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

Katharina Görlach

Report 2013/01

(Version 2.0)
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;
- \( \Sigma \) 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 $\Sigma$ 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 $\varepsilon$.

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

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

**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:

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

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

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

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

$$\alpha \sqcup \beta = (\alpha_1, \alpha_2, \ldots, \alpha_m) \sqcup (\beta_1, \beta_2, \ldots, \beta_n) := \alpha \oplus \beta \ominus (\alpha_1, \alpha_2, \ldots, \alpha_k) \land$$
\[(\alpha_i \in \alpha \land \alpha_i \in \beta) \land (\alpha_j \in \alpha \land \alpha_i \in \beta) \land \ldots \land (\alpha_k \in \alpha \land \alpha_k \in \beta), 1 \leq i \leq j \leq k \leq n\]

### Definition 7 (Multiset)

A multiset \(M\) over a set \(A\) is a function \(M : A \rightarrow \mathbb{N}_0\) represented by a finite collection \(M = [x_1, x_2, x_3, \ldots, x_n]\) 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 empty multiset is denoted \(\varepsilon\).

### Definition 8 (Multiset Union \(\oplus\))

The union of two multisets \(W = [x_1, x_2, \ldots, x_n]\) and \(V = [y_1, y_2, \ldots, y_m]\) is defined by:

\[W \oplus V = [x_1, x_2, \ldots, x_n] \oplus [y_1, y_2, \ldots, y_m] := [x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m]\]

### Definition 9 (Multiset Subtraction \(\ominus\))

A multiset \(W\) over a set \(A\) minus a multiset \(V\) can be calculated if the multiset \(V\) is included in the multiset \(W\) and is defined by:

\[W \ominus V := U \iff \exists U : B \rightarrow \mathbb{N}_0 \land W = V \oplus U\]

### Definition 10 (Operation multiset \(\cap\) set)

Assuming a multiset \(M = [x_1, x_2, \ldots, x_n]\) and a set of symbols \(\Phi\). Then the operation \(M \cap \Phi\) calculates the partial multiset \(M_\Phi : \Phi \rightarrow \mathbb{N}_0\) containing all symbols of \(M\) that are part of \(\Phi\):

\[M \cap \Phi = M_\Phi(x_i) := \begin{cases} 0, & x_i \in \Phi \land x_i \notin M; \\ M(x_i), & x_i \in \Phi \land x_i \in M. \end{cases}\]

### Definition 11 (Multiset Union \(\sqcup\))

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

\[U(x_i) := \begin{cases} M(x_i), & x_i \in A \land ((x_i \in B \land M(x_i) \geq N(x_i)) \lor x_i \notin B); \\ N(x_i), & x_i \in B \land ((x_i \in A \land M(x_i) < N(x_i)) \lor x_i \notin A). \end{cases}\]
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;
- \(d_2 = (\text{Type}, (\text{Input}, \text{Output}))\) specifies the associated service operation (Type) including input and output parameters where Input and Output may be empty,
- \(d_3 : \text{String} \rightarrow V\) 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 $m(w)$ of a word $w = (x_1, x_2, ..., x_n)$ over an alphabet $\Sigma$ is defined by the multiset $M : \Sigma \rightarrow \mathbb{N}_0$ with:

$$m(w) = [x_1, x_2, ..., x_n]$$

**Definition 14 (c-Interpretation of Production Rules)**

The $c$-interpretation $i_c$ of a production rule $p = (\alpha, \beta)$ specifying the words $\alpha, \beta$ results in a $c$-interpreted production rule $p_c = (m(\alpha), m(\beta))$ with the multiset-interpretation $m$.

$$i_c : \text{Words} \times \text{Words} \rightarrow \text{Multisets} \times \text{Multisets}$$

$$i_c((\alpha, \beta)) = (m(\alpha), m(\beta))$$

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.

\[
\forall x, y \in P \land N \in V \land N \in LHS(x) \land N \in LHS(y) \land LHS(x) \setminus \{N\} \neq LHS(y) \setminus \{N\} \neq \emptyset \\
\forall w_i \in \sigma = S \rightarrow w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n \land LHS(x) \in w_i \land LHS(y) \in w_i
\]

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 \( \text{lhs} \) 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
   \[
   p \in P \land N \in LHS(p) \land LHS(p) \setminus \{N\} \neq \emptyset \Rightarrow \forall q \in P \exists \alpha \in (\Sigma \cup V)^+ \land N \in LHS(q) \land \alpha \in LHS(q)
   \]

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

Property (1) in definition 17 ensures the deterministic selection of production rules with different \( \text{lhs} \) 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 \( \text{lhs} \) 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 $N$:

$$\forall r \in \{ p \in P \mid \exists q \in P \land q \neq p \land N \in V \land LHS(q) = LHS(p) \land N \in LHS(p) \land N \in LHS(q) \} :$$

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

$\Sigma$ - 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 \( \Gamma \) following the rules in \( P \) is defined by the function \( \text{substitute}_P : V^* \Sigma^* \rightarrow V^* \Sigma^* \) with:

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

with:

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

**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 \( \text{ruleSelection}_P : \Sigma^* \times V \rightarrow (\Sigma^* V \times (\Sigma \cup V)^*) \) with:

\[
\text{ruleSelection}(\alpha, N) = p \in P : \\
\begin{align*}
N \in \text{LHS}(p) & \land (\text{LHS}(p) \cap \Sigma) \in \alpha, & 1\text{-dimensional } N; \\
N \in \text{LHS}(p) & \land (\text{LHS}(p) \cap \Sigma) \in \alpha \land \text{invoke}(\text{dim}_2(N)), & 2\text{-dimensional } N; \\
N \in \text{LHS}(p) & \land (\text{LHS}(p) \cap \Sigma) \in \alpha \land M \in \text{RHS}(p) \\
& \land \alpha = \text{invoke}(\text{dim}_2(N)) \land \text{dim}_3(N, \alpha) = M, & 3\text{-dimensional } N.
\end{align*}
\]

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 \textit{lhs} and before writing the \textit{rhs}, service invocation might be required if the automaton’s current processing symbol is a two- or three-dimensional non-terminal. In detail:

\textbf{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.

<table>
<thead>
<tr>
<th>Type of Non-Terminals</th>
<th>Complexity of Associated Non-Terminals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helpers</td>
<td>1-dimensional</td>
<td>Non-Terminals representing navigator-related information without association to an external service.</td>
</tr>
<tr>
<td>ExpressionEvaluator</td>
<td>2- or 3-dimensional</td>
<td>A (synchronous) web service operation that is responsible for expression evaluation, e.g. for expression-based data assignments or conditional control flow.</td>
</tr>
<tr>
<td>InsertData</td>
<td>2-dimensional</td>
<td>A (synchronous) web service operation of a reference resolution system that allows the creation of a data reference (cf. [8]).</td>
</tr>
<tr>
<td>GetData</td>
<td>2-dimensional</td>
<td>A (synchronous) web service operation of a reference resolution system that returns data related to a reference (cf. [8]).</td>
</tr>
<tr>
<td>Type of Non-Terminals</td>
<td>Complexity of Associated Non-Terminals</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>UpdateData</td>
<td>2-dimensional</td>
<td>A (synchronous) web service operation of a reference resolution system that allows to change the data related to a reference (cf. [8]).</td>
</tr>
<tr>
<td>DeleteData</td>
<td>2-dimensional</td>
<td>A (synchronous) web service operation of a reference resolution system that allows the deletion of a data reference (cf. [8]).</td>
</tr>
<tr>
<td>AlarmService_For</td>
<td>2-dimensional</td>
<td>An (asynchronous) operation of a web service providing alarm events that are determined by for expressions (cf. section 3.2.6).</td>
</tr>
<tr>
<td>EventX</td>
<td>3-dimensional</td>
<td>A (synchronous) web service operation that provides information about the occurrence of event X to the service composition (cf. section 3.2.6).</td>
</tr>
<tr>
<td>GetEventX</td>
<td>2-dimensional</td>
<td>A (synchronous) web service operation that provides an occurring event X to the service composition (cf. section 3.2.6).</td>
</tr>
<tr>
<td>AlarmX</td>
<td>3-dimensional</td>
<td>A (synchronous) web service operation that provides information about the occurrence of an alarm X to the service composition (cf. section 3.2.6).</td>
</tr>
<tr>
<td>FaultX</td>
<td>3-dimensional</td>
<td>A (synchronous) web service operation that provides information about the occurrence of a fault X to the service composition (cf. section 3.2.6).</td>
</tr>
<tr>
<td>GetFaultX</td>
<td>2-dimensional</td>
<td>A (synchronous) web service operation that provides an occurring fault X to the service composition (cf. section 3.2.6).</td>
</tr>
<tr>
<td>(Services)</td>
<td>(2-dimensional)</td>
<td>(An abstract type representing associated services that are composed by a service composition.)</td>
</tr>
<tr>
<td>Calculator</td>
<td>2-dimensional</td>
<td>A synchronous web service operation for calculations with numbers, e.g. operation add is provided for adding two numbers.</td>
</tr>
<tr>
<td>Calculatorin</td>
<td>2-dimensional</td>
<td>An asynchronous web service operation for calculating with numbers.</td>
</tr>
<tr>
<td>Calculatorest</td>
<td>2-dimensional</td>
<td>Callback of the asynchronous web service operation for calculating with numbers.</td>
</tr>
</tbody>
</table>
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 CalculatorOut 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
(a) WSDL description of the service composition that synchronously calls the calculator service (Composition.wsdl).

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

Figure 1: Original service composition model specifying a synchronous call of the calculator service.
(a) Non-Terminal $S_1$ representing a synchronous service call.

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

(c) Non-terminal type Calculator

Figure 2: Unified service composition model specifying a synchronous call of the calculator service.
(a) Extension to the WSDL description of the service composition presented in figure 1 for enabling an asynchronously call of the calculator service (Composition.wsdl).

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

Figure 3: Original service composition model specifying an asynchronous call of the calculator service.
(a) Non-Terminal $S_1$ representing an asynchronous service call and non-terminal $S_2$ representing the receiving of the response.

(1) $\text{Start} \rightarrow S_1 \quad \text{with:} \quad S_1 \leftarrow \text{Calculator}_{in}$

(2) $S_1 \rightarrow s_1H \quad \quad \quad S_2 \leftarrow \text{Calculator}_{out}$

(3) $H \rightarrow h_1S_2 \quad \quad \quad H \leftarrow \text{Helpers}$

(4) $S_2 \rightarrow s_2 \quad \quad \quad S_1, H \in V$

\[ s_i \in \Sigma \]

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

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 $S_2$.

$$
\begin{align*}
(1) & \quad \text{Start} \rightarrow S_1 \\
(2) & \quad S_1 \rightarrow s_1S_2 \\
(3) & \quad S_2 \rightarrow s_2S_3 \\
(4) & \quad S_3 \rightarrow s_3
\end{align*}
$$

with: $S_i \leftarrow \text{Services}$

- $S_i \in V$
- $s_i \in \Sigma$

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

(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.

\[
\begin{align*}
(1) & \quad \text{Start} \rightarrow S_1 \quad \text{with: } S_i \in \text{Services} \\
(2) & \quad S_1 \rightarrow s_1S_2 \quad s_i \in \Sigma \\
(3) & \quad S_1 \rightarrow s_1S_3 \\
(4) & \quad S_2 \rightarrow s_2 \\
(5) & \quad S_3 \rightarrow s_3
\end{align*}
\]

**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 $S_3$. Additionally, a 1-dimensional non-terminal $H_2$ representing the need for synchronization is activated. Rule (5) realizes

\[\text{The order of symbols is specified w.l.o.g. in the c-interpreted production rules.}\]
(a) BPEL-based if activity specifying condition control flow concerning the alternative service calls $S_2$ and $S_3$.

(1) $Start \rightarrow C_1$ with $S_i \leftarrow Services$
(2) $C_1 \rightarrow T_1$ $C_i \leftarrow ExpressionEvaluator$
(3) $C_1 \rightarrow F_1$ $T_1, F_1 \leftarrow Helpers$
(4) $T_1 \rightarrow S_2$ $S_i, C_1, T_1, F_1 \in V$
(5) $F_1 \rightarrow S_3$ $s_i \in \Sigma$
(6) $S_2 \rightarrow s_2$
(7) $S_3 \rightarrow s_3$

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

(c) Non-terminal $C_1$ representing condition evaluation.

Figure 6: Unified model for conditional control flow.
(a) BPEL-based while activity specifying the successive service calls $S_1$ and $S_2$ in the loop body and a successive service call $S_3$.

(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.

(1) \( \text{Start} \rightarrow S_1S_2S_3H_2 \)
(2) \( S_1 \rightarrow s_1 \)
(3) \( S_2 \rightarrow s_2 \)
(4) \( S_3 \rightarrow s_3 \)
(5) \( s_1s_2s_3H_2 \rightarrow s_1s_2s_3S_4 \)
(6) \( S_4 \rightarrow s_4 \)

(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):

(2') \( S_1 \rightarrow s_1S_x \)
(5') \( s_1s_2s_3s_x \rightarrow s_2s_3s_xS_4 \)
(7) \( S_x \rightarrow s_x \)

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
Figure 9: BPEL-based flow activity specifying links with explicit transition and join condition.
Production rules realizing an internal evaluation of a join condition and an external evaluation of a transition condition.

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

(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>
    </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 $S_5, S_6, \text{ and } S_7$ as well as explicit transition conditions.

1. $Start \rightarrow S_5J_6H_7$ with:
   $S_i \leftarrow \text{Services}$
   $C_i \leftarrow \text{ExpressionEvaluator}$
2. $S_5 \rightarrow s_5C_5$
3. $C_5 \rightarrow C_5$
4. $C_5 \rightarrow F_5$
5. $T_5 \rightarrow t_5$
6. $F_5 \rightarrow f_5$
7. $t_5J_6 \rightarrow S_6J_7$
8. $F_6 \rightarrow f_6J_7$
9. $S_6 \rightarrow s_6C_6$
10. $C_6 \rightarrow T_6$
11. $C_6 \rightarrow T_6$
12. $T_6 \rightarrow t_6$
13. $F_6 \rightarrow f_6$
14. $t_6J_7 \rightarrow S_7$
15. $F_7 \rightarrow f_7J_7$
16. $S_7 \rightarrow s_7J_7$
17. $j_7H_7 \rightarrow ...$

(b) Production Rules illustrating Dead-Path Elimination in a link-based sequence of service calls $S_5, S_6, \text{ and } S_7$ 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 \texttt{InsertData}, \texttt{GetData}, \texttt{UpdateData}, and \texttt{DeleteData} are associated with the data management service but address different operations. However, the output of non-terminals of type \texttt{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 \texttt{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.

\textbf{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 \texttt{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 \texttt{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 \texttt{DeleteData} representing the deallocation of the storage needs to be introduced for each variable. The non-terminals of type \texttt{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 \texttt{ExpressionEvaluator} that is processed immediately after the storage allocation, i.e the non-terminal of type \texttt{insertData}. However, the initialization by use of expressions is equivalent to data assignments using expressions.

\textbf{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. \texttt{D_1}) of type \texttt{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-
(a) BPEL-based variable declaration and initialization with constants.

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

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

(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.

(a) BPEL specification of an assignment of constants.

(b) Production rules for the assignment of constants.

(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.

(b) Production rules for assignments using expressions.

```
(1) Start → E_3
(2) E_3 → ...
with: E_3 ← ExpressionEvaluator
      E_3 ∈ V
```

(c) NonTerminals for assignments 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 be applied for changing
BPEL specification of an assignment of a variable part.

Production rules for the assignment of a variable part.

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 used 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 scull workflow with data flow from service call $S_1$ to service call $S_3$.

(1) $\text{Start} \rightarrow S_1 S_2$
(2) $S_1 \rightarrow s_1 S_3$
(3) $S_2 \rightarrow s_2$
(4) $S_3 \rightarrow s_3$

with $S_i \leftarrow \text{Services}$
$S_i \in V$
$s_i \in \Sigma$  

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

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

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

Event Handlers Event handlers are executed concurrently to the regular logic of the corresponding scope. Figure 18 shows a BPEL scope $R_i$ 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.
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 rhs 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).

(1) $\text{Start} \rightarrow D_1 t_1 x_1 R_1$ with: $D_1 \leftarrow \text{InsertData}$
(2) $D_1 \rightarrow S_1$ $R_1 \leftarrow \text{Helpers}$
(3) $\ldots t_1 x_1 \ldots \rightarrow \ldots f_1 \ldots$ $S_1 \leftarrow \text{Services}$
(4) $t_1 S_1 \rightarrow t_1 s_1 S_2$ $D_2 \leftarrow \text{DeleteData}$
(5) $f_1 S_1 \rightarrow f_1$
(6) $k_1 S_1 \rightarrow k_1$
(7) $t_1 x_1 S_2 \rightarrow t_1 s_1 b_1$
(8) $f_1 S_2 \rightarrow f_1$
(9) $k_1 S_2 \rightarrow k_1$
(10) $t_1 b_1 R_1 \rightarrow r_1 D_2$
(11) $f_1 R_1 \rightarrow k_1 D_2$
(12) $D_2 \rightarrow \varepsilon$

$D_1, R_1, S_i \in V$
$t_1, f_1, s_1, b_1, r_1, k_i \in \Sigma$

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

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

Figure 21: Unified model for user-defined termination handlers.
(a) BPEL specification of a scope with user-defined fault handlers.

<table>
<thead>
<tr>
<th>Step</th>
<th>Transition</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start → t₁x₁N₂N₃S₁H₁R₁</td>
<td>Fault handler for other faults</td>
</tr>
<tr>
<td>2</td>
<td>t₁S₁ → t₁s₁S₂</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f₁S₁ → f₁</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>q₁S₁ → q₁</td>
<td>Prepare completion of the scope</td>
</tr>
<tr>
<td>5</td>
<td>t₁S₂ → t₁s₂S₃</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>f₁S₂ → f₁</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>q₁S₂ → q₁</td>
<td>Final fault handling</td>
</tr>
<tr>
<td>8</td>
<td>t₁x₁S₃ → t₁s₁b₁</td>
<td>Check for fault F₂</td>
</tr>
<tr>
<td>9</td>
<td>f₁S₃ → f₁</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>q₁S₃ → q₁</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>N₂ → T₂</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>N₂ → F₂</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>t₁x₁F₂ → t₁x₁N₂</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>t₁b₁F₂ → t₁b₁n₂</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>f₁F₂ → f₁n₂</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>t₁x₁T₂ → t₁D₁n₂</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>t₁b₁T₂ → t₁b₁n₂</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>f₁T₂ → f₁n₂</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>D₁ → G₁</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>G₁ → S₄</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>S₁ → s₁D₂</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>D₂ → h₂</td>
<td></td>
</tr>
</tbody>
</table>

(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 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 rethrown to the parent scope of $R_1$. In the following the unified model for the default compensation 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>
  </sequence>
  <invoke name="S_3"/>
</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
with:  
\[ H_i, X_2, R_i, T_i, F_i \leftarrow Helpers \]
\[ N_{3i}, N_5 \leftarrow Fault_{R1} \]
\[ N_3, N_7 \leftarrow Fault_{R2} \]
\[ D_1 \leftarrow InsertData \]
\[ G_2 \leftarrow GetFault_{R2} \]
\[ W_2 \leftarrow SetFault_{R1} \]
\[ D_2 \leftarrow DeleteData \]
\[ S_i \leftarrow Services \]

\[ H_i, X_i, R_i, T_i, F_i, N_i, D_i, G_i, W_i, S_i \in V \]
\[ b_i, d_i, t_i, f_i, x_i, y_i, n_i, r_i, q_i, k_i, c_i, z_i \in \Sigma \]

Figure 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>
      <scope name="R_6">
      </scope>
    </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 Con-Dec [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
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.
Figure 27: Continuation of the unified model in figure 26.
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 $\varepsilon$ 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 $\text{response}(A, B)$ 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 $\text{response}(A, B)$ 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,
- $\text{Tasks} = \{A, B, C\}$ 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 $\text{existence}(A)$. 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 $\epsilon$ on the rhs.

$$
\begin{align*}
(1-3) & \quad S_1 \rightarrow A_1 \mid B_1 \mid C_1 \\
(4) & \quad A_1 \rightarrow aS_2 \\
(5) & \quad B_1 \rightarrow bS_1 \\
(6) & \quad C_1 \rightarrow cS_1 \\
(7-9) & \quad S_2 \rightarrow A_2 \mid B_2 \mid C_2 \\
(10) & \quad A_2 \rightarrow aS_3 \\
(11) & \quad B_2 \rightarrow bS_2 \\
(12) & \quad C_2 \rightarrow cS_2 \\
(13-16) & \quad S_3 \rightarrow A_3 \mid B_3 \mid C_3 \mid \epsilon \\
(17) & \quad A_3 \rightarrow aS_3 \\
(18) & \quad B_3 \rightarrow bS_3 \\
(19) & \quad C_3 \rightarrow cS_3
\end{align*}
$$

Figure 29: Unified model for the constraint $\text{existence}_2(A)$, 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 $\text{absence}(A)$, 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 $\text{existence}_N(A)$ are analogous to the rules in figure 29 specifying the concrete constraint $\text{existence}_2(A)$.

\(^3\)Rules for the generic constraint $\text{absence}_N(A)$ are analogous to the rules in figure 30 specifying the concrete constraint $\text{absence}_2(A)$.\footnote{\textsuperscript{2}Rules for the generic constraint $\text{existence}_N(A)$ are analogous to the rules in figure 29 specifying the concrete constraint $\text{existence}_2(A)$.}
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.

\[
\begin{align*}
(1 - 4) & \quad S_1 & \rightarrow & A_1 | B_1 | C_1 | \varepsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \\
5 & \quad A_1 & \rightarrow & aS_2 \\
(6) & \quad B_1 & \rightarrow & bS_1 \\
(7) & \quad C_1 & \rightarrow & cS_1 \\
(8 - 11) & \quad S_2 & \rightarrow & A_2 | B_2 | C_2 | \varepsilon \\
12 & \quad A_2 & \rightarrow & aS_3 \\
13 & \quad B_2 & \rightarrow & bS_2 \\
14 & \quad C_2 & \rightarrow & cS_2 \\
(15 - 17) & \quad S_3 & \rightarrow & B_3 | C_3 | \varepsilon \\
18 & \quad B_3 & \rightarrow & bS_3 \\
19 & \quad C_3 & \rightarrow & cS_3
\end{align*}
\]

Figure 30: Unified model for the constraint \( \text{absence}_2(A) \), i.e. activity A occurs at most 2 times.

The constraint \( \text{exactly}_N(A) \) 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 \( \text{exactly}_2(A) \). Characteristically, the rules do not allow the finishing of the composition until task A is executed twice. In particular, the symbol \( \varepsilon \) is only allowed to be selected if \( S_3 \) is activated (cf. rule(16)).

The constraint \( \text{init}(A) \) 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) switches to the start symbol version \( S_1 \). Afterwards, the rules (4–7) allow to activate all tasks in \( \text{Tasks} \). 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:

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

\[5\text{The order of symbols is presented w.l.o.g. However, each order of specified symbols is covered by the presented production rule.}\]
Figure 31: Unified model for the constraint $\text{exactly}_2(A)$, i.e. activity $A$ occurs exactly 2 times.

Figure 32: Unified model for the constraint $\text{init}(A)$, i.e. activity $A$ must be the first executed activity in the service composition.
\begin{align*}
(1) \quad & S_0 \rightarrow A_0B_0H_0 \quad \text{with: } A_i, B_i \in Services \\
(2) \quad & A_0 \rightarrow a \quad \quad \quad \quad S_i, H_i \in Helpers \\
(2') \quad & B_0 \rightarrow b \quad \quad \quad \quad a, b, c \in \Sigma \\
(3) \quad & abH_0 \rightarrow abS_1
\end{align*}

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 \textit{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 \textit{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 \textit{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
Figure 33: Unified model for the constraint \( \text{responded existence}(A, B) \), 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 \( \text{response}(A, B) \) 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 \( \text{response} \). 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 \( \text{alternate} \) supplement requires that participating tasks alternate whereas the \( \text{chain} \) supplement requires that executions of participating tasks are next to each other.

Figure 35(b) presents the unified model for the constraint \( \text{alternate response}(A, B) \). 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
(1 – 4) \[ S_1 \rightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon \] with: \( A_i, B_i, C_i \in \text{Services} \) \( S_i \in \text{Helpers} \) \( a, b, c \in \Sigma \)

(5) \[ A_1 \rightarrow aS_2 \]

(6) \[ B_1 \rightarrow bS_3 \]

(7) \[ C_1 \rightarrow cS_1 \]

(8 – 10) \[ S_2 \rightarrow A_2 \mid B_2 \mid C_2 \]

(11) \[ A_2 \rightarrow aS_2 \]

(12) \[ B_2 \rightarrow bS_4 \]

(13) \[ C_2 \rightarrow cS_2 \]

(14 – 16) \[ S_3 \rightarrow A_3 \mid B_3 \mid C_3 \]

(17) \[ A_3 \rightarrow aS_4 \]

(18) \[ B_3 \rightarrow bS_3 \]

(19) \[ C_3 \rightarrow cS_3 \]

(20 – 23) \[ S_4 \rightarrow A_4 \mid B_4 \mid C_4 \mid \varepsilon \]

(24) \[ A_4 \rightarrow aS_4 \]

(25) \[ B_4 \rightarrow bS_4 \]

(26) \[ C_4 \rightarrow cS_1 \]

Figure 34: Unified model for the constraint \( \text{co-existence}(A,B) \), 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 \( \text{chain response}(A,B) \) 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 \( \text{precedence}(A,B) \) 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 \( \text{Tasks excepts B} \) 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 \( \text{Tasks} \) as well as the finishing of the service composition is allowed.

Figure 36(b) presents the unified model for the constraint \( \text{alternate precedence}(A,B) \). 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
Constraint \textit{response}(A,B), i.e. activity B must be executed in future when A is executed at least once.

\begin{align*}
(1 - 4) & \quad S_1 \rightarrow A_i \mid B_i \mid C_i \mid \varepsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \\
(5) & \quad A_i \rightarrow aS_2 \quad \text{with: } S_i \in \text{Helpers} \\
(6) & \quad B_i \rightarrow bS_1 \\
(7) & \quad C_i \rightarrow cS_1
\end{align*}

Constraint \textit{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{align*}
(1 - 4) & \quad S_1 \rightarrow A_i \mid B_i \mid C_i \mid \varepsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \\
(5) & \quad A_i \rightarrow aS_2 \quad \text{with: } S_i \in \text{Helpers} \\
(6) & \quad B_i \rightarrow bS_1 \\
(7) & \quad C_i \rightarrow cS_1 \\
(8 - 9) & \quad S_2 \rightarrow B_2 \mid C_2 \\
(10) & \quad B_2 \rightarrow bS_1 \\
(11) & \quad C_2 \rightarrow cS_2
\end{align*}

Constraint \textit{chain response}(A,B), i.e. activity B has to be executed directly after A.

\begin{align*}
(1 - 4) & \quad S_1 \rightarrow A_i \mid B_i \mid C_i \mid \varepsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \\
(5) & \quad A_i \rightarrow aS_2 \quad \text{with: } S_i \in \text{Helpers} \\
(6) & \quad B_i \rightarrow bS_1 \\
(7) & \quad C_i \rightarrow cS_1 \\
(8) & \quad S_2 \rightarrow B_2 \\
(9) & \quad B_2 \rightarrow bS_1
\end{align*}

Figure 35: Unified model for the response constraints.
(1 − 3) \( S_1 \rightarrow A_1 \mid C_1 \mid \varepsilon \)  
with: \( A_i, B_i, C_i \in \text{Services} \)

(4) \( A_1 \rightarrow aS_2 \)  
(5) \( C_1 \rightarrow cS_1 \)  
\( S_i \in \text{Helpers} \)  
\( a, b, c \in \Sigma \)

(6 − 9) \( S_2 \rightarrow A_2 \mid B_2 \mid C_2 \mid \varepsilon \)

(10) \( A_2 \rightarrow aS_2 \)
(11) \( B_2 \rightarrow bS_1 \)
(12) \( C_2 \rightarrow cS_1 \)

(a) Constraint \textit{precedence}(A,B), i.e. activity A needs to be executed when B begins to execute.

(1 − 3) \( S_1 \rightarrow A_1 \mid C_1 \mid \varepsilon \)  
with: \( A_i, B_i, C_i \in \text{Services} \)

(4) \( A_1 \rightarrow aS_2 \)  
(5) \( C_1 \rightarrow cS_1 \)  
\( S_i \in \text{Helpers} \)  
\( a, b, c \in \Sigma \)

(6 − 8) \( S_2 \rightarrow A_2 \mid B_2 \mid C_2 \)
(9) \( A_2 \rightarrow aS_2 \)
(10) \( B_2 \rightarrow bS_1 \)
(11) \( C_2 \rightarrow cS_1 \)

(b) Constraint \textit{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.

(1 − 3) \( S_1 \rightarrow A_1 \mid C_1 \mid \varepsilon \)  
with: \( A_i, B_i, C_i \in \text{Services} \)

(4) \( A_1 \rightarrow aS_2 \)  
(5) \( C_1 \rightarrow cS_1 \)  
\( S_i \in \text{Helpers} \)  
\( a, b, c \in \Sigma \)

(6 − 9) \( S_2 \rightarrow A_2 \mid B_2 \mid C_2 \mid \varepsilon \)
(10) \( A_2 \rightarrow aS_2 \)
(11) \( B_2 \rightarrow bS_1 \)
(12) \( C_2 \rightarrow cS_1 \)

(c) Constraint \textit{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 $\text{chain precedence}(A,B)$. 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 $\text{succession}(A,B)$ 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 $\text{not responded existence}(A,B)$ 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 $\text{not co-existence}(A,B)$ 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 $\text{not response}(A,B)$ is used to specify that the task $B$ cannot be executed after the execution of task $A$. Similar, the constraint $\text{not precedence}(A,B)$ 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 $\text{not succession}(A,B)$ is also equivalent to the not response and the not precedence constraint.
Constraint *succession*(A,B), i.e. response(A,B) AND precedence(A,B).

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

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

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

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

(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.
(1 - 4) \( S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon \) with: \( A_i, B_i, C_i \in \text{Services} \)

(5) \( A_1 \rightarrow aS_2 \)

(6) \( B_1 \rightarrow bS_3 \)

(7) \( C_1 \rightarrow cS_1 \)

(8 - 10) \( S_2 \rightarrow A_2 | C_2 | \varepsilon \)

(11) \( A_2 \rightarrow aS_2 \)

(12) \( C_2 \rightarrow cS_2 \)

(13 - 15) \( S_3 \rightarrow B_3 | C_3 | \varepsilon \)

(16) \( B_3 \rightarrow bS_3 \)

(17) \( C_3 \rightarrow cS_3 \)

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\((B, A)\).

---

(1 - 4) \( S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon \) with: \( A_i, B_i, C_i \in \text{Services} \)

(5) \( A_1 \rightarrow aS_2 \)

(6) \( B_1 \rightarrow bS_3 \)

(7) \( C_1 \rightarrow cS_1 \)

(8 - 10) \( S_2 \rightarrow A_2 | C_2 | \varepsilon \)

(11) \( A_2 \rightarrow aS_2 \)

(12) \( C_2 \rightarrow cS_2 \)

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 \textit{not chain response} is used to specify that task \( A \) is not allowed to be directly followed by the task \( B \). Equivalently, the constraint \textit{not chain precedence} is used to specify that task \( B \) is not allowed to be directly preceded by the task \( A \). The constraint \textit{not chain succession} the constraints not chain response and not chain precedence.

\begin{align*}
(1 - 4) & \quad S_1 \rightarrow A_1 \mid B_1 \mid C_1 \mid \varepsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \\
(5) & \quad A_1 \rightarrow aS_2 \\
(6) & \quad B_1 \rightarrow bS_1 \\
(7) & \quad C_1 \rightarrow cS_1 \\
(8 - 10) & \quad S_2 \rightarrow A_2 \mid C_2 \mid \varepsilon \\
(11) & \quad A_2 \rightarrow aS_2 \\
(12) & \quad C_2 \rightarrow cS_1
\end{align*}

Figure 40: Unified model for the constraint \textit{not chain response}(\( A,B \)), i.e. \( A \) should never be followed directly by \( B \), the constraint \textit{not chain precedence}(\( A,B \)), i.e. \( B \) should never be preceded directly by \( A \), and the constraint \textit{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 \( \text{Tasks} = \{ A, B, C, D \} \) be the set of all possible tasks. Then, the constraint \textit{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 \textit{1of3}. Actually the constraint \textit{1of3} is an instance of the generic constraint \textit{NofM}. Additionally other instances can be specified, e.g. \textit{1of2}, \textit{1of3}, \textit{1of8}, \textit{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 \textit{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 \textit{NofM}. In detail, the
The algorithm creates a grammar storing the set of non-terminals \( V \), the set of terminals \( \Sigma \), 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: $G = (V, \Sigma, P, S)$
1: $S = \{S_{(0,0,...,0)}\}$
2: $S = S_{(0,0,...,0)}$
3: while $S \neq \emptyset$ do
4: $V = V \cup S$
5: for all $q : S_q \in S$ do
6: $\exists X_{j_q} \in V \forall j \in \{1, 2, ..., M\}$
7: $\exists (S_q, X_{j_q}) \in P \forall j \in \{1, 2, ..., M\}$
8: $\exists C_q \in V$
9: $\exists (S_q, C_q) \in P$
10: $\exists (C_q, cS_q) \in P$
11: end for
12: $T = \{X_{j_q} | \forall q : S_q \in S, \forall j \in \{1, 2, ..., M\}\}$
13: $S = \emptyset$
14: for all $X_{j_q} \in T$ do
15: $i = (r_{1}, r_{2}, ..., r_{M})$
16: if $\text{card}(1 \in i) = N - 1$ then
17: $S_k = S_F$
18: else if $\text{card}(1 \in i) < N - 1$ then
19: $S_k = S_q$ with $q = \begin{cases} (1, r_{2}, ..., r_{M}) & j=1 \\ (r_{1}, 1, ..., r_{M}) & j=2 \\ ... \\ (r_{1}, r_{2}, ..., 1) & j=M \end{cases}$
20: end if
21: $\exists (X_{j_q}, x_jS_k) \in P$
22: if $S_k \neq S_F \land S_k \not\in V$ then
23: $S = S \cup \{S_k\}$
24: end if
25: end for
26: end while
27: $\exists S_F \in V$
28: $\exists X_{j_F} \in V \forall j \in \{1, 2, ..., M\}$
29: $\exists (S_F, X_{j_F}) \in P \forall j \in \{1, 2, ..., M\}$
30: $\exists (S_F, c) \in P$
31: $\exists (X_{j_F}, x_jS_F) \in P \forall j \in \{1, 2, ..., M\}$
32: $\exists (S_F, C_F) \in P$
33: $\exists (C_F, cS_F) \in P$

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.

\[
\begin{align*}
(1 - 5) & \quad S_{(0,0,0,0)} \rightarrow X_{1(0,0,0,0)} \mid X_{2(0,0,0,0)} \mid X_{3(0,0,0,0)} \mid X_{4(0,0,0,0)} \mid C_{(0,0,0,0)} \\
(6) & \quad X_{1(0,0,0,0)} \rightarrow x_1 S_{(1,0,0,0)} \\
(7) & \quad X_{2(0,0,0,0)} \rightarrow x_2 S_{(0,1,0,0)} \\
(8) & \quad X_{3(0,0,0,0)} \rightarrow x_3 S_{(0,0,1,0)} \\
(9) & \quad X_{4(0,0,0,0)} \rightarrow x_4 S_{(0,0,0,1)} \\
(10) & \quad C_{(0,0,0,0)} \rightarrow c S_{(0,0,0,0)} \\
(11 - 15) & \quad S_{(1,0,0,0)} \rightarrow X_{1(1,0,0,0)} \mid X_{2(1,0,0,0)} \mid X_{3(1,0,0,0)} \mid X_{4(1,0,0,0)} \mid C_{(1,0,0,0)} \\
(16) & \quad X_{1(1,0,0,0)} \rightarrow x_1 S_{(1,0,0,0)} \\
(17) & \quad X_{2(1,0,0,0)} \rightarrow x_2 S_F \\
(18) & \quad X_{3(1,0,0,0)} \rightarrow x_3 S_F \\
(19) & \quad X_{4(1,0,0,0)} \rightarrow x_4 S_F \\
(20) & \quad C_{(1,0,0,0)} \rightarrow c S_{(1,0,0,0)} \\
(21 - 25) & \quad S_{(0,1,0,0)} \rightarrow X_{1(0,1,0,0)} \mid X_{2(0,1,0,0)} \mid X_{3(0,1,0,0)} \mid X_{4(0,1,0,0)} \mid C_{(0,1,0,0)} \\
(26) & \quad X_{1(0,1,0,0)} \rightarrow x_1 S_F \\
(27) & \quad X_{2(0,1,0,0)} \rightarrow x_2 S_{(0,1,0,0)} \\
(28) & \quad X_{3(0,1,0,0)} \rightarrow x_3 S_F \\
(29) & \quad X_{4(0,1,0,0)} \rightarrow x_4 S_F \\
(30) & \quad C_{(0,1,0,0)} \rightarrow c S_{(0,1,0,0)} \\
... \\
(51 - 55) & \quad S_F \rightarrow X_{1_F} \mid X_{2_F} \mid X_{3_F} \mid X_{4_F} \mid C_F \mid \varepsilon \\
(56) & \quad X_{1_F} \rightarrow x_1 S_F \\
(57) & \quad X_{2_F} \rightarrow x_2 S_F \\
(58) & \quad X_{3_F} \rightarrow x_3 S_F \\
(59) & \quad X_{4_F} \rightarrow x_4 S_F \\
(60) & \quad C_F \rightarrow c S_F \\
\end{align*}
\]

with:  
\[X_i, C_i \in Services\]  
\[S_i \in Helpers\]  
\[a, b, c \in \Sigma\]

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.

\[
\begin{align*}
(1-4) & \quad S_1 \rightarrow A_1 \mid B_1 \mid C_1 \mid D_1 \quad \text{with:} \quad A_i, B_i, C_i \in \text{Services} \\
(5) & \quad A_1 \rightarrow aS_2 \\
(6) & \quad B_1 \rightarrow bS_3 \\
(7) & \quad C_1 \rightarrow cS_1 \\
(8) & \quad D_1 \rightarrow dS_4 \\
(9-11) & \quad S_2 \rightarrow A_2 \mid C_2 \mid \varepsilon \\
(12) & \quad A_2 \rightarrow aS_2 \\
(13) & \quad C_2 \rightarrow cS_2 \\
(14) & \quad B_3 \rightarrow bS_3 \\
(15) & \quad C_3 \rightarrow cS_3 \\
(9-13) & \quad S_3 \rightarrow B_3 \mid C_3 \mid \varepsilon \\
(14) & \quad D_4 \rightarrow dS_4 \\
(15) & \quad C_4 \rightarrow cS_4
\end{align*}
\]

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 $T$ 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: $G = (V, \Sigma, P, S)$
1: $S = \{S_{(0,0,...,0)}\}$
2: $S = S_{(0,0,...,0)}$
3: while $S \neq \emptyset$ do
4: $V = V \cup S$
5: for all $q : S_q \in S$ do
6: if $\text{card}(1 \in q) = N$ then
7: $\exists(S_q, \epsilon) \in P$
8: for all $j \in \{1, 2, ..., M\}$ do
9: if $q(j) = 1$ then
10: $\exists(S_q, X_{jq}) \in P$
11: end if
12: end for
13: $T = \{X_{jq} | \forall q : S_q \in S \land \forall j \in \{1, 2, ..., M\} : q(j) = 1\}$
14: else
15: $\exists X_{jq} \in V \land \forall j \in \{1, 2, ..., M\}$
16: $\exists(S_q, X_{jq}) \in P \land \forall j \in \{1, 2, ..., M\}$
17: $T = \{X_{jq} | \forall q : S_q \in S \land \forall j \in \{1, 2, ..., M\}\}$
18: end if
19: $\exists C_q \in V$
20: $\exists(S_q, C_q) \in P$
21: $\exists(C_q, cS_q) \in P$
22: end for
23: $S = \emptyset$
24: for all $X_{jq} \in T$ do
25: $i = (r_1, r_2, ..., r_M)$
26: if $\text{card}(1 \in i) = N$ then
27: $S_k = S_i$
28: else if $\text{card}(1 \in i) < N$ then
29: $S_k = S_q$ with $q = \begin{cases} (1, r_2, ..., r_M) & j=1 \\
(r_1, 1, ..., r_M) & j=2 \\
... \\
(r_1, r_2, ..., 1) & j=M 
\end{cases}$
30: end if
31: $\exists(X_{jq}, x_jS_k) \in P$ with
32: if $S_k \notin V$ then
33: $S = S \cup \{S_k\}$
34: end if
35: end for
36: end while
37: 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.
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:

$$V' = V \setminus \{X\} \cup \{X_1, X_2\}$$

$$\Sigma' = \Sigma \setminus \{x\} \cup \{x_1, x_2\}$$

$$P' = P \setminus \{(\alpha, \beta) \mid X \in \alpha \lor X \in \beta\} \cup Q$$

$$Q = \{(\alpha', \beta') \mid \exists (\alpha, \beta) \in P \land (X \in \alpha \lor X \in \beta)\}$$

with:

$$\alpha' = \text{substitute } X \text{ by } X_i \text{ and } x \text{ by } x_i \text{ in } \alpha$$

$$\beta' = \text{substitute } X \text{ by } X_i \text{ and } x \text{ by } x_i \text{ in } \beta$$

$$i \in \{1, 2\}$$
Constraint \(\text{response}(A, (B_1, B_2))\), i.e. both tasks \(B_1\) and \(B_2\) are the target of the dependency specified by the constraint.

\[
\begin{align*}
(1 - 5) & \quad S_1 \rightarrow A_{1_1} \mid B_{1_1} \mid B_{2_1} \mid C_1 \mid \epsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \quad S_i \in \text{Helpers} \\
(6) & \quad A_{1_1} \rightarrow aS_2 \\
(7) & \quad B_{1_1} \rightarrow b_1S_1 \\
(8) & \quad B_{2_1} \rightarrow b_2S_1 \\
(9) & \quad C_1 \rightarrow cS_1 \\
(10 - 13) & \quad S_2 \rightarrow A_{2_1} \mid B_{1_2} \mid B_{2_2} \mid C_2 \\
(14) & \quad A_{2_1} \rightarrow aS_2 \\
(15) & \quad B_{1_2} \rightarrow b_1S_1 \\
(16) & \quad B_{2_2} \rightarrow b_2S_1 \\
(17) & \quad C_2 \rightarrow cS_2
\end{align*}
\]

Constraint \(\text{response}((A_1, A_2), B)\), i.e. both tasks \(A_1\) and \(A_2\) are the source of the dependency specified by the constraint.

\[
\begin{align*}
(1 - 4) & \quad S_1 \rightarrow A_{1_1} \mid A_{2_1} \mid B_{1} \mid C_1 \mid \epsilon \quad \text{with: } A_i, B_i, C_i \in \text{Services} \\
(5) & \quad A_{1_1} \rightarrow a_1S_2 \\
(5) & \quad A_{2_1} \rightarrow a_2S_2 \\
(6) & \quad B_{1} \rightarrow bS_1 \\
(7) & \quad C_1 \rightarrow cS_1 \\
(8 - 10) & \quad S_2 \rightarrow A_{1_2} \mid A_{2_2} \mid B_{2} \mid C_2 \\
(11) & \quad A_{1_2} \rightarrow a_1S_2 \\
(11) & \quad A_{2_2} \rightarrow a_2S_2 \\
(12) & \quad B_{2} \rightarrow bS_1 \\
(13) & \quad C_2 \rightarrow cS_2
\end{align*}
\]

Figure 47: Production rules for branched versions of the constraints \(\text{response}(A, B)\), i.e. for the version \(\text{response}(A, (B_1, B_2))\) and the version \(\text{response}((A_1, A_2), B)\).
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
Figure 48: Combining two grammars, i.e. production rules for combining constraint-based process models.

Require: $G_1 = (V_1, \Sigma_1, P_1, S_u) \land G_2 = (V_2, \Sigma_2, P_2, S_v)$

Ensure: $G_3 = \text{combine}(G_1, G_2) = (V_3, \Sigma_3, P_3, S_{(u,v)})$

1: $S_{init} = S_u$
2: $V_{init_1} = \{X_i \mid X_i \in V_1 \land i = u\}$
3: $\Sigma_{init_1} = \{x \mid x \in \Sigma_1 \land \exists(X_i, \alpha x \beta) \in P_1 \land i = u \land X_i \in V_1 \land \alpha, \beta \in (V_1 \cup \Sigma_1)^*\}$
4: $P_{init_1} = \{(aX_i, \gamma) \mid \forall X_i \in V_1 \land i = u \land \exists(aX_i, \alpha) \in P_1 \land \alpha, \beta \in (V_1 \cup \Sigma_1)^*\}$
5: $S_{init_1} = S_v$
6: $V_{init_2} = \{X_i \mid X_i \in V_2 \land i = v\}$
7: $\Sigma_{init_2} = \{x \mid x \in \Sigma_2 \land \exists(X_i, \alpha x \beta) \in P_2 \land i = v \land X_i \in V_2 \land \alpha, \beta \in (V_2 \cup \Sigma_2)^*\}$
8: $P_{init_2} = \{(aX_i, \gamma) \mid \forall X_i \in V_2 \land i = v \land \exists(aX_i, \alpha) \in P_2 \land \alpha, \beta \in (V_2 \cup \Sigma_2)^*\}$
9: if $u = 0 \land v = 0$ then
10: $(G_4, \Omega) = \text{join}((V_{init_1}, \Sigma_{init_1}, P_{init_1}, S_{init_1}), (V_{init_2}, \Sigma_{init_2}, P_{init_2}, S_{init_2}))$
11: else if $u = 0 \land v \neq 0$ then
12: $(G_4, \Omega) = \text{join}(G_{init_1}, u, v)$
13: else if $u \neq 0 \land v = 0$ then
14: $(G_4, \Omega) = \text{join}(G_{init_2}, u, v)$
15: end if
16: Select $S_{(u',v')} \in \Omega$
17: $G_5 = (V_1 \setminus V_{init_1} \cup \{S_{u'}\}, \Sigma_1, P_1 \setminus P_{init_1}, S_{u'})$
18: $G_6 = (V_2 \setminus V_{init_2} \cup \{S_{v'}\}, \Sigma_2, P_2 \setminus P_{init_2}, S_{v'})$
19: if $u = 0 \land v = 0$ then
20: $G_7 = \text{merge}(G_5, G_6)$
21: else if $u = 0 \land v \neq 0$ then
22: $G_7 = \text{merge}(G_5, G_2)$
23: else if $u \neq 0 \land v = 0$ then
24: $G_7 = \text{merge}(G_1, G_6)$
25: end if
26: $V_3 = V_4 \cup V_7$
27: $\Sigma_3 = \Sigma_4 \cup \Sigma_7$
28: $P_3 = P_4 \cup P_7$
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:** $G_1 = (V_1, \Sigma_1, P_1, S_u) \land G_2 = (V_2, \Sigma_2, P_2, S_v)$

**Ensure:** $G_3 = \text{merge}(G_1, G_2) = (V_3, \Sigma_3, P_3, S_{(u,v)})$

1. $S = \{S_{(u,v)}\}$
2. while $S \neq \emptyset$ do
3.     $S = S \setminus \{S_{(i,j)}\}$
4.     $\exists S_{(i,j)} \in V_3$
5.     for all $(S_i, Y_i) \in P_1 \land (S_j, Y_j) \in P_2$ do
6.         $\exists Y_{(i,j)} \in V_3$
7.         $\exists (S_{(i,j)}, Y_{(i,j)}) \in P_3$
8.     end for
9.     for all $(X_i, xS_k) \in P_1 \land (X_j, xS_l) \in P_2$ with $x \in \Sigma_1 \cup \Sigma_2$ do
10.        $\exists x \in \Sigma_3$
11.        $\exists (X_{(i,j)}, xS_{(k,l)}) \in P_3$
12.        if $S_{(k,l)} \notin V_3$ then
13.            $S = S \cup \{S_{(k,l)}\}$
14.        end if
15.     end for
16. 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
(1−3) \[ S_{(1,1)} \rightarrow A_{(1,1)} \mid B_{(1,1)} \mid \varepsilon \] with: \( A_i, B_i, C_i \in Services \)
\( S_i \in Helpers \)
\( a, b, c \in \Sigma \)

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

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: \( G_1 = (V_1, \Sigma_1, P_1, S_u) \land G_2 = (V_2, \Sigma_2, P_2, S_v) \)
Ensure: \( G_3 = \text{join}(G_1, G_2) = (V_3, \Sigma_3, P_3, S_{u,v}) \)

1: \( \exists S_{(u,v)} \in V_3 \)
2: \( \text{for all } (S_u, \alpha_1) \in P_1 \text{ do} \)
3: \( \text{for all } (S_v, \alpha_2) \in P_2 \text{ do} \)
4: \( \text{Tmp}_{V} = \{X_i | (X_i \in \alpha_1 \lor X_i \in \alpha_2) \land \exists (X_i, x) \in P_1 \cup P_2 \land x \in \Sigma_1 \cup \Sigma_2 \} \)
5: \( \text{Tmp}_{2V} = \{X_{(i,j)} | X_i \in Tmp_V \land X_j \in Tmp_V \land i \neq j \} \cup \{X_{(i,i)} | X_i \in Tmp_V \land \exists X_j \in Tmp_V \land i \neq j \} \)
6: \( \exists X_{(i,j)} \in V_3 \land X_{(i,j)} \in Tmp_{2V} \)
7: \( \text{// Closing rules (synchronizing)} \)
8: \( \text{for all } H_k \in \alpha_1 \land H_k \notin \text{Tmp}_V \text{ do} \)
9: \( \text{for all } H_l \in \alpha_2 \land H_l \notin \text{Tmp}_V \text{ do} \)
10: \( \text{for all } (\gamma H_l \delta, \gamma Y_m \delta) \in P_2 \text{ do} \)
11: \( \exists (\gamma \cup \gamma' H_l \delta', \gamma' Y_m \delta') \in P_3 \)
12: \( \exists H_{(k,l)} \in V_3 \land \exists Y_{(m,n)} \in V_3 \)
13: \( \exists Y_{(m,n)} \in \Omega \)
14: \( \text{end for} \)
15: \( \text{end for} \)
16: \( \text{// Entry rules} \)
17: \( \exists (S_{(u,v)}, \alpha_3) \in P_3 \text{ with:} \)
18: \( \text{for all } X_{(i,j)} \in Tmp_{2V} \text{ do} \)
19: \( \exists X_{(i,j)} \in \alpha_3 \)
20: \( \text{end for} \)
21: \( \text{// Intermediate rules} \)
22: \( \text{for all } X_{(i,j)} \in Tmp_{2V} \land (X_i, x) \in P_1 \cup P_2 \land x \in \Sigma_1 \cup \Sigma_2 \text{ do} \)
23: \( \exists X_{(i,j)}, x \in P_3 \)
24: \( \exists x \in \Sigma_3 \)
25: \( \text{end for} \)
26: \( \text{end for} \)
27: \( \text{return } ((V_3, \Sigma_3, P_3, S_{(u,v)}), \Omega) \)

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:** $G_1 = (V_1, \Sigma_1, P_1, S_u) \land m \land n$

**Ensure:** $(G_3, \Omega) = \text{join}(G_1, m, n) = ((V_3, \Sigma_3, P_3, S_{u,u}), \Omega)$

1. $V_3 = \{X_{(i,i)} \mid X_i \in V_1\}$
2. $\Sigma_3 = \Sigma_1$
3. $\exists(\alpha', \beta') \in P_3$ with:
4. **for all** $(\alpha, \beta) \in P_1$ **do**
5. **for all** $X_i \in \alpha \land X_i \in V_1$ **do**
6. $\exists X_{(i,i)} \in \alpha'$
7. **end for**
8. **for all** $x \in \alpha \land x \in \Sigma_1$ **do**
9. $\exists x \in \alpha'$
10. **end for**
11. **for all** $X_i \in \beta \land X_i \in V_1$ **do**
12. **if** $X = S$ **then**
13. **if** $m = 0 \land n \neq 0$ **then**
14. $\exists S_{(i,n)} \in \beta'$
15. $\exists S_{(i,n)} \in \Omega$
16. **else if** $m \neq 0 \land n = 0$ **then**
17. $\exists S_{(m,i)} \in \beta'$
18. $\exists S_{(m,i)} \in \Omega$
19. **end if**
20. **else**
21. $\exists X_{(i,i)} \in \beta'$
22. **end if**
23. **end for**
24. **for all** $x \in \beta \land x \in \Sigma_1$ **do**
25. $\exists x \in \beta'$
26. **end for**
27. **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


### A Files

#### A.1 Calculator.wsdl

```xml
<wSDL:definitions xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/
xmlns:ns="http://test.de"
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:portType>
</wSDL:definitions>
```
</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>
                  <startConditions/>
               </outputProcessorPorts>
            </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>
         </workflow>
      </workflows>
   </tavernaResearchObject>
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

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

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

(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).
Figure 54: Types of the non-terminals in figure 53.
(a) Non-terminals for checking the occurrence of an alarm (see figure 56 for the non-terminal type).

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

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

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