Institute of Architecture of Application Systems

Unified Execution of Service Compositions

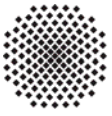
Katharina Görlach[1] and Frank Leymann[1] and Volker Claus[2]

[1]Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{goerlach, leymann}@iaas.uni-stuttgart.de

[2]Institute for Formal Methods in Computer Science,
University of Stuttgart, Germany
claus@informatik.uni-stuttgart.de

BibTeX:

```
@inproceedings{GLC13,
  author    = {Katharina Görlach and Frank Leymann and Volker Claus},
  title     = {Unified Execution of Service Compositions},
  booktitle = {Proceedings of the 6th IEEE International
               Conference on Service Oriented Computing &
               Applications, SOCA 2013,
               16-18 December 2013, Koloa, Hawaii, USA},
  year      = {2013},
  pages     = {162--167},
  publisher = {IEEE Computer Society}
}
```

Universität Stuttgart
Germany

Unified Execution of Service Compositions

Katharina Görlach
Institute of Architecture of Application Systems
University of Stuttgart
goerlach@iaas.uni-stuttgart.de

Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
leymann@iaas.uni-stuttgart.de

Volker Claus
Institute for Formal Methods in Computer Science
University of Stuttgart
claus@informatik.uni-stuttgart.de

***Abstract*—This paper discusses the unification of service composition based on formal specifications. The approach aims for a unified execution of service compositions that can be modeled by various specification languages covering different modeling paradigms. The unification of service composition models is realized based on formal grammars whereas the unification of service composition execution is realized based on formal queued automata. The approach introduces a classification of context-sensitive grammars for determining an optimized automaton class for the execution of service compositions. Finally, a prototype providing transformations of various modeling languages to formal grammars as well as the grammar-based execution of service compositions is presented.**

Keywords—Web Service, Service Composition, Unification, Formal Grammars, Formal Automata

I. INTRODUCTION

For the modeling of service compositions various specification languages exist, e.g. BPEL, BPMN, ConDec, UML Activity Diagrams. Furthermore, various engines for service compositions exist supporting one or more specification languages. For the processing each engine uses its own internal model covering the service composition logic [7]. That means, there exists no common reference model for processing service compositions. The approach at hand aims at a unified execution of service compositions that are allowed to be modeled by different specification languages. Therefore, the approach introduces a unified model that is suitable to be an internal model as well as an engine for processing the unified model. The introduced unified model covers different modeling paradigms, e.g. imperative and declarative languages. However, the unified model is intended to be an internal model, i.e. it is the target of transformations and human modelers are not expected to directly specify the unified model.

Figure 1 illustrates the idea of the approach: Each specification language requires an engine that implements the operational semantics of the language. Analogously, formal grammars and automata are correlated in language theory. The approach at hand proposes formal grammars for the unified model of service compositions and formal automata for the unified execution of service compositions. In general, a formal grammar coordinates symbols (non-terminals and terminals) whereas a service composition coordinates service calls. That means, formal grammars are suitable to cover service compositions by correlating the symbols of a grammar with service calls in a service composition. In detail, a service call is represented by a non-terminal (possibly in combination with a terminal). The creation of the non-terminal (e.g. A in Figure 2) represents the activation of the service call. The processing of a production rule specifying the particular non-terminal on the left hand side (lhs) represents the execution of a service call (e.g. rule $A \rightarrow a$ in Figure 2). The right hand side (rhs) of this production rule may specify a terminal representing lasting information about the finishing of the service call.

Various approaches covering a unified model for service compositions already exist with different purpose. For example, [1] aims for a unified modeling in UML and a transformation to executables languages like BPEL. A formal unified model is introduced in [2] with intent to analyze semantic properties, e.g. behavioral equivalence of service compositions. Other approaches aim for defining a unified model to be a reference model for service compositions. For example, [3] introduces workflow patterns allowing the unification of service composition models based on a set of generic constructs. More accurate, [4] and [5] introduce unified meta-models by ontologies for the domain service composition. Similarly, [6] introduces an interchange format representing a unified model supporting all the information that is typically covered in existing specification languages. The approach at hand introduces a unified model that is suitable to be a native model [7] for service composition. That means, the unified model is intended to be used as internal processing model in engines. The relation between the unified model and typical specification languages for service compositions is similar to the relation between assembler and high programming languages. Furthermore, the approach especially aims for the unification of different modeling paradigms and allows to improve the scalability of engines.

II. UNIFIED MODEL

A formal grammar $G=(V, \Sigma, P, S)$ is a specialized rewrite system $(V \cup \Sigma, P)$ separating the symbols of a rewrite system into non-terminals and terminals, providing a specific start symbol, and restricting the structure of production rules by requiring at least one non-terminal on the lhs of each rule. The service grammars introduced in this section specialize formal grammars by further separation of non-terminals and further restrictions to the production rule structure. Additionally,

production rules of service grammars are specifically interpreted, i.e. the order of symbols is abstracted.

Definition 1: A service grammar $G_S=(V, \Sigma, P, S)$ is a formal grammar $G=(V,\Sigma, P, S)$ where:

- V is a set of complex non-terminals
- P is a set of c-interpreted production rules with: $P \subseteq \Sigma^* V \Sigma^* \times (V \cup \Sigma)^*$

Complexity of Non-Terminals

In service grammars non-terminals can have multiple dimensions. Conventional non-terminals are 1-dimensional symbols exclusively specifying the name of the symbol. The 1-dimensional non-terminals are helper symbols in service compositions that are required for example for the synchronization of parallel control flow (cf. non-terminal H in Figure 2 and Figure 3). For enabling service-oriented computing 2-dimensional non-terminals are associated with service operations by a non-terminal type (cf. non-terminal A in Figure 2 and Figure 3). Additionally, the second dimension of non-terminals covers input and output parameters if necessary. Hence, data is given by an absolute term or handled by reference, i.e. input and output data of a service call is typically stored in an external database (cf. section IV). The correlation of non-terminals and service operations allows a classification of non-terminals, i.e. different non-terminals of the same type represent calls of the same service operation. However, input and output parameters are specific to a service call, i.e. non-terminal. For example, two different conditions create two different non-terminals sharing the same type (e.g. XPathSolver) but specifying different input parameters, i.e. conditions. Finally, the third dimension of non-terminals covers the feedback of a service call if the service operation result represents required information for the execution of a service composition. In detail, the third dimension specifies a mapping of service operation return values and 1-dimensional non-terminals (cf. non-terminal C in Figure 2 and Figure 3).

C-Interpretation of Production Rules

Production rules in formal grammars handle words, i.e. cover the concurrent existence as well as the order of symbols. For example, the production rule $abX \rightarrow abC$ requires the terminal a to be placed immediately left to the terminal b. In contrast, 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. Furthermore, c-interpreted production rules cover a set of conventionally interpreted production rules. For example, the c-interpreted production rule $abX \rightarrow abC$ collects a set of conventionally interpreted production rules $\alpha x \beta y \gamma z \delta \rightarrow \alpha x' \beta y' \gamma z' \delta$ with $x,y,z \in \{a,b,X\}$, $x',y',z' \in \{a,b,C\}$, $x \neq y \neq z$, $x' \neq y' \neq z'$, and $\alpha,\beta,\gamma,\delta \in (V \cup \Sigma)^*$.

Definition 2: The multiset-interpretation of a word $w=(x_1,x_2,\ldots,x_n)$ over an alphabet Σ is defined by a multiset $\mathfrak{M}:\Sigma \rightarrow IN_0$ with: multiset(w)= $[x_1, x_2, \ldots, x_n]$

where a multiset is notated by square brackets, e.g. [a,a,b,c].

Definition 3: The c-interpretation of a production rule $p=(\alpha,\beta)$ results in a c-interpreted production rule with the function i_c:

$$i_c(p)=(\text{multiset}(\alpha), \text{multiset}(\beta))$$

Considering the language the c-interpretation of production rules creates another language covering the commutative closure of single words of the actual conventional language. [8] discusses the commutativity of symbols in words and languages in detail. Hence, dependencies are introduced restricting the commutativity of symbol pairs. In contrast, the approach at hand allows commutativity for all symbols at all time. However, the approach in [8] is suitable to define the language corresponding to a c-interpretation of a grammar.

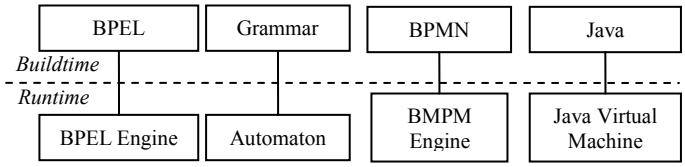


Figure 1: Grammars and automata for unification

Context Types

As mentioned before, formal grammars require at least one non-terminal on the lhs of production rules. The approach at hand calls this non-terminal the *processing symbol* of the particular rule that is not necessarily uniquely determined. Context-sensitive and higher classes of grammars allow additional symbols next to the processing symbol on the lhs of production rules, i.e. *context symbols*. The following context types create sub-classes of context-sensitive grammars based

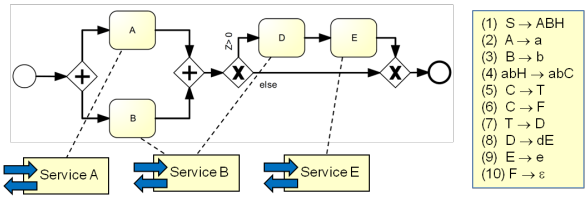


Figure 2: Example

```
<nonTerminal> <name> H </name> </nonTerminal>
<nonTerminal> <name> A </name>
  <type> Service A Operation A1 </type>
  <input> <reference>Y </reference> </input>
  <output> <reference>Z </reference> </output>
</nonTerminal>
<nonTerminalType name="Service A Operation A1">
  <wsa:EndpointReference> <wsa:Address>
       http://localhost:9763/services/ServiceA
  <wsa:Address> </wsa:EndpointReference>
…<operation> A1 </operation>
</nonTerminalType>
<nonTerminal> <name> C </name>
  <type> XPathSolver </type>
  <input> <reference>Z </reference>
          <value> Z>0 </value> </input>
   <relations>
     <relation> <outputValue> True </outputValue>
                <nonTerminalREF> T </nonTerminalREF>
     </relation>
     <relation> <outputValue> False </outputValue>
                <nonTerminalREF> F </nonTerminalREF>
     </relation>
   </relations>
</nonTerminal>
<nonTerminalType name="XPathSolver">
  <wsa:EndpointReference> <wsa:Address>
    http://localhost:8080/services/XPathSolver
    </wsa:Address></wsa:EndpointReference>
…<operation> evaluate </operation>
</nonTerminalType>
```

Figure 3: A one-dimensional non-terminal H, a two-dimensional non-terminal A, and a three-dimensional non-terminal C

on the structure of production rules:

Definition 4: Assuming a formal grammar (V, Σ, P, S) context types are defined as follows:

Terminal-based Context: Production rules are restricted to specify exactly one non-terminal on the lhs but are allowed to specify multiple terminals on the lhs.

$$P \subseteq \Sigma^* V \Sigma^* \times (V \cup \Sigma)^*$$

Non-Terminal-based Context: Production rules are allowed to specify multiple non-terminals and terminals on the lhs.

$$P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$$

Invariant Context: Context symbols that are specified on the lhs are not allowed to be changed, i.e. need to be specified on the lhs as well as on the rhs.

$$\forall(\alpha,\beta) \in P \; \exists x,y: \alpha = xMy \wedge \beta = xNy \wedge M \in V \wedge N \in (V \cup \Sigma)^*$$

with $x,y \in \Sigma^*$ for terminal-based context and

$x,y \in (V \cup \Sigma)^*$ for non-terminal-based context.

Variant Context: Context symbols that are specified on the lhs are allowed to be changed, i.e. are not required to be specified on the rhs.

$P \subseteq \Sigma^* V \Sigma^* \times (V \cup \Sigma)^*$ for terminal-based context and

$P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ for non-terminal-based context

Table 1: Combining context types

	Invariant Context	*Variant Context*
Terminal-based Context	Unique processing symbol Terminals are not allowed to be deleted after their creation	Unique processing symbol Terminals are allowed to be deleted after their creation
Non-Terminal-based Context	Unique Processing symbol Terminals are not allowed to be deleted after their creation	Multiple processing symbols Terminals are allowed to be deleted after their creation

Table 1 summarizes the effects of combining the context types introduced in Definition 4. The symbol kind creating the context (i.e. terminal- or non-terminal-based context) impacts on the uniqueness of the processing symbol. A non-terminal-based context allows to uniquely determine the processing symbol only if the context is invariant where the single non-terminal that is allowed to be changed (i.e. exists on the lhs but not on the rhs) represents the processing symbol. The variance of context impacts on the ability to delete symbols after their creation. Non-terminals basically have the ability to be deleted by the ability to be a processing symbol. However, terminals are only allowed to be deleted as context symbols if they are not intended to be really terminal. Note that the ability to delete terminals after their creation has no impact on the corresponding language. Assuming a fixed language there always exists another grammar that doesn't require the deletion of terminals after their creation. For example, the following grammars specify the same language but provide different kinds of context.

G_1: $S \rightarrow BA$, $B \rightarrow v \mid x$, $vA \rightarrow ab$, $xA \rightarrow ba$

G_2: $S \rightarrow BA$, $B \rightarrow a \mid b$, $aA \rightarrow ab$, $bA \rightarrow ba$

G_3: $S \rightarrow BA$, $B \rightarrow V \mid X$, $VA \rightarrow Vb$, $XA \rightarrow Xa$, $V \rightarrow a$, $X \rightarrow b$

Language theory considers the above grammars to be equivalent as they cover the same language. In contrast, the approach at hand considers the grammars to be different as they specify different instructions to create the same words. In service grammars, a non-terminal occurring at a specific time is of high importance whereas the resulting word is of low interest. That means, the approach at hand focuses on the grammar but neglects the corresponding language. Furthermore, the grammars G_1, G_2, and G_3 are considered to be not equivalent as they show different runtime behavior that needs to be reflected by corresponding automata (cf. section III.A).

Transformations

This section outlines transformations of existing specification languages for service compositions to service grammars. For a detailed description of the presented transformation please see [9]. The grammars generated by the presented transformation cover words representing traces of service calls and scope states. Data is mainly handled by reference. That means, data is mostly not specified by explicit symbols in the grammar but an external database is responsible for storing the data. Consequently, data assignment is represented by a simple service invocation.

BPEL [10] is an imperative, i.e. flow-based language that is highly developed for the specification of service compositions. A BPEL sequence activity containing the service calls D and E is represented by simple production rules creating one 2-dimensional non-terminal on the rhs in maximum (cf. rule (8) in Figure 2). In contrast, parallel control flow requires to create multiple non-terminals in one production rule representing the activation of multiple service calls at the same time (cf. rule (1) in Figure 2). That means, in contrast to sequential control flow that is satisfied by regular grammars parallel control flow requires at least context-free grammars. If synchronization is required for parallel control flow paths the grammar even needs to be context-sensitive (cf. rule (4) in Figure 2). BPEL Scopes are specified by one 1-dimensional non-terminal in combination with one terminal and require context-sensitive production rules. The non-terminal of a scope indicates the activation, i.e. the non-terminal is created/deleted when the scope is activated/finished. The terminal of a scope indicates the regular or the fault mode of an activated scope. If the scope is already finished the terminal indicates the finishing mode of the scope (e.g. successfully finished or compensated). After the finishing of a scope the terminal needs to exist until the end of the processing as the ability to compensate the scope possibly needs to be decided by context-sensitive rules. *Service compositions require grammars to specify variant context as the deletion of multiple symbols in one production rule is required (e.g. for scope finishing)*[1]*. The variant context needs to be terminal-based to optimize the corresponding runtime effort in service compositions, i.e. terminal-based context ensures the simplest automaton for variant context.*

ConDec [11] is a declarative language that is used for the specification of constraint-based service compositions. Hence, constraints specify dependencies between service calls, i.e. activities. Typically, multiple activities are allowed to be executed at a specific point in time and a (human) user is expected to select a specific activity for execution. After the execution of the selected activity a new set of activities that are allowed to be executed next needs to be calculated based on all

[1] Scope finishing requires even more general grammars (type0) as the non-terminal is deleted without substitution.

(1-3) $S_1 \rightarrow A_1 \mid C_1 \mid \varepsilon$
(4) $A_1 \rightarrow a\ S_2$
(5) $C_1 \rightarrow c\ S_1$
(6-8) $S_2 \rightarrow A_2 \mid B_2 \mid C_2$
(9) $A_2 \rightarrow a\ S_2$
(12-15) $S_3 \rightarrow A_3 \mid B_3 \mid C_3 \mid \varepsilon$
(10) $B_2 \rightarrow b\ S_3$
(11) $C_2 \rightarrow c\ S_2$
(16) $A_3 \rightarrow a\ S_2$
(17) $B_3 \rightarrow b\ S_3$
(18) $C_3 \rightarrow c\ S_3$

Figure 5: A declarative service composition

given constraints. Figure 5 shows the production rules for a service composition specifying three service calls A, B, and C as well as two constraints. The constraint response(A,B) specifies that the call B must be executed in future when A is executed at least once. The constraint precedence(A,B) specifies that the call A needs to be executed when B begins to execute. In between all other service calls C are allowed to be executed. At the very beginning the start symbol S_1 allows the activation of the service calls A and C additionally to the symbol ε indicating the finishing of the service composition. The index for the start symbol needs to be introduced for covering different sets of service calls that are allowed to be executed at the same point in time. The index for service calls A, B, and C is introduced to cover different effects of service call executions at different points in time. Obviously, declarative service compositions create grammars with a lot of non-deterministic alternatives. The non-determinism reflects the fact, that a user is expected to be responsible for selecting a specific activity, i.e. service call for execution at runtime. Service grammars are allowed to specify non-determinism exclusively for covering dynamic information that can be provided by an external component (i.e. oracle) at runtime.

III. UNIFIED ENGINE

Service compositions create breadth-first grammars, i.e. the least recently produced non-terminal must be processed first ensuring the processing of parallel control flow paths nearly at the same time. In contrast, a depth-first search on service composition grammars would process parallel paths successively. Grammars with breadth-first semantics are typically covered by queued automata [12]. Queued automata are equivalent to Turing machines, i.e. suitable to cover service grammars. In contrast to Turing machines queued automata provide the separation of the tape and the working storage (i.e. the queue) allowing a classification of automata based on access types to different storages corresponding to different context types.

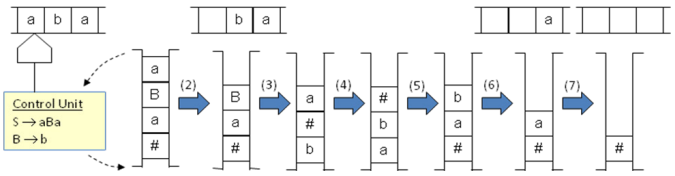


Figure 4: Configurations of a queued automaton

Generating queued automata require to delay the processing of symbols. Figure 4 shows some example configurations of an accepting queued automaton. The first terminal a in the queue is allowed to be accepted from the tape in step (2) but the second terminal a in the queue needs to be delayed until step (7). In accepting automata the input word is be used for the decision about the accepting or the delay of a terminal. In generating automata no input word exists that contributes in deciding about the handling of terminals. Instead, a special symbol # needs to be introduced ensuring the right order of terminals that are generated to the tape: Terminals immediately following the symbol # are allowed to be generated. After the first non-terminals following terminals need to be delayed until the symbol # is processed again.

A. *Hierarchy for Queued Automata*

This section presents a hierarchy for queued automata enabling the measuring of runtime efforts for service compositions. Hence, the separation of tape and working storage (i.e. queue) is an important issue for the classification. The tape is considered to store exclusively terminals whereas the working storage is considered to primarily store non-terminals. The terminals occurring in the workings storage are considered to be immediately processed, i.e. accepted/generated from/to the tape. The introduced hierarchy characterizes automata classes by storage access types in addition to storage types that are already covered by the Chomsky hierarchy. For the tape an automaton can provide reading access with and without deletion as well as writing access. For the working storage an automaton can provide simple or complex access indicating the number of symbols that need to be processed in the context of the a single production rule.

Service grammars require at most automata of class 4.

1. Automata for regular grammars
- Simple tape access (e.g. reading with deletion for accepting words)
- Simple working storage access (i.e. exactly one symbol in the working storage is processed)
- Working storage size = 2

2. Automata for context-free grammars
- Simple tape access
- Simple working storage access
- Working storage size > 1

3. Automata for context-sensitive grammars with invariant and terminal-based context
- Moderate tape access (e.g. reading with and without deletion for accepting a word)
- Simple working storage access
- Working storage size > 1

4. Automata for context-sensitive grammars with variant and terminal-based context
- Complex tape access (i.e. reading and writing access)
- Simple working storage access
- Working storage size > 1

5. Automata for context-sensitive grammars with invariant and non-terminal-based context
- Moderate tape access
- Complex working storage access (i.e. more than one symbol in the working storage needs to be processed)
- Working storage size > 1

6. Automata for context-sensitive grammars with variant and non-terminal-based context
- Complex tape access
- Complex working storage access
- Working storage size > 1

B. Generating Automata

Service Grammars specify how to create valid runs of service compositions. An automaton for the execution of grammar-based service composition is required to generate a valid run. That means, service compositions require generating automata instead of accepting automata. Existing generating automata, i.e. transducer (cf. Moore machine, Mealy machine) generate output words but also operate on input words. The input words are required for determinism while processing but represent static information. The approach at hand uses generating automata covering output words but no input words. The input that is required for determinism is provided by an oracle at runtime.

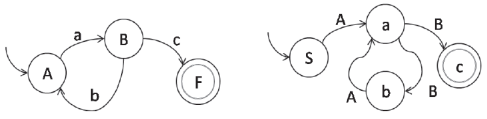
Accepting and generating automata have some fundamental differences. At first, accepting a word is satisfied by restricting to one *derivation* as only one derivation needs to exist in order to prove the membership of a word to a language. For example, a pushdown automaton is restricted to the left-most derivation. Generating automata are required to support all derivations of a word specified by a grammar if the grammar is focused instead of the language. Therefore, automata corresponding to grammar-based service compositions are required to support all alternatives for creating a word as all execution paths specified in a service composition need to be covered. *Queued automata implementing a breath-first search in combination with the ability to delay symbols for later processing allow the support of all possible derivations.*

Secondly, a given input word in accepting automata impacts on the *determinism* whereas generating automata cannot use this information as the word doesn't exist yet. For example, the accepting automaton in Figure 6(a) is deterministic although the corresponding grammar is static non-deterministic. The generating automaton in Figure 6(b) illustrates the same grammar without further information, i.e. the automaton is non-deterministic as the following state to state "a" cannot be uniquely determined.

Remark 1: A service grammar is deterministic at buildtime, i.e. static deterministic iff the lhs of contained production rules are pairwise distinct: $\forall x,y \in P , x \neq y , x=(\alpha_1,\beta_1) , y=(\alpha_2,\beta_2) : \alpha_1 \neq \alpha_2$

Remark 2: A service grammar is deterministic at runtime, i.e. dynamic deterministic iff a single rule is deterministically selected out of production rules specifying the same lhs based on dynamic information at runtime.

Service grammars need to be dynamic deterministic for enabling alternative runs. Figure 2 shows a dynamic deterministic grammar, i.e. the dynamic information Z is needed to evaluate the condition. Statically, the grammar allows both alternative control flow paths by the rules (5) and (6). At runtime only one rule is selected based on the evaluation of the condition Z>0. For providing dynamic deterministic grammars corresponding automata need to be extended by oracles. An oracle machine [13] is an automaton that uses an additional component (i.e. oracle) implementing a particular function. Hence, the automaton doesn't need to implement the function itself. Instead the oracle provides the function value to the automaton. Therefore, an oracle is suitable to provide dynamic information to an automaton at runtime. The approach at hand implements the required oracle by a service in the presented prototype. A call of the oracle is implemented by a 3-dimensional non-terminal. *Production rules in service grammars with the same lhs are allowed to exclusively specify 3-dimensional non-terminals on the lhs for ensuring dynamic determinism.*



(a) **(b)**

Figure 6: Accepting and generating automaton for the grammar G=({A,B},{a,b,c},{(A,aB),(B,bA),(B,c)},A)

Finally, a given word impacts on the finishing requirement of accepting automata but generating automaton requires to cover other factors. Accepting automata successfully finish iff no terminal is left on the tape that is required to be accepted (additional a final state needs to be reached if final states are specified by the corresponding automaton). A generating automaton cannot decide finishing based on terminals but uses non-terminals.

Definition 5: Generating automata successfully finish iff there exists no non-terminal anymore.

In summary, the automaton that is used for the execution of service compositions is required to be a (1) queued automaton with at most complex tape access and simple working storage access, (2) generating automaton providing a set of oracles for dynamic determinism.

Definition 6: A generating queued automaton M for service compositions is a 5-tuple (Σ, Γ, δ, S, O) where
Σ is a (finite) set of symbols (i.e. the tape alphabet)
Γ is a (finite) set of symbols (i.e. the queue alphabet)
δ is a transition mapping with $\delta:\Sigma^* \times \Gamma \rightarrow \Sigma^* \times \Gamma^*$
S is the start symbol for the queue
O is an oracle

Definition 7: A configuration of a generating queued automaton is a tuple (w,γ) where w represents the content of the tape and γ represents the content of the queue.

Definition 8: Let G=(V,Σ,P,S) be a dynamic deterministic service grammar with an oracle O. The generating queued automaton for the service grammar is defined by M=(Σ,V$\cup\Sigma$,δ,S,O) with: $\forall(\alpha X\beta,\gamma)\in P, X\in V \; \exists \; \delta(\alpha\beta,X)=(\varepsilon,\gamma)$
$\forall \; X\in V \; \exists \; \delta(\varepsilon,X)=(\varepsilon,X)$
$\forall \; y\in \Sigma \; \exists \; \delta(\varepsilon,y)= (y,\varepsilon)$

Property 1: A service grammar is deterministic with a queued automaton if the grammar is static or dynamic deterministic and provides dynamic exclusive context, i.e. for each pair (p_1,p_2) of context-sensitive production rules with the same processing symbol exist context symbols $t_1 \in p_1$, $t_2 \in p_2$ that are mutually exclusive at runtime.

IV. PROTOTYPE

This section introduces a prototype for the grammar-based execution of service composition. Figure 7 shows the architecture of the prototype. A formal queued automaton is a generic component that is used for the execution of a service composition instance. A service grammar specifies the model of the service composition and multiple instances of the model are provided by multiple automata covering the same grammar. The formal automaton needs to be extended by a component implementing the service invocation. If the automaton

processes a 2- or 3-dimensional non-terminal the service invocation component is responsible for executing the service calls. Hence, the service invocation component uses a parameter resolution component for determining the concrete input and output parameters of service calls. As data is handled by reference a reference resolution system [14] is responsible for storing data and managing references to data. The reference resolution system is not part of the service composition instance but is provided by a service that is invoked at runtime similar to the composed services. In detail, the automaton interacts with the reference resolution system for parameter resolution. The parameter resolution component manages the internal reference names specified in non-terminals and correlates these names with reference identifiers assigned by the reference resolution system. Therefore, multiple automata, i.e. service composition instances are allowed to use the same reference resolution system service for storing data. Similar to the reference resolution system the presented prototype provides an expression evaluation service, i.e. an XPath solver for XPath expressions. The expression evaluation service implements the oracle in the automaton that is required if decisions about control flow alternatives need to be taken (cf. non-terminal C in Figure 2). Finally, the prototype implements a management component that is responsible for the transformation of a service composition model to a grammar-based specification as well as the creation and deployment of a service composition instance. Hence, a dynamic distribution algorithm [15] is used to determine the best location for the service composition instance at runtime.

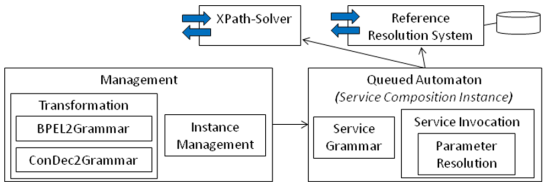


Figure 7: Architecture

V. CONCLUSION

The presented approach uses formal grammars and automata for the modeling and execution of service composition. Used formal grammars and automata need to be adapted: Complex non-terminals need to be provided covering the relation to services. Furthermore, production rules are allowed to ignore the order of symbols while focusing on the concurrent existence of symbols. Automata that are used for the execution of service compositions need to be extended by service invocation and need have generating character supporting all derivations of a word. A formal automaton represents a generic and simple implementation component in comparison to conventional service composition engines. Hence, the approach allows to easily measure the runtime effort of a service composition by the particular automaton class. For example, "simple" parallel execution paths require context-free production rules whereas parallel paths with the need for synchronization require context-sensitive production rules. In particular, different kinds of production rules (i.e. service composition logic) require different automaton classes for the processing. Furthermore, the approach improves the flexibility of service composition execution by using a simple and generic automaton covering exactly one service composition instance. In particular, different instances can be easily deployed on different nodes in a distributed runtime environment in contrast to managing all instances in one engine running on one node (cf. [15]). Formal grammars are well suited for service composition models as service compositions are associated with the concept programming in the large. In particular, service compositions are "only" considered to coordinate services. In contrast, programming in the small covers information on a more detailed level that is typically specified by high programming languages (e.g. Java). The approach at hand uses formal grammars exclusively for the programming in the large concepts. That means, concrete data as well as operations on data are not necessarily covered by the grammar but by external services. In summary, formal grammars are suitable to cover service compositions as data is mostly required to be transferred from service to service whereas only a few concrete data values are required for the execution of service composition (cf. control flow alternatives). Furthermore, grammars already compose symbols emphasizing the ability to compose services. Introducing service invocation and dynamic data in formal grammars is similar to the extension of automata by oracles that can be properly implemented by services.

REFERENCES

[1] D. Skogan, R. Grønmo, I. Solheim: „Web service Composition in UML" Proceedings of the EDOC 2004

[2] M Mazzara, I. Lanese: "Towards a Unifying Theory for Web Services Composition" Web Services and Formal Methods, LNCS Vol. 4182, 2006, pp 257-272

[3] W. van der Aalst, A. Barros, A. Hofstede, B. Kiepuszewski: „Advanced Workflow patterns" Cooperative Information Systems. Springer Berlin Heidelberg, 2000.

[4] F. Heidari, P. Loucopoulos, F.Brazier, J. Barjis: "A Meta-Meta-Model For Seven Business Process Modeling Languages" CBI 2013

[5] R. Hull "Towards a Unified Model for Web Service Composition" Advances in Computer Science–ASIAN 2005. Data Management on the Web. Springer Berlin Heidelberg, 2005. 1-10

[6] J. Mendling, G. Neumann, and M. Nüttgens. "A comparison of XML interchange formats for business process modelling." Workflow handbook (2005): 185-198.

[7] F. Leymann. "BPEL vs. BPMN 2.0: Should You Care?" Business Process Modeling Notation. Springer Berlin Heidelberg, 2011. 8-13.

[8] V. Diekert, G. Rozenberg "The Book of traces. " World Scientific, 1995.

[9] K.Görlach: "A Generic Transformation of Existing Service Composition Models to a Unified Model" University of Stuttgart, Technical Report 2013/01 ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2013-01/TR-2013-01.pdf

[10] Web Services Business Process Execution Language Version 2.0, OASIS Standard, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html (August 6, 2013)

[11] M Pesic: "Constraint-Based Workflow Management Systems: Shifting Control to Users" PhD thesis, Technische Universiteit Eindhoven, 2008

[12] E. Allevi, A. Cherubini, S. Crespi Reghizzi. "Breadth-first phrase structure grammars and queue automata." Mathematical Foundations of Computer Science 1988. Springer Berlin Heidelberg, 1988.

[13] D. van Melkebeek. "Randomness and completeness in computational complexity" ACM Doctoral Dissertation Award Series. LNCS 1950, Springer 2000

[14] M. Wieland, K. Görlach, D.Schumm, F. Leymann. „Towards reference passing in web service and workflow-based applications" Proceedings of the EDOC 2009

[15] K. Görlach, F. Leymann: "Dynamic Service Provisioning for the Cloud." Proceedings of the SCC 2012