Using Graph Grammars for Interaction Style Description. Application to Service-Oriented Architectures*

Karim Guennoun and Khalil Drira
LAAS-CNRS, 7 Avenue du Colonel Roche 31077 Toulouse Cedex 4 FRANCE,
guennoun,khalil@laas.fr

Abstract. Applications with run-time changing architectures constitute a challenge for both modelling and reasoning. Their description is not limited to the specification of a unique static topology but must cover the scope of all the correct configurations. We develop, in this paper, the concept of architectural styles to achieve this goal. We elaborate and specify the basic architectural styles for the design of service-oriented applications. For this purpose we develop an appropriate formal framework using graph grammars. Our approach enables both generating architectures in conformance with a given style and checking conformance of ad-hoc architectures. We first, describe formally the basic interaction style involving elementary interactions between a service requestor and a service provider. Then we consider the orchestrated interaction style where an orchestrator manages the workflow of several service requestors and providers. Finally, we define a complex architectural style to address the compositional aspect of service-oriented architectures considering composite services. We also provide rules for composing the previous styles to define the composite basic invocation style and the composite orchestrated style.

1 Introduction

Research activities related to software architecture aim to answer requirements such as adaptability, mobility and reuse by promoting the distinction between the design level and the implementation level. The IEEE 1471 standard [1] defines the architecture description as "*a collection of products to document an architecture*". Architecture description is supposed to include aspects such as the organization of the system, the decomposition into components, the description of these components functionalities and the manner with which these components interact [2].

Existent research and standardization works addressing service-oriented applications and web services focus on service description and interface specification through *WSDL* (Web Service Definition Language) and *UDDI* (Universal Description, Discovery and Integration) [3], and on behavioral and workflow aspects through languages such as *BPEL4WS* (Business Process Execution Language for Web Services) for orchestration, *WSCI* (Web Service Choreography Interface) for choreography [4,5], and *WS-QoS* (Web Service Quality of Service) for QoS description [6].

Future complex software for various application domains should rely on dynamic architectures to fulfill evolving-related requirements. Architecture reconfiguration is used for policy-based security management. [7] Presents two approaches: the proactive reconfiguration that aims to increase the resilience of the system to face a specific type of attacks and the reactive reconfiguration that aims to restore the integrity of the system when an intrusion is detected and the system suspected to be in a corrupted state. Architecture dynamic evolving can also be used as a fault tolerance mechanism [8] making it possible to offer reliable services by reconfiguring the system in response to fault occurrence. [9] Implements the reconfiguration of architecture by deploying new components and by changing component behavior. These architecture

* This work has been developed within the framework of the IST project WS-DIAMOND.

transformations are used like means for load balancing and mobility management while [10] uses reconfiguration to provide quality of service.

We propose, in this paper, to extend the formal scope of SOA-related existing work. We analyze the architectural characteristics considering the dynamic aspects in SOA within a formal framework based on graph grammars [11]. The goal is to provide a new point of view for the architectural style of service-oriented applications. We believe this will be useful to specify and check architecture-related properties at design-time.
We illustrate our approach by specifying the basic architecture styles of service-oriented applications. We focus, here, on the execution model and do not consider the discovery step. Extending our model to cover this step is possible by introducing additional components representing, for instance, the *UDDI* registry. This paper is organized as follows. In the following section, we will have an overview of the state of the art for the architecture style description and of the graph grammar literature. In section 3.1, we introduce the basic interaction style involving elementary interactions between a service requestor and a service provider. Then, we specify the orchestrated interaction style in section 3.2. Within this architectural style, we introduce the orchestrator, a component which is dedicated to managing the workflow between a set of service requestors and providers. In section 3.3, we consider service composition and introduce the composite style involving composite service providers. Finally, based on the combination of the two first styles with the composite one, we specify the composite basic interaction in subsection 3.3.1 and the composite orchestrated interaction styles in section 3.3.2.

2 State of the art

2.1 Architecture style description

Suitable description languages and formalisms for avoiding ambiguities are necessary for correct architectural design, management and analysis. In the last decade, many architecture description languages (ADLs) were introduced providing rigorous syntax and semantic to define architectural entities and relations. General approaches for ADLs provide formalisms to specify the components, the connectors, the ports and the interfaces belonging to the described architecture and the way to put them together to build the architecture in a consistent way. ADL-based approaches generally rely on formal languages such as process algebra for architecture constraints verification and analysis. Such approaches are not sufficient when dealing with dynamic architectures considering advanced requirements for run-time reconfiguration. In such cases, the set of architectural entities is not fixed and may evolve during the execution. The architectural description should define the set of correct architectural configurations rather than enumerating the elements of a unique configuration.
The concept of architectural styles was defined by the IEEE 1471 standard as "*a set of patterns or rules for creating one or more architectures in a consistent fashion*". In concrete terms, this concept is concerned with the specification of acceptable or consistent architecture instances of a system. Few ADLs consider architectural styles but the dynamicity-related specification expressiveness remains limited. In the following paragraphs we will give an overview of style description in three of the most popular ADLs.

The concept of architectural style is explicitly defined by *Wright* [12,13]. A style is composed of two parts: 1) the first part is specifying *component* and *connector* types that might compose the application architecture where 2) the second part is dedicated to the specification of constraints (expressed in terms of the first-order logic) over components and connectors bindings. Architecture instances (called configurations in *Wright* terminology) are defined by declaring the entities of the architecture and describing their interactions with respect to the constraints of the style binding. The entities represent instances of component and connector types belonging to the architectural style.

C2SADL [14,15] is an architecture description language that allows the definition of architectural styles. As for *Wright*, a style is defined by declaring its component and connector types. *C2SADL* is based on a generic style called *C2*. In this style, components have exactly two ports as interface while connectors may have as much ports as needed. The ports of components are called *top-domain* and *bottom-domain* and the ports of connectors are called *top-ports* and *bottom-ports*. Styles defined in *C2SADL* are sub-styles of the *C2* style constraining the bindings so that one component is connected to a unique connector for each port while connectors may be connected to more than one component for each ports side. The specification of instances belonging to architectural styles includes the enumeration of its component and connector instances (belonging to the style description) and the description of their bindings following the *C2* style constraints.

The description of architectural styles is not explicitly expressed in *Darwin* [16,17]. But, this description may be partially achieved using parameterization and array declaration mechanisms. Architectural styles could be described by the definition of configurations where component parameters are used to control size and topology of replicated structures. Architecture instances belonging to these generic structures are obtained by instantiating these parameters.

To conclude about these architecture description languages, we can say that their approaches for the specification of architectural styles are interesting but remains unsatisfactory. The *C2* style is constraining insofar as the communication between components can be arranged only in layers. This excludes, for example, the description of two components being mutually sending requests or architectures containing cycles. Moreover the fact that the components are constrained with exactly a *top-domain* and a *bottom-domain* prevents them from being connected to several connectors which adds an additional restriction for the communication structure between components. In *Wright*, the constraints that we can express for an architectural style are limited to those that we can express in the first-order logic. We cannot express, for instance, constraints such as maximum or minimum of bindings on server. *Darwin* parameterization approach is also not suitable for the description of sophisticated architectural styles.

2.2 Graph grammars

Graph grammars [18, 11, 19] constitute a powerful theoretical framework for describing complex mechanisms for structure transformation. This formalism was introduced at the end of the Sixties to address problems such as compiler building, or data types specification. It constitute also a formal basis for solving problems related to various application domains including facial identification [20, 21], object recognition of [22, 23], symbol recognition [24], and character identification and graphology [25].

The basic way to transform a graph G into a graph G' is to replace a subgraph m of G by a graph d. G' is the graph resulting from these two operations. G is called the host graph, m is called mother graph and d is called daughter graph. In this approach, a grammar production is described in the basic model by a pair of graphs: $<L;R>$. This rule is applicable to a graph G if there is an

occurrence of *L* in *G*. Its application has, as a consequence, the removal of the occurrence of *L* from the graph *G* and its substitution by a copy (isomorphous) of *R*. This type of basic definition introduces the problem of dangling edges as shown in figure 1. To solve this problem, two approaches were introduced with different choices concerning productions specification and dangling edge management [26].

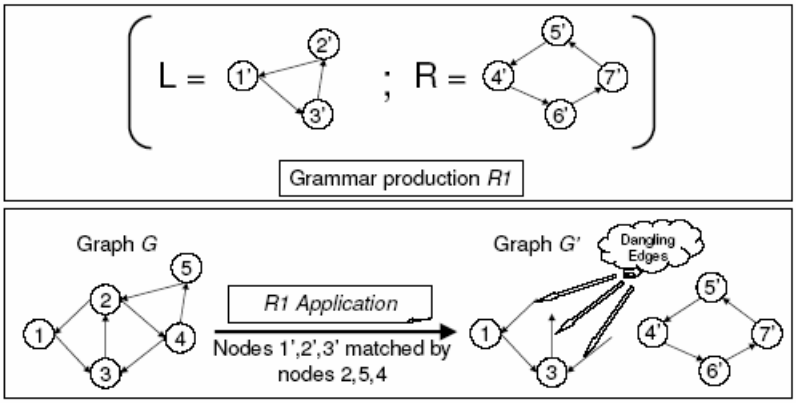

Figure 1. Dangling edge problem

In the *Single PushOut* (*SPO*) approach, a grammar production is specified by a pair of graphs *<L;R>*. Its application to a graph *G* requires the existence of an occurrence of *L* in *G*. The difference, compared to the basic approach presented previously, comes from parts (i.e. nodes and edges) of graph *L* that will be maintained. These parts have to be clearly specified in the two graphs *L* and *R*. Then, the application of this production implies removing the graph corresponding to *Del= (L∩(L\R))* and inserting the graph corresponding to *Add = (R∩(L\R))*. The way *SPO* approach deals with dangling edges is to simply remove them. An example of graph rewriting with an *SPO* production is given in the figure 2. The application of the grammar production is composed of two steps: 1) the graph is transformed by removing the occurrence of the graph *Del* and by inserting a copy of the graph *Add*, and 2) the single dangling edge (connecting node 4 to node 3), resulting from the removal of node 4, is removed.

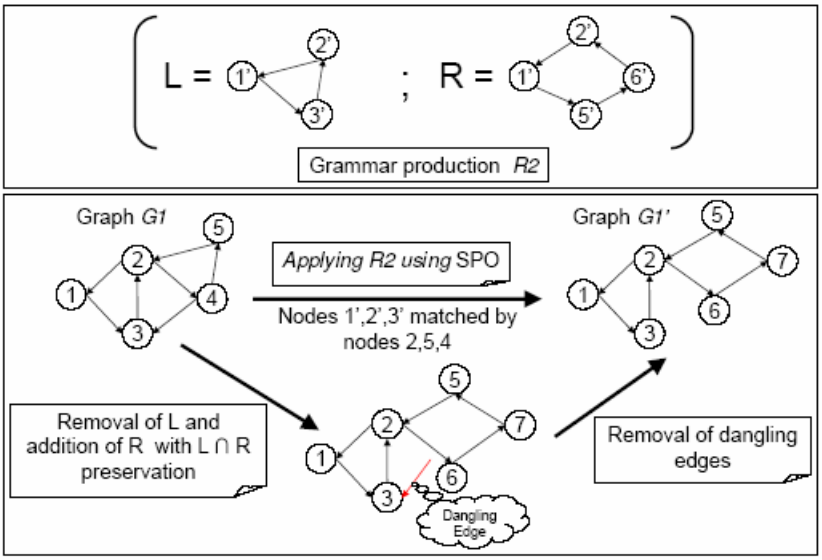

Figure 2. SPO Approach

Within the *Double PushOut* approach (*DPO*), grammar productions are specified by a 3-uplet *<L;K;R>*. The pattern *K* specifies the part to be maintained after the application of the rule rather than deducing it by the operation *L\R*. The other difference with the *SPO* approach is that the application of this production requires an additional condition called the dangling condition. This condition states that the production is applicable only if its application will not lead to dangling edges. If the two conditions (i.e. existence of an occurrence of *L* and absence of dangling edges) hold, the application of the production implies the removal of the graph corresponding to the occurrence of *Del = (L∩K)* and the insertion of a copy of the graph *Add = (R∩K)*. A simple example of the *DPO* transformation approach is given in figure 3. We note that the host graph of this example (noted *G2*) is different from the one given for the *SPO* approach (noted *G1*). The difference lies in the fact that *G2* does not contain any more the edge connecting nodes 4 and 3 in *G1*. This is due to the fact that if we maintain this edge in *G2*, *R3* would not be applicable any more because its application would violate the dangling condition.

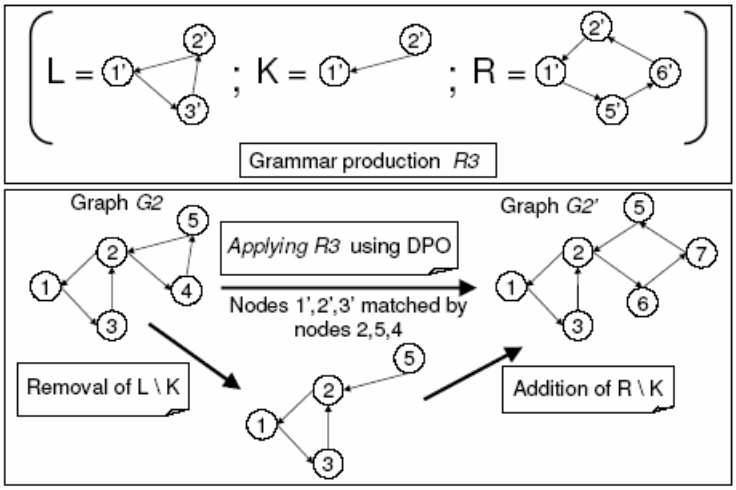

Figure 3. DPO Approach

3 Describing Service-Oriented Architecture styles using the Graph Grammar formalism

Graphs are used here to describe service-oriented architectures. Following the commonly used conventions for standard graphical descriptions, we consider that vertices represent services and edges correspond to their related interdependencies. The use of graphs is relevant since we address the specification of architectural styles where declarative aspects corresponding to the description of all the possible instances can be correctly specified by graph grammars. Theoretical work on the field of graphs and graph grammars provides us formal means to specify and check constraints on these architectures. Important work was achieved using graphs and graph grammars for component-oriented architecture description. We can quote, for instance, [27] for the coordination of dynamic architectures, [28] for the description of architectures and their communication and [29] for diagnosis and repair.
We present, in the next sections, our graph-based approach for the description of architectural style in the context of SOA where graphs are used to describe single architectural configurations and graph grammars are used to specify architectural styles. We define graph grammars as a classical system <AX;NT;T;P> where *AX* is the axiom, *NT* is the set of the non-terminal vertices, *T* the set of terminal vertices, and *P* the set of grammar productions. An instance belonging to the graph grammar is a graph containing only terminal vertices and obtained starting from axiom *AX* by applying a sequence of productions in *P*. For the specification of the grammar productions, we consider the explicit structure of the *DPO* approach that we associate with the more expressive *SPO* approach when dealing with grammar production application. This supposes that there is no dangling condition to apply a grammar production and that dangling edges are removed. For easy understanding, a node is specified in the form *T(i)* where *T* corresponds to the node type and *i* denotes its identifier. Edges are pairs of the form *(n1→n2)* where *n1* is the identifier of the tail node and *n2* is the identifier of the head node. A graph is defined by the two sets of nodes and edges composing its structure.

3.1 Basic Interaction (BI) architecture style

The BI architectural style can be considered as the basic style for service-oriented architectures. Figure 4 presents the basic infrastructure composed of a service requestor and a service provider exchanging, for example, SOAP messages on the top of Internet application layer protocols.

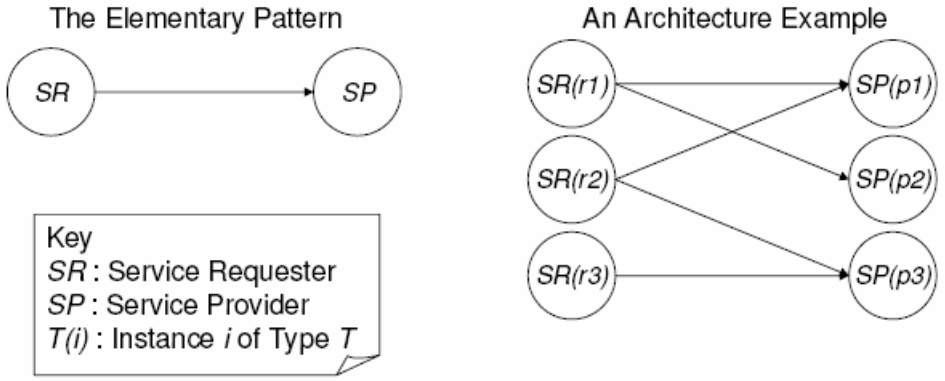

Figure 4. The elementary pattern and an architecture example of the direct interaction service-oriented style

The BI architecture style is described by the following graph grammar:
<AX;{BI};{SR; SP};P_{BI}> Where P_{BI} is the following set of productions:

< { AX } ; { } ; { BI } > (BI.1)
< { BI } ; { } ; { } > (BI.2)
< { BI } ; { BI } ; { SR(r) , SP(p), (r→p)} > (BI.3)
< { BI , SR(r) } ; { BI , SR(r) } ; { SP(p) , (r→p)} > (BI.4)
< { BI , SP(p) } ; { BI , SP(p)} ; { SR(r) , (r→p)} > (BI.5)
< { BI , SR(r) , SP(p) } ; { BI , SR(r) , SP(p) } ; { (c→p)} > (BI.6)

Productions *(BI.4)*, *(BI.5)* and *(BI.6)* allow a requestor to be bound to multiple providers and a provider to be bound to multiple requestors. The architecture example presented in figure 4 could be obtained by applying the following productions: *BI.1*, then *BI.3* with r=r1 and p=p1, then *BI3* with r=r3 and p=p3, then *BI.4* with r=r1 and p=p2, then *BI.5* with r=r2 and p=p1, then *BI.6* with r=r2 and p=p3 and finally *BI.2*.

3.2 Orchestrated Interaction (OI) architecture style

The above BI architectural style provides a simple interaction model involving a single independent operation on a single service provider. In the case of interactions involving multiple sequences of operations, BI style is not sufficient to specify the associated architectures. Indeed, to manage the workflow of these services, we may need to execute invocations in conformance with their ordering constraints. We may also need to root service requests to the appropriate provider and to root the service provider response to the right service requestor. Orchestration [5, 30, 31] addresses the issue of creating business processes at the message level including business logic, task execution order and transactional aspects. We introduce, in this section, in order to consider orchestration issues, the orchestrator element and define the orchestrated interaction style.

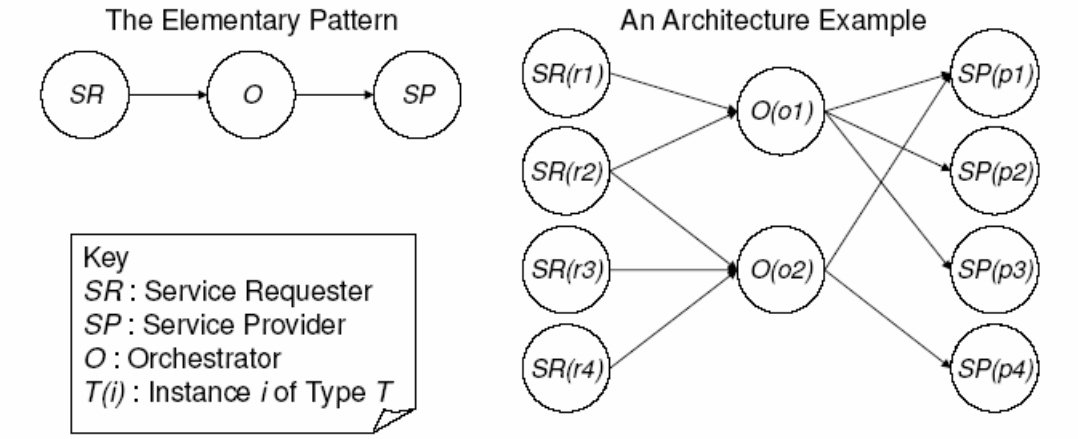

Figure 5. Consistent architecture instances of the orchestrated interaction style

The OI architectural style is described by the following graph grammar:
<AX; {OI}; {SR;O; SP}; P_{OI} > where P_{OI} is the set of the grammar productions defined below:

< { AX } ; { } ; { OI } > (OI.1)
< { OI } ; { } ; { } > (OI.2)
< { OI } ; { OI } ; { SR(r) , O(o) , SP(p) , (r→o) , (o→p) } > (OI.3)
< { OI , O(o) } ; { OI , O(o) } ; { SR(r) , (r→o) } > (OI.4)
< { OI , O(o) } ; { OI , O(o) } ; { SP(p) , (o→p) } > (OI.5)
< { OI , O(o) , SR(r) } ; { OI , O(o) , SR(r) } ; { (r→o) } > (OI.6)
< { OI , O(o) , SP(r) } ; { OI , O(o) , SP(r) } ; { (o→p) } > (OI.7)
< { OI , SR(r) } ; {OI , SR(r)} ; { O(o) , SP(p) , (r→o) , (o→p) } > (OI.8)
< { OI , SP(p) } ; { OI , SP(p) } ; { SR(r) , O(o) , (r→o) , (o→p)} > (OI.9)

The defined graph grammar allows an orchestrator to coordinate multiple providers *(OI.4)* and multiple service requestors *(OI.5)*. The OI grammar also involves the general case where an orchestrator may share its service requestors and providers with other orchestrators (resp. *(OI.6&OI.8)* and *(OI.7&OI.9)*). The architectural style restricting the framework to independent orchestrated patterns can be obtained using the same graph grammar as for the OI style by removing the *OI.6* to *OI.9* productions.

3.3 Composite architecture styles

The OI architectural style was defined to address the problem of service interactions that consist of sequence operation invocations. In this section, we consider the composition scenario where the invocation of a service operation involves different operations offered by different other services. One of the key differences between composition [32, 34] and orchestration is related to the fact that composition is concerned with internal implementation of operations [35]. Unlike orchestration protocols which are public documents described by standardized languages such as BPEL4WS, the specification of composite services is generally done within a single company and the composition schema still transparent from the client perspective for privacy considerations. We define, here, a composite service style by introducing a new interdependency link between service providers. We distinguish, now, two kinds of links: invocation links (denoted by the continuous arrow symbol "→") and composition links (denoted by the dashed arrow symbol `-->'). The second kind of links expresses the composition relation between service providers and their containing composite services.
The graph grammar producing patterns of composite service providers is described as follows: <AX; {Comp} ; {CP; SP}; P_{Comp}> where P_{Comp} is the set of the following grammar productions:

< {AX} ; { } ; {Comp}> (Comp.1)
< {Comp} ; { } ; { } > (Comp.2)
< {Comp} ; {Comp} ; {SP(p)} > (Comp.3)
< {Comp} ; {Comp} ; {CP(p1) , SP(p2) , p1 --> p2} > (Comp.4)
< {Comp , CP(p1)} ; {Comp , CP(p1)} ; {SP(p2) , p1 -->p2} > (Comp.5)
< {Comp , CP(p1) , SP(p2)} ; {Comp , CP(p1) , SP(p2)} ; {p1 -->p2} > (Comp.6)
< {Comp , CP(p1)} ; {Comp , CP(p1)} ; {CP(p2) , p2 --> p1} > (Comp.7)
< {Comp , CP(p1) , CP(p2)} ; {Comp , CP(p1) , CP(p2)} ; {p2 -->p1} > (Comp.8)
< {Comp , SP(p2)} ; {Comp , SP(p2)} ; {CP(p1) , p1-->p2} > (Comp.9)

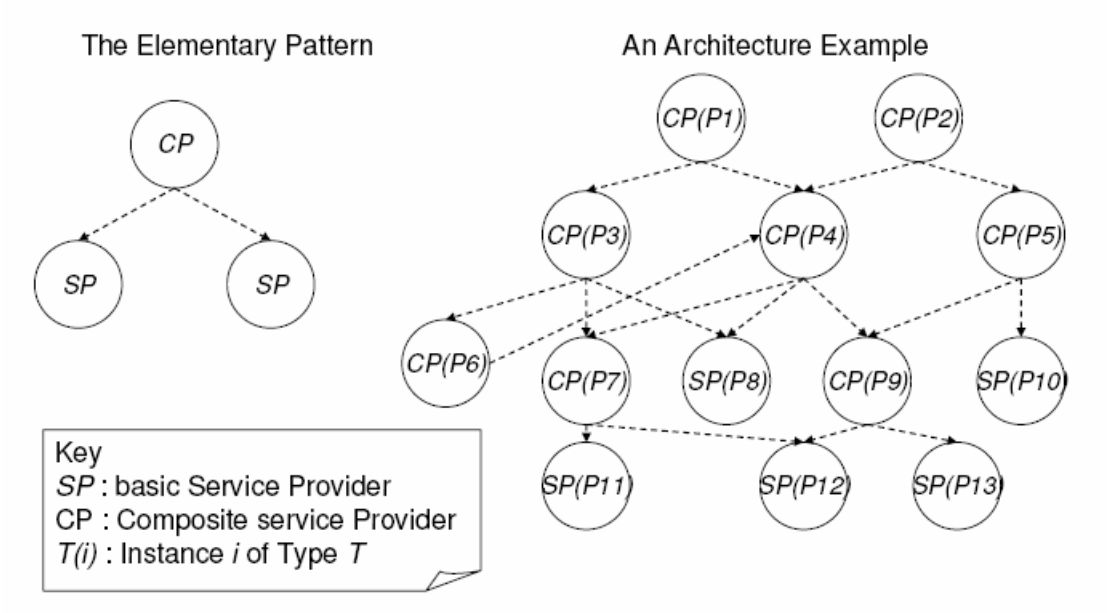

Figure 6 . Elementary composite service and an example of a conforming associated architecture instance of the composite style.

The defined graph grammar, for composite architectures, allows a composite provider to be composed of several atomic providers *(Comp.4&Comp.5)* and several composite providers *(Comp.7&Comp.8)*. It also implies that multiple levels of composition may be considered *(Comp.7&Comp.8)* but that the lowest level contains exclusively atomic services (because of *Comp.3* and *Comp.4* and because of absence of a production of type *< {Comp ,CP(p1)} ; {Comp, CP(p1)} ; {CP(p2) , p1 --> p2} >*. This production is different from production *Comp.7* where we add a composite service of a higher level. An atomic or a composite service may compose different composite services (respectively *Comp.6&Comp.8*).

In the sequel, we extend the styles introduced for the BI and OI architectures to consider the composition issue. According to whether we place the architecture within the framework of the BI or the OI architectures, we obtain two new composite architecture styles.

3.3.1 Composite Basic Invocation (CBI) architecture style

The CBI architecture style is described by the graph grammar: <AX;{CBI};{SR; SP;CP};P_{CBI}> where P_{CBI} is the set of the following grammar productions:

< {AX} ; { } ; {CBI}>	(CBI.1)
< {CBI} ; { } ; { } >	(CBI.2)
< {CBI} ; {CBI} ; {SP(p)} >	(CBI.3)
< {CBI} ; {CBI} ; {CP(p1) , SP(p2) , p1 -->p2} >	(CBI.4)
< {CBI , CP(p1)} ; {CBI , CP(p1)} ; {SP(p2) , p1-->p2} >	(CBI.5)
< {CBI , CP(p1) , SP(p2)} ; {CBI , CP(p1) , SP(p2)} ; {p1-->p2} >	(CBI.6)
< {CBI , CP(p1)} ; {CBI , CP(p1)} ; {CP(p2) , p2 -->p1} >	(CBI.7)
< {CBI , CP(p1) , CP(p2)} ; {CBI , CP(p1) , CP(p2)} ; {p2 -->p1} >	(CBI.8)
< {CBI , SP(p2)} ; {CBI , SP(p2)} ; {CP(p1) , p1-->p2} >	(CBI.9)
< {CBI , SP(p)} ; {CBI , SP(p)} ; {SR(r) , r→p}>	(CBI.10)
< {CBI , SP(p) , SR(r)} ; {CBI , SP(p) , SR(r)} ; {r→p}>	(CBI.11)

< {CBI , CP(p)} ; {CBI , CP(p)} ; {SR(r) , r→p}> (CBI.12)
< {CBI , CP(p) , SR(r)} ; {CBI , CP(p) , SR(r)} ; {r→p}> (CBI.13)

We notice that the previous grammar contains both the composite grammar and the BI grammar: productions {*CBI.i*, i = 1…9} are equivalent to productions *Comp.i* and productions *BI.1*, *BI.2*, *BI.5*, and *BI.6* are respectively equivalent to *CBI.1*, *CBI.2*, *CBI.10*, and *CBI.11* while *BI.3* is equivalent to the composition of productions *CBI.3* and *CBI.10* and *BI.4* is equivalent to the composition of productions *CBI.3* and *CBI.11*. In addition to the requirements and the constraints related to the composite and to the direct interaction styles (which are still valid because of productions inclusion and equivalence), the graph grammar of the CBI style allows a requestor to interact with a provider at any level of composition *(CBI.12 & CBI.13)* including the lowest level *(CBI.10 & CBI.11)*. This specification also allows requestors to interact with different services belonging to different composition levels *(CBI.10 & CBI.11 & CBI.12 & CBI.13)*.

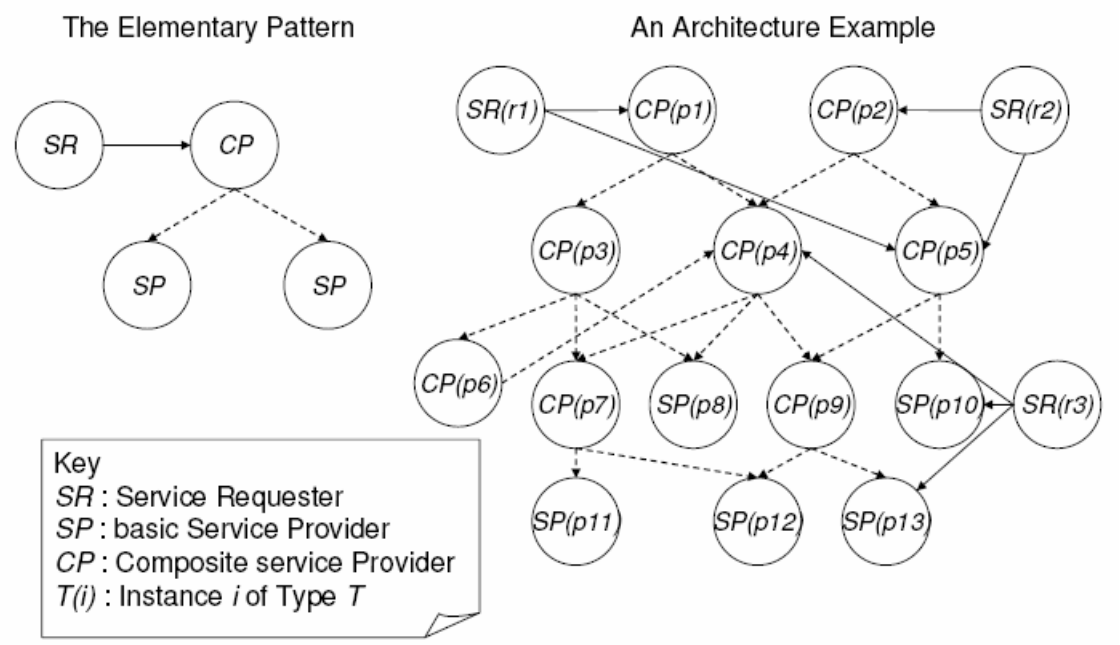

Figure 7. The elementary pattern and an example of architecture that conforms to the composite basic interaction style.

3.3.2 Composite Orchestrated Invocation (COI) architecture style

The composite orchestrated cooperation architecture style is described by:
<AX; {COI}; {SR;O;CP; SP}; P_{COI} > where P_{COI} is the set of the following productions:

< {AX} ; { } ; {COI}> (COI.1)
< {COI} ; { } ; { } > (COI.2)
< {COI} ; {COI} ; {SP(p)} > (COI.3)
< {COI} ; {COI} ; {CP(p1) , SP(p2) , p1 -->p2} > (COI.4)
< {COI , CP(p1)} ; {COI , CP(p1)} ; {SP(p2) , p1-->p2} > (COI.5)
< {COI , CP(p1) , SP(p2)} ; {COI , CP(p1) , SP(p2)} ; {p1-->p2} > (COI.6)
< {COI , CP(p1)} ; {COI , CP(p1)} ; {CP(p2) , p2 -->p1} > (COI.7)
< {COI , CP(p1) , CP(p2)} ; {COI , CP(p1) , CP(p2)} ; {p2 -->p1} > (COI.8)

< {COI , SP(p2)} ; {COI , SP(p2)} ; {CP(p1) , p1-->p2} > (COI.9)
< {COI , SP(p)} ; {COI , SP(p)} ; {SR(r) , O(o) , r→o;o→p} > (COI.10)
< {COI , SP(p) , O(o)} ; {CDI , SP(p) , O(o)} ; {o→p} > (COI.11)
< {COI , CP(p)} ; {COI , CP(p)} ; {SR(r) , O(o) , r→o;o→p} > (COI.12)
< {COI , CP(p) , O(o)} ; {COI , CP(p) , O(o)} ; {o→p}> (COI.13)
< {COI , orch(o)} ; {COI , O(o)} ; {SR(r) , r→o}> (COI.14)
< {COI , SR(r) , O(o)} ; {COI , SR(r) , O(o)} ; {r→o}> (COI.15)

Productions from *COI.1* to *COI.9* are respectively equivalent to productions from *Comp.1* to *Comp.9* and the considerations and constraints related to the composite style are still consequently valid for the COI style. Considering the COI graph grammar we allow a unique orchestrator to coordinate many composite services *(COI.13)*, many atomic services *(COI.11)* and many service requestors *(COI.14)*. An orchestrator may share its atomic providers *(COI.11)*, its composite providers *(COI.13)*, and its service requestors *(COI.15)*.

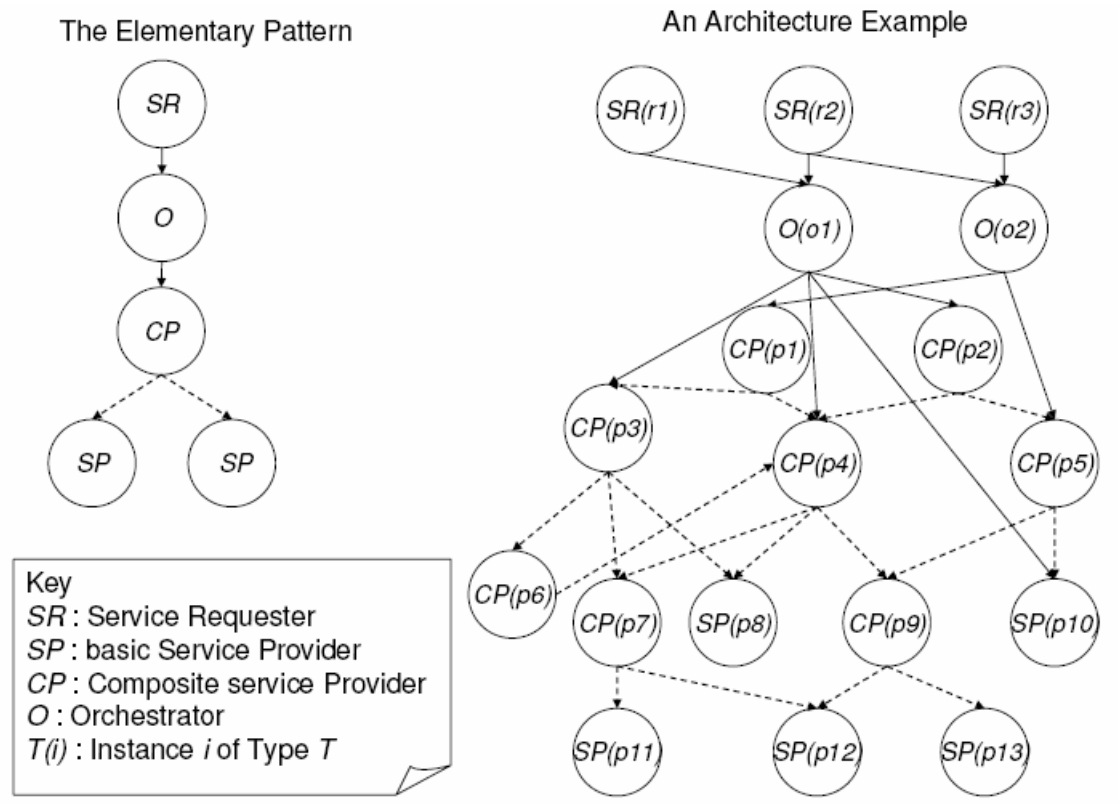

Figure 8. The elementary pattern and an architecture example of the composite orchestrated interaction style.

4 Conclusion

In this paper, we have defined a formal framework for service-oriented architectural style description. We have proposed the graph grammar formalism to specify the associated architectural styles. The presented elementary styles classification rises from considering two major concepts which are service composition and orchestration. We specified the basic interaction style, the orchestrated interaction style, the composite basic interaction style and the composite orchestrated style.

Moreover the proposed approach also makes it possible to check the conformance of an architecture instance to a given style. It is possible to verify if the graph representing a given ad-hoc architecture can be produced from the axiom by generating partially the development tree to overcome scalability problems. We have implemented a graph transformation engine in C++ that supports executing graph grammar productions. We are presently working on applying our approach for web services in the framework of the IST WS-DIAMOND project. Our models will be used and extended to check architecture conformance in case of reconfiguration as a repair action for QoS management.

References

1. IEEE Recommended practice for architectural description of software-intensive systems, In *IEEE Std 1471-2000,* pages i--23, 2000.
2. M. Shaw, R. Deline, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions On Software Engineering*, 21(4):314--335, April 1995.
3. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Untraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2), March-April 2002.
4. *Business Execution Language for Web Services Version 1.1*. http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel/.
5. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46--52, october 2003.
6. M. Tian, A. Gramm, H. Ritter, and J. Schiller. Efficient selection and monitoring of QoS-aware web services with the WS-QoS framework. In *2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*, ISBN 0-7695-2100-2, pages 152--158, Beijing, China, September 2004.
7. A. Wolf, D. Heimbigner, J. Knight, P. Devanbu, M. Getz, and A. Carzaniga. Bend, Don't Break: Using reconfiguration to achieve survivability. In *The* 3*rd IEEE Information Survivability workshop*, Kluwer Academic Publishers , ISBN 1-4020-7043-8 2002, Boston, Ma, USA, October 2000.
8. Y. Ren, D. Bakken, T. Courtney, M. Cukier, A. Karr, P. Rubel, C. Sabnis, W. Sanders, R. Schantz, and M. Seri. AQUA: An adaptative architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31--50, January 2003.
9. I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in Hadas. *IEEE Transactions on Software Engineering*, 27(9):769--787, September 2001.
10. T. Nadour, N. Simoni, and G. Du-Chene. Towards dynamic vertical self-organization. In *12th IEEE International Conference on Networks (ICON'04)*, volume 1, ISBN 0-7803-8783-X, pages 432--436, Singapore, November 2004.
11. J. Engelfriet and G. Rozenberg. Handbook of Graph Grammars and Computing by Graph Transformation, ISBN 981-02-2884-8, chapter Node Replacement Graph Grammars, pages 1--94. World Scientific Publishing, 1997.
12. R. Allen. A Formal Approach to Software Architecture. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.
13. D. Garlan. Wright Web Site, *Carnegie Mellon University*. http://www-2.cs.cmu.edu/~able/wright.
14. R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, J.E Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390--406, 1996.
15. R.N. Taylor, N. Medvidovic, and P. Oreizy. C2SADL Web Site, *ISR-University of California, Irvine*. http://www.isr.uci.edu/architecture/c2.html.
16. J. Magee, N. Dulay, S. Eisenbagh, and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conference, ESEC'95*, LNCS 989, ISBN 3-540-60406-5, Springer-Verlag, Barcelona, Spain, September 1995.

17. J. Magee and J. Kramer. Darwin Web Site, *Imperial College London*. http://wwwdse.doc.ic.ac.uk/Research/architecture.html.
18. G. Rozenberg, editor. Handbook of Graph Grammars and Computing by Graph Transformation, ISBN 981-02-2884-8. World Scientific Publishing, 1997.
19. H. Ehrig and H.-J. Kreowski, editors. Graph grammars and their application to computer Science, ISBN 3-540-54478-X. Springer-Verlag, 1990.
20. L. Wiskott, J. Fellous, N. KrÄuger, and C. von der Malsburg. Face recognition by elastic bunch graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):775--779, 1997.
21. C. Kotropoulos, A. Tefas, and I. Pitas. Frontal face authentication using morphological elastic graph matching. *IEEE Transactions on Image Processing*, 9(4):555--560, 2000.
22. H. Bunke. Graph matching for visual object recognition. *Saptial vision*, 13:333-- 340, 2000.
23. N.M. Nasrabadi and W. Li. Object recognition by a hopfield neural network. *IEEE Transactions on systems, man, and cybernetics, 21(6):1523--1535, 1991.*
24. J. Llados and G. Sanchez. Symbol recognition using graphs. In *IEEE International Conference on Image Processing*, volume 3, pages II- 49--52, Barcelona, Spain, September 2003.
25. S.W. Lu, Y. Ren, and C.Y. Suen. Hierarchical attributed graph representation and recognition of handwritten Chinese characters. *Pattern Recognition*, 24:617-- 632, 1991.
26. H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single PushOuts. In *4th International Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, ISBN 3-540-54478-X, pages 24--37, Bremen, Germany, March 1990. Springer-Verlag.
27. D. Le Metayer. Describing software architecture styles using graph grammars. *IEEE Transactions On Software Engineering*, 24(7):521--533, July 1998.
28. D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architectures and styles with graph grammars and constraint solving. In *1st Working IFIP Conference on Software Architecture*, ISBN 0-7923-8453-9 , Kluwer pages 127--142, San Antonio, TX, USA, February 1999.
29. H. Fahmy and R. Holt. Using graph rewriting to specify software architectural transformations. In *15th IEEE international Conference on Automated Software Engineering*, ISBN 0-7695-0710-7, pages 187--196, Grenoble, France, September 2000.
30. J. Estublier and S. Sanlaville. Business processes and workflow coordination of web services. In *2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, ISBN 0-7695-2073-1, pages 85--88, Hong Kong, April 2005.
31. T. Lemniotes, G.A. Papadopoulos, and F. Arbab. Coordinating web services using channel based communication. In *Computer Software and Applications Conference COMPSAC*, volume 1, ISBN 0-7695-2413-3, pages 486--491, Hong Kong, September 2004.
32. N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51--59, November-December 2004.
33. S.A. Chun, V. Atluri, and N.R. Adam. Policy-based web service composition. In *14th International Workshop on Research Issues on Data Engineering Web Services for E-Commerce and E-Government Applications RIDE-WS-ECEG'2004,* ISBN 0-7695-2095-2, pages 85--92, Boston, USA, March 2004.
34. J. Liu, J. Cui, and N. Gu. Composing web services dynamically and semantically. In *IEEE International Conference on E-Commerce Technology for Dynamic E-Business*, ISBN 0-7695-2206-8, pages 234--241, Beijing , China, September 2004.
35. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. Web Services. Concepts, Architectures and Applications, ISBN 3540440089, Springer-Verlag, 2004.