An SOA-based architecture framework

Wil M.P. van der Aalst

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, Eindhoven 5600 MB, The Netherlands
E-mail: W.M.P.v.d.Aalst@tue.nl

Michael Beisiegel

IBM,
Somers, New York, USA
E-mail: mbgl@us.ibm.com

Kees M. van Hee

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, Eindhoven 5600 MB, The Netherlands
E-mail: k.m.v.hee@tue.nl

Dieter König

IBM Böblingen Laboratory,
Schönaicher Strasse 220, Böblingen 71032, Germany
E-mail: dieterkoenig@de.ibm.com

Christian Stahl*

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, Eindhoven 5600 MB, The Netherlands
E-mail: c.stahl@tue.nl
*Corresponding author

Abstract: We present an Service-Oriented Architecture (SOA)-based architecture framework. The architecture framework is designed to be close to industry standards, especially to the Service Component Architecture (SCA). The framework is language independent 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. Through the framework, concepts of an SOA such as wiring, correlation and instantiation can be clarified.

Keywords: service-oriented architecture; SOA; architecture framework; service component architecture; SCA

Reference to this paper should be made as follows: van der Aalst, W.M.P., Beisiegel, M., van Hee, K.M., König, D. and Stahl, C. (XXXX) 'A SOA-based architecture framework', *Int. J. Business Process Integration and Management*, Vol. X, No. Y, pp.XXX–XXX.

Biographical notes: Wil M.P. van der Aalst is a Full Professor of Information Systems at the Technische Universiteit Eindhoven (TU/e) having a position in both the Department of Mathematics and Computer Science and the Department of Technology Management. He is also an Adjunct Professor at Queensland University of Technology (QUT) working within the BPM group. His research interests include workflow management, process mining, Petri nets, business process management, process modelling and process analysis. He has published more than 60 journal papers, 10 books (as author or editor), 150 refereed conference publications and 20 book chapters. He has been a co-chair of many conferences and is an Editor/Member of the Editorial Board of several journals.

Michael Beisiegel is a Distinguished Engineer with IBM Software Group's Strategy and Technology division in Somers, NY. He is responsible for the Service Component Architecture (SCA) programming model development. He has worked on systems management products for z/OS, VisualAge for Smalltalk CICS and IMS Connection, VisualAge for Java Enterprise Access Builder (EAB), Common Connector Framework (CCF) for Component Broker and WebSphere, WebSphere Studio Application Developer Integration Edition, WebSphere Process Server SCA runtime. He was IBM's expert on the J2EE Connector Architecture JSR that is modelled after CCF. Currently, he is working with the Open SOA collaboration (http://www.osoa.org) on the standardisation of SCA. He joined IBM in 1989 and began working for the 390 software development organisations at the IBM Laboratory in Böblingen, Germany. Later, he took an assignment (1998–2000) working on Java-based integration tools and connector technology at the IBM Toronto Laboratory in Canada. He joined IBM, US in 2000. He received a Masters (Dipl. Ing.) in Electrical Engineering from the University of Kaiserslautern, Germany.

Kees M. van Hee received a PhD in Operations Research from the Technische Universiteit Eindhoven (TU/e). From 1994 till 2004, he was Managing Partner of Bakkenist Management Consultants and Deloitte Consultancy. During 1991–1992, he was a Visiting Professor at the University of Waterloo, Ontario. From 1984 till 1994 and again since 2004, he is Professor of Computer Science at TU/e. He published papers and books on the following topics: Markov decision processes, Applications of queuing theory, Decision support systems, Formal specification methods and tools, Petri nets, Database systems and Workflow management systems.

Dieter König is a Senior Technical Staff Member with IBM Software Group's laboratory in Böblingen and Architect for workflow products. He joined the laboratory in 1988. He has worked on Resource Measurement Facility for z/OS, WebSphere MQ Workflow and WebSphere Process Server. He is a Member of the OASIS WS-BPEL Technical Committee, which is working on an industry standard-based on the Specification of the Web Services Business Process Execution Language (WS-BPEL). He has published many papers and has given talks at conferences about Web services and workflow technology and is co-author of two books about web services. He received a Masters (Dipl. inform.) in Computer Science from the University of Bonn, Germany.

Christian Stahl studied Computer Science at Humboldt-Universität zu Berlin, Germany. He received a Masters in 2004. Since then, he is working as a Research Assistant in the group of Wolfgang Reisig in Berlin, and since 2006, also in the group of Kees M. van Hee and Wil M.P. van der Aalst in Eindhoven. His research interests include process modelling, process analysis, formal methods, in particular Petri nets and model checking.

1 Introduction

The concept of *modularisation* can be used to master the complexity of large (software) systems. Modules have different names like 'function', 'class' or 'component'. The principle of *compositionality* is one of the most desirable 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 some property and they are connected properly, then the system as a whole should satisfy the same property. 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 the *architecture* of a system. There are several languages to define components and to glue them together. There are also different *architectural styles*. In this paper, we concentrate on a style based on the Service-Oriented Architecture (SOA) (High et al., 2005). SOA can be seen as one of the key technologies to enable flexibility and reduce complexity in software systems. Today, SOA is a set of ideas for architectural design and there are some proposals for SOA frameworks, including a concrete architectural language: the Service Component Architecture (SCA) (Beisiegel et al., 2007) and software tools to design systems in the SOA style.

In this paper, we present an SOA-based architecture framework by means of a *meta model* and discuss its concepts in detail. The architecture framework consists of *three* models each representing a particular view.

The *component model* presents an abstract view on the components of the system and shows which components interact with each other by message exchange.

Every component contains a process, which is a set of activities. The *process model* provides a view on these activities and their relation to the data entities. An activity can access data entities that are located within and outside its component by using the concepts of method call and message exchange, respectively. We further show that our process model is generic and thus it can be specialised by process models such as WS-BPEL (Alves et al., 2007) or Petri nets.

The *data model* is a view on data entities and their relationships. The architecture framework allows for internal relationships between data entities (i.e. within a component) and external relationships between data entities (i.e. across the borders of components). These two different relationships introduce hierarchy in the data model.

Besides these three views, the architecture framework also covers important concepts such as component *instantiation* and *message correlation* (i.e. deliver messages to their correct component instance).

To enable the verification of systems on the level of the architecture, we collect a number of constraints for the architecture framework and specify them using the Object Constraint Language (OCL) (Object Management Group, 2003). These constraints can be implemented and automatically checked during the system design.

Our architecture framework should be close to industrial standards, especially SCA. Therefore, we compare the concepts of our architecture framework with those of SCA and show that it extends SCA.

The outline of this paper is as follows. In Section 2, we sketch the practice of component-based software systems. We also introduce software architectures, in particular, the SOA. Our main contribution, the architecture framework including component, process and data model, is presented in Section 3. In Section 4, we compare our proposed framework with SCA. Finally, Section 5 summarises this paper, discusses related work and describes how our work will be continued.

2 Context

2.1 The Component-based world

The idea to use components in software development was already published by McIlroy (1968). In that paper, McIlroy presented his idea of mass-produced software components. Even though much progress has been booked since then, today there is still no universally accepted definition of what a component is. Most cited is the definition of Szyperski (1998):

> "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."

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 or *coarse grained* like a debtors register in an Enterprise Resource Planning (ERP) system.

Components can be classified based on their *functionality*: There are *application-specific* and *generic* components. An SAP component is an example of an application-specific component whereas a workflow engine is a generic component. Another classification of components is based on the *configuration* of their parameters. In a *predefined* 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.

Components may specify non-functional properties. Non-functional properties are also named Quality of Service (QoS). Examples are response time and the usage of resources.

A component may have relevance from a business perspective (which is the primary focus of SOA) or from an IT perspective (as in traditional software systems).

A system that is developed by composing components is a *component-based system*. Component-based systems will evolve in an *organic* way. There may never be a total renewal or an upgrade of the overall system. Instead, components will be replaced periodically by better ones, for example, because the performance was not good enough anymore. Adding new functionality to the system will also be realised by either adding new components or replacing components by better ones. This will reduce the total cost of ownership of component-based systems.

Customers will use component-based systems, because component-based design has two major benefits when the component-based system fulfils the compositionality principle. Firstly, it structures the design and the development of systems and thus reduces the amount of effort needed to verify and maintain systems. Secondly, the reuse of components reduces the development effort and thus time and costs (Bouyssounouse and Sifakis, 2005).

In the component-based world, the architecture is of crucial importance. 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 as a white-box (i.e. the internal details of the component are visible, too). 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 independent of each other.

2.2 Architecture frameworks

Let us now shift our focus from components and component-based systems to *software architectures*. We start with a definition of software architecture in general and introduce then the SOA.

2.2.1 Software architectures

Just like for the term 'component', everyone knows roughly what a software architecture is, but there is also no universally accepted definition. A modern definition of software architecture is the one of Bass et al. (2003):

> "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."

Referring to this definition, an architecture shows the elements of the system. In case of a component-based system, it shows the components and their relationships. We restrict us to 'the structure of the 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*; that is, no view should contradict another view on the system. Furthermore, views should also be *complete*. That means, every property of the system should be modelled by at least one view.

Based on these facts, we elaborate the definition of a software architecture to the following which is used throughout this 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 organise and develop a software architecture in a specific style is an *architecture framework*. Some examples for software architecture frameworks are UML, CORBA, Koala (Ommering et al., 2000), SENSORIA Reference Modelling Language (Fiadeiro et al., 2006) and SCA (Beisiegel et al., 2007) to name a few. Later, in Section 4.1, we will give a short introduction to SCA.

2.2.2 Service-oriented architecture

One of today's most popular architecture frameworks is a SOA. SOA is seen as one of the key technologies to enable flexibility and reduce complexity in software systems (High et al., 2005). It follows the paradigm to explicitly *separate* an implementation from its interface. Such an interface is *well-defined*; that is, it is based on standards such as the Web Service Description Language (WSDL) (Christensen et al., 2001). Implementation and interface together form a component. In a SOA, a component is referred to a *service*, but we prefer to use the term component. Components are independent of applications and the computing platforms on which they run. Components in a SOA can be connected without having knowledge of their technical details; they are *loosely coupled*. To connect components during runtime, SOA supports *dynamic binding*. For the message exchange between components, standardised communication protocols are used. Further, all the standards, which are used in a SOA, are *extensible*, meaning they are not limited to current standards and technologies.

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

A component has two kinds of interfaces. *Buy* interfaces specify which services are required by the component. *Sell* interfaces specify which services are provided by the component. So, in terms of the component roles, in a SOA, a component plays the consumer's role at the buy interfaces, and at the sell interfaces, it plays the provider's role.

3 A SOA-based architecture framework

In this section, we present a meta model of our architecture framework. We introduce its concepts including the three views, component model, process model and data model.

3.1 Component model

Figure 1 shows the complete meta model of our architecture framework. In the following, we concentrate on the component model; that is, we have a detailed look at the concepts of components, the interface concept and the wiring of components.

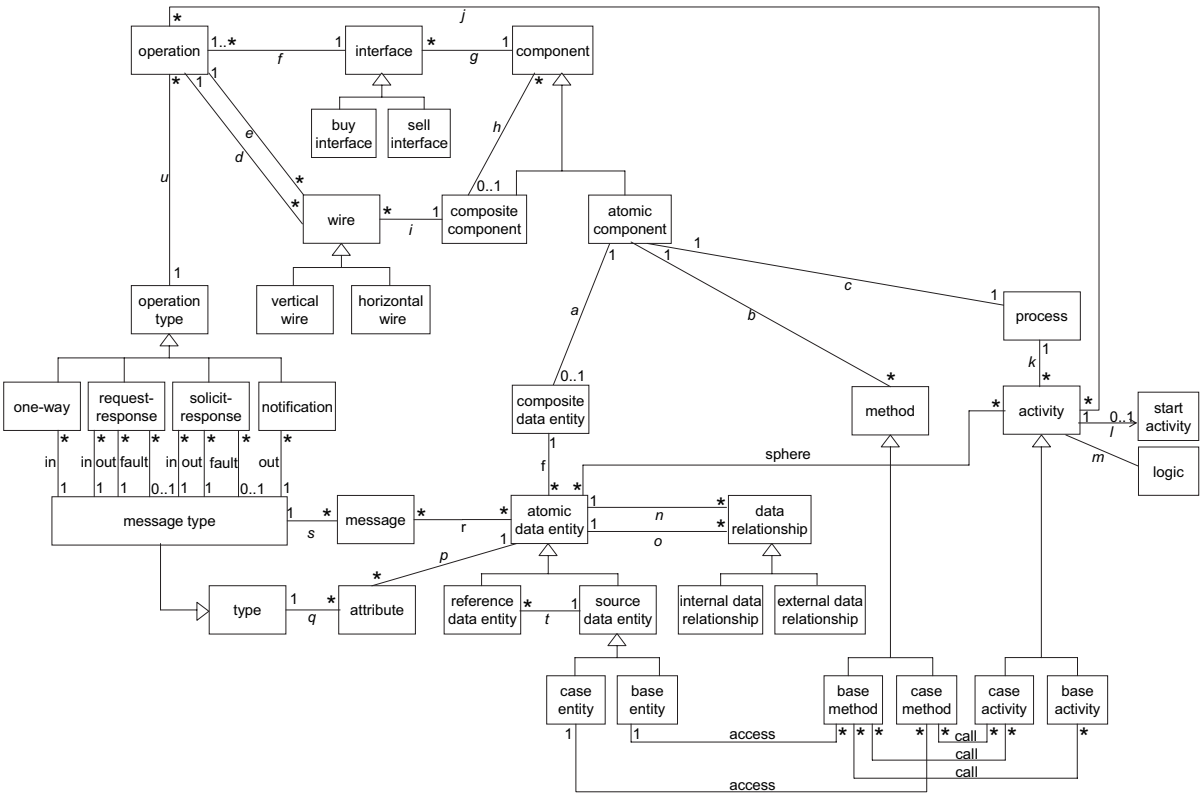
The basic concept of the architecture framework is a *component*. We distinguish *atomic components* and *composite components*. An atomic component consists of a *process*, which is a set of *activities* (c is the name of the relationship between the entities 'atomic component' and 'process' and k between 'process' and 'activity' in Figure 1), zero or one *composite data entities* (relationship a) and *methods* (relationship b). A composite component, however, describes a hierarchical relationship between components. It is a container for components; that is, it may contain atomic components and other composite components (relationship h).

Each component has one or more *interfaces* (sometimes called *port*) with its environment (relationship g). An interface is either a *buy* or a *sell* interface and consists of a set of *operations* (relationship f). An operation describes a message exchange between two participants. However, it can be used by any number of components. An operation follows a given *operation type* (relationship u) which describes a message exchange pattern between the participants. We distinguish the four operation types presented in the WSDL 1.1 specification (Christensen et al., 2001): *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 Figure 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. Both operation types differ in their message order. In case of a solicit-response, the component first sends a message and then receives a message whereas in case of a request-response it is the other way around. In practice, operation types one-way and request-response are predominantly used and solicit-response and notification are less relevant.

An activity may exchange messages through one or more operations (relationship j) with other components. It may also access some of the (atomic) data entities of its atomic component by means of *method calls* (relationship *call*). These method calls may change the value of the data entities. A more detailed look at processes and data entities is presented in Sections 3.2 and 3.3, respectively.

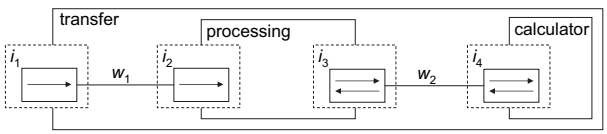
Besides wrapping components (relationship h), a composite component also defines one or more *wires* (relationship i). In general, a wire connects interfaces of components. More precisely, a wire connects two operations depicted by relationships d and e. These two operations have either the same operation type or they have complementing operation types, for example, one-way and notification.

Figure 1 Meta model of the proposed architecture framework in UML notation



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*. It always connects an operation of a component by its direct parent operation. In contrast, wiring two operations with complementing operation types shows the *connection* of two components. We call such a wire a *horizontal wire*. A wire represents only an abstract view on the communication of a component. It only shows the invocation dependencies of a component and there can be any number of calls along a wire. Figure 2 illustrates the different wires.

Figure 2 Component model of a bank transfer: component *transfer* contains two components *processing* (receives the customer's bank transfer) and *calculator* (calculates the bank transfer)



Note: It defines a vertical wire w_1 and a horizontal wire w_2 depicted by a solid line. Interfaces i_1,…,i_4 are depicted by a dashed frame. A box visualises an operation. Its operation type is depicted by one or two arcs inside the box. Interface i_1 and i_2 have an operation with operation type one-way, i_3 request-response and i_4 solicit-response. Interface i_3 is a buy interface. All other interfaces are sell interfaces.

Most of the information about wiring operations cannot be derived from the meta model in Figure 1. Later, in Section 3.4, we will therefore define the wiring characteristics using OCL.

3.2 Process model

Let us now shift our attention from components to processes. First of all, we clarify the relation between process and data entities and we introduce the two concepts an activity can access to a data entity. Subsequently, the concept of instantiation is explained. Instantiation makes it necessary to deliver a message to a concrete component instance. Therefore, we develop a concept of message correlation. Finally, we introduce with WS-BPEL and Petri nets two specialisations of our process model.

3.2.1 Activities and data entities

A process contains a set of activities (relationship k in Figure 1). Every activity consists of zero or more method calls and some additional logic (relationships *call* and m, respectively). Methods are used to read and write the value of data elements. Logic controls the method calls and evaluates their return values. Logic can be specified by functions and their signatures. The construction of the logic is the work of programmers after the software architect has designed the architecture.

The *sphere* of an activity is defined as the set of data entities this activity can access using a method call (relationship *sphere*). Clearly, the sphere only contains data entities that are defined in the same atomic component as the activity. In our architecture framework, method calls are restricted to activities. As a consequence, no data entity can have access to another data entity. Methods and activities are defined in the same component and activities can only call methods which are defined in that component. A method can be used by several activities. Therefore, it is defined at the component level.

The architecture framework also supports a second mechanism which is mainly used to access data outside an atomic component. Instead of calling a method directly, an activity sends a message to an operation (relationship j) that passes it to another activity which contains the respective data entity in its sphere.

3.2.2 Instantiation

One of the most important concepts of an architecture framework is instantiation. In our architecture framework, components can be instantiated multiple times (not shown in Figure 1). For the purpose of instantiation, atomic components distinguish between *case activities* and *case entities* on the one hand and *base activities* and *base entities* on the other hand (cf. Figure 1). The set of case and base activities is also called *case process* and *base process*, respectively. To understand the difference between case and base, we need to consider the lifecycle of a component.

Once an (atomic) component is initialised, its base process and its base entities are initialised, too. This initialisation can be seen as the instantiation of the base process. Afterwards, the component can be instantiated. To create a *case*; that is, a new instance, a *start activity* (cf. Figure 1) is used. We distinguish two possibilities to create a case, depending on the start activity being a base or a case activity. A base start activity can create any number of cases. Every case is identified by a *case id*. A case start activity, in contrast, needs to be triggered by the component's environment. For this purpose, an atomic component has to be invoked via its interface. A message is received by the start activity, which then creates a case. A process may have more than one start activity, but no process may have both a base and a case start activity. This fact will be specified with the help of an OCL constraint in Section 3.4. The instantiation implies the creation of case activities and case entities which belong to exactly one case. Their lifecycle is restricted to its respective case. When a case has been finished, it can be destroyed. The lifecycle of base activities and base entities, in contrast, only ends once the component is deactivated.

Base activities and entities are independent of a specific case. A base activity may create cases and may access base entities within its sphere. A case activity, however, may access case and base entities as shown in Figure 1. It may also trigger base activities. In contrast, a base activity can neither access case entities nor trigger case activities. On this account, a base entity can be seen as a configuration parameter. Base activities, however, are typically used for monitoring and configuration of a component.

3.2.3 Message correlation

In a component interaction, several cases of the components may be involved. With it, the problem arises how a message can be delivered to the correct case of a component. Therefore, our architecture framework provides the concept of *message correlation*, which is known from WS-BPEL, for instance. Every case is identified by its case id. In WS-BPEL, a case id is called *correlation property*. Typical examples of correlation properties are customer numbers, order ids and invoice numbers. A message can be delivered to the correct case if the case id can be either determined from the content of the message or it is an explicit part of the message. We restrict ourselves to interaction between case processes. The differences, when base processes are also involved in the interaction, are subject of (van der Aalst et al., 2007). Before we introduce our concept of message correlation, we present possible scenarios of component interaction to demonstrate the requirements of message correlation.

In an interaction between two components S and R, there is one component, say S, that starts the interaction. There are two possibilities for S to start the interaction with R. S can either *create* a case of R or it can *find* a case of R. A case can be found if S has a *reference* (i.e. the case id) to a case of R (e.g. from a third party) or vice versa. Another possibility to find a case of R is to decide from the message content whether there exists a case of R that can handle the request. This *criterion* either specifies requirements of S that have to be fulfiled by R or it contains information that have to fulfil the requirements of at least one case of R. Obviously, if S sends a criterion, there might be no matching case in R or there are several cases that can handle the request sent by S. If several cases match, one case has to be chosen, for example non-deterministically.

Now, we define the term *correlation*. Correlation of a case c is *firstly the set of cases to which c knows their respective case id and secondly the set of components to which c sent its case id*. A correlation of all cases spans a graph where each node is a case. This graph has two kinds of directed arcs: *reference arcs* and *send arcs*. Let c_1 and c_2 be two cases. A reference arc is drawn from case c_1 to case c_2 if c_1 knows the case id of c_2. A send arc, in contrast, is drawn from case c_1 to case c_2 if c_1 sent its case id to component C_2 and the message was delivered to c_2. In the latter case, c_1 has only knowledge about component C_2 and not about case c_2. We call the resulting graph a *correlation set*.

In our concept of message correlation, we formalise a *message format* by a six tuple that consists of the sender's address, the receiver's address, the sender's case id, the receiver's case id, the correlation information (i.e. the criterion used to decide whether a message matches a case) and finally, the message content. Addresses and message content are mandatory whereas case ids and correlation information are optional.

3.2.4 Example process models

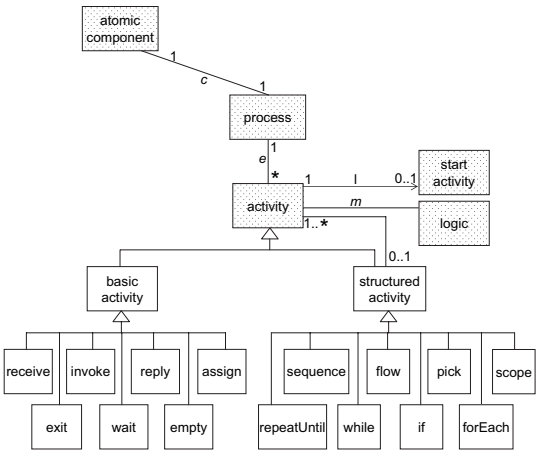
Our proposed architecture framework in Figure 1 is highly generic and thus it is easy to fit in specific language proposals. In the following, we demonstrate that we can easily link two example process models, WS-BPEL and Petri nets, into our architecture framework. These example process models specialise the process model in Figure 1.

The *Web Services Business Process Execution Language* (WS-BPEL) (Alves et al., 2007) is a widely used language for describing the behaviour of business processes based on web services. For the specification of a business process, WS-BPEL provides *activities* and distinguishes between *basic* and *structured* activities. A basic activity can communicate with other WS-BPEL processes by message exchange, for instance. A structured activity defines a causal

order on the basic activities and can be nested in another structured activity. For the sake of simplicity, we restrict our view on WS-BPEL to activities and do not go into the details of WS-BPEL's advanced concepts like fault and compensation handling.

The meta model in UML notation for this restricted part of WS-BPEL is depicted in Figure 3 (activities throw, rethrow, compensate, compensateScope, validate and extensionActivity are not shown in Figure 3). The relation between entities *activity* and *structured activity* is most relevant for our architecture: Every WS-BPEL activity can be contained in a structured activity and every structured activity can contain one or more activities. Entity 'activity' in Figure 1 coincides with a WS-BPEL activity. Thus, WS-BPEL can be easily linked into our framework. This is shown by connecting entity 'activity' with the already known entities 'process' and 'atomic component' (cf. Figure 1). A 'data entity' (not shown in Figure 3) corresponds to a WS-BPEL variable. If WS-BPEL is used for describing the process model for a component, then this process also implicitly describes the data model through its WS-BPEL variable definitions. WS-BPEL does not distinguish base and case; that is, WS-BPEL activities, variables and also WS-BPEL's advanced concepts like fault and compensation handling always belong to exactly one case (i.e. to a process instance). The concept of start activities is also supported in WS-BPEL. Activities 'receive' and 'pick' can be used to create an instance of a BPEL process if the attribute *createInstance* is set to *yes*. In our framework, every activity can be a start activity. So, we have to add this fact by a constraint. Our concept of logic can be mapped to WS-BPEL, too. It is possible to specify XPath expressions in WS-BPEL and there exists extensions of WS-BPEL that allow, for instance, the integration of Java code into the WS-BPEL code (Blow et al., 2004).

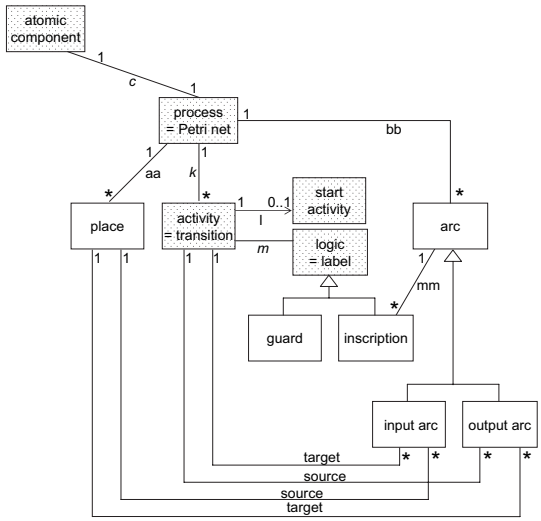
Figure 3 Meta model for WS-BPEL activities



The formalism of Petri nets has been proven to be an adequate model for business processes (e.g. van der Aalst, 1998). A Petri net (see e.g. (Reisig, 1985) for a formal definition) is a bipartite graph. It consists of two different nodes, *places* and *transitions* and (directed) *arcs*. An arc connects either a place and a transition (input arc) or a transition and a place (output arc). Places can contain (black) tokens which represent a data value. We consider Coloured Petri Nets (CPNs) (Jensen, 1992), an extension of usual Petri nets. In a CPN, tokens have a value (i.e. a colour). That way, 'real' data values can be modelled.

The Petri net meta model in UML notation is presented in Figure 4. In the meta model, input arcs and output arcs are distinguished. Like WS-BPEL, Petri nets can be easily linked into our architecture framework. Entity *Petri net* and entity *transition* coincide with entities 'process' and 'activity' in Figure 1, respectively. A start activity can also be modelled by a Petri net transition. More precisely, the transition has to generate a new case id. Entity 'logic' in Figure 1 coincides with entity *label*. A label is either a transition guard (i.e. a Boolean expression) or an (arc) inscription. A data element can be modelled by a place and the data value by a token on this place. If we think of a Petri net as a model for the base and the case process, different cases can be expressed by different colours, where each colour represents exactly one case id, for instance.

Figure 4 Meta model for Petri nets



3.3 Data model

Many architects consider only the information architecture of a system (i.e. the database schema) when they use the term architecture. The information architecture is actually a *data model*. It is a view on data entities and their relationships. The information architecture is very important, because it facilitates the structuring and organising of data entities. Often, architects start the system design with the development of the information architecture. In the following, we introduce the general concepts of the data model, in particular, its hierarchy concept.

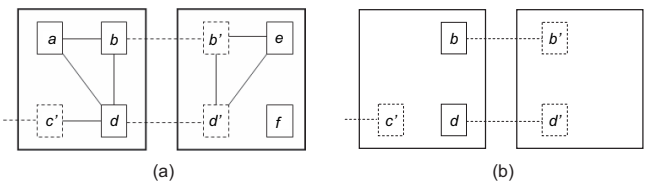
The starting point of the data model is again an atomic component. Entity *composite data entity* in Figure 1 is an ER-model. A composite data entity is a set of *atomic data entities* (relationship *f* in Figure 1) where every atomic data entity consists of a set of *attributes* (relationship *p*). Relationship *q* shows that every attribute has a *type*. Atomic data entities can be accessed by activities (relationship *sphere*) or exchanged by messages (relationship *r*).

The data model allows for relationships between atomic data entities. Entity *data relationship* illustrates this fact. Two atomic data entities can be related (relationships

n and *o*). We distinguish between *internal data relationship* and *external data relationship*. An internal data relationship relates two atomic data entities within a composite data entity. To provide a relationship between two atomic data entities located in different composite data entities and thus in different atomic components, the meta model distinguishes between *source data entity* and *reference data entity*. A reference data entity is a reference to a source data entity. For every source data entity, there can be any number of references (relationship *t*). That way, it is possible to define a source data entity in one composite data entity (i.e. in an atomic component) and to have references (with the help of reference data entities) in other atomic components. A reference data entity and its corresponding source data entity are related by an external data relationship. These constraints are specified in Section 3.4 using OCL.

The use of reference data entities introduces hierarchy in the data model. We distinguish two different views on the data model. The first and detailed view visualises the relationship of all atomic entities in a composite entity. On one hand, it shows the internal data relationships between atomic data entities, that means, how entities within an atomic component are connected. On the other hand, this view also presents the external data relationships; that is, for each reference data entity, its source data entity (relationship *t* in Figure 1) is depicted. Relationship *t* shows how an atomic component is related on the data level to other atomic components by help of reference data entities. The second and abstract view, however, is restricted to the external data relationship only. This hierarchy concept is, in fact, similar to the concept of atomic and composite components. As an example, a data model of two atomic components is shown in Figure 5. It is possible to have more levels of hierarchy by having a deeper hierarchy of (composite) components.

Figure 5 Two levels of hierarchy in the data model: (a) data model – concrete level and (b) data model – abstract level



Note: A solid frame depicts an atomic component. Inside this frame, the composite data entity is shown. Boxes *a–f* depict atomic data entities. Solid boxes and dashed boxes visualise source data entities (e.g. *b*) and reference data entities (e.g. b'), respectively. Undirected solid arcs connecting two atomic data entities model an internal data relationship between these entities (e.g. *a* and *b*, d' and *e*). In contrast, dashed arcs that cross the border of an atomic component depict external data relationships and thus which reference data entity is related to which source data entity. Examples are *b* and b', *d* and d', and also c' with a source data entity not depicted in Figure 5(a). The detailed view is shown in Figure 5(a). All atomic data entities and their internal and external data relationships are visible. The abstract view is shown in Figure 5(b). Only the three reference data entities, the two corresponding source data entities and their external data relationships are visible.

Now, we have a look at the relation between internal and external data relationship on one hand and method call and message exchange on the other hand. This is also a relation between data model and process model. From the details given in Section 3.2, it is known that activities can change the value of data elements by method call. An activity has, however, only access to a restricted set of data elements, namely, to the data elements within its sphere (cf. relationship *sphere* in Figure 1). In the data model, a method call is reflected by an internal data relationship between two atomic data entities. External data relationships, in contrast, reflect message exchange between activities. This is defined by an OCL constraint in Section 3.4.

3.4 Constraints

The meta model in Figure 1 is in some sense quite general, because specific constraints cannot be expressed in UML. Therefore, it is possible to create errors during the design of the system. However, constraints of UML models can be specified using the OCL (Object Management Group, 2003). They can be implemented in a Computer-Aided Software Engineering (CASE) tool which can be used to check the system at design time. Thus, the architect can be prevented from creating such errors.

In the following, we present several constraints that help to formalise concepts like wiring or the relationships between entities. Due to the page limit, we only specify one constraint as an OCL invariant and describe all other constraints informally. For a complete overview of all these constraints specified in OCL, we refer to van der Aalst et al. (2007). OCL keywords are depicted in bold font. For the parameters used, we refer to the relationships in Figure 1.

1 Two atomic data entities, which are related by an external data relation, are located in different atomic components and one of them is a source data entity and the other one its reference data entity:

context x: external data relationship **inv**:

$x.n.f \neq x.o.f$ **and**

(($x.n.oclIsTypeOf$ (*reference data entity*) **and**

$x.o.oclIsTypeOf$ (*source data entity*) **and** $x.n.t = x.o$)
or ($x.n.oclIsTypeOf$ (*source data entity*) **and**
$x.o.oclIsTypeOf$ (*reference data entity*) **and**
$x.o.t = x.n$))

The keyword **inv** means that this OCL expression is an OCL invariant. This invariant is introduced for the context of an 'external data relationship'. Informally spoken, it specifies that an external data relationship between two data entities only exists if these entities are located in different composite data entities and thus in different atomic components. Furthermore, one of the entities has to be a source data entity and the other entity, one of its reference data entities. The second line of this invariant specifies that both atomic data entities are located in different composite data entities.
The remaining lines specify a disjunction. Either $x.n$ is a reference data entity and $x.o$ is a source data entity (lines three and four) or vice versa (lines five and six). To check the type of an atomic data entity, we use OCL's

oclIsTypeOf operator. $x.n.t = x.o$ then specifies that the reference data entity $x.n$ has to be a reference of the source data entity $x.o$.

2 Two atomic data entities which are related by an internal data relation are located in the same atomic component.

3 External data relationship means message exchange.

4 There is no process having a base start activity and a case start activity.

5 A horizontal wire connects two components within an enclosing component.

6 A vertical wire connects a component with its enclosing component.

7 A vertical wire connects two operations with the same operation type.

8 A horizontal wire connects two operations with matching operation types.

In conclusion, these constraints can be merely seen as examples. Once they are implemented, they can be automatically checked during the design phase of the system. At this level of design, it is faster and cheaper to fix errors than in later design phases. Nevertheless, constraints are not sufficient to guarantee the correctness of systems.

4 Comparing the architecture framework with SCA

4.1 Introduction to SCA

The SCA (Beisiegel et al., 2007) provides a model for the composition of services, the creation of service components and the reuse of existing applications within service compositions. Service components can be implemented in different programming languages and accessed via different protocols, including web services, asynchronous messages or synchronous remote procedure calls.

The following paragraphs describe the SCA component model. SCA components use a simple interface contract to describe their partner relationships.

The most important construct of SCA is the *component* consisting of

1 *services* (i.e. business functions offered to other components)

2 *references* (i.e. dependencies on business functions needed from other components)

3 *properties* (i.e. values that influence the component implementation)

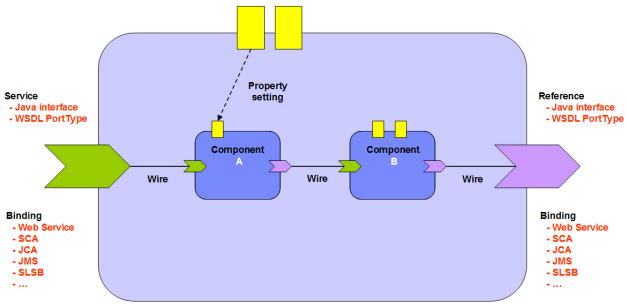
4 *implementation* (i.e. concrete realisation of the provided services).

SCA provides the concept of a *component type* defining the configurable aspects (or points of variability) of an implementation. A component is a configured instance of an implementation.

An SCA component may be implemented using traditional programming languages like C++ or Java, scripting languages like PHP or JavaScript, declarative languages like XQuery or SQL, or as a business process using WS-BPEL.

The *SCA Assembly Model* describes how components can be assembled into *composites*, containing the aggregated components, services, references and properties. Composites can be viewed as an implementation of a higher-level component and can be nested. Composites also contain *wires*. The *source* of a wire may be a component reference or a composite service. The *target* of a wire may be a component service or a composite reference. Figure 6 illustrates an example SCA composite. Note that the wires (as in Figure 1) describe a dependency relationship and not control flow.

Figure 6 An example SCA composite (for colours see online version)



An SCA *system* represents the configuration of an SCA run-time environment. It represents a region of configuration and control and defines the scope of what can be connected via SCA wires. In general, an SCA run-time environment is distributed and heterogeneous. It has a logical system level composite of running components that are implemented by simple implementations or composites.

In SCA, services and references can be associated with *bindings* and *policies*.

References use bindings to describe the mechanism used to call a service and services use bindings to describe the access mechanism that clients have to use in order to call the service. Examples for bindings are a web service, a stateless session EJB, a database stored procedure or an EIS service binding.

A policy is a declaration of a specific set of behaviours, and applies to the implementation of a component or to an interaction with a component. Policies may be aggregated into *profiles*. Examples for policies are WS-ReliableMessaging or WS-Addressing policies associated with web service bindings, or a conversation policy associated with JMS bindings.

The interface model is extensible such that detailed partner interaction semantics could be captured as well, for example, by using concepts like WS-BPEL Abstract Processes. SCA components may be stateless or stateful; however, SCA does not provide an explicit data model describing data managed by a component.

4.2 Comparison to the architecture framework

To compare SCA with our architecture framework, we first of all present in Table 1 a comparison of the terms used. Then, we step into the details of both the frameworks.

Table 1 Comparing the terms of SCA and the architecture framework

SCA	*Framework as depicted in Figure 1*
Component	Atomic component
Composite	Composite component
System	Outmost composite component
Implementation	Process implementation like WS-BPEL, Petri nets
Service	Sell interface
Reference	Buy interface
Property	Base entity
Wire	Wire

The component concepts in both frameworks are very similar. Both frameworks support atomic and composite components, wires and processes. In SCA, it is possible to specify a property for a composite, whereas in our framework, composite components do not have data entities.

In SCA, the term implementation is used for the choice of a process technology like WS-BPEL, Java or Petri nets. So, an SCA implementation coincides with a process implementation in our architecture framework.

At the level of the interface, SCA is more extendable than our interface concept which is restricted to WSDL 1.1. However, we can also easily extend our interface concept.

Both frameworks are very general and thus support different process models. For our framework, we showed this in Section 3.2.4 and the process models supported by SCA are listed in the last section.

SCA specifies bindings, QoS and policies. This is, so far, not integrated in our framework. As our meta model in Figure 1 is general, we could easily add an entity for each of the three concepts. The semantics had to be defined by adding relationships and additional OCL constraints.

Finally, our architecture framework has a data model. SCA, in contrast, has no data model yet. It only supports the configuration of components with the help of properties. In our framework, we use base activities for the configuration of components (cf. Section 3.2.2).

To summarise, both frameworks are very similar, in particular in the component and process model. However, SCA does not support a data model yet, which is in our opinion, a very important model as we mentioned in Section 3.3.

5 Related work and outlook

In this paper, we addressed our efforts in developing an architecture framework for SOA. We introduced the architecture framework by means of a meta model that focused on three different views on software systems: a component view, a process view and a data view. The proposed architecture framework also covers other important concepts such as instantiation and message correlation.

We aim at formally verifying systems on the level of the architecture. For this purpose, we collected a number of constraints for our architecture framework and specified them using the OCL. These constraints can be implemented and checked by a CASE tool. That way, architects have tool support during the system design. In (van der Aalst et al., 2007), we also presented rules to translate the architecture framework into CPNs. On the level of CPNs, formal verification techniques can be applied.

The presented architecture framework is required to be language independent and close to industry standards, in particular to SCA. We have shown that our architecture framework extends SCA, since SCA does not provide an explicit data model yet.

Another architectural framework which has been inspired by SCA is the SENSORIA Reference Modelling Language (SRML). SRML presents a formal model for components and their composition. The process model specifies the language of interaction of the process but there is no data model so far. Axenath et al. (this special issue) present a meta model for business processes modelling (AMFIBIA) which captures the aspects, control flow, data and organisation. As in our approach, these aspects can be modelled independently of each other and it is possible to integrate them later on. A component model is missing so far, but the approach is extensible. To summarise, the main contribution of our framework, the integration of the component, the data and the process views, is neither provided by SRML and AMFIBIA nor by state-of-the-art architecture frameworks such as CORBA, UML and Koala.

In ongoing research, we want to extend the architecture framework, for example with the concept of inheritance which allows the reuse of parts of the system. Inheritance is one of the most important concepts in object-oriented programming and should therefore be adapted on the level of architecture frameworks. We also want to spend more effort on the verification of the architecture and as a long-term objective on the development of tools for the design and management of component-based systems.

References

Alves, A. et al. (2007) *Web Services Business Process Execution Language Version 2.0*, OASIS Standard, 11 April 2007, OASIS.

Axenath, B., Kindler, E., Rubin, V. (this special issue) 'AMFIBIA: a meta-model for integrating business process modelling aspects', *Int. J. Business Process Integration and Management*.

Bass, L., Clements, P. and Kazman, R. (2003) '*Software Architecture in Practice*', 2nd edition, Addison Wesley Professional.

Beisiegel, M., et al. (2007) *Service Component Architecture – Assembly Model Specification*, SCA Version 1.00, 15 March 2007, IBM, SAP et al.

Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D. and Rowley, M. (2004) *BPELJ: BPEL for Java, Whitepaper*, BEA, IBM.

Bouyssounouse, B. and Sifakis, J. (Ed). (2005) '*Embedded Systems Design – The ARTIST Roadmap for Research and Development*, Vol. 3436 of *Lecture Notes in Computer Science*, Heidelberg: Springer Berlin.

Christensen, E., Curbera, F., Meredith, G. and Weeravarana, S. (2001) *Web Service Discription Language (WSDL) 1.1.*, W3C Note 15 March 2001, Ariba, International Business Machines

Corporation, Microsoft.

Fiadeiro, J.L., Lopes, A. and Bocchi, L. (2006) 'A formal approach to service component architecture', in M. Bravetti, M. Núñez and G. Zavattaro (Eds). *Web Services and Formal Methods (WS-FM 2006), Proceedings*, Vol. 4184 of *Lecture Notes in Computer Science*, pp.193–213, Springer-Verlag.

High, R., Kinder, S. and Graham, S. (2005) 'IBM's SOA foundation – an architectural introduction and overview', *Technical Report 1.0*, IBM.

Jensen, K. (1992) *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Vol. 1 of *Monographs in Theoretical Computer Science*, Springer-Verlag.

McIlroy, M.D. (1968) 'Mass produced software components', in P. Naur and B. Randell (Eds). *Proceedings of NATO Software Engineering Conference*, Vol. 1, pp.138–150, Garmisch, Germany.

Object Management Group (2003) *UML2.0 Object Constraint Language (OCL) Specification*, Specification, Object Management Group (OMG).

Ommering, R.C., van der Linden, F., Kramer, J. and Magee, J. (2000) 'The Koala component model for consumer electronics software', *IEEE Computer*, Vol. 33, No. 3, pp.78–85.

Reisig, W. (1985) *Petri Nets*, Berlin, Heidelberg, Springer-Verlag, Tokyo, EATCS Monographs on Theoretical Computer Science edition.

Szyperski, C. (1998) *Component Software–Beyond Object-Oriented Programming*, Addison-Wesley and ACM Press.

van der Aalst, W.M.P., Beisiegel, M., van Hee, K.M., König, D. and Stahl, C. (2007) 'A SOA-based architecture framework', *Computer Science Report 07/02*, Technische Universiteit Eindhoven, The Netherlands.

van der Aalst, W.M.P. (1998) 'The application of Petri nets to workflow management', *Journal of Circuits, Systems and Computers*, Vol. 8, No. 1, pp.21–66.