A SOA-Based Architecture Framework

Wil van der Aalst[1], Michael Beisiegel[2], Kees van Hee[1], Dieter König[3], and Christian Stahl[1]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
W.M.P.v.d.Aalst@tm.tue.nl, {k.m.v.hee, c.stahl}@tue.nl
[2] IBM Application and Integration Middleware (AIM) Division
Somers, New York, USA
mbgl@us.ibm.com
[3] IBM Böblingen Laboratory
Schönaicher Strasse 220, 71032 Böblingen, Germany
dieterkoenig@de.ibm.com

Abstract. In this paper we present first results of a SOA-based architecture framework. The architecture framework is required to be close to industry standards, especially to service component architecture (SCA), language independent (i.e. it is adoptable) and the building blocks of each system, activities and data, are first class citizens. We present a *meta model* of the architecture framework and discuss its concepts in detail.

1 Introduction

Since the early days of computer science it is well-known that mastering the complexity of large (software) systems is the major challenge. On the level of programming many methods and techniques, such as structured programming, stepwise refinement, functional programming, logical programming and object-oriented programming, were developed. Another attempt to master complexity was to introduce more levels of abstraction in the development. So techniques for structural analysis and design were introduced and later formal specifications with languages as Z [1] and Vienna Development Method (VDM) [2].

One very successful approach for handling complexity is *modularization*. Already from the beginning of computer science, programming languages have facilities to split systems into modules that hide details you do not need when you use or reuse the module. Modules have different names like procedure, subroutine, function, class, object, capsule or component. There are many differences in the properties of these modules concerning the way they are invoked, if they are stateless and if they have side effects. The principle of *compositionality* is one of the most wanted requirements for modular systems: A collection of modules that are properly connected to each other, should behave as one module itself. Often we require more: If we have verified that all modules of a system satisfy

Dagstuhl Seminar Proceedings 06291
The Role of Business Processes in Service Oriented Architectures
http://drops.dagstuhl.de/opus/volltexte/2006/827

some property and they are connected properly, then the system as a whole should have the same property. In object-oriented programming modules, called classes or objects, are first class citizens. During the last decade modularization is considered as the most important feature of a design of a system. In the rest of this paper we will use the term *component* for a module.

At a high level a system is described by its components and their relationships. Such a description is called the *architecture* of a system. Software architects are the most wanted specialists in the software industry. There are several languages to define components and to glue them together. There are also different *architectural styles*. In this paper we focus on a style based on the *service-oriented architecture* (SOA). SOA is seen as one of the key technologies to enable flexibility and reduce complexity in software systems (see [3]). Today SOA is a set of ideas for architectural design and there are some proposals for SOA-frameworks, including an architectural language, the *service component architecture* (SCA) [4] and software tools to design systems in the SOA-style.

In this paper we present first results of a SOA-based architecture framework which is close to industry standards, especially to SCA. The architecture framework is language independent, i.e. it is adoptable and the building blocks of each system, activities and data, are first class citizens. We present this architecture framework by means of a *meta model* and discuss its concepts in detail.

The outline of the paper is as follows: In Sect. 2 we sketch the practice of component-based software systems. Next, in Sect. 3, we introduce software architectures and in particular the service-oriented architecture. Based on SOA we formulate a set of requirements for a SOA-based architecture framework and present an architectural framework covering most of these requirements in Sect. 4. Finally, Sect. 5 summarizes the paper and describes how this work will be continued.

2 The component-based world

The idea to use components in software development was already published by McIlroy in 1968 [5]. In this paper McIlroy presented his idea of mass-produced software components. Even though much research was achieved since then, today there is still no universally accepted definition of what a component is. Most cited is the definition of Szyperski [6]: "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." In [7] Messerschmitt and Szyperski present a more enhanced definition: A software component is a reusable module suitable for composition into multiple applications. A component fulfills five properties:

- it can be used in multiple projects,
- it is designed independently of any specific project and system context,
- it can be composed with other components,
- it is encapsulated, i.e. only the interfaces are visible and the implementation cannot be modified,

– it can be deployed and installed as an independent atomic unit and later upgraded independently of the remainder of the system.

As there is no consensus about what a component is there is also no agreement on the *granularity* of components. A component can be *small grained*, like a graphical object in a user interface, for instance or *coarse grained*, like a debtors register in an Enterprise Resource Planning (ERP) system, for instance.

A component has four different interfaces: (1) a *software interface* to compose the component with other software components, (2) a *user interface* which allows the communication between the component and a human user, (3) a *configuration interface* that is used to configure the component, e.g. set parameters and (4) a *monitoring interface* to provide runtime diagnostic statements of the component's internal, e.g. values of the messages that are sent or received by the component. So far components often have neither a user nor a monitoring interface, but in near future they will become an inherent part of a component's interface.

Components can be classified based on their *functionality*: There are *application specific* and *generic* components. A general ledger component or an SAP component is an example of an application specific component whereas a document manager or a workflow engine is a generic component. A synonym for generic and application specific, respectively, is *horizontal* and *vertical*, respectively, because components address either a horizontal or a vertical market [6]. A vertical market, also known as a niche market, meets the interest of their customers by offering custom-tailored products. In contrast, a horizontal market tries to attempt most of the needs of a broader community of customers. Another classification of components is based on the *configuration* of their parameters. In a *predefined* (or hard-wired) component the version is hard-coded and the parameters are selected from an option list. An inventory control rule like FIFO or LIFO would be an example. In contrast, the parameters in a *programmable* component are database schemes, process models or business rules.

A system which is developed by composing components is a *component-based system*. Component-based systems will evolve in an *organic* way. There will never be a total renewal nor an upgrade of the overall system. Instead components will be replaced periodically by better ones, e.g. because the performance was not good enough anymore. Adding new functionality to the system will also be realized by either adding new components or replacing components by better ones. This will save the total cost of ownership of component-based systems.

At the time the market expectations are low, but during the next 5-10 years they will increase. Reasons are the enormously growing of software systems during the last decades and the globalization which demands greater flexibility – in particular from the software systems. So for today's IT it is most challenging to respond quickly to new business requirements while reducing the IT costs. One possibility to overcome this problem is to buy software (components) from third parties. This is in fact usually cheaper and more effective than doing the work itself. Therefore the software development in many companies has been outsourced. Building software from reusable components rather than from scratch

is another possibility to decrease IT costs and develop more flexible software systems.

Components may have a vendor. Vendors will compete with each other to offer the best functionality. For instance, they will offer components with different levels of quality and/or functionality at a different level of price. Furthermore components will be customized. For this purpose, vendors might offer *compound components*, i.e. prepacked solutions or combinations of components with parameters that can be used as a new predefined component. For example, the software of set-top boxes offered by telecommunication companies consists of components. These components have parameters, e.g. video format and resolution, that are initially configured.

In the component-based world the architecture becomes an important role. Firstly, the architecture can be used as a blue print for the development of a component. For example, a component can be seen as a black-box, i.e. only the interface is visible, or the internal details of the component are visible. As the architecture supports such different views on a component, it may help to develop software in a more structured way. Secondly, an architecture facilitates the work distribution in the software development process: If the interfaces are specified, different components can be developed in parallel and independent of each other. Simulation, testing and verification of components is an important, but very difficult task. Components are replaced by other components or added to the system and this change must preserve the properties of the system. We further believe that in the near future customers will require a guarantee that at least some safety properties hold in a component. Such a safety property might be for example: "if the component fails to function, it will never jeopardize the overall system". Therefore (computer-aided) verification of components becomes increasingly important. Finally, as mentioned above, a component-based system is not upgraded, but components have to be added and exchanged during the runtime of the system. To this end, approaches are needed that incorporate this requirement into the architecture.

3 Architecture frameworks

We are shifting our attention now from components and component-based systems to *software architectures*. We start with a definition of software architectures in general and introduce then the service-oriented architecture.

3.1 Software architectures

Just like for the term "component" everyone knows roughly what a software architecture is, but there is also no universally accepted definition. We therefore start this section with two definitions from the literature. Based on these definitions we elaborate our own definition.

The first definition of software architecture, we present, is a modern one [8]: "The software architecture of a program or computing system is the structure or

structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." The second, but also well-known definition is presented by the IEEE Standards Association for Recommended Practice for Architectural Description of Software-Intensive Systems: "Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."

Referring to both definitions an architecture shows the elements of the system, i.e. in case of a component-based system the components, and their relationships. We restrict us to "the structure of the system" or "the fundamental organization of a system" and we define this as a set of *views*. A view is a model of a part or an aspect of a system. Views should be *consistent*, i.e. no view should contradict another view of the system. Furthermore views should also be *complete*. That means, every property of the system should be modelled by at least one view. As a view is a model, i.e. a simplification of the system, it is therefore possible that the view does not make a statement whether a specific property holds or not.

Based on this facts we elaborate the definition of a software architecture to the following which is used throughout the paper: "An architecture of a system is a set of descriptions that present different views of the system. These views should be consistent and complete. Each view models a set of components of the system, one or more functions of each component and the relationships between these components." For example, a view could show a data model of some components and the inheritance relationship between the components.

A specification to organize and develop a software architecture in a specific style is an *architecture framework*. Some examples for software architecture frameworks are UML, CORBA, Turbine, Avalon and SCA to name a few. The Unified Modeling Language (UML)[4] serves the (graphical) description of models. It can be used to describe the structure, e.g. by help of class diagrams or the behavior, e.g. by help of use-case diagrams or sequence diagrams. Common Object Request Broker Architecture (CORBA)[5] enables the interaction of heterogenous applications by providing an interface definition language, object models and communication protocols. Apache's Turbine[6] is a servlet-based framework to develop web applications. Avalon[7], also from the Apache foundation, is a framework for building server side applications. It allows to create components, manage them and use them in applications. Service Component Architecture (SCA) [4] provides a programming model for building applications, components and systems based on SOA. The programming model describes the relationships, the composition and the deployment of components. It also applies infrastructure capabilities to components, such as security and transaction.

[4] www.uml.org
[5] www.corba.org
[6] http://jakarta.apache.org/turbine/
[7] http://avalon.apache.org

3.2 Service-oriented architecture

The example architecture frameworks presented above reveal that the use of reusable software components becomes more and more popular. One of today's most popular architecture frameworks is the *service-oriented architecture* (SOA). SOA is seen as one of the key technologies to enable flexibility and reduce complexity in software systems (see [3]). It follows the paradigm to explicitly *separate* an implementation from its interface. Such an interface is *well-defined*, i.e. it is based on standards like the Web Service Description Language (WSDL) [9]. Implementation and interface form together a component. In a SOA a component is called *service*. Services are independent of applications and the computing platforms on which they run. Services in a SOA can be connected without having knowledge of their technical details, they are *loosely coupled*. To connect services during runtime SOA supports *dynamic binding*. For the message exchange between services standardized communication protocols are used. Further, all the standards used in a SOA are *extensible*, i.e. they are not limited to current standards and technologies.

SOA distinguishes three different roles of services: *service provider*, *service consumer* and *service registry*. It postulates a general protocol for interaction: A service provider registers at the service registry by submitting information about how to interact with its service. The service registry manages such information about all registered service providers and allows a service consumer to find an adequate service provider. Then, the service of the provider and the service of the consumer may bind and start interaction.

A service has two kinds of interfaces: *required* and *provided* interfaces. Required interfaces specify which services are used by the service. In contrast, provided interfaces specify which services are offered by the service. So in terms of the service roles in SOA a service plays the consumer's role at the required interfaces and at the provided interfaces it plays the provider's role.

Apart from these technical paradigms services in SOA are also based on an *economical paradigm*: a service is comparable with a business unit. So it should create *value* for its environment. Therefore the two kinds of interfaces can be seen as the *buy side* and the *sell side* of the service. On the buy side a service behaves as a service consumer or *client* and buys other services. On the sell side a service behaves as the service provider and offers its service to other services. Services are operating as actors on a market place. This means, they offer their services to any consumer who needs it and they buy services from providers with the best value proposition. So both parties publish their needs and offerings at a repository, respectively.

4 A SOA-based architecture framework

Starting from the service-oriented architecture we collect and discuss requirements for a SOA-based architecture framework. Then we present a meta model of our architecture framework. We introduce its concepts and show that it covers most of the collected requirements.

4.1 General requirements

The following list of required features is distilled from the variety of proposals for architecture frameworks. They can be seen as *general requirements* that should be satisfied by an architectural framework:

- The basic concept of an architecture framework should be a *component*.
- A component should have an *interface* with its environment. The interface should facilitate an easy "plug and play" of components.
- A component should also have an internal *structure* that consists of
  - a partially ordered set of *activities*. Activities describe the component's behavior.
  - zero or more *data elements*, which are global to the component. Data elements can be used to configure the component.
- A component should have a mechanism to catch and handle faults. It should also support an orthogonal mechanism, namely the roll back of already executed activities.
- A component should offer a monitoring service which logs the execution of the component. For this purpose, the monitoring interface of the component (see Sect. 2) is used.
- It should support *relationships* between components:
  1. Interaction relationships with facilities for *synchronous* and *asynchronous* communication by *message exchange* on the one hand and *shared data elements* on the other hand.
  2. Hierarchical relationships between components to support *refinement* as a design technique.
- The architecture framework is *open* in the sense that the following three elements are left undefined, they can be considered as "plug-ins":
  - A *process formalism* describes the ordering of the activities in a component. Such a formalism usually separates the activities from the data elements of a component. We allow for different formalisms, e.g. labelled transition systems, various kinds of process algebras, Petri nets, or industry standards as UML activity diagrams, Business Process Modeling Notation (BPMN) [10] and Business Process Execution Language for Web Services (BPEL) [11]. Programming languages like Java or C++ can also be used.
  - A *data model* defines the data elements, their types and their methods. We may use here algebraic formalisms such as abstract state machines [12] or industry standards as UML class diagrams, entity relationship model or the relation model. As industry standards are often not refined enough to provide all the relevant aspects of a data model we use the *object constraint language* (OCL) [15] to specify constraints between entities of a data model. It is also possible to use programming languages like Java or C++ as a data model.
  - A *language* defines the operations of activities. Here we may apply formal specification languages such as abstract state machines, B [13], VDM [2] or Z [1], but also programming languages like Java or C++.

– An architecture framework should have a formal semantics.
– It should be close to existing industrial (graphical) description techniques, such as the UML family, BPMN and process models as BPEL. With "close" we mean that it is easy to translate the models used in the architecture framework into existing industrial description techniques and vice versa.

Not every process formalism separates the activities from the data elements of a component. For example, BPEL provides a combined view on activities (BPEL activities) and on data elements (BPEL variables). So an architecture should offer both views, the combined view, showing data elements and activities together, and also a view restricted to the ordering of activities without data aspects.

4.2 A meta model of the architecture framework

In the following we present our SOA-based architecture framework. It is based on the general requirements presented in the last section. Figure 1 shows the meta model of the architecture framework in UML notation. After a general explanation of this meta model we have a more detailed look at the concepts of the architecture framework.

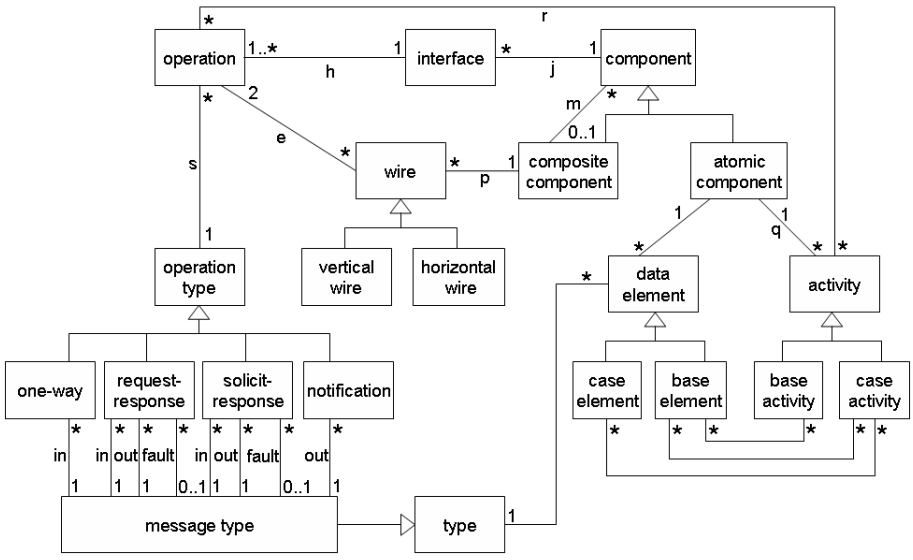

Fig. 1. Meta model of the architecture framework.

The basic concept of the architecture framework is a *component*. We distinguish between *atomic components* and *composite components*. An atomic component consists of a set of *activities* (q is the name of the relationship between

atomic component and activity in Fig. 1) and zero or more *data elements*. Every data element has a *type*. A composite component, however, describes hierarchical relationship between components. It is a container for components, i.e. it may contain atomic components and other composite components (see relationship m in Fig. 1).

Each component has one or more *interfaces* with its environment (see relationship j in Fig. 1). An interface consists of a set of *operations* (relationship h). An operation describes message exchange between two participants. It follows a given *operation type* (s is the name of the relationship between operation and operation type) which describes a message exchange pattern between the participants. We allow for the four operation types presented in the WSDL 1.1 specification [14]: *one-way*, *request-response*, *solicit-response* and *notification*. In general an operation type consists of zero or one input and/or zero or one output messages and an optional fault message. Each message has a *message type*. As can be seen from Fig. 1 the operation type of one-way and notification has an input and an output message, respectively. Operation types solicit-response and request-response define an input message, an output message and optionally a fault message. The difference between both operation types is the message order. In case of a solicit-response the component sends a message and then receives a correlated message (i.e. first an outgoing message, then an incoming message) whereas in case of a request-response the component first receives a message and then sends a correlated message (i.e. first an incoming message, then an outgoing message).

An activity may exchange messages through one or more operations (see relationship r in Fig. 1) with other components. It may also access to data elements of its atomic component by means of *method calls* (not visible in Fig. 1). Please note that these method calls may change the value of the data elements.

Besides wrapping components, a composite component also defines one or more *wires* (relationship p). In general, a wire connects interfaces of components. More detailed, a wire connects two operations depicted by relationship e. These two operations have either the same operation type or they have complementing operation types, e.g. one-way and notification. Wiring two operations with the same operation type can be seen as a *reference*. The operation of a component is propagated to the enclosing composite component. Such a wire is therefore called a *vertical wire*. In contrast, wiring two operations with complementing operation types shows the *connection* of two components. We call such a wire a *horizontal wire*. Most of these information about wiring operations cannot be derived from the meta model in Fig. 1. Later in this section we will therefore define the wiring using the object constraint language (OCL) [15].

Components support the concept of instances (not depicted in Fig. 1). If an atomic component is invoked via its interface, a *case*, i.e. a new instance, is created. This implies the creation of *case activities* and *case elements* which belong to exactly one case. Besides case activities and case elements, an atomic component may also contain *base activities* and *base elements*. These activities and elements are static, i.e. all cases can access to them. The set of case and

base activities is also called *case process* and *base process*, respectively. A base process starts when the component becomes live and stops when the component "dies". The life-cycle of a case process, in contrast, is restricted to its respective case.

The *state* of a component is determined by the value of data elements, the received or sent messages and the state of its activities.

Components and activities A component is required to consists of activities (see Sect. 4.1). From Fig. 1 it can be derived that a composite component does not (directly) fulfill this requirement. A composite component, however, can be flattened to a component that contains atomic components only. Consequently, it also contains a set of activities – the set of all activities of its inclosing atomic components and thus it still fulfills the above requirement.

As already mentioned, a component consists of case and base activities which form a base and a case process, respectively. Both processes can communicate with the component's environment by message exchange. Base and case processes are connected in only one direction. The case process can trigger the base process but not vice versa. Together, the activities form the process layer of the component. The process layer can be seen as a *workflow model*. The second layer of a component is the data layer. It can be seen as a *class model*. It consists of the component's data elements and methods. Methods are used to read and write the value of data elements. The relationships between activities and data elements in Fig. 1 show that process layer and data layer are connected.

An activity consists of one or more method calls and some additional logic (not visible in Fig. 1). The logic controls the method calls and evaluates their return values. In our architecture framework method calls are restricted to activities. As a consequence, no data element can access to another data element. Methods are defined in the same component as the activities and activities can only call methods which are defined in their component. Furthermore a method is never part of an activity. Otherwise there would be too much overhead, because a method can be used by several activities.

The meta model of our proposed architecture framework in Fig. 1 is very general. Thus it supports different process models. In the following we will present, as an example, two possible process models, BPEL and Petri nets.

The first example of a process model, we present, is the widely-used Business Process Execution Language for Web Service (BPEL) [11]. BPEL is a language for describing the behavior of business processes based on web services. For the specification of a business process, BPEL provides *activities* and distinguishes between *basic* and *structured* activities. A basic activity can communicate with other services by message exchange or it can manipulate data, for instance. A structured activity defines a causal order on the basic activities and can be nested in another structured activity. The structured activities include sequential execution, parallel execution, branching, and repeated execution. The most important structured activity is a *scope*. It links an activity to a transaction management subsystem and provides fault, compensation, and event handling.

In our meta model this subsystem is part of a case process, because it needs access to *all* data elements, e.g. to reverse some effects caused by the execution of an activity. For the sake of simplicity we restrict our view on BPEL to activities and do not go into the details of BPEL's advanced concepts like fault and compensation handling. The meta model in UML notation for this restricted part of BPEL is depicted in Fig. 2. So back to our meta model in Fig. 1, in case of BPEL an activity in our architecture framework is a synonym for a BPEL activity and a data element corresponds to a BPEL variable.

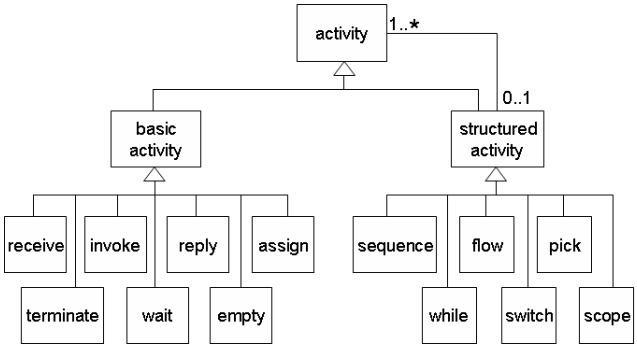

Fig. 2. Meta model for BPEL activities.

Petri nets are another example of a process model supported by our architecture framework. The formalism of Petri nets has been proven to be an adequate model for business processes (e.g. [16]). A Petri net (see e.g. [17] for a formal definition) is a bipartite graph. It consists of two different nodes, *places* and *transitions*, and (directed) *arcs*. An arc connects two nodes of different kind. That means, it connects either a place and a transition or a transition and a place.[8] The meta model for Petri nets in UML notation is presented in Fig. 3. In the meta model an arc that connects a place and a transition is called *P-T-Arc* and an arc that connect a transition and a place is called *T-P-Arc*. The entity *Transition* in this meta model coincides with the entity *activity* in the meta model of the architecture framework in Fig. 1.

Interface and wiring Now we have a more detailed look at the interface concept. The similarity of our interface concept to WSDL is intended. As WSDL is a widely-used industry standard, it is necessary that the interface definition in our architecture framework is at least adoptable to WSDL. The interface concept and the wiring of components is visualized in Fig. 4. Three components, c_1, c_2 and c_3, are depicted in Fig. 4. Component c_1 contains components c_2 and c_3. It

[8] Please note, there are Petri net classes which allow arcs between nodes of the same kind. For a general meta model for Petri nets we refer to Billington et al. [18].

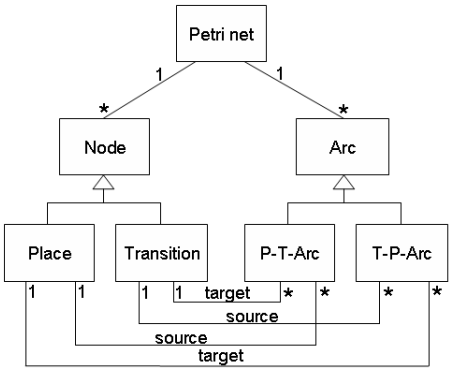

Fig. 3. Meta model for Petri nets.

defines four wires w_1,...,w_4. Each of them is depicted by a solicit line. Interfaces i_1,...,i_6 of the components are depicted by a dashed frame. A box inside of an interface visualizes an operation. Its operation type is depicted by one or two arcs inside the box. Interface i_2 has an operation with operation type one-way, i_3 notification, i_5 solicit-response and i_6 request-response. A horizontal wire connects operations of two components that have the same enclosing component. For instance, wire w_4 in Fig. 4 connects the operations of the interfaces i_5 and i_6. The interfaces are part of components c_2 and c_3 whose enclosing component is c_1. A special case is wire w_1 which connects operations of the interfaces i_2 and i_3. Both interfaces are part of component c_2 and consequently share the same enclosing component c_1. A vertical wire, in contrast, connects the operation of a component with an operation of its enclosing component. Wires w_2 and w_3 in Fig. 4 are examples for vertical wires. It can be seen that w_2 and w_3 connect operations of component c_2 with operations of its enclosing component c_1. A wire represents an abstract view on the communication of a component, i.e. it only shows the invocation dependencies of a component. As shown by wire w_1, it is not excluded to wire an operation to another operation of the same component.

From the relationship e in the meta model in Fig. 1 can be seen, one operation may be part of several wires. Thus at runtime it is possible to wire this operation to more than one operation. This is an important feature as we may want to wire a request-response operation with a notification operation and a one-way operation, for instance. However, it is also possible to wire a request-response operation with two solicit-response operations, for instance. Such a choice should be solved non-deterministically. Nevertheless, the framework should throw a warning to the developer, because this behavior might not be its intention. Furthermore it is also possible that two or more activities share one operation. As long as these two activities exclude each other, i.e. only one of them is activated and the other cannot be activated anymore (e.g. each activity is a different OR-branch), this is a valid behavior. Otherwise it is not valid. As there is no general answer whether such a construct is correct or erroneous, we

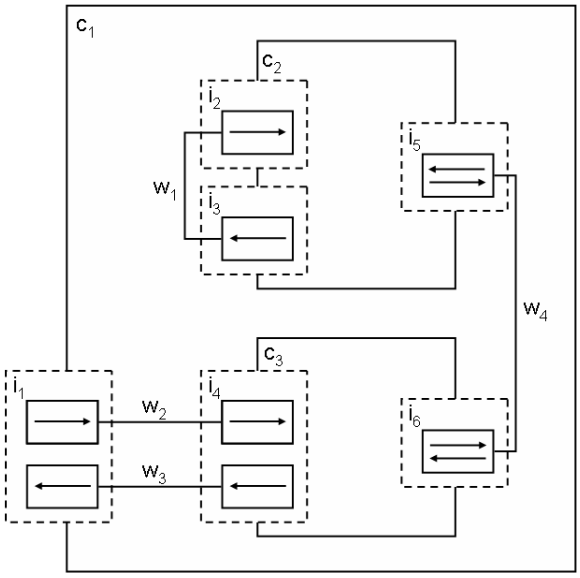

Fig. 4. Wiring of components.

do not exclude it from the level of our meta model. These examples show that the architecture framework firstly needs a formal semantics (as proposed in the general requirements in Sect. 4.1). The semantics defines the behavior of the architecture framework formally. Secondly, the architecture framework should also offer tool support to detect such design flaws.

Instantiation One of the most important concepts of an architecture framework is instantiation. In our architecture framework components can be instantiated (not shown in the meta model in Fig. 1). For the purpose of instantiation, atomic components distinguish between case activities and case elements on the one hand and base activities and base elements on the other hand. A case is created if a specific activity of the component receives a message. Then case activities and case elements of the component are copied and "fresh" values are assigned to the data elements. Base activities and base elements, in contrast, are independent of a case. That means, every case may execute base activities and may have access to base elements. Base activities and base elements are initialized once the component is initialized. After this initialization cases of the component can be created. The life-cycle of the base activities and base elements ends once the component is deactivated. A base element can therefore be seen as a parameter. Base activities are typically used for monitoring and configuration of a component.

If we think of a Petri net as a model for the base and case process, for instance, different cases can be expressed by different colors, where each color

represents the identifier of exactly one case. Thus the resulting Petri net is a Colored Petri net.

Messages sent to an atomic component need to be delivered not only to the correct operation, but also to the correct case of the component. Therefore we use the concept of *correlation* (known from BPEL, for instance). Every case gets an identifier. A message can be delivered to the correct case if the identifier can be determined from the content of the message. As it is possible to create an instance (by receiving a message), it is also possible to destroy an instance when it has been finished.

Constraints In the previous sections we have mentioned that the presented meta model is in some sense quite general. Therefore it is possible to design erroneous components (remember the example where two activities share one operation). We also motivated that it is not possible to exclude such behavior at the level of the meta model as it would also restrict some valid behavior. In the following we present five constraints that help to formalize in particular the concept of wiring. Consequently, the constraints will restrict the meta model in Fig. 1 as well as the interface concept depicted in Fig. 4. All constraints are invariants specified in the Object Constraint Language (OCL) [15]. OCL keywords are depicted in bold font. For the parameters used we refer to the relationships in Fig. 1.

1. Connection between activity and operation:

   context a : activity, o : operation **inv**:
   if $o.r \rightarrow select(a' \mid a' = a)$ **and** $a.r \rightarrow select(o' \mid o' = o)$ **then** $a.q = o.h.j$

   From the keyword **inv** it can be derived that this OCL expression is an OCL invariant. The invariant is introduced for the context of activity and operation. It specifies that the relation r (see Fig. 1) between an activity a and an operation o is redundant. This relation can be expressed using the component that encloses a, because the component is related to the operation o by its interface. As $o.r$ ($a.r$) is the set of activities (operations) that are related to o (a), we use the select operator to select activity a (operation o) from this set of activities (operations).
2. A horizontal wire connects components having the same enclosing component:

   Let f be a relationship between operation and wire (see Fig. 1) with $e \neq f$.
   context w: horizontal wire **inv**: $w.e.h.j.m = w.f.h.j.m = w.p$

   A horizontal wire w connects two operations. The component(s) of these two operations are embedded in the same composite component. $w.e.h.j.m$ specifies the composite component for the component of one operation and $w.f.h.j.m$ the composite component for the component of the other operation. The composite components are the same and finally $w.p$ specifies that the wire w is defined in this composite component. Wires w_1 and w_4 in Fig. 4 are examples of a horizontal wire.

3. A vertical wire connects a component with its enclosing component:

   Let f be a relationship between operation and wire (see Fig. 1) with $e \neq f$.
   context w: vertical wire **inv**: $w.e.h.j = w.p$ **and** $w.f.h.j.m = w.p$

   A vertical wire w connects two operations. One of these operations is defined in a component and the other operation in that component's enclosing component. The first part of the conjunction specifies the enclosing component and the second part its inclosed component. $w.p$ specifies that the wire is defined in the enclosing component. Wires w_2 and w_3 in Fig. 4 are examples of vertical wires.
4. A vertical wire connects two operations with the same operation type:

   Let f be a relationship between operation and wire (see Fig. 1) with $e \neq f$.
   context w: vertical wire **inv**:
   $w.e.s.oclIsKindOf(one-way) = w.f.s.oclIsKindOf(one-way)$ **or**
   $w.e.s.oclIsKindOf(notification) = w.f.s.oclIsKindOf(notification)$ **or**
   $w.e.s.oclIsKindOf(request-response) =$
   $w.f.s.oclIsKindOf(request-response)$ **or**
   $w.e.s.oclIsKindOf(solicit-response) =$
   $w.f.s.oclIsKindOf(solicit-response)$

   A vertical wire always connects two operation of the same operation type. Wires w_2 and w_3 in Fig. 4 are examples for vertical wires.
5. Horizontal wire connects two operations with matching operation types:

   Let f and g be relationships between operation and wire (see Fig. 1) with $e \neq f \neq g$.
   context w: horizontal wire **inv**:
   $w.e.s.oclIsKindOf(one-way) = w.f.s.oclIsKindOf(notification)$ **or**
   $w.e.s.oclIsKindOf(request-response) =$
   $w.f.s.oclIsKindOf(solicit-response)$ **or**
   $w.e.s.oclIsKindOf(request-response) =$
   $w.f.s.oclIsKindOf(one-way)$ **and** $w.g.s.oclIsKindOf(notification)$ **or**
   $w.e.s.oclIsKindOf(solicit-response) =$
   $w.f.s.oclIsKindOf(one-way)$ **and** $w.g.s.oclIsKindOf(notification)$

   A horizontal wire connects two operations of complementing operation type. Complementing operation types are one-way and notification as well as request-response and solicit-response (first three lines of the disjunction). The four remaining lines of the disjunction specify that a horizontal wire can also connect an operation with a synchronous operation type to two operations with asynchronous operation type. More detailed, a solicit-response operation can be wired with a notification operation and a one-way operation. The same holds for a request-response operation.

5 Outlook

In this paper we addressed our efforts in developing an architecture framework based on the service-oriented architecture (SOA). The architecture framework is

required to be language independent and close to industry standards, in particular to the service component architecture (SCA) [4]. We further require activities and data, which are the building blocks of each system, to be first class citizens.

The architecture framework was presented by means of a meta model. The main concepts include components as building blocks, interaction relationship between components by synchronous and asynchronous message exchange, hierarchical relationship between components and the concept of instantiation. Components have two connected layers, a process layer (consisting of activities) and an data layer (data elements and methods).

The development of the architecture framework, however, is not finished yet. We are currently working on the integration of an inheritance concept. Inheritance is an important concept as it allows the reuse of parts of the system. Furthermore we have to formally define our architecture framework. To this end, we are aiming at building a Colored Petri net (CPN) model.

As mentioned above, our architecture framework is required to be close to existing industry standards. Therefore further research includes a comparison of our architecture framework with the service component architecture, its data model, *service data objects* (SDO) [19] and BPEL [11] as a process model. Finally, having formalized the architecture framework we want to spend effort in the verification of components and as a long-term objective in the development of tools for the design and management of component-based systems.

References

1. Abrial, J.R., Schuman, S.A., Meyer, B.: Specification language. In: On the Construction of Programs. (1980) 343–410
2. Jones, C.: Systematic Software Development using VDM. Prentice Hall (1990)
3. High, R., Kinder, S., Graham, S.: IBM's SOA Foundation – An Architectural Introduction and Overview. Technical Report 1.0, IBM (2005)
4. Beisiegel, M., Booz, D., Edwards, M., Hurley, O., Ielceanu, S., Karmarkar, A., Malhotra, A., Marino, J., Nally, M., Newcomer, E., Patil, S., Pavlik, G., Rowley, M., Tam, K., Vorthmann, S., Waterman, L.: Service Component Architecture – Assembly Model Specification. SCA Version 0.96 draft 1, August 2006, BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue Wave, SAP, Software AG., Sybase, TIBCO (2006)
5. McIlroy, M.D.: Mass Produced Software Components. In Naur, P., Randell, B., eds.: Proceedings of NATO Software Engineering Conference. Volume 1., Garmisch, Germany (1968) 138–150
6. Szyperski, C.: Component Software–Beyond Object-Oriented Programming. Addison-Wesley and ACM Press (1998)
7. Messerschmitt, D., Szyperski, C.: Software Ecosystem–Understanding an Indispensable Technology and Industry. MIT Press (2003)
8. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. 2 edn. Addison Wesley Professional (2003)
9. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Candidate Recommendation 27 March 2006, W3C (2006)

10. White, S.A.: Business Process Modelling Notation Version 1.0. Technical report, Business Process Management Initiative (BPMI) (2004)
11. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, International Business Machines Corporation, Microsoft Corporation (2003)
12. Börger, E., Stärk, R.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
13. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
14. Christensen, E., Curbera, F., Meredith, G., Weeravarana, S.: Web Service Discription Language (WSDL) 1.1. W3C Note 15 March 2001, Ariba, International Business Machines Corporation, Microsoft (2001)
15. Object Management Group: UML2.0 Object Constraint Language (OCL) Specification. Specification, Object Management Group (OMG) (2003)
16. Aalst, W.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
17. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo (1985)
18. Billington, J., Christensen, S., Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, Technology, and Tools. In Aalst, W., Best, E., eds.: Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings. Volume 2679 of Lecture Notes in Computer Science., Springer (2003) 483–505
19. Beatty, J., Blohm, H., Boutard, C., Brodsky, S., Carey, M., Dubray, J.J., Ellersick, R., Ho, M., Karmarkar, A., Kearns, D., Brettevillois, R.L., Nally, M., Preotiuc-Pietro, R., Rowley, M., Smith, S., Yan, H.: Service Data Objects For Java. Specification, Version 2.01, IBM, BEA, Oracle, SAP, Siebel, Xcalia, Sybase (2005)