{(0,0,1,0)} \\
(9) & X4_{(0,0,0,0)} & \longrightarrow & x_4 S_{(0,0,0,1)} \\
(10) & C_{(0,0,0,0)} & \longrightarrow & c S_{(0,0,0,0)} \\
(11-15) & S_{(1,0,0,0)} & \longrightarrow & X1_{(1,0,0,0)} \mid X2_{(1,0,0,0)} \mid X3_{(1,0,0,0)} \mid X4_{(1,0,0,0)} \mid C_{(1,0,0,0)} \\
(16) & X1_{(1,0,0,0)} & \longrightarrow & x_1 S_{(1,0,0,0)} \\
(17) & X2_{(1,0,0,0)} & \longrightarrow & x_2 S_F \\
(18) & X3_{(1,0,0,0)} & \longrightarrow & x_3 S_F \\
(19) & X4_{(1,0,0,0)} & \longrightarrow & x_4 S_F \\
(20) & C_{(1,0,0,0)} & \longrightarrow & c S_{(1,0,0,0)} \\
(21-25) & S_{(0,1,0,0)} & \longrightarrow & X1_{(0,1,0,0)} \mid X2_{(0,1,0,0)} \mid X3_{(0,1,0,0)} \mid X4_{(0,1,0,0)} \mid C_{(0,1,0,0)} \\
(26) & X1_{(0,1,0,0)} & \longrightarrow & x_1 S_F \\
(27) & X2_{(0,1,0,0)} & \longrightarrow & x_2 S_{(0,1,0,0)} \\
(28) & X3_{(0,1,0,0)} & \longrightarrow & x_3 S_F \\
(29) & X4_{(0,1,0,0)} & \longrightarrow & x_4 S_F \\
(30) & C_{(0,1,0,0)} & \longrightarrow & c S_{(0,1,0,0)} \\
... & & & \\
(51-55) & S_F & \longrightarrow & X_{1_F} \mid X_{2_F} \mid X_{3_F} \mid X_{4_F} \mid C_F \mid \varepsilon \\
(56) & X_{1_F} & \longrightarrow & x_1 S_F \\
(57) & X_{2_F} & \longrightarrow & x_2 S_F \\
(58) & X_{3_F} & \longrightarrow & x_3 S_F \\
(59) & X_{4_F} & \longrightarrow & x_4 S_F \\
(60) & C_F & \longrightarrow & c S_F \\
 & & & \\
with: & X_i, C_i & \in & Services \\
 & S_i & \in & Helpers \\
 & a, b, c & \in & \Sigma
\end{array}
$$

Figure 43: Creating production rules for the constraint 2of4(X_1, X_2, X_3, X_4), i.e. at least 2 of the 4 tasks X_1, X_2, X_3, X_4 have to be executed.

The constraint *exclusive NofM*($X_1, X_2, ..., X_M$) is similar to the constraint NofM but restricts the tasks that can be executed in the final version of the start symbol. For example, the constraint exclusive 1of3(A,B,D) is used to specify that one fixed task of the three tasks A, B, and D can be executed one or multiple times but the other ones cannot be executed at all. Similar, the constraint exclusive 2of4(X_1, X_2, X_3, X_4)

is used to specify that two of the four tasks can be executed but the remaining two tasks cannot be executed at all. Figure 44 presents the unified model for the constraint *exclusive 1of3(A,B,D)*. Obviously, the execution of one of the tasks covered by the constraint causes the switch to a start symbol version only allowing the execution of the same task, the execution of the other task C, and the finishing of execution but not the execution of another task covered by the constraint. For example the execution of the task A causes the switch to the start symbol version S_2 that does not allow the execution of the tasks B and D in further processing.

$$
\begin{array}{clll}
(1-4) & S_1 & \longrightarrow A_1 \mid B_1 \mid C_1 \mid D_1 & \quad with: \; A_i, B_i, C_i \in Services \\
(5) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(6) & B_1 & \longrightarrow bS_3 & \quad a, b, c \in \Sigma \\
(7) & C_1 & \longrightarrow cS_1 & \\
(8) & D_1 & \longrightarrow dS_4 & \\
(9-11) & S_2 & \longrightarrow A_2 \mid C_2 \mid \varepsilon & \\
(12) & A_2 & \longrightarrow aS_2 & \\
(13) & C_2 & \longrightarrow cS_2 & \\
(9-13) & S_3 & \longrightarrow B_3 \mid C_3 \mid \varepsilon & \\
(14) & B_3 & \longrightarrow bS_3 & \\
(15) & C_3 & \longrightarrow cS_3 & \\
(9-13) & S_4 & \longrightarrow D_4 \mid C_4 \mid \varepsilon & \\
(14) & D_4 & \longrightarrow dS_4 & \\
(15) & C_4 & \longrightarrow cS_4 &
\end{array}
$$

Figure 44: Unified model for the constraint exclusive 1of3(A,B,D), i.e. one of the three tasks A, B and D has to be executed, while the others cannot be executed at all.

The algorithm generating the production rules for all instances of the constraint exclusive NofM is shown in figure 45. The algorithm is similar but not equal to algorithm for the constraint NofM shown in figure 42. In particular, there exists no single final version S_F of the start symbol for the constraint exclusive NofM. Therefore, the lines 6–14 are required for the constraint exclusive NofM in figure 45 instead of the lines 27–33 in figure 42. As shown in line 14 the varied number of final states also affects to the set $\mathcal{T}$ storing newly introduced non-terminals for tasks in *Tasks* requiring the creation of further production rules. Figure 46 shows the production rules generated by the presented algorithm for the constraint exclusive 2of4(X_1, X_2, X_3, X_4).

3.3.5 Branching of Constraints

Each of the previously introduced constraints can be extended in order to deal with more tasks than predefined. For example the constraint response(A, B) can be extended

Require: Constraint exclusive NofM$(X_1, X_2, ..., X_M)$
Ensure: $G = (V, \Sigma, P, S)$
$\mathcal{S} = \{S_{(0,0,\ldots,0)}\}$
$S = S_{(0,0,\ldots,0)}$
while $\mathcal{S} \neq \emptyset$ **do**
  $V = V \cup \mathcal{S}$
  for all $q : S_q \in \mathcal{S}$ **do**
    if $card(1 \in q) = N$ **then**
      $\exists (S_q, \varepsilon) \in P$
      for all $j \in \{1, 2, ..., M\}$ **do**
        if $q(j) = 1$ **then**
          $\exists X_{j_q} \in V$
          $\exists (S_q, X_{j_q}) \in P$
        end if
      end for
      $\mathcal{T} = \{X_{j_q} \mid \forall q : S_q \in \mathcal{S} \wedge \forall j \in \{1, 2, ..., M\} : q(j) = 1\}$
    else
      $\exists X_{j_q} \in V \ \forall j \in \{1, 2, ..., M\}$
      $\exists (S_q, X_{j_q}) \in P \ \forall j \in \{1, 2, ..., M\}$
      $\mathcal{T} = \{X_{j_q} \mid \forall q : S_q \in \mathcal{S} \ \forall j \in \{1, 2, ..., M\}\}$
    end if
    $\exists C_q \in V$
    $\exists (S_q, C_q) \in P$
    $\exists (C_q, cS_q) \in P$
  end for
  $\mathcal{S} = \emptyset$
  for all $X_{j_i} \in \mathcal{T}$ **do**
    $i = (r_1, r_2, ..., r_M)$
    if $card(1 \in i) = N$ **then**
      $S_k = S_i$
    else if $card(1 \in i) < N$ **then**
      $$S_k = S_q \text{ with } q = \begin{cases} (1, r_2, ..., r_M) & \text{j=1} \\ (r_1, 1, ..., r_M) & \text{j=2} \\ ... & \\ (r_1, r_2, ..., 1) & \text{j=M} \end{cases}$$
    end if
    $\exists (X_{j_i}, x_j S_k) \in P$ with
    if $S_k \notin V$ **then**
      $\mathcal{S} = \mathcal{S} \cup \{S_k\}$
    end if
  end for
end while

Figure 45: Creating production rules for the constraint exclusive NofM$(X_1, X_2, ..., X_M)$, i.e. N of the M tasks $X_1, ..., X_M$ have to be executed while the remaining ones cannot be executed at all.

$$
\begin{array}{cll}
(1-5) & S_{(0,0,0,0)} & \longrightarrow X1_{(0,0,0,0)} \mid X2_{(0,0,0,0)} \mid X3_{(0,0,0,0)} \mid X4_{(0,0,0,0)} \mid C_{(0,0,0,0)} \\
(6) & X1_{(0,0,0,0)} & \longrightarrow x_1 S_{(1,0,0,0)} \\
(7) & X2_{(0,0,0,0)} & \longrightarrow x_2 S_{(0,1,0,0)} \\
(8) & X3_{(0,0,0,0)} & \longrightarrow x_3 S_{(0,0,1,0)} \\
(9) & X4_{(0,0,0,0)} & \longrightarrow x_4 S_{(0,0,0,1)} \\
(10) & C_{(0,0,0,0)} & \longrightarrow c S_{(0,0,0,0)} \\
(11-15) & S_{(1,0,0,0)} & \longrightarrow X1_{(1,0,0,0)} \mid X2_{(1,0,0,0)} \mid X3_{(1,0,0,0)} \mid X4_{(1,0,0,0)} \mid C_{(1,0,0,0)} \\
(16) & X1_{(1,0,0,0)} & \longrightarrow x_1 S_{(1,0,0,0)} \\
(17) & X2_{(1,0,0,0)} & \longrightarrow x_2 S_{(1,1,0,0)} \\
(18) & X3_{(1,0,0,0)} & \longrightarrow x_3 S_{(1,0,1,0)} \\
(19) & X4_{(1,0,0,0)} & \longrightarrow x_4 S_{(1,0,0,1)} \\
(20) & C_{(1,0,0,0)} & \longrightarrow c S_{(1,0,0,0)} \\
(21-25) & S_{(0,1,0,0)} & \longrightarrow X1_{(0,1,0,0)} \mid X2_{(0,1,0,0)} \mid X3_{(0,1,0,0)} \mid X4_{(0,1,0,0)} \mid C_{(0,1,0,0)} \\
(26) & X1_{(0,1,0,0)} & \longrightarrow x_1 S_{(1,1,0,0)} \\
(27) & X2_{(0,1,0,0)} & \longrightarrow x_2 S_{(0,1,0,0)} \\
(28) & X3_{(0,1,0,0)} & \longrightarrow x_3 S_{(0,1,1,0)} \\
(29) & X4_{(0,1,0,0)} & \longrightarrow x_4 S_{(0,1,0,1)} \\
(30) & C_{(0,1,0,0)} & \longrightarrow c S_{(0,1,0,0)} \\
\ldots & & \\
(51-54) & S_{(1,1,0,0)} & \longrightarrow X1_{(1,1,0,0)} \mid X2_{(1,1,0,0)} \mid C_{(1,1,0,0)} \mid \varepsilon \\
(55) & X1_{(1,1,0,0)} & \longrightarrow x_1 S_{(1,1,0,0)} \\
(56) & X2_{(1,1,0,0)} & \longrightarrow x_2 S_{(1,1,0,0)} \\
(57) & C_{(1,1,0,0)} & \longrightarrow c S_{(1,1,0,0)} \\
(58-61) & S_{(1,0,1,0)} & \longrightarrow X1_{(1,0,1,0)} \mid X3_{(1,0,1,0)} \mid C_{(1,0,1,0)} \mid \varepsilon \\
(62) & X1_{(1,0,1,0)} & \longrightarrow x_1 S_{(1,0,1,0)} \\
(63) & X3_{(1,0,1,0)} & \longrightarrow x_3 S_{(1,0,1,0)} \\
(64) & C_{(1,0,1,0)} & \longrightarrow c S_{(1,0,1,0)} \\
(65-68) & S_{(1,0,0,1)} & \longrightarrow X1_{(1,0,0,1)} \mid X4_{(1,0,0,1)} \mid C_{(1,0,0,1)} \mid \varepsilon \\
(69) & X1_{(1,0,0,1)} & \longrightarrow x_1 S_{(1,0,0,1)} \\
(70) & X4_{(1,0,0,1)} & \longrightarrow x_4 S_{(1,0,0,1)} \\
(71) & C_{(1,0,0,1)} & \longrightarrow c S_{(1,0,0,1)} \\
(72-75) & S_{(0,1,1,0)} & \longrightarrow X2_{(0,1,1,0)} \mid X3_{(0,1,1,0)} \mid C_{(0,1,1,0)} \mid \varepsilon \\
(76) & X2_{(0,1,1,0)} & \longrightarrow x_2 S_{(0,1,1,0)} \\
(77) & X3_{(0,1,1,0)} & \longrightarrow x_3 S_{(0,1,1,0)} \\
(78) & C_{(0,1,1,0)} & \longrightarrow c S_{(0,1,1,0)} \\
(79-82) & S_{(0,1,0,1)} & \longrightarrow X2_{(0,1,0,1)} \mid X4_{(0,1,0,1)} \mid C_{(0,1,0,1)} \mid \varepsilon \\
(83) & X2_{(0,1,0,1)} & \longrightarrow x_2 S_{(0,1,0,1)} \\
(84) & X4_{(0,1,0,1)} & \longrightarrow x_4 S_{(0,1,0,1)} \\
(85) & C_{(0,1,0,1)} & \longrightarrow c S_{(0,1,0,1)} \\
(86-89) & S_{(0,0,1,1)} & \longrightarrow X3_{(0,0,1,1)} \mid X4_{(0,0,1,1)} \mid C_{(0,0,1,1)} \mid \varepsilon \\
(90) & X3_{(0,0,1,1)} & \longrightarrow x_3 S_{(0,0,1,1)} \\
(91) & X4_{(0,0,1,1)} & \longrightarrow x_4 S_{(0,0,1,1)} \\
(92) & C_{(0,0,1,1)} & \longrightarrow c S_{(0,0,1,1)} \\
 & & \\
\textit{with}: & X_i, C_i & \in \textit{Services} \\
 & S_i & \in \textit{Helpers} \\
 & a, b, c & \in \Sigma
\end{array}
$$

Figure 46: Creating production rules for the constraint exclusive 2of4(X_1, X_2, X_3, X_4), i.e. 2 of the 4 tasks X_1, X_2, X_3, X_4 have to be executed while the remaining ones cannot be executed at all.

to response(A_1, A_2, B), response(A, B_1, B_2), or response$(A_1, A_2, B_1, B_2, B_3)$. However, ConDec calls this extension mechanism branching which is mainly driven by the graphical representation. The constraint response(A, B) is graphically represented by a specific arrow from the task A to the task B. This arrow can be branched in order to include further tasks in the dependency. For example, an arrow can have a single source A and two targets B_1 and B_2 in order to represent the constraint response(A, B_1, B_2). However, the branching requires to explicitly specify the sources distinct from the targets.

Branching fundamentally implements an OR-dependency between tasks. Therefore, branching requires additional alternatives in production rules. Considering the constraint response(A, B) the branching of the target into the alternatives B_1 and B_2 requires to provide production rules specifying B_1 and B_2 instead of the rules specifying B. That means, each production rule that is valid for the constraint response(A, B) is also valid for the constraint response(A, B_1, B_2) if the occurrence of B is substituted with B_1 and B_2 whereas the occurrences on the *lhs* as well on the *rhs* need to be covered. However, the method is also valid for a branching covering the source A of the constraint. Figure 47 shows the production rules for the branched versions *response*$(A, (B_1, B_2))$ and *response*$((A_1, A_2), B)$.

In general, definition 24 can be used to derive a grammar G' that is valid for a branched version of a constraint given by a grammar G. For example, the production rules for the constraint responded existence$((A_1, A_2), B)$ can be derived by using definition 24 with the given grammar G specifying the constraint responded existence(A, B) as introduced in figure 33 by substituting the non-terminal A by the non-terminals A_1 and A_2.

Definition 24 (Branching Constraints)
Let $G = (V, \Sigma, P, S)$ be a grammar that is valid for a constraint covering a set of tasks $T = Y_1, ..., X, ..., Y_n$. Then the grammar $G' = (V', \Sigma', P', S)$ is valid for the branched version of the constraint substituting the task X by the tasks X_1 and X_2 with:

$$
\begin{aligned}
V' &= V \setminus \{X\} \cup \{X_1, X_2\} \\
\Sigma' &= \Sigma \setminus \{x\} \cup \{x_1, x_2\} \\
P' &= P \setminus \{(\alpha, \beta) \mid X \in \alpha \vee X \in \beta\} \cup Q \\
Q &= \{(\alpha', \beta') \mid \exists (\alpha, \beta) \in P \wedge (X \in \alpha \vee X \in \beta)\} \text{ with:} \\
&\quad \alpha' = \text{ substitute } X \text{ by } X_i \text{ and } x \text{ by } x_i \text{ in } \alpha \\
&\quad \beta' = \text{ substitute } X \text{ by } X_i \text{ and } x \text{ by } x_i \text{ in } \beta \\
&\quad i \in \{1, 2\}
\end{aligned}
$$

⌟

$$
\begin{array}{rrll}
(1-5) & S_1 & \longrightarrow A_1 \mid B_{1_1} \mid B_{2_1} \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(6) & A_1 & \longrightarrow aS_2 & \quad S_i \in Helpers \\
(7) & B_{1_1} & \longrightarrow b_1S_1 & \quad a, b, c \in \Sigma \\
(8) & B_{2_1} & \longrightarrow b_2S_1 & \\
(9) & C_1 & \longrightarrow cS_1 & \\
(10-13) & S_2 & \longrightarrow A_2 \mid B_{1_2} \mid B_{2_2} \mid C_2 & \\
(14) & A_2 & \longrightarrow aS_2 & \\
(15) & B_{1_2} & \longrightarrow b_1S_1 & \\
(16) & B_{2_2} & \longrightarrow b_2S_1 & \\
(17) & C_2 & \longrightarrow cS_2 &
\end{array}
$$

(a) Constraint *response*($A, (B_1, B_2)$), i.e. both tasks B_1 and B_2 are the target of the dependency specified by the constraint.

$$
\begin{array}{rrll}
(1-4) & S_1 & \longrightarrow A_{1_1} \mid A_{2_1} \mid B_1 \mid C_1 \mid \varepsilon & \quad with: \; A_i, B_i, C_i \in Services \\
(5) & A_{1_1} & \longrightarrow a_1S_2 & \quad S_i \in Helpers \\
(5) & A_{2_1} & \longrightarrow a_2S_2 & \quad a, b, c \in \Sigma \\
(6) & B_1 & \longrightarrow bS_1 & \\
(7) & C_1 & \longrightarrow cS_1 & \\
(8-10) & S_2 & \longrightarrow A_{1_2} \mid A_{2_2} \mid B_2 \mid C_2 & \\
(11) & A_{1_2} & \longrightarrow a_1S_2 & \\
(11) & A_{2_2} & \longrightarrow a_2S_2 & \\
(12) & B_2 & \longrightarrow bS_1 & \\
(13) & C_2 & \longrightarrow cS_2 &
\end{array}
$$

(b) Constraint *response*($(A_1, A_2), B$), i.e. both tasks A_1 and A_2 are the source of the dependency specified by the constraint.

Figure 47: Production rules for branched versions of the constraints response(A, B), i.e. for the version response$(A, (B_1, B_2))$ and the version response$((A_1, A_2), B)$.

3.3.6 Combination Algorithm

Typically, a declarative service composition model specifies multiple tasks and different constraints between the tasks. The approach at hand firstly transforms single constraints to grammars, i.e. production rules. In order to combine the grammars covering constraints in the same service composition model the approach at hand calculates the cross product of production rules in the grammars covering single constraints. Therefore, all constraints excepts the init-constraints need to be merged, i.e. the intersection of related production rules need to be calculated in order to satisfy the constraints concurrently. In contrast, production rules specifying the init-constraints need to be joined, i.e. the union of related production rules need to be calculated in order to satisfy the combination of init constraints (i.e. constraint instances covering different tasks). The combination of grammars assume that the same tasks in the grammars are represented by the same symbols, e.g. the task A is always represented by the non-terminals A_i and the finishing of the task is always represented by the terminal a.

Figure 48 shows the algorithm for combining two grammars specifying declarative service composition models. At first the algorithm extracts the production rules from both grammars covering the init constraint. Afterwards, the production rules of grammar G_1 exclusively covering the init constraint are joined with the production rules of grammar G_2 exclusively covering the init constraint. Then, the remaining production rules of grammar G_1 (covering all other constraints excepts the init constraint) are merged with the remaining production rules of grammar G_2. Finally, the resulting grammars from the join and the merge are combined in order to create the combined grammar G_3. Note that the combination algorithm is not commutative as the used merging algorithm is not commutative. However, the combination algorithm becomes commutative if the used merging algorithm is commutative.

Merging Algorithm The merging of constraints typically strengthen the restrictions as all participating constraints need to be satisfied and the particular restrictions are concurrently valid. Therefore, the introduced merging algorithm creates the intersection of related sets of production rules. In particular, production rules that provide the same *lhs* and that specify different constraints are related to each other. The merging algorithm only covers tasks and relations that are specified in the given rules, i.e. no further information is added.

Note that the merging operation is not commutative. In particular, the merging algorithm allows to merge a grammar G_1 covering one or multiple constraints with a grammar G_2 that is allowed to cover exactly one single constraint. Figure 49 shows the merging algorithm for constraint-based grammars G_1 and G_2. The algorithm uses complex indexes for non-terminals representing the correlation between the two grammars G_1 and G_2. In particular, the index is implemented by a tuple where the first value

Require: $G_1 = (V_1, \Sigma_1, P_1, S_u) \wedge G_2 = (V_2, \Sigma_2, P_2, S_v)$
Ensure: $G_3 = combine(G_1, G_2) = (V_3, \Sigma_3, P_3, S_{(u,v)})$

$S_{init_1} = S_u$
$V_{init_1} = \{X_i \mid X_i \in V_1 \wedge i = u\}$
$\Sigma_{init_1} = \{x \mid x \in \Sigma_1 \wedge \exists(X_i, \alpha x \beta) \in P_1 \wedge i = u \wedge X_i \in V_1 \wedge \alpha, \beta \in (V_1 \cup \Sigma_1)^*\}$
$P_{init_1} = \{(\alpha X_i \beta, \gamma) \mid \forall X_i \in V_1 \wedge i = u \wedge \exists(\alpha X_i \beta, \gamma) \in P_1 \wedge \alpha, \beta \in \Sigma_1^* \wedge \gamma \in (V_1 \cup \Sigma_1)^*\}$
$S_{init_2} = S_v$
$V_{init_2} = \{X_i \mid X_i \in V_2 \wedge i = v\}$
$\Sigma_{init_2} = \{x \mid x \in \Sigma_2 \wedge \exists(X_i, \alpha x \beta) \in P_2 \wedge i = v \wedge X_i \in V_2 \wedge \alpha, \beta \in (V_2 \cup \Sigma_2)^*\}$
$P_{init_2} = \{(\alpha X_i \beta, \gamma) \mid \forall X_i \in V_2 \wedge i = v \wedge \exists(\alpha X_i \beta, \gamma) \in P_2 \wedge \alpha, \beta \in \Sigma_2^* \wedge \gamma \in (V_2 \cup \Sigma_2)^*\}$
if $u = 0 \wedge v = 0$ **then**
    $(G_4, \Omega) = join((V_{init_1}, \Sigma_{init_1}, P_{init_1}, S_{init_1}), (V_{init_2}, \Sigma_{init_2}, P_{init_2}, S_{init_2}))$
else if $u = 0 \wedge v \neq 0$ **then**
    $(G_4, \Omega) = join(G_{init_1}, u, v)$
else if $u \neq 0 \wedge v = 0$ **then**
    $(G_4, \Omega) = join(G_{init_2}, u, v)$
end if
Select $S_{(u',v')} \in \Omega$
$G_5 = (V_1 \backslash V_{init_1} \cup \{S_{u'}\}, \Sigma_1, P_1 \backslash P_{init_1}, S_{u'})$
$G_6 = (V_2 \backslash V_{init_2} \cup \{S_{v'}\}, \Sigma_2, P_2 \backslash P_{init_2}, S_{v'})$
if $u = 0 \wedge v = 0$ **then**
    $G_7 = merge(G_5, G_6)$
else if $u = 0 \wedge v \neq 0$ **then**
    $G_7 = merge(G_5, G_2)$
else if $u \neq 0 \wedge v = 0$ **then**
    $G_7 = merge(G_1, G_6)$
end if
$V_3 = V_4 \cup V_7$
$\Sigma_3 = \Sigma_4 \cup \Sigma_7$
$P_3 = P_4 \cup P_7$

Figure 48: Combining two grammars, i.e. production rules for combining constraint-based process models.

represents the particular start symbol version in G_1 and the second value represents the particular start symbol version of G_2. For example, the index $S_{(8,9)}$ combines the version S_8 from G_1 and S_9 from G_2.

Require: $G_1 = (V_1, \Sigma_1, P_1, S_u) \wedge G_2 = (V_2, \Sigma_2, P_2, S_v)$
Ensure: $G_3 = merge(G_1, G_2) = (V_3, \Sigma_3, P_3, S_{(u,v)})$
$\mathcal{S} = \{S_{(u,v)}\}$
while $\mathcal{S} \neq \emptyset$ **do**
  $\mathcal{S} = \mathcal{S} \setminus \{S_{(i,j)}\}$
  $\exists S_{(i,j)} \in V_3$
  for all $(S_i, Y_i) \in P_1 \wedge (S_j, Y_j) \in P_2$ **do**
    $\exists Y_{(i,j)} \in V_3$
    $\exists (S_{(i,j)}, Y_{(i,j)}) \in P_3$
  end for
  for all $(X_i, xS_k) \in P_1 \wedge (X_j, xS_l) \in P_2$ with $x \in \Sigma_1 \cup \Sigma_2$ **do**
    $\exists x \in \Sigma_3$
    $\exists (X_{(i,j)}, xS_{(k,l)}) \in P_3$
    if $S_{(k,l)} \notin V_3$ **then**
      $\mathcal{S} = \mathcal{S} \cup \{S_{(k,l)}\}$
    end if
  end for
end while

Figure 49: Merging algorithm for grammars specifying constraints, i.e. calculating the intersection of contained production rules.

The merging algorithm relates similar production rules in both input grammars in order to create correlated production rules in the merged grammar G_3. Therefore, the restrictions given by both grammars are fulfilled in combination in the resulting grammar. Figure 50 shows the production rules created by the merging algorithm by merging the constraints response(A,B) and precedence(A,C).

Join Algorithm As mentioned before, the combination of production rules covering the init constraint requires a special operator with different semantics than introduced by the merging algorithm in figure 49. In detail, the combination of init constraints require the union of correlated production rules instead of the intersection. Using the union of production rules ensure the semantics of satisfying the init constraint of both input grammars concurrently in the output grammar.

Figure 51 shows the join algorithm for grammars specifying the init constraint. The join algorithm assumes that a tasks can be initially executed at most one time in a single grammar. That means, corresponding to the init constraint in ConDec a

$$
\begin{array}{lllll}
(1-3) & S_{(1,1)} & \longrightarrow A_{(1,1)} \mid B_{(1,1)} \mid \varepsilon & \quad with: & A_i, B_i, C_i \in Services \\
(4) & A_{(1,1)} & \longrightarrow aS_{(2,2)} & & S_i \in Helpers \\
(5) & B_{(1,1)} & \longrightarrow bS_{(1,1)} & & a, b, c \in \Sigma \\
(1-4) & S_{(2,2)} & \longrightarrow A_{(2,2)} \mid B_{(2,2)} \mid C_{(2,2)} & & \\
(4) & A_{(2,2)} & \longrightarrow aS_{(2,2)} & & \\
(4) & B_{(2,2)} & \longrightarrow aS_{(1,2)} & & \\
(4) & C_{(2,2)} & \longrightarrow aS_{(2,2)} & & \\
(1-4) & S_{(1,2)} & \longrightarrow A_{(1,2)} \mid B_{(1,2)} \mid C_{(1,2)} \mid \varepsilon & & \\
(4) & A_{(1,2)} & \longrightarrow aS_{(2,2)} & & \\
(4) & B_{(1,2)} & \longrightarrow aS_{(1,2)} & & \\
(4) & C_{(1,2)} & \longrightarrow aS_{(1,2)} & &
\end{array}
$$

Figure 50: Resulting production rules for the merged constraints response(A,B) and precedence(A,C).

task A is allowed to be specified in a single init(A) constraint. The specification of two init constraints regarding the same task (A) is not allowed, i.e. would result in a redundant specification. The join algorithm uses complex indexes in the output grammar indicating the symbol version of both input grammars similar to the merging algorithm. However, the join algorithm is required to simultaneously activate multiple tasks at the very beginning. That means, synchronization is required afterwards. The synchronization requirements need to be newly calculated as the output grammar needs to synchronize over all tasks at the very beginning whereas the input grammars "only" synchronize over the tasks that are part of the particular input grammar. In figure 51 the lines (7–18) calculate the production rules specifying the synchronization of tasks in the output grammar. Each synchronization rule of the first input grammar is combined with each synchronization rule of the second input grammar, i.e. the cross product of synchronization rules is calculated. The concrete tasks that are required to be activated at the very beginning of the output grammar are given by the union of tasks that are activated at the very beginning of both input grammars. Line (20–26) in figure 51 calculates all symbols for tasks that need to be activated at the very beginning as well as symbols required for synchronization in order to specify the production rule(s) for the very beginning of the output grammar. Finally, rules specifying the service calls for the tasks are calculated in the lines (28–31).

Figure 52 shows the joining algorithm for a single non-empty grammar with the empty grammar. In particular, the joining algorithm in figure 51 is applicable only if both grammars that are intended to be joined specify an init constraint. In case only one grammar specifies the init constraint when the combination of two grammars needs to be calculated requires the application of the algorithm in figure 52 instead of the algorithm in figure 51. In detail, the algorithm in figure 52 doesn't need to correlate

Require: $G_1 = (V_1, \Sigma_1, P_1, S_u) \wedge G_2 = (V_2, \Sigma_2, P_2, S_v)$
Ensure: $G_3 = join(G_1, G_2) = (V_3, \Sigma_3, P_3, S_{(u,v)})$
$\exists S_{(u,v)} \in V_3$
for all $(S_u, \alpha_1) \in P_1$ **do**
  for all $(S_v, \alpha_2) \in P_2$ **do**
    $Tmp_V = \{X_i | (X_i \in \alpha_1 \vee X_i \in \alpha_2) \wedge \exists (X_i, x) \in P_1 \cup P_2 \wedge x \in \Sigma_1 \cup \Sigma_2\}$
    $Tmp2_V = \{X_{(i,j)} | X_i \in Tmp_V \wedge X_j \in Tmp_V \wedge i \neq j\} \cup \{X_{(i,i)} | X_i \in Tmp_V \wedge \nexists X_j \in Tmp_V \wedge i \neq j\}$
    $\exists X_{(i,j)} \in V_3 \; \forall X_{(i,j)} \in Tmp2_V$
    // Closing rules (synchronizing)
    for all $H_k \in \alpha_1 \wedge H_k \notin Tmp_V$ **do**
      for all $(\gamma H_k \delta, \gamma Y_m \delta) \in P_1$ **do**
        for all $H_l \in \alpha_2 \wedge H_l \notin Tmp_V$ **do**
          for all $(\gamma' H_l \delta', \gamma' Y_n \delta') \in P_2$ **do**
            $\exists (\gamma \sqcup \gamma' H_{(k,l)} \delta \sqcup \delta', \gamma \sqcup \gamma' Y_{(m,n)} \delta \sqcup \delta') \in P_3$
            $\exists H_{(k,l)} \in V_3 \wedge \exists Y_{(m,n)} \in V_3$
            $\exists H_{(k,l)} \in Tmp_H$
            $\exists Y_{(m,n)} \in \Omega$
          end for
        end for
      end for
    end for
    // Entry rules
    $\exists (S_{(u,v)}, \alpha_3) \in P_3$ with:
    for all $X_{(i,j)} \in Tmp2_V$ **do**
      $\exists X_{(i,j)} \in \alpha_3$
    end for
    for all $X_{(i,j)} \in Tmp_H$ **do**
      $\exists X_{(i,j)} \in \alpha_3$
    end for
    // Intermediate rules
    for all $X_{(i,j)} \in Tmp2_V \wedge (X_i, x) \in P_1 \cup P_2 \wedge x \in \Sigma_1 \cup \Sigma_2$ **do**
      $\exists (X_{(i,j)}, x) \in P_3$
      $\exists x \in \Sigma_3$
    end for
  end for
end for
return $((V_3, \Sigma_3, P_3, S_{(u,v)}), \Omega)$

Figure 51: Joining algorithm for grammars specifying the init-constraint, i.e. calculating the union of contained production rules.

production rules of both input grammars. Instead, only the indexes of non-terminals need to be substituted by complex indexes.

Require: $G_1 = (V_1, \Sigma_1, P_1, S_u) \wedge m \wedge n$
Ensure: $(G_3, \Omega) = join(G_1, m, n) = ((V_3, \Sigma_3, P_3, S_{(u,u)}), \Omega)$
$V_3 = \{X_{(i,i)} \mid X_i \in V_1\}$
$\Sigma_3 = \Sigma_1$
$\exists(\alpha', \beta') \in P_3$ with:
for all $(\alpha, \beta) \in P_1$ **do**
  for all $X_i \in \alpha \wedge X_i \in V_1$ **do**
    $\exists X_{(i,i)} \in \alpha'$
  end for
  for all $x \in \alpha \wedge x \in \Sigma_1$ **do**
    $\exists x \in \alpha'$
  end for
  for all $X_i \in \beta \wedge X_i \in V_1$ **do**
    if $X = S$ **then**
      if $m = 0 \wedge n \neq 0$ **then**
        $\exists S_{(i,n)} \in \beta'$
        $\exists S_{(i,n)} \in \Omega$
      else if $m \neq 0 \wedge n = 0$ **then**
        $\exists S_{(m,i)} \in \beta'$
        $\exists S_{m,i)} \in \Omega$
      end if
    else
      $\exists X_{(i,i)} \in \beta'$
    end if
  end for
  for all $x \in \beta \wedge x \in \Sigma_1$ **do**
    $\exists x \in \beta'$
  end for
end for

Figure 52: Joining a non-empty grammar with an empty grammar.

Index Transformation Note that the resulting grammar of the merging algorithm needs to be prepared before the grammar can create an input for the algorithm. In particular, the complex index of non-terminals needs to be transferred to a single number. The approach at hand recommends to transfer complex indexes specifying the same number in each part to the particular number, e.g. $S_{(1,1)}$ should be transferred to S_1

and $A_{(2,2)}$ should be transferred to A_2. Indexes specifying different numbers in each part should be transferred to indexes that are successively numbered.

4 References

[1] Katharina Görlach, Frank Leymann, and Volker Claus. Unified execution of service compositions. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, Koloa, HI, USA, December 16-18, 2013*, pages 162–167. IEEE, 2013.

[2] OASIS. *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*. Organization for the Advancement of Structured Information Standards, 2007.

[3] Thomas M. Oinn, R. Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole A. Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip W. Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[4] Maja Pesic. *Constraint-Based Workflow Management Systems: Shifting Control to Users.* PhD thesis, Technische Universiteit Eindhoven, 2008.

[5] W3C. *XSL Transformations (XSLT) Version 1.0.* World Wide Web Consortium (W3C), 1999.

[6] W3C. *Web Services Description Language (WSDL) 1.1.* World Wide Web Consortium (W3C), 2001.

[7] W3C. *Web Services Addressing (WS-Addressing).* World Wide Web Consortium (W3C), 2004.

[8] Matthias Wieland, Katharina Görlach, David Schumm, and Frank Leymann. Towards reference passing in web service and workflow-based applications. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2009, 1-4 September 2009, Auckland, New Zealand*, pages 109–118. IEEE Computer Society, 2009.

A Files

A.1 Calculator.wsdl

```
</wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:ns="http://test.de"
                  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  targetNamespace="http://test.de">

    <wsdl:documentation>Calculator.wsdl</wsdl:documentation>

    <wsdl:types>
        <xs:schema attributeFormDefault="qualified"
                  elementFormDefault="qualified"
                  targetNamespace="http://test.de">
            <xs:element name="add">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="x" type="xs:int"/>
                        <xs:element minOccurs="0" name="y" type="xs:int"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="addResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="return"
                                    type="xs:int"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </wsdl:types>

    <wsdl:message name="addRequest">
        <wsdl:part name="parameters" element="ns:add"/>
    </wsdl:message>
    <wsdl:message name="addResponse">
        <wsdl:part name="parameters" element="ns:addResponse"/>
    </wsdl:message>

    <wsdl:portType name="CalculatorPT">
        <wsdl:operation name="add">
            <wsdl:input message="ns:addRequest"
                        wsaw:Action="urn:add"/>
            <wsdl:output message="ns:addResponse"
                        wsaw:Action="urn:addResponse"/>
```

```
        </wsdl:operation>
        <wsdl:operation name="add2">
            <wsdl:input message="ns:addRequest"
                        wsaw:Action="urn:add2"/>
        </wsdl:operation>
    </wsdl:portType>

   <wsdl:binding name="CalculatorSOAPbinding" type="ns:CalculatorPT">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                        style="document"/>
        <wsdl:operation name="add">
            <soap:operation soapAction="urn:add" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="Calculator">
        <wsdl:port name="CalculatorHttpSOAPEndpoint"
                    binding="CalculatorSOAPBinding">
            <soap:address location=
            "http://localhost:9763/services/Calculator.CalculatorHttpSoapEndpoint/"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

A.2 ExampleDF.scufl2.xml


```
<tavernaResearchObject>
    <workflows>
        <workflow>
            <name>myWorkflow</name>
            <inputWorkflowPorts/>
            <outputWorkflowPorts/>
            <processors>
                <processor>
                    <name>S1</name>
                    <configurableProperties/>
                    <dispatchStack/>
                    <inputProcessorPorts/>
                    <outputProcessorPorts>
                        <outputProcessorPort>
                            <name>S1_out</name>
                            <depth>0</depth>
                            <configurableProperties/>
                        </outputProcessorPort>
                    </outputProcessorPorts>
                    <startConditions/>
                 </processor>
                <processor>
                    <name>S2</name>
                    <configurableProperties/>
                    <dispatchStack/>
                    <inputProcessorPorts/>
                    <outputProcessorPorts/>
                    <startConditions/>
                 </processor>
                <processor>
                    <name>S3</name>
                    <configurableProperties/>
                    <dispatchStack/>
                    <inputProcessorPorts>
                        <inputProcessorPort>
                            <name>S3_in</name>
                            <depth>0</depth>
                            <configurableProperties/>
                        </inputProcessorPort>
                    </inputProcessorPorts>
                    <outputProcessorPorts/>
                    <startConditions/>
                 </processor>
            </processors>
            <datalinks>
                <datalink>
                    <senderPortReference>
                        <identification>
```

```
          workflow/myWorkflow/processor/S1/outputprocessorport/S1_out
                      </identification>
                  </senderPortReference>
                  <receiverPortReference>
                      <identification>
          workflow/myWorkflow/processor/S3/inputprocessorport/S3_in
                      </identification>
                  </receiverPortReference>
              </datalink>
           </datalinks>
           <configurableProperties/>
        </workflow>
    </workflows>
</tavernaResearchObject>
```

B Non-Terminal Specifications

```
<nonTerminal>
    <name> M_1 </name>
    <type> Event1 </type>
    <parameters />
    <relations>
        <relation>
            <outputValue>
                True </outputValue>
            <nonTerminalRef>
                T_1 </nonTerminalRef>
        </relation>
        <relation>
            <outputValue>
                False </outputValue>
            <nonTerminalRef>
                F_1 </nonTerminalRef>
        </relation>
    </relations>
</nonTerminal>
```

```
<nonTerminal>
    <name> M_3 </name>
    <type> Event1 </type>
    <parameters />
    <relations>
        <relation>
            <outputValue>
                True </outputValue>
            <nonTerminalRef>
                T_3 </nonTerminalRef>
        </relation>
        <relation>
            <outputValue>
                False </outputValue>
            <nonTerminalRef>
                F_3 </nonTerminalRef>
        </relation>
    </relations>
</nonTerminal>
```

(a) Non-terminals for checking the occurrence of an event (see figure 54 for the non-terminal type).

```
<nonTerminal>
    <name> G_1 </name>
    <type> GetEvent_1 </type>
    <parameters>
        <output>
            <reference>
                myEvent
            </reference>
        </output>
    </parameters>
</nonTerminal>
```

(b) Non-terminal for receiving an event (see figure 54 for the non-terminal type).

```
<nonTerminal>
    <name> U_1 </name>
    <type> CancelEvent_1 </type>
    <parameters />
</nonTerminal>
```

(c) Non-terminal for setting the point in time where following events are rejected (see figure 54 for the non-terminal type).

Figure 53: Non-Terminals for the message event handlers (cf. unified model for event handlers in figure 19).

```
<nonTerminalType name="Event_1">
    <service>CompositionUtils</service>
    <operation>  event1  </operation>
    <port>CompositionUtilsHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
    http://localhost:9763/services/CompositionUtils.CompositionUtilsHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
</nonTerminalType>

<nonTerminalType name="GetEvent_1">
    <service>CompositionUtils</service>
    <operation>  event1op  </operation>
    <port>CompositionUtilsHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
    http://localhost:9763/services/CompositionUtils.CompositionUtilsHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
</nonTerminalType>

<nonTerminalType name="CancelEvent1">
    <service>CompositionUtils</service>
    <operation>   cancelEvent1  </operation>
    <port>CompositionUtilsHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
    http://localhost:9763/services/CompositionUtils.CompositionUtilsHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
</nonTerminalType>
```

Figure 54: Types of the non-terminals in figure 53.

```
<nonTerminal>
    <name> M_2 </name>
    <type> Alarm1 </type>
    <parameters />
    <relations>
        <relation>
            <outputValue>
                True </outputValue>
            <nonTerminalRef>
                T_2 </nonTerminalRef>
        </relation>
        <relation>
            <outputValue>
                False </outputValue>
            <nonTerminalRef>
                F_2 </nonTerminalRef>
        </relation>
    </relations>
</nonTerminal>
```

```
<nonTerminal>
    <name> M_4 </name>
    <type> Alarm1 </type>
    <parameters />
    <relations>
        <relation>
            <outputValue>
                True </outputValue>
            <nonTerminalRef>
                T_4 </nonTerminalRef>
        </relation>
        <relation>
            <outputValue>
                False </outputValue>
            <nonTerminalRef>
                F_4 </nonTerminalRef>
        </relation>
    </relations>
</nonTerminal>
```

(a) Non-terminals for checking the occurrence of an alarm (see figure 56 for the non-terminal type).

```
<nonTerminal>
    <name> E_1 </name>
    <type> ExpressionEvaluator </type>
    <parameters>
        <input>
            <reference position="1">
                duration
            </reference>
            <value position="2">
                duration
            </value>
        </input>
        <output>
            <reference>
                myAlarm
            </reference>
        </output>
    </parameters>
</nonTerminal>
```

(b) Non-terminal for evaluation of an alarm expression.

```
<nonTerminal>
    <name> A_1 </name>
    <type> AlarmService_For </type>
    <parameters>
        <input>
            <reference>
                myAlarm
            </reference>
        </input>
    </parameters>
</nonTerminal>
```

(c) Non-terminal A_1 for starting the alarm service.

```
<nonTerminal>
    <name> U_2 </name>
    <type> CancelAlarm_1 </type>
    <parameters />
</nonTerminal>
```

(d) Non-terminal for setting the point in time where following alarms are rejected (see figure 56 for the non-terminal type).

Figure 55: Non-Terminals for alarm event handlers (cf. unified model for event handlers in figure 19).

```
<nonTerminalType name="Alarm_1">
    <service>CompositionUtils</service>
    <operation>  alarm1  </operation>
    <port>CompositionUtilsHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
    http://localhost:9763/services/CompositionUtils.CompositionUtilsHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
</nonTerminalType>

<nonTerminalType name="CancelAlarm1">
    <service>CompositionUtils</service>
    <operation>   cancelAlarm1  </operation>
    <port>CompositionUtilsHttpSOAPEndpoint</port>
    <wsa:EndpointReference>
        <wsa:Address>
    http://localhost:9763/services/CompositionUtils.CompositionUtilsHttpSoapEndpoint/
        </wsa:Address>
    </wsa:EndpointReference>
</nonTerminalType>
```

Figure 56: Types of the non-terminals in figure 55.