

# A SOA-Based Architecture Framework

Wil M. P. van der Aalst<sup>1</sup>, Michael Beisiegel<sup>2</sup>, Kees M. van Hee<sup>1</sup>,  
Dieter König<sup>3</sup>, and Christian Stahl<sup>1</sup>

<sup>1</sup>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,k.m.v.hee,c.stahl}@tue.nl

<sup>2</sup>IBM  
Somers, New York, USA  
mbgl@us.ibm.com

<sup>3</sup>IBM Böblingen Laboratory  
Schönaicher Strasse 220, 71032 Böblingen, Germany  
dieterkoenig@de.ibm.com

We present a 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 such as wiring, correlation, and instantiation can be clarified. This allows us to demystify some of the confusion related to SOA.

**Key words:** Service-Oriented Architecture (SOA), Architecture framework, Service Component Architecture (SCA)

## 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

followed by specification languages such as Z [ASM80] and Vienna Development Method (VDM) [Jon90].

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 types of properties the modules differ in, for example, the way they are invoked, whether they are stateless or not, and if they have side effects. 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 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 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 concentrate on a style based on the *Service-Oriented Architecture* (SOA) [HKG05]. 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) [BBE<sup>+</sup>06], and software tools to design systems in the SOA style.

In this paper, we present a 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. Therefore, the component model shows the components, their interfaces, and how these interfaces are wired. The component model allows for a concept of hierarchy, too.

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 to 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 specialized by process models such as WS-BPEL [AAA<sup>+</sup>06] and 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 support the concept of instantiation, we distinguish in the process model between case and base activities and in the data model between case and base entities. A case activity (entity) belongs to a single instance whereas a base activity (entity) is independent of a specific 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) [OMG03]. These constraints can be implemented and automatically checked during the system design. We further present rules to transform the architecture framework into Colored Petri net models. This is only shown by example, but a fully worked out semantics for the architecture framework can be easily derived from it. The formal semantics is the basis to make the architecture framework applicable for formal verification (e.g., model checking).

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 the paper is as follows: In Sect. 2, we sketch the practice of component-based software systems. We also 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. Next, in Sect. 3, we present our two running examples, the dating service and the container transport system. Our main contribution, the architecture framework including component, process, and data model, is presented in Sect. 4. We introduce the architecture framework by means of a meta model and show that it covers most of the requirements for a SOA-based architecture framework. Subsequently, in Sect. 5, we show how the architecture can be formalized by transforming the dating service example into Colored Petri net models. Afterwards, in Sect. 6, we compare our proposed framework with the Service Component Architecture. Finally, Sect. 7 summarizes the 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 in 1968 [McI68]. In this 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 [Szy98]:

“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.”

Messerschmitt and Szyperski present in [MS03] 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 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, for example, 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 is *horizontal* and *vertical*, respectively, because components address either a horizontal or a vertical market [Szy98]. 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.

Components may specify nonfunctional properties. Nonfunctional 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, which 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 nor 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 realized by either adding new components or replacing components by better ones. This will reduce the total cost of ownership of component-based systems.

At the time, component-based systems are in particular used in the area of web services. However, to make these systems for customers attractive, the industry has to make tool support available and define standards, for example, for components and component architectures. In fact, companies like IBM, Microsoft, Oracle, and SAP spend a lot of effort in this field. Consequently, the market expectations of component-based systems will increase during the next 5–10 years. Reasons, why customers will use component-based systems, 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 out-sourced. Building software from reusable components rather than from scratch is another possibility to reduce IT costs, design time, and develop more flexible software systems. Component-based design has two major benefits when the component-based system fulfills 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 [BS05].

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 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.

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. Sometimes the replacement has to be done at runtime which makes this task even more difficult. 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 such as in [SCCS05] becomes increasingly important. Mainly important are especially verification methods to predict systems properties from component properties. 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.

## 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 Service-Oriented Architecture.

### 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. 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 [BCK03]:

“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*; 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. 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 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 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, Koala, SCA, and SRML to name a few. The Unified Modeling Language (UML)<sup>1</sup> serves the (graphical) description of models. It can be used to describe the structure (e.g., using class diagrams) or the behavior (e.g., using use-case diagrams or sequence diagrams). Common Object Request Broker Architecture (CORBA)<sup>2</sup> enables the interaction of heterogenous applications by providing an interface definition language, object models, and communication protocols. Apache’s Turbine<sup>3</sup> is a servlet-based framework to develop web applications. Avalon<sup>4</sup>, 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. At Philips, the Koala framework [OLKM00, Omm02] is designed and used. It is a component model for electronic devices. Components interact with each other through interfaces and can be connected using connectors. The Service Component Architecture (SCA) [BBE<sup>+</sup>06] 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. Later, in Sect. 6.1, we will give a short introduction to SCA. Finally, the *SENSORIA Reference Modelling Language* (SRML) [FLB06] is an architectural framework which has been inspired by SCA. The language presents a formal model for components and their composition.

## 2.2.2 Service-Oriented Architecture

The example architecture frameworks, which are 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) [HKG05]. SOA is seen as one of the key technologies to enable flexibility and reduce complexity in software systems. It follows the paradigm to explicitly *separate* an implementation from

---

<sup>1</sup>[www.uml.org](http://www.uml.org)

<sup>2</sup>[www.corba.org](http://www.corba.org)

<sup>3</sup><http://jakarta.apache.org/turbine/>

<sup>4</sup><http://avalon.apache.org>

its interface. Such an interface is *well-defined*; that is, it is based on standards such as the Web Service Description Language (WSDL) [CCMW01, CMRW06]. Implementation and interface form together a component.

In 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, standardized 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* and *sell* interfaces. Buy interfaces specify which services are required by the component. In contrast, sell interfaces specify which services are provided by the component. So in terms of the component roles, in SOA, a component plays the consumer's role at the buy interfaces and at the sell 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.

### 2.3 General Requirements of an Architecture Framework

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 four *interfaces*: a software interface, a user interface, a configuration interface, and a monitoring interface. Interfaces, in particular the software interface, should facilitate an easy “plug and play” of components.
- A component should also have an internal *structure* that consists of

- a *process*, which is a partially ordered set of *activities*. Activities describe the component’s behavior.
  - *data entities*, which are global to the component. These data entities can be used to configure the component.
- Components should support the concept of *instantiation*. For this purpose, the architecture framework should distinguish between activities and data entities that belong to a single instance and those that can be used by all instances.
  - An architecture framework should support the concept of *message correlation* to deliver a message to its correct component instance.
  - 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 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 entities* on the other hand.
    2. Hierarchical relationships between components to support *refinement* as a design technique.
    3. Inheritance relationships to facilitate *reuse* in the (re)design.
  - 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 entities of a component. We should allow for different formalisms, for example, labelled transition systems, various kinds of process algebras, Petri nets, or industry standards as UML activity diagrams, Business Process Modeling Notation (BPMN) [Whi04], and Web Services Business Process Execution Language (WS-BPEL) [AAA<sup>+</sup>06]. Programming languages such as Java or C++ can also be used.
    - A *data model* defines the data entities, their types, and the methods to access to them. We may use here algebraic formalisms such as abstract state machines [BS03] 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) [OMG03] to specify constraints between

entities of a data model. It is also possible to use programming languages such as 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 [Abr96], VDM [Jon90], or Z [ASM80], but also programming languages such as Java, C++, or Standard ML (the programming language used in CPN Tools [RWL<sup>+</sup>03]).
- 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 WS-BPEL. With “close” we mean that the models used in the architecture framework can be easily translated into existing industrial description techniques and vice versa.

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

### 3 Running Examples

In this section, we present our two running examples, the dating service and the container transport system.

The dating service administrates a database which stores information about boys and girls looking all for the “right” partner. The business idea is that the customer (i.e., a boy or a girl) registers and after paying a fee, the system searches for the best matching partner in its database. The contact data of this resulting match is send to the customer. If the customer is not happy with this match, he can go on trying until he is satisfied with the partner selected by the system or he gives up.

Let us now have a more detailed look a the dating service. After receiving the registration information of the customer, the system checks its identification, for example, whether the customer is a known marriage swindler. If the check is positive, the registration of the customer is confirmed, else it is rejected. After receiving the confirmation, the customer can pay the fee. Then, the system searches for the best matching partner and its contact information are sent to the customer. If the customer is not happy with this match, he can pay again and thus initiate a new partner search. This can be repeated until the customer is satisfied with his partner or decides to give up. If the customer finds a partner, he is asked to send a success story to the dating service which will be published. Then, the service is finished.

The container transport system is more complex than the dating services. In this example, there exists containers, ships, and trucks. The idea is that containers are stuffed (i.e., loaded) and then transported to their strip address. After stripping (i.e., unloading), the container has to be transported to its stuff address where it is stuffed

and so on. Means of transport are trucks for short distances and ships for long distances. Trucks transport containers from a harbor to its strip address and after stripping to its stuff address. Next, a truck transports the container to the harbor.

## 4 A SOA-Based Architecture Framework

Starting from the Service-Oriented Architecture, we collected requirements for a SOA-based architecture framework in Sect. 2.3. 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. We further show that the architecture framework covers most of the collected requirements.

### 4.1 Component Model

In the following, we present our SOA-based architecture framework. It is based on the general requirements presented in Sect. 2.3. Figure 1 shows the abstract 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 components, the interface concept, and the wiring of components.

#### 4.1.1 General Overview of the Architecture Framework

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 entities “atomic component” and “process” and  $k$  between “process” and “activity” in Fig. 1), and zero or more *data entities* (relationship  $a$  in Fig. 1). Every data entity has a *type* (relationship  $q$ ). A composite component, however, describes 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 allow for the four operation types presented in the WSDL 1.1 specification [CCMW01]: *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 first sends a message and then receives a message (i.e., an outgoing message

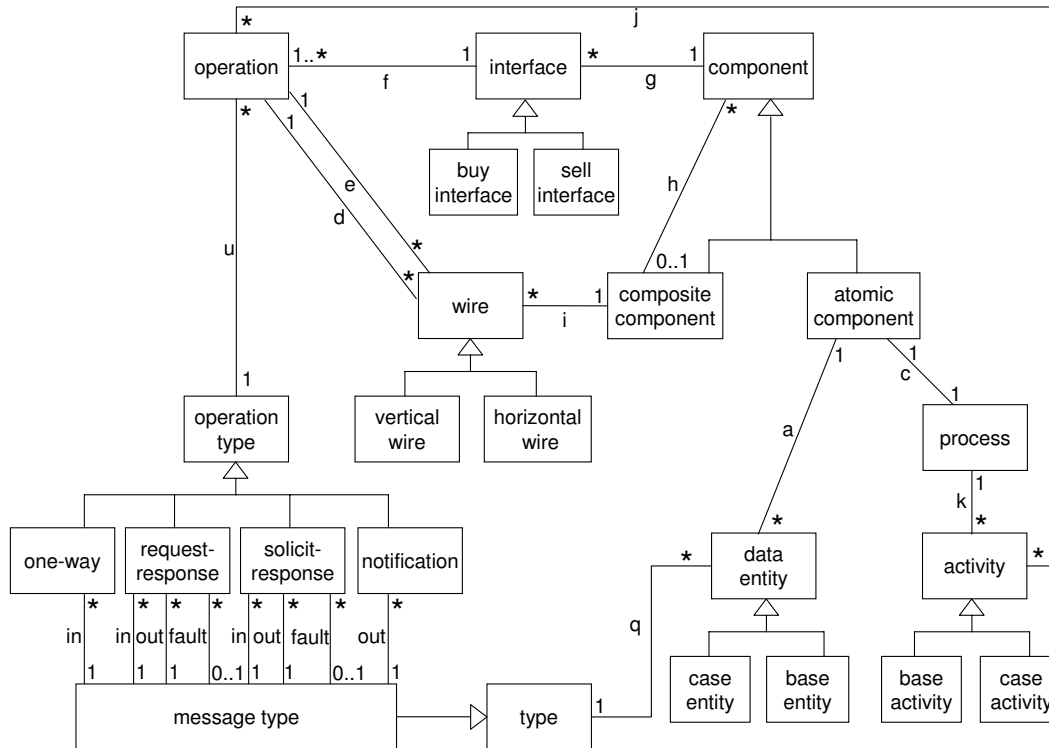


Figure 1: Abstract meta model of the architecture framework.

followed by an incoming message) whereas in case of a request-response, the component first receives a message and then sends a message (i.e., an incoming message followed by an outgoing message). In practise, 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 data entities of its atomic component by means of *method calls* (not shown in Fig. 1). These method calls may change the value of the data entities. A more detailed look at processes and data elements is presented in Sect. 4.2 and Sect. 4.3, 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. 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*. The two different wires are visualized

in Fig. 2.

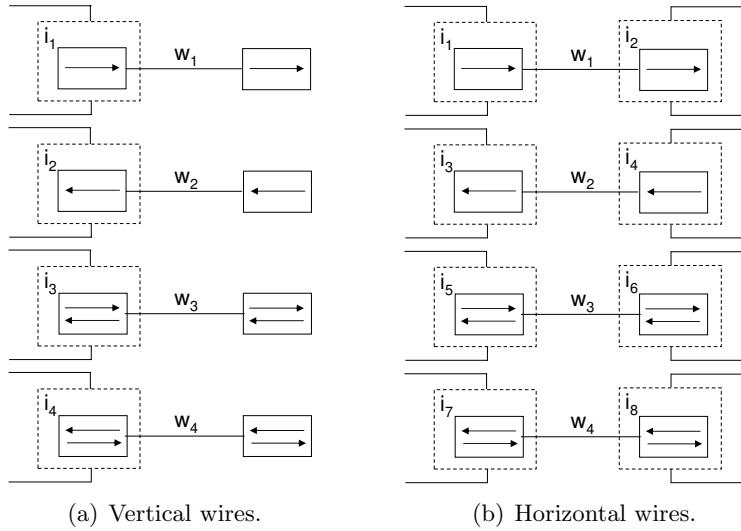


Figure 2: Different wiring concepts: In Fig. 2(a), for every pair of operations with the same operation type a vertical wire is shown which connects these operations. Wires  $w_1, \dots, w_4$  are depicted by a solid line. Interfaces  $i_1, \dots, i_4$  are depicted by a dashed frame. A box visualizes an operation. Its operation type is depicted by one or two arcs inside the box. Interface  $i_1$  has an operation with operation type notification,  $i_2$  one-way,  $i_3$  request-response, and  $i_4$  solicit-response. On the left hand of a wire, the interface of a component is shown. The component is sketched by a solid frame. On the right hand, only the propagated operation is shown. As can be seen from this figure, every wire connects only operations of the same operation type. Figure 2(b) presents the four different pairs of operations, this time connected by a horizontal wire. Again, a component is sketched by a solid frame. In contrast to a vertical wire, a horizontal wire connects two operations with complementing operation types. Wire  $w_1$ , for instance, connects a notification operation (interface  $i_1$ ) with a one-way operation (interface  $i_2$ ).

Most of the information about wiring operations cannot be derived from the meta model in Fig. 1. Later, in Sect. 4.5, we will therefore define the wiring using the Object Constraint Language.

Components support the concept of instances (not depicted in Fig. 1). As this mainly affects the process of a component, we introduce instantiation in Sect. 4.2.

The *state* of a component is determined by the value of data entities, the received or sent messages, and the state of its process (i.e., its activities). Each activity is changing the state. So, an architecture describes a transition system.

### 4.1.2 Interface and Wiring

Now, we take a more detailed look at the framework’s 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 adaptable to WSDL. In Fig. 3, an example component model is visualized. We use the same graphical notation as in Fig. 2.

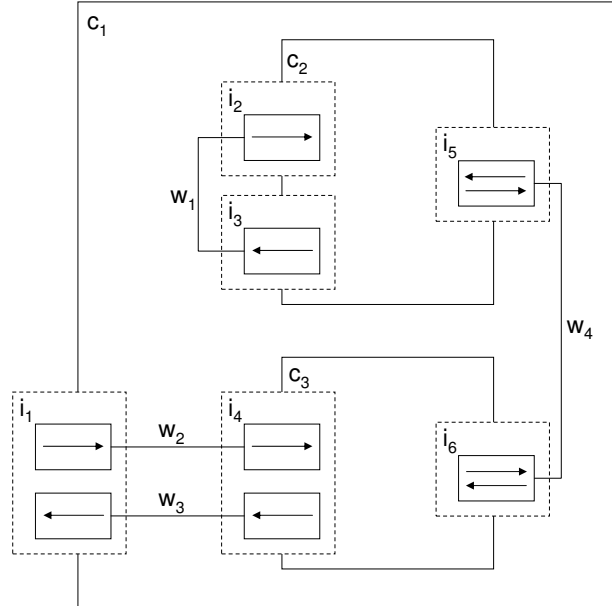


Figure 3: Example component model consisting of three components,  $c_1$ ,  $c_2$ , and  $c_3$ . Component  $c_1$  contains components  $c_2$  and  $c_3$ . It defines four wires  $w_1, \dots, w_4$  and six interfaces  $i_1, \dots, i_6$ . A horizontal wire connects operations of two components that have the same enclosing component. For example, wires  $w_1$  and  $w_4$  are horizontal wires. Wire  $w_4$  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  $w_3$  are examples of vertical wires. It can be seen that  $w_2$  and  $w_3$  connect operations of component  $c_2$  with operations of its enclosing component  $c_1$ . Interfaces  $i_3$  and  $i_6$  are buy interfaces. All other interfaces are sell interfaces.

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. As shown by wire  $w_1$ , it is not excluded to wire an operation to another operation of the same component. In order to clarify the difference between horizontal

and vertical wires, we present in Sect. 4.5 constraints that specify their behavior.

From relationships  $d$  and  $e$  in the meta model in Fig. 1, it can be derived that one operation may be part of several wires. The different possibilities of wiring are shown in Fig. 4. All the wires  $w_1, \dots, w_4$  are horizontal wires. Wiring one operation with another operation is depicted in Fig. 4(a). Wire  $w_1$  connects a one-way operation with a notification operation and  $w_2$  a request-response operation with a solicit-response operation. It is also possible to wire an operation to several operations having the same complementing operation type. As an example, consider the wiring in Fig. 4(b). In this figure, wire  $w_3$  and  $w_4$  wire a request-response operation (interface  $i_3$ ) with two solicit-response operations (interface  $i_4$ ).

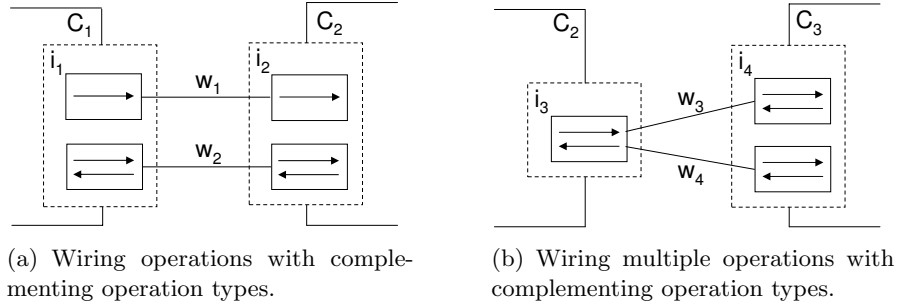


Figure 4: Different wiring concepts.

A wiring, as presented in Fig. 4(b), can be seen as a choice. If at runtime the corresponding activities of both operations are enabled, this choice is solved, for example, nondeterministically. If only the activity of one operation is enabled, the choice can be solved deterministically. Nevertheless, the framework should throw a warning to the developer, because this behavior might not be its intention. However, it is not possible to wire a request-response operation with a one-way and a notification operation, for instance.

On the level of activities and operations, it is possible that two or more activities share one operation. This is a valid behavior, but as the wiring in Fig. 4(b), it can be seen as a choice; that is, in case of an incoming message each activity can receive this message. If at runtime only one of these activities is enabled, the choice can be deterministically solved. Otherwise, the choice can be solved nondeterministically, for instance. Again, in such a situation, the framework should throw a warning to the developer. A deterministic choice, however, can be enforced by the static structure of the process. As an example, consider each activity being in a different OR-branch. Then, only one of these branches is executed and thus only one activity is enabled.

These examples show that the architecture framework relies on precise interpretations, hence we need formal semantics (as proposed in the general requirements in Sect. 2.3). The semantics formally defines the behavior of the architecture framework and is subject of Sect. 5. In addition, the architecture framework should also offer tool support to detect such design flaws.

## 4.2 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. Subsequent, 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 specializations of our process model.

### 4.2.1 Activities and Data Elements

As required in Sect. 2.3, a component has a set of activities. 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.

Figure 5 presents a detailed meta model of an atomic component in UML notation. It is used, in the following, to explain the relationship between entities “data entity” and “process”. A process contains a set of activities (relationship  $k$  in Fig. 5). 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. It 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 elements 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.

We have seen that for each activity data access by a method call is restricted to data elements within the activity’s sphere. However, this is often too restrictive as an activity might also need access data entities located in other components. For example, the ship process needs information about the container loaded on the ship. Therefore, the architecture framework supports a second mechanism to access data. To access data outside of an atomic component, the concept of message exchange can be used. This concept is not visible in Fig. 5, but in Fig. 1 (see relationship  $j$ ). Instead of calling a method directly, an activity sends a message to an operation (relationship  $j$  in Fig. 1) that passes it to another activity which contains the respective data entity in its sphere.

A component consists of two connected layers: a process layer and an data layer. The component’s activities form a process (see entity “process” in Fig. 5) and thus the process layer. The process layer can be seen as a *workflow model*. The data layer, in contrast, can be seen as a data base schema. It consists of the component’s data

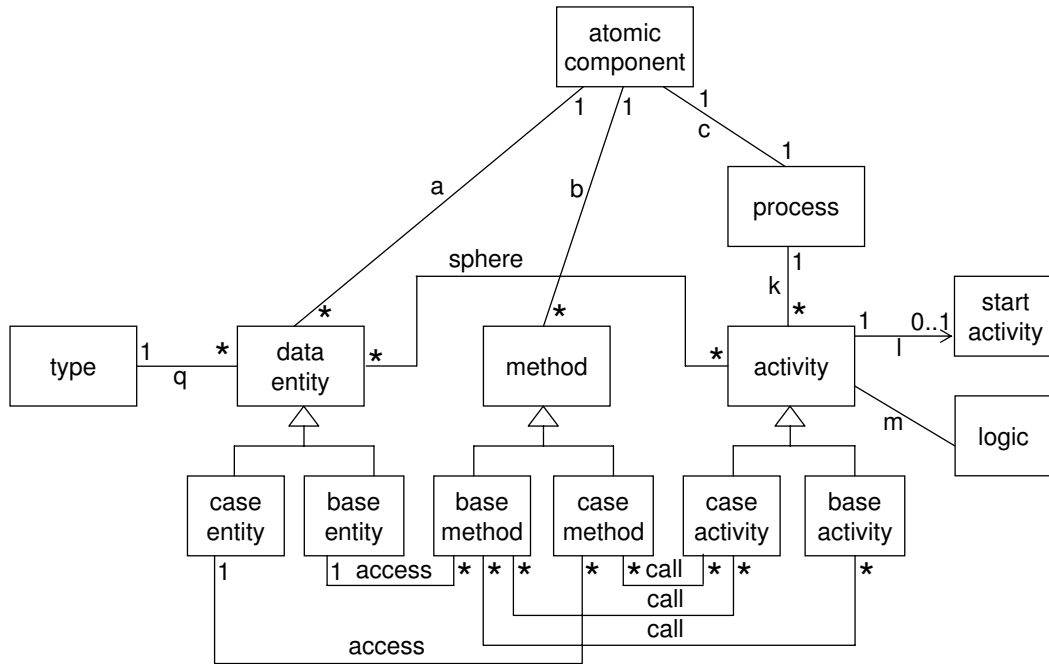


Figure 5: More detailed meta model of an atomic component with respect to the entity process.

entities. The process can communicate with the component’s environment by message exchange and it can access to data entities by method call. The relationships between entities “process” and “data entity” in Fig. 1, more precisely, the relationships between “activity”, “method”, and “data entity” presented in Fig. 5 show that process layer and data layer are connected. Neither entity “method” nor “message” (and the corresponding interface concept) are part of one of these two layers, but they can be seen as the glue between these two layers.

#### 4.2.2 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 the meta models in Fig. 1 and Fig. 5). 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. Fig. 1 and Fig. 5). 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 life-cycle of a component.

Once an (atomic) component is initialized its base process and its base elements (if the component contains them) are initialized, too. Thus, the initialization can be seen as the instantiation of the base process. Afterwards, the component can be instantiated. To create a *case*, i.e. a new instance, a *start activity* (see Fig. 5) 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. The message is received by the start activity, which then creates a case. A process may have more than one start activity, but there is no process which has a base and a case start activity. This fact will be specified by help of an OCL constraint in Sect. 4.5. The instantiation implies the creation of case activities and case elements which belong to exactly one case. Their life-cycle is restricted to its respective case. When a case has been finished, it can be destroyed. The life-cycle of base activities and base entities, in contrast, only ends once the component is deactivated.

In Fig. 6(a), an abstract view on the process of the dating service is shown. The process is drawn by a solid frame. On this frame, the two interfaces  $i_1$  and  $i_2$  are depicted. The dashed line inside the frame divides the process into case (on the left) and base (on the right). The case process consists of three activities,  $ca_1$ – $ca_3$ . Each activity is drawn as a rectangle and the arcs visualize the direction of their execution order; that is,  $ca_1$ – $ca_3$  are executed sequentially starting from  $ca_1$ . Activity  $ca_1$  is highlighted visualizing that it is a start activity. The case entity  $ce$  is depicted by a cylinder. If an activity has access to a data element or it is connected to an operation, then this is represented by a dotted line between the two entities (e.g., the line connecting  $ca_2$  and  $ce$ ). The base consists of a base activity  $ba_1$  and a base entity  $be$ . The pictograms are the same as for the case. The idea of the dating service is as follows: The customer registers at the dating service ( $ca_1$ ). Its profile is saved in the customer database  $be$ . Next, the dating service sends the confirmation together with the bill to the customer who replies by paying the fees ( $ca_2$ ). The payment information are stored in  $ce$ . Finally, the dating service looks for a matching partner in  $be$  and delivers the result to the customer ( $ca_3$ ). Figure 6(a) is an example of a process being instantiated by a start activity. More precisely, the registration which a customer sends to the first operation in interface  $i_1$  implies that  $ca_1$  creates a case.

A slightly different version of the process of the dating service is shown in Fig. 6(b). There are two additional activities, base activity  $ba_0$  and case activity  $ca_0$ , in this process. All other elements are identical with respect to Fig. 6(a). In contrast to Fig. 6(a), the process is instantiated by the base process.  $ba_0$  is a start activity and creates any number of cases. Each case can execute its case activity  $ca_0$  which can be seen as a preparation of the respective case. Afterwards, each case has to wait for a request. If a request arrives at the operation, it will be assigned to one of the cases. This assignment can be nondeterministically, for instance.

For a case created by a base process, we divide its life-cycle into two different phases. The first phase is called *preservation time*. During its preservation time, the case does not interact with other components. In the second phase, the *engage time*, the case interacts with other components (i.e., with cases of other components). The engage time ends when the case is destroyed. A case, which is created by a start activity, in contrast, has only an engage time and no preservation time. Back to our example in Fig. 6, the engage time of  $DS_1$  and  $DS_2$  is the execution of case activities  $ca_1$ – $ca_3$ . The

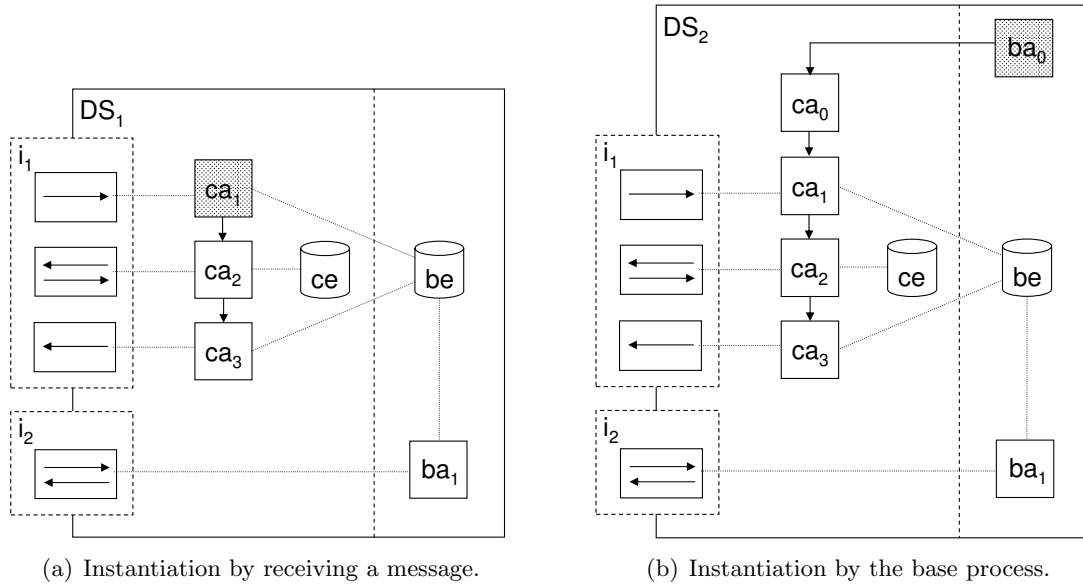


Figure 6: Different instantiation concepts.

preservation time of  $DS_2$  is the execution of case activity  $ca_0$ .  $DS_1$  has no preservation time, because it is instantiated by a start case activity.

Base activities and entities are independent of a specific case. More precisely, if a case has access to a base, then all cases have access, too. A base activity may create cases and may access base entities within its sphere. A case activity, however, may access case entities and base entities as shown in Fig. 5. It may also trigger base activities. In contrast, a base activity can neither access to case entities nor trigger case activities. Therefore, the connection between base and case processes is stronger in direction from case to base than the other way around. On this account, a base entity can be seen as a parameter. Base activities, however, are typically used for monitoring and configuration of a component (see the four interfaces mentioned in Sect. 2.1). In the example presented in Fig. 6, all customer profiles are saved in base entity  $be$ . This makes sense, as every case needs to add new profiles and get information about possible partners. However, for each case the information about the payment of a customer is stored locally in case entity  $ce$ . That means, after the customer received the contact information of a partner, its profile is saved in  $be$ , but information about its payment are deleted as  $ce$  is destroyed together with the case. Interface  $i_2$  in Fig. 6(a) and Fig. 6(b) is an example for a monitoring interface. Base activity  $ba_1$  is triggered by an incoming message and replies information about the process which are taken from the customer database  $be$ . Depended on the information saved in  $be$ , it might be also possible to monitor all cases. For this purpose, every case had to write its state information into  $be$ .

### 4.2.3 Message Correlation – Interacting Case Processes

In this section, we take a look at component interaction by message exchange. During the interaction, messages have to be delivered to the correct component's operation. However, component interaction is a more difficult task as several cases of every component 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. 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. In this section, we restrict ourselves to interaction between case processes. The differences, when base processes are also involved in the interaction, are discussed in the next section. Before we introduce our concept of message correlation, we present possible scenarios of component interaction to show the requirements of message correlation.

In the literature (e.g., in [Pap01]), *orchestration* and *choreography* are two widely used terms for service or component interaction. Each term describes a specific view on the interaction.

An orchestration describes the interaction from the perspective of one component. It is a view on the process model and the message exchange of this component. In contrast, a choreography is described from the perspective of all parties. It defines the observable behavior between all participating components. Thus, it is a view on the component model which shows all interacting components. A choreography does not necessarily consist of isolated cases only, but several cases of a component might be involved. For example, in our container transport system a single ship can load plenty of containers; that is, a single ship case interacts with several container cases.

Usually, the number of components, which are involved in the choreography, is fixed. We call such a choreography *static*. A static choreography is typically designed in a way that all parties arrange the interfaces of the components needed. Afterwards, each party can develop its component(s) and eventually – after all components are implemented – the interaction can start. However, it is also possible to design a *dynamic* choreography. For example, when a component searches for other components (maybe by help of a provider), the number of components involved in the choreography grows. The number of components may also decrease as a component might leave the choreography. So in a dynamic choreography, the number of involved components is not fixed, but it can increase or decrease.

Independent whether the choreography is dynamic or static, the interaction between the involved components starts and eventually ends. Let  $S$  and  $R$  be components in a choreography. In an interaction between  $S$  and  $R$ , there is one component,  $S$ , that takes the *initiative*; that is,  $S$  sends the first message and thus it starts the interaction between both components. 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* 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 fulfilled by  $R$  or it contains information that have to fulfill the requirements of at least one case of  $R$ . Obviously, if  $S$  sends a criterion, it is possible that there is 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, nondeterministically. If  $S$  creates a case of  $R$ , then  $R$  has a case start activity. Otherwise, the interaction between  $S$  and  $R$  starts by finding a case. As  $S$  has no reference to a case of  $R$ , it has to fulfill a criterion. This criterion can also be the empty set, meaning the message is matched by any case.

We now take a look at some examples for the start of an interaction. In our dating service, a person, say a boy, might call the service in order to find a partner. This is an example of a case creation, as the case of the boy's process creates a case of the dating service. For the second example, consider the case that the dating service finds a possible partner for that boy. The service sends the girl's contact information to the boy. So the boy has a reference to call the girl. That means, the case of the boy's process knows the case id of the girl's process and thus can interact with that case. As an example for a criterion, we refer to the container transport system. A shipper might transport a container to the harbor and publish information of the container's destination and the date when it has to reach its destination at the latest. Then, every ship that fulfills these criteria (destination and date) can decide to load this container.

If a case of  $R$  replies a message to  $S$ , the case has to decide whether it wants to send information about its case id or it wants to keep it anonymous. A shipper in the container transport system, for instance, does not need to know on which ship its container is transported. Thus, the ship can send an acknowledgement that the container has arrived at the destination harbor without publishing the ship's case id. It is also possible that  $R$ , the receiver, has no information about the sender  $S$ , because  $S$  did not send its case id and no other sufficient information can be derived from the message content to identify its case. For example, a shipper might call a ship transport company to ask for price information. For this purpose, the shipper does not need to identify.

Now, we define the term *correlation*. Correlation of a case  $c$  is *firstly the set of cases to which  $c$  has a reference to and secondly the set of components to which  $c$  sent its case id*. That means, the correlation contains all cases that can be directly called by  $c$ , because it knows their respective case ids. Furthermore, it contains a set of components and each component has at least one case which has a reference to  $c$ . 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*. The graph can be a complete graph. A complete graph is a graph in which each node has a directed arc (of each kind) to all other nodes. In component interaction, every state can be described by a correlation set.

As an example, we consider a choreography of three components  $C_1$ ,  $C_2$ , and  $C_3$ . These

components and their wiring are depicted in Fig. 4.  $C_1$  is the component on the left in Fig. 4(a) (with interface  $i_1$ ) and  $C_3$  the component on the right in Fig. 4(b) (with interface  $i_4$ ). Component  $C_2$  is the component that has an interface to  $C_1$  (right component in Fig. 4(a) with interface  $i_2$ ) and it also has an interface to  $C_3$  (left component in Fig. 4(b) with interface  $i_3$ ). The possible interaction of this choreography is illustrated in Fig. 7

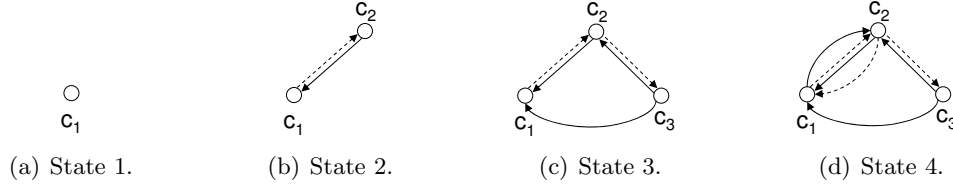


Figure 7: Correlation set for components in Fig. 4. The states of the interaction are described by the four correlation sets shown in 7(a)–7(d). Each node  $c_i$  depicts a case of component  $C_i$  ( $1 \leq i \leq 3$ ). A send arc is depicted by a dotted arc and a reference arc by a solid arc. In Fig. 7(a)  $c_1$ , the initiator case of component  $C_1$ , is shown.  $c_1$  sends a message to component  $C_2$  and it creates case  $c_2$ . This message contains the case id of  $c_1$ . The resulting state is shown in Fig. 7(b). For the created case  $c_2$  a node is drawn. There is a reference arc from  $c_2$  to  $c_1$  meaning that  $c_2$  knows the case id of  $c_1$ . The send arc from  $c_1$  to  $c_2$  visualizes the knowledge of case  $c_1$  about sending its case id to component  $C_2$ . Afterwards,  $c_2$  creates  $c_3$  by sending a message containing its case id and the case id of  $c_1$ . This correlation set is shown in Fig. 7(c). Case  $c_3$  has a reference arc to every case in this choreography and  $c_2$  a send arc to  $c_3$ .  $c_3$  replies, but it does not send its case id. Thus, the correlation set in Fig. 7(c) does not change. Finally,  $c_2$  replies to  $c_1$ . As this message contains the case id of  $c_2$ , a send arc from  $c_2$  to  $c_1$  and a reference arc from  $c_1$  to  $c_2$  is added in Fig. 7(d).

In our concept of message correlation, we formalize a *message format* by the following six tuple:

$$msg = (address_{snd}, address_{rec}, caseID_{snd}, caseID_{rec}, corrInfo, msgCnt) \quad (1)$$

The six tuple 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.

#### 4.2.4 Interacting Base Processes

In the last section, we introduced message correlation, a concept that makes it possible to deliver messages to a case, more precisely, to a case process. However, a process also consists of a base process and base activities may also communicate by message exchange. Therefore, this section deals with component interaction where the message is received by the component’s base process.

The base process is part of every running case of its enclosing component. For a sender, it is therefore sufficient to know the address of the receiving component. That means, the concept of message correlation has not to be applied to base processes. As an example, consider the monitoring interface  $i_2$  in Fig. 6(a). To monitor component  $DS_1$  it is not necessary to know the case id of a running case. This fact makes it necessary to define the semantics of an operation connected to a base and a case activity. Figure 8 presents an example component for this scenario. The process of component  $C$  consists of a base activity  $ba$  and two case activity  $ca_1$  and  $ca_2$ . Both,  $ca_2$  and  $ba$  are connected with the request-response operation in interface  $i_2$ . The behavior of this connection can be seen as a choice. Independent whether the message contains a case id, it is always possible that  $ba$  receives this message. If both activities are activated at runtime, the choice can be solved nondeterministically, for instance.

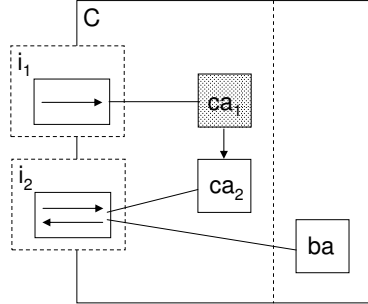


Figure 8: Operation connecting a case activity and a base activity.

#### 4.2.5 Specification of the Middleware

Components in our architecture can be wired. Usually, in complex systems the wiring of components is decoupled from the components itself. A software that connects components is a *middleware*. A middleware is a component itself. It helps to hide the complexity of a component from its environment. Possible tasks of a middleware are the routing of incoming and outgoing messages, for instance. In Fig. 9, an abstract model of the middleware is shown. In this figure, two components  $A$  and  $B$  are depicted. Each component consists of one case activity  $ca_A$  and  $ca_B$  and one interface  $i_A$  and  $i_B$ , respectively. The two operations are wired by  $w$ . For purposes of simplification, there is just a simple request-response interaction between these components.

The middleware in Fig. 9 is drawn as a cloud. It should realize the wiring of the two operations. More detailed, the middleware (virtually) connects the interfaces of both components and takes care that an outgoing message of the left component reaches the respective operation in the right component. The same holds for the reply message. Furthermore, the middleware has to evaluate the message in order to route it to the right case of the receiving component. For this purpose, the case id of the receiver (see the message format on page 22) has to be compared with the case ids of all running cases of the receiver's component. If an incoming message neither creates a case nor specifies

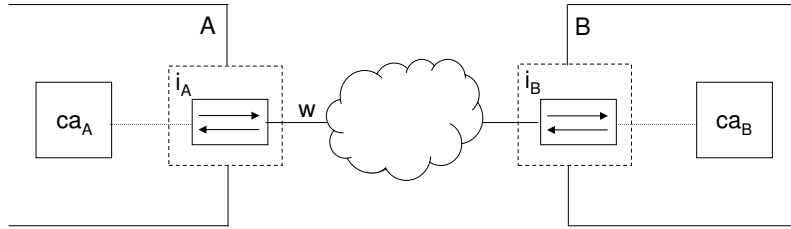


Figure 9: Abstract model of the middleware.

the receiver’s case id, the middleware has to evaluate whether there is a criterion that matches one of the running cases. With it, the correlation information and the sender’s case id (see the message format on page 22) are evaluated. If no match is found, the middleware sends an error message to the sender.

The routing is of course more complex as shown in Fig. 9. In Sect. 5, we will present an implementation of the middleware using a Petri net model. The model in Fig. 9, in contrast, can be seen as a specification of the middleware.

#### 4.2.6 Example Process Models

Our proposed architecture framework in Fig. 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 specialize the process model in Fig. 5.

The *Web Services Business Process Execution Language* (WS-BPEL) [AAA<sup>+</sup>06] is a widely-used language for describing the behavior 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 (invoke, receive, reply), manipulate or check data (assign, validate), wait for some time (wait), just do nothing (empty), signal faults (throw), or end the entire process instance (exit). 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 (sequence), parallel execution (flow), data-dependent branching (if), timeout- or message-dependent branching (pick), and repeated execution (while, repeatUntil, forEach). The most important structured activity is a *scope*. It links an activity to a transaction management subsystem and provides fault, compensation, and event handling. 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 Fig. 10.<sup>5</sup> 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

<sup>5</sup>Activities throw, rethrow, compensate, compensateScope, validate, and extensionActivity are not shown in Fig. 10.

and every structured activity can contain one or more activities. Entity “activity” in our architecture framework (see Fig. 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” (see Fig. 1 and Fig. 5). A “data entity” (not shown in Fig. 10) 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 WS-BPEL process if the attribute *createInstance* is set to *true*. In our framework, every activity can be a start activity. So we have to add this fact by a constraint. Also our concept of logic can be mapped to WS-BPEL: It is possible to specify XPATH expressions in WS-BPEL and there also exist extensions of WS-BPEL that allow, for instance, the integration of Java code into the WS-BPEL code [BGK<sup>+</sup>04].

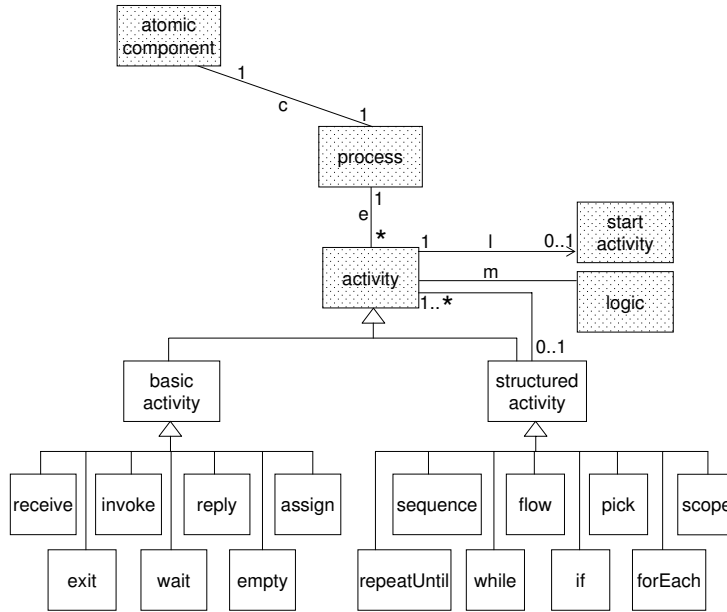


Figure 10: Meta model for WS-BPEL activities.

The formalism of Petri nets has been proven to be an adequate model for business processes (e.g., [Aal98]). A Petri net (see e.g., [Rei85] 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)<sup>6</sup>. Places can contain (black) tokens which represent a data value. We

<sup>6</sup>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. [BCH<sup>+</sup>03].

consider here Colored Petri nets (CPNs) [Jen92], an extension of usual Petri nets. In a CPN, tokens have a value (i.e., a color). That way, “real” data values can be modeled.

The Petri net meta model in UML notation is presented in Fig. 11. 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 Fig. 1, respectively. A start activity can also be modeled by a Petri net transition. More precisely, the transition has to generate a new case id. Entity “logic” in Fig. 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 modeled 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 colors, where each color represents exactly one case id, for instance.

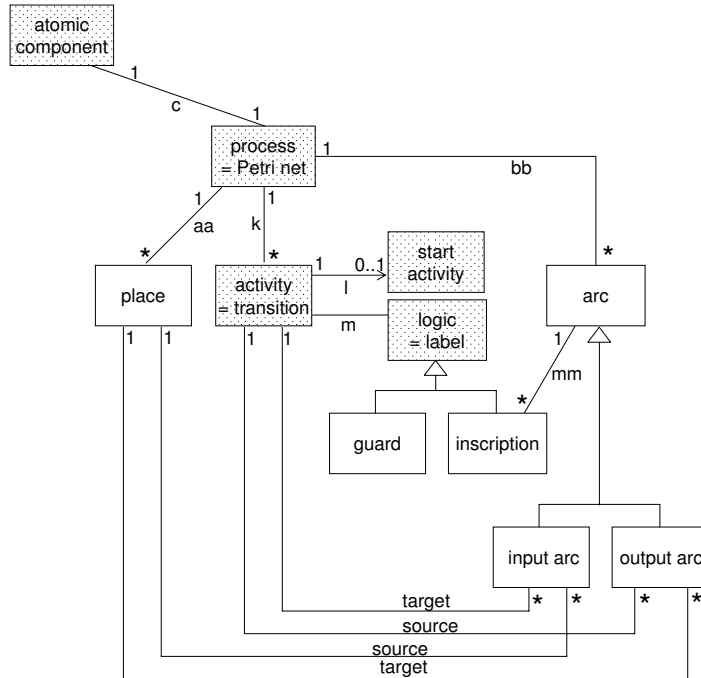


Figure 11: Meta model for Petri nets.

The WS-BPEL and Petri net meta models be seen as specializations of the original meta model in Fig. 5. In principle, other specializations are possible. As a kind of intermezzo we briefly discuss the relation between WS-BPEL and Petri nets. Petri nets are well-suited to model the control flow aspects of WS-BPEL’s activities. As WS-BPEL lacks of a formal semantics the transformation of WS-BPEL to Petri nets results in a formal model of the WS-BPEL process. With Petri nets several elegant technologies such as the theory of workflow nets [Aal98], a theory of controllability [Mar04, Sch05], a long list of verification techniques (e.g., [Mur89, McM93, DE95, Sch00]), and tools (e.g., [RWL<sup>+</sup>03, SR00, Sch00, Mäk02, VBA01, DMV<sup>+</sup>05]) become directly applicable.

Figure 12 shows a Petri net model for most of the WS-BPEL activities of Fig. 10. We use the common graphical notation. A place is denoted by a circle and a transition by a box. These Petri net models can be seen as patterns. Every pattern has an initial and a final place. A token on place *initial* models that the respective activity is activated. A token on place *final* shows that the activity was executed. Places annotated with *ch* on the left in Figures 12(b), 12(c), 12(d), and 12(i) model communication channels. A token on a channel place models a message. As an example, the transition in the receive pattern (Fig. 12(b)) can fire if there is a token on *initial* and *ch* modelling the activation of the pattern and a message in the channel, respectively. If the transition fires, the tokens on *initial* and *ch* are consumed (i.e., the message is received) and a token is produced on *final*.

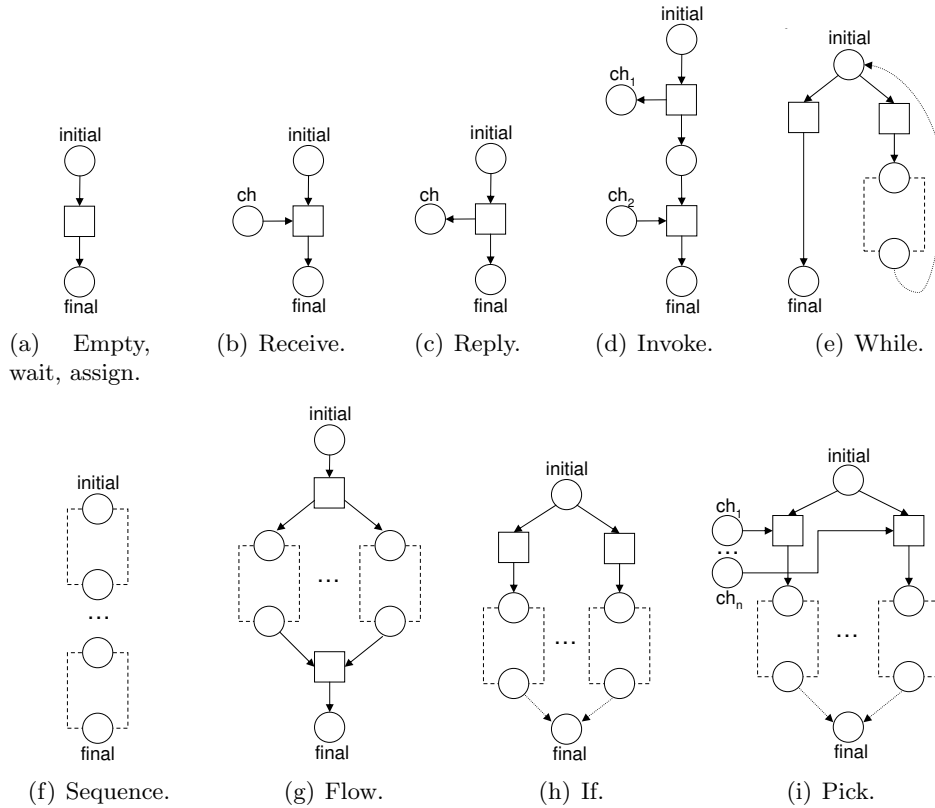


Figure 12: Some of WS-BPEL's activities modelled with Petri nets.

In the patterns of the structured activities (Figures 12(e)–12(i)) dashed frames are depicted. On each such frame an initial and a final place is drawn. Such a frame presents a wrapper for an arbitrary activity pattern. Every pattern can be plugged into this frame by merging its initial and final place with the initial and final place of the wrapper, respectively. For example, the empty pattern in Fig. 12(a) could be plugged into the wrapper of the while pattern in Fig. 12(e) by merging the initial place and the final place of empty and the frame, respectively.

A dotted arc connects two places being merged. In the while pattern, for instance, initial and the wrapper’s final place are merged. In the pick and if pattern, the final places of each wrapper are merged with the final place of the pattern. The patterns presented in Fig. 12 are simplified versions of the ones in [SHS05, OVA<sup>+</sup>05, LMW06]. In these papers, a feature-complete Petri net semantics for BPEL 1.1 [SHS05, OVA<sup>+</sup>05] and WS-BPEL [LMW06] was developed.

### 4.3 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 organizing of data entities. Often, architects start the system design with the development of the information architecture.

Besides component model and process model, our proposed architecture framework also offers a data model as a third view on the system. The data model is also a model of an atomic component. In contrast to the process model (cf. Fig. 5), where the focus was the entity “process”, the main focus of a data model is entity *data entity* in Fig. 1. In the following, we introduce the general concepts of the data model, in particular its hierarchy concept.

The data model of an atomic component is shown as a meta model in UML notation in Fig. 13. Like in the process model (compare Fig. 5), the starting point of the data model is again an atomic component. As a main difference, the entity *data element* is renamed to *composite data entity*. This change is needed to model the data aspect in a more refined way. A composite data entity forms the data layer of the atomic component which was introduced in Sect. 4.2.

A composite data entity is a set of *atomic data entities* (relationship *f* in Fig. 13) 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*). However, relationships *sphere* and *r* in Fig. 13 are on a higher level of abstraction, because activities have access to data by help of method calls (cf. Fig. 5 in Sect. 4.2) and messages are sent between activities (cf. Fig. 1 in Sect. 4.1).

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 possible to define a source data entity in one composite data entity (i.e., in an atomic component) and to have references

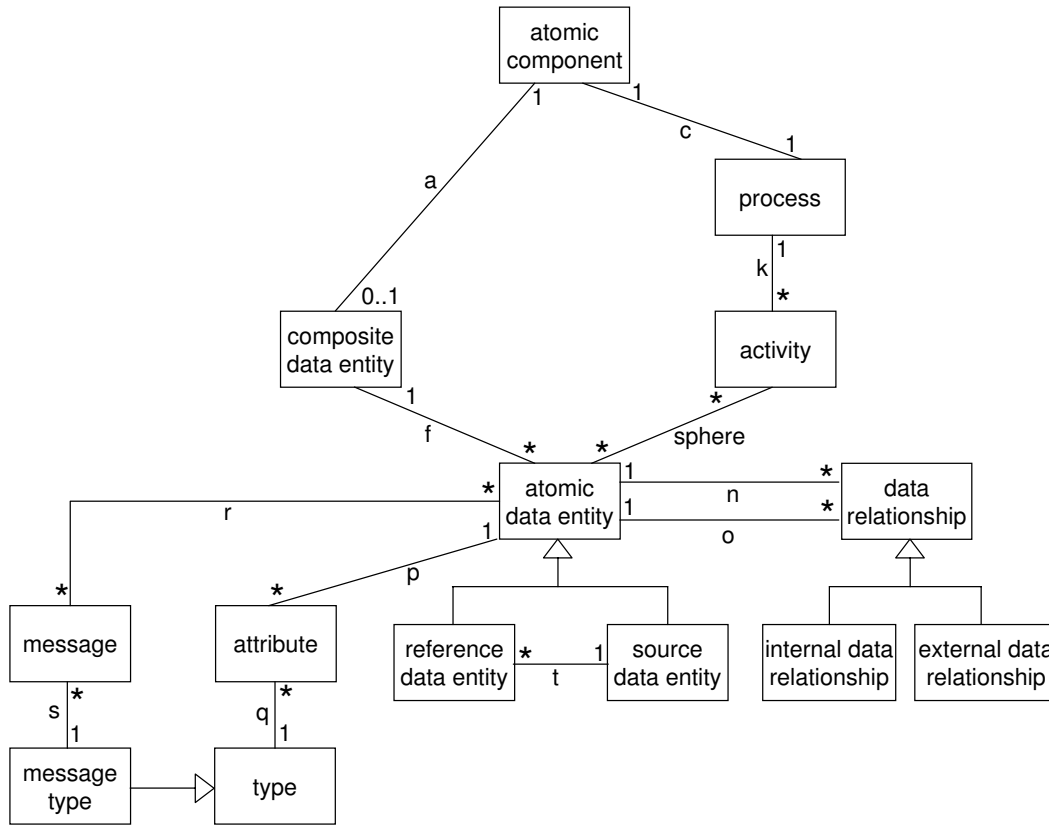


Figure 13: More detailed meta model of an atomic component with respect to the entity data element.

(by 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 Sect. 4.5 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 visualizes the relationship of all atomic entities in a composite entity. On the 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 Fig. 13) 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 Fig. 14. Note that we can have more levels of hierarchy by having a deeper hierarchy of (composite) components.

Now, let us have a look at the relation between internal data relationship and external

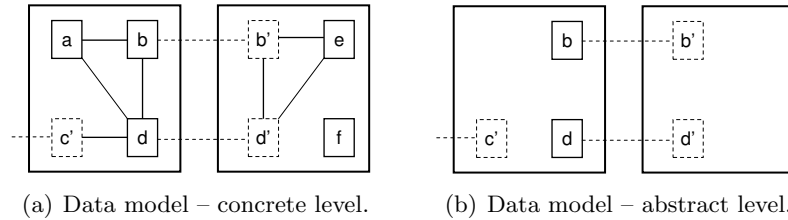


Figure 14: Two levels of hierarchy in the data model: A solid frame depicts an atomic component. Inside this frame the composite data entity is shown. Boxes  $a$ – $f$  depict atomic data entities. Solid boxes and dashed boxes visualize source data entities (e.g.,  $b$ ) and reference data entities (e.g.,  $b'$ ), 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 Fig. 14(a). The detailed view is shown in Fig. 14(a). All atomic data entities and their internal and external data relationships are visible. The abstract view is shown in Fig. 14(b). Only the three reference data entities, the two corresponding source data entities, and their external data relationships are visible.

data relationship on the 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 Sect. 4.2 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 Fig. 5). 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 Sect. 4.5.

The concept of message exchange between activities in combination with method calls is a very powerful concept. Message patterns, like message pull and message push, can be easily modelled.

So far, we restricted our attention to more complex structures like relationships between entities. Our data model, however, also allows for the specification of ordinary variables, used in programming languages, for instance. Such a variable can be specified as an atomic data entity which is not related to other entities. Entity  $f$  in Fig. 14(a) is an example.

#### 4.4 Relationship Between Component, Process, and Data Model

Not only the entities in the three meta models in Figures 1, 5, and 13 are related, but there are also relationships between these meta models. In the following, we will clarify what these relationships are.

Entities “atomic component”, “process” and thus “activity” are part of every meta model mentioned above. The process model and the data model presents the process and the data entities for one atomic component, respectively. This atomic component is part of the component model that shows the relationship between all components. Entity “message” is also contained in all three meta models. In the component and process model, messages are (indirectly) expressed by the relationship between activities and operations. The data model, in contrast, contains this entity explicitly.

Looking at the process and data model, further relationships can be found. Both meta models have an entity “data entity”. In the data model in Fig. 13, this entity is refined to atomic and composite data entity. The concept of an activity’s sphere then relates entities data element and activity. All these relations are visualized in Fig. 15.

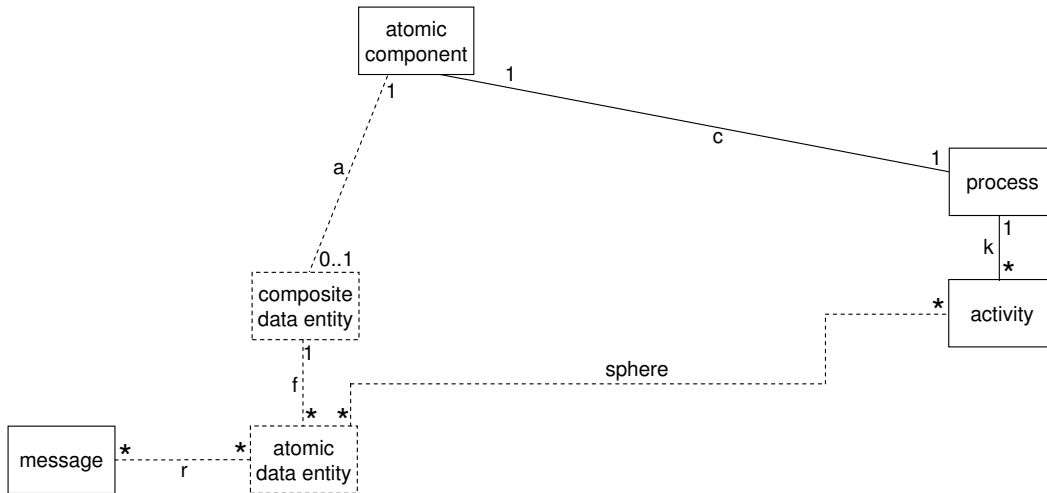


Figure 15: Relationship between the three meta models of the architecture framework. Entities and relationships drawn by a solid line are part of the component, process, and data model. Entities and relationships drawn by a dashed line are only part of the process and data model.

## 4.5 Constraints

In the previous sections, we introduced a meta model of our architecture framework (see Fig. 1). We further presented two refined meta models, one for the entity “process” in Fig. 5 and another one for the entity “data element” in Fig. 13. Figure 16 presents the detailed meta model of the architecture framework. It integrates the two refined meta models into Fig. 1.

The meta model in Fig. 16 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 *Object Constraint Language* (OCL) [OMG03]. 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 formalize concepts like wiring or the relationships between entities. Consequently, the constraints will restrict the meta model in Fig. 16 as well as the interface concept depicted in Fig. 2. All constraints are invariants specified in OCL. OCL keywords are depicted in bold font. For the parameters used we refer to the relationships in Fig. 16.

1. Relationship  $j$  between the entities activity and operation is redundant:

**context**  $ac$  : activity,  $op$  : operation **inv**:  
**if**  $op.j \rightarrow select(ac' \mid ac' = ac) \rightarrow size() = 1$  **and**  
 $a.j \rightarrow select(op' \mid op' = op) \rightarrow size() = 1$  **then**  $ac.k.c = op.f.g$

The keyword **inv** means that this OCL expression is an OCL invariant. This invariant is introduced for the context of an activity and an operation. Informally spoken, it specifies that whenever an activity  $ac$  is connected to an operation  $op$ , then  $op$  is part of an interface of a component which has a process containing  $ac$ . Thus, relationship  $j$  is redundant. In OCL, this condition can be expressed by an **if then** construct. The **if** condition specifies that  $ac$  and  $op$  have to be connected.  $op.j$  returns a set of activities connected to  $op$ . OCL offers a large number of predefined operators on sets (e.g., *select*, *size()*) which can be accessed by  $\rightarrow$ . By help of OCL's *select* operator we select from the set  $op.j$  all activities  $ac$ . As all activities have a unique name, we select only one activity. Using OCL's *size()* operator we check the existence of activity  $ac$  (i.e., whether only one element was selected). Similar we select from the set of operations connected to activity  $ac$  the operation  $op$ . In the **then** part of the invariant, we show that  $op$  and  $ac$  are elements of the same component. More precisely, the (atomic) component given by  $ac.k.c$  is the same as the one given by  $op.f.g$ .

2. A horizontal wire connects two components with the same enclosing component:

**context**  $w$ : horizontal wire **inv**:  $w.d.f.g.h = w.e.f.g.h = w.i$

A horizontal wire  $w$  connects two operations. The component(s) of these two operations are embedded in the same composite component.  $w.d.f.g.h$  specifies the composite component for the component of one operation and  $w.e.f.g.h$  the composite component for the component of the other operation. These composite components are equivalent and finally,  $w.i$  specifies that the wire  $w$  is defined in exactly this composite component. Wires  $w_1$  and  $w_4$  in Fig. 3 are examples of a horizontal wire.

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

**context**  $w$ : vertical wire **inv**:  $w.d.f.g = w.i$  **and**  $w.e.f.g.h = w.i$

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 left part of the conjunction specifies the enclosing component and the right part the inclosed component. More precisely,  $w.i$  specifies the enclosing component

which embeds the wire  $w$ .  $w.d.f.g$  specifies the component of one of the operations. This component is equivalent to  $w.i$  and thus the enclosing component. For the right part  $w.e.f.g$  specifies the component of the second operation and  $w.e.f.g.h$  returns its enclosing scope which is equivalent to  $w.i$ . Wires  $w_2$  and  $w_3$  in Fig. 3 are examples of vertical wires.

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

**context**  $w$ : vertical wire **inv**:

$$\begin{aligned} w.d.u.oclIsTypeOf(one - way) &= w.e.u.oclIsTypeOf(one - way) \text{ or} \\ w.d.u.oclIsTypeOf(notification) &= w.e.u.oclIsTypeOf(notification) \text{ or} \\ w.d.u.oclIsTypeOf(request - response) &= \\ w.e.u.oclIsTypeOf(request - response) &\text{ or} \\ w.d.u.oclIsTypeOf(solicit - response) &= \\ w.e.u.oclIsTypeOf(solicit - response) \end{aligned}$$

A vertical wire always connects two operations of the same operation type. Informally spoken, this invariant defines both operation types to be either of type one-way, notification, request-response, or solicit-response (compare Fig. 2(a)). This is realized by an OCL disjunction using **or**. In each disjunction,  $w.d.u$  and  $w.e.u$  are the two operation types of the operations wired by  $w$ . As there are four possible operation types, we use OCL's  $oclIsTypeOf()$  expression to specify the operation type in detail.  $w.d.u.oclIsTypeOf(one - way)$  returns true if the type of one-way and  $w.d.u$  are equivalent and false else. Wires  $w_2$  and  $w_3$  in Fig. 3 are examples for vertical wires.

5. A horizontal wire connects two operations with matching operation types:

**context**  $w$ : horizontal wire **inv**:

$$\begin{aligned} w.d.u.oclIsTypeOf(one - way) &= w.e.u.oclIsTypeOf(notification) \text{ or} \\ w.d.u.oclIsTypeOf(request - response) &= \\ w.e.u.oclIsTypeOf(solicit - response) \end{aligned}$$

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. Thus, in this invariant we specify that the two operation types must be either one-way and notification or request-response and solicit-response. Wires  $w_1$  and  $w_4$  in Fig. 3 are examples of a horizontal wire.

6. An activity can only call methods that have access to data elements within its sphere:

**context**  $ac$ : activity **inv**:

$$\begin{aligned} ac.call \rightarrow &forAll(md \mid md.access.sphere \rightarrow \\ select(ac' \mid ac' = ac) &\rightarrow size() = 1) \end{aligned}$$

Every activity  $ac$  can only call methods  $md$  that have access to atomic data entities within the sphere of  $ac$ . The respective OCL constraint consists of two parts. First, it collects the set of all methods  $md$  activity  $ac$  can call.  $ac.call$  returns the set of all methods  $ac$  can call. The  $forAll$  operation in OCL allows specifying a Boolean

expression, which must hold for all objects in this set. Thus, for every  $md$ , a Boolean expression must hold. This Boolean expression is described in the second part of the constraint. Every method  $md$  can access to one atomic data entity, which must be in the sphere of  $ac$ .  $md.access.sphere$  returns the set of activities which contain the respective atomic data entity within their sphere. Using the *select* operator we select all activities  $ac$  of this set. The existence of an activity  $ac$  is checked by the *size()* operator.

7. Every process has at least one start activity:

**context**  $pr$ : process **inv**:

$pr.k \rightarrow select(ac \mid ac.l \rightarrow notEmpty()) \rightarrow size() > 0$

In a process  $pr$ , there is at least one activity  $ac$  which is a start activity. The existence of such an activity is checked by applying OCL's *select* operator.  $ac.l$  is a set of either 0 or 1 elements and applying OCL's *isEmpty()* operator, we check whether this set is empty. Using the *size*, we check whether the set of activities  $ac$  is nonempty.

8. There is no process which has a base and a case start activity:

**context**  $pr$ : process **inv**:

**if**( $pr.k \rightarrow select(ca \mid ca.oclIsTypeOf(case\ activity) \mathbf{and} ca.l \rightarrow notEmpty())$ )

**then** ( $pr.k \rightarrow forAll(ba \mid ba.oclIsTypeOf(base\ activity) \mathbf{and}$

$ba.l \rightarrow isEmpty())$ ) **endif and**

**if**( $pr.k \rightarrow select(ba \mid ba.oclIsTypeOf(base\ activity) \mathbf{and} ba.l \rightarrow notEmpty())$ )

**then** ( $pr.k \rightarrow forAll(ca \mid ca.oclIsTypeOf(case\ activity) \mathbf{and}$

$ca.l \rightarrow isEmpty())$ ) **endif**

A process  $pr$  can never have a base start activity and a case start activity. This invariant is expressed by two OCL **if** expressions. In the first if expression, we check whether there exists a case activity  $ca$  in process  $pr$  which is a start activity.  $ca.l$  is a set of either 0 or 1 elements. Applying OCL's *notEmpty()* operator, we check whether the resulting set is nonempty. If this holds, there is no base start activity in  $pr$ .  $ba.l$  is a set of either 0 or 1 elements and applying OCL's *isEmpty()* operator, we check whether this set is empty. In the second if statement, we do the same check for a base activity  $ba$ .

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

**context**  $x$ : internal data relationship **inv**:  $x.n.f = x.o.f$

An internal data relationship between two atomic data entities only exists if the entities are contained in the same composite entity and thus in the same atomic component. As an example see the relationship between  $a$  and  $b$  in Fig. 14(a).

10. 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**  
 $((x.n.oclIsTypeOf(\text{reference data entity}) \text{ and}$   
 $x.o.oclIsTypeOf(\text{source data entity}) \text{ and } x.n.t = x.o) \text{ or}$   
 $(x.n.oclIsTypeOf(\text{source data entity}) \text{ and}$   
 $x.o.oclIsTypeOf(\text{reference data entity}) \text{ and } x.o.t = x.n))$

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 four and five). To check the type of an atomic data entity, we use the already known *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  $x.o$ .

11. External data relationship means message exchange:

**context**  $A, B$ : activity,  $x$ : external data relationship,  $y$ : message,  $z$ : wire **inv**:  
**if**  $x.n.f \neq x.o.f$  **and**  
 $x.n.sphere \rightarrow \text{select}(a_1 \mid a_1 = A) \rightarrow \text{size}() = 1$  **and**  
 $x.o.sphere \rightarrow \text{select}(a_2 \mid a_2 = B) \rightarrow \text{size}() = 1$  **then**  
 $z.d.j \rightarrow \text{select}(a_3 \mid a_3 = A) \rightarrow \text{size}() = 1$  **and**  
 $z.e.j \rightarrow \text{select}(a_4 \mid a_4 = B) \rightarrow \text{size}() = 1$  **and**  
 $z.d.u.s \rightarrow \text{select}(m_1 \mid m_1 = y) \rightarrow \text{size}() = 1$  **and**  
 $z.e.u.s \rightarrow \text{select}(m_2 \mid m_2 = y) \rightarrow \text{size}() = 1$  **and**  
 $(y.r \rightarrow \text{select}(e_1 \mid e_1 = x.n) \rightarrow \text{size}() = 1$  **and**  
 $y.r \rightarrow \text{select}(e_2 \mid e_2 = x.o) \rightarrow \text{size}() = 1)$

Whenever there are two atomic data entities related by an external data relationship  $x$ , one in the sphere of activity  $A$  and the other in the sphere of activity  $B$ , then  $A$  and  $B$  exchange a message  $y$ . More precisely, there exists a wire  $z$  wiring an operation of  $A$  to an operation of  $B$  and there is a message that exchanges the value of the atomic data entities. The invariant specifies in the **if**-part that  $x$  is an external data relationship (line three) and entity  $x.n$  and  $x.o$  are in the sphere of activity  $A$  and  $B$ , respectively (line four and five). To check the existence of  $A$  and  $B$ , we use OCL's *select* and *size()* operators. Line six and seven specify that  $A$  and  $B$  are wired. At least one message  $y$  is sent using wire  $z$ . Thus,  $y$  has to pass the operation of  $A$  ( $z.d$ ) and  $B$  ( $z.e$ ). This is specified in line eight and nine. In the last two lines, the relationship between message  $y$  and both atomic data entities  $x.n$  and  $x.o$  is specified.

In conclusion, these constraints can merely be seen as examples. Once these constraints 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. There are many types of errors which cannot be checked on this abstract system level. For example, the soundness property [Aal98] has to be checked on the process model (e.g., on the Petri net). To apply formal verification techniques on the level of our architecture framework, it is necessary to develop a formal semantics for the architecture framework.

## 4.6 Comparing the Architecture Framework with the Requirements

During the last sections, we introduced our architecture framework and its concepts. In the following, we compare these concepts with the general requirements we collected in Sect. 2.3. Table 1 summarizes this comparison.

As can be seen from this table, the concept of inheritance is the only requirement which is not supported by our architecture framework so far. Subsequent, we comment every column of Table 1.

Components are the basic concept in our architecture framework. The four kinds of interfaces are not directly supported, but they can be modeled. Software and user interface can be modeled by an interface whose operations are either connected to base or case activities. In contrast, the operations of a configuration and monitoring interface are only connected with base activities (see Fig. 6, for instance). The concept of wiring components at their interfaces facilitates an easy “plug and play” of interfaces. Furthermore, a component has a process (i.e., a set of activities) and data entities.

Instantiation is an important concept in our framework. The framework offers two possibilities: Either a case start activity creates the case or a base start activity does so. A case start activity is triggered by an incoming message whereas a base start activity has no trigger. To make the concept of instantiation more expressive, the framework distinguishes between activities and data related to a single case (i.e., case activities and case entities) on the one hand and activities and data related to all cases (i.e., base activities and base entities) on the other hand. Messages in our architecture framework can be delivered to their correct case using the concept of message correlation.

Components in our framework do not explicitly support fault and compensation handling. However, process models like WS-BPEL offer fault and compensation handling which can then be used by the component. There is also no direct support for a monitoring service, but, as mentioned above, a monitoring service can be easily modeled by base process.

As a general requirement, an architecture framework should support relationships between components. Interaction relationship is facilitated by activities. Activities may exchange messages by using operations. Message exchange can be either synchronous or asynchronous depended on the message exchange pattern (i.e., the operation type) used. One-way and notification can be used for asynchronous message exchange and request-response and solicit-response for synchronous message exchange. Communication by shared data entities is also supported, because spheres of activities do not have to be disjoint. That means, multiple activities within a component can share their data entities. Reference and source data entities support shared data between different

Table 1: Almost all general requirements, which were presented in Sect. 2.3, are supported in the proposed architecture framework.

<b>Requirement</b>	<b>How supported in architecture framework</b>
components are the basic concept	yes
software, user, configuration, and monitor interface	no explicit support, but can be modeled
“plug and play” of interfaces	yes, by wiring
process (set of activities)	yes
data entities	yes, atomic and composite data entities
instantiation	two concepts, either by receiving a message or by the base process
distinguish between data and activities belonging to a single instance and to all instances	base and case entities; base and case activities
message correlation	yes, concept for base and case processes
component has fault and compensation mechanism	no explicit support, but can be part of the process model (e.g., WS-BPEL)
component has monitoring service	no explicit support; can be easily modeled by a base process
interaction relationship (communication by message exchange and shared data entities)	both concepts supported; components can communicate and data is shared within a component and across the borders of components by reference data entities that can be updated with their respective source data entity by message exchange
refinement as a design technique	yes, in the component model by help of atomic and composite components and in data model by help of reference and source data entities
inheritance	not integrated
open to plug in process formalisms	yes, shown for WS-BPEL and Petri nets in Sect. 4.2, for instance
open to plug in data models	yes, use OCL to define constraints
open to plug in a language to define operations of activities	yes, in the meta model we only speak about logic which can be specified and implemented in any language
support of a formal semantics	yes, but not fully worked out
close to industrial description techniques	yes, because process and data model are pluggable and so can be either exchanged or transformed

components.

The architecture framework also supports refinement as a design technique. Refinement is offered in the component and the data model. Components are either composite components and embed other components or atomic components. In the data model, the framework offers composite data entities and basic data entities on the one hand and source and reference data entities on the other hand.

Further, the framework is required to be open to plug in different formalisms. In Sect. 4.2, we presented that WS-BPEL and Petri nets, two process models, can be easily linked into the architecture framework. The same holds for the data model which is as the process model very general. Operations for activities are modeled by entity “logic” in Fig. 5 and thus can be defined in every programming language.

In the next section, we will formalize the proposed architecture framework. We present rules, how the architecture framework can be transformed in to the formalism of CPNs [Jen92]. The framework is also close to industrial description techniques. So implementations of the data and the process model can be easily linked into the framework. In Sect. 6.2 we also compare our framework with SCA.

## 5 Formalization of the Architecture Framework

In this section, we formalize our proposed architecture framework. To this end, we present rules that show how to transform the architecture framework into the formalism of Colored Petri nets (CPN) [Jen92]. The aim of this paper is not to present a formal semantics which is worked out in every detail. Instead, we show this formalization by example. We present a transformation of the dating service (cf. Sect. 3) into CPN models. From this example, the general transformation rules can be easily derived. Before we present the CPN models, we show the three views on the dating service. For it, we use our already known graphical notation.

### 5.1 Component Model of the Dating Service

In Fig. 17, an abstract component model of the dating service,  $C_{DS}$ , is shown. It has four interfaces:  $i_{DS-Cust}$ ,  $i_{DS-MF}$ ,  $i_{DS-Ch1}$ , and  $i_{DS-Ch2}$ . The first interface is the interface to the customer. All the other interfaces are used to monitor or configure the component.

Before we present the detailed component model of the dating service, we will introduce the component models of its most important components, *Match*, *Story*, and *Check* which are all depicted in Fig. 18.

Component  $C_{Match}$  in Fig. 18(a) implements the main functionality of the dating service. It receives the customer’s login information, administrates this information, and is also responsible to accept the payment of the customer and to send information of a matching partner to the customer. The matching component communicates with the customer via interface  $i_{M-Cust}$ . Via interface  $i_{M-Ch}$ , the matching component invokes component  $C_{Check1}$  which checks whether the customer is a known marriage swindler. Interface  $i_{M-St}$  is used to invoke component  $C_{Story}$  to prepare everything such that the

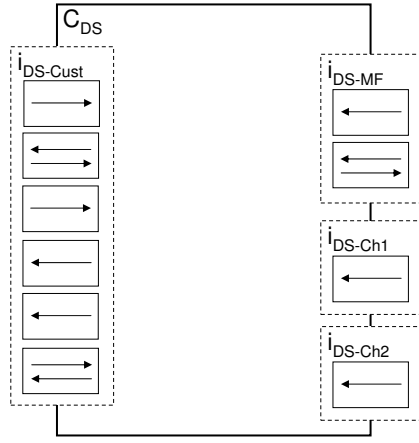
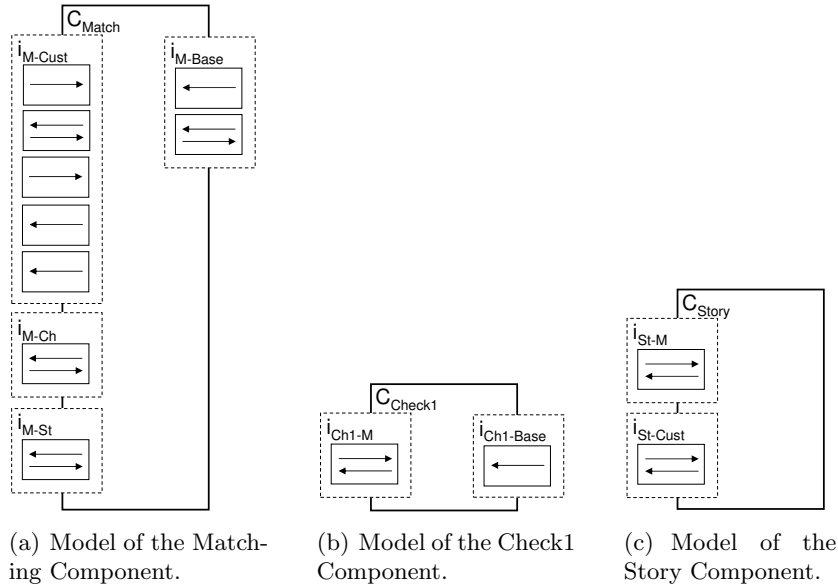


Figure 17: Abstract component model of the dating service.



(a) Model of the Matching Component. (b) Model of the Check1 Component. (c) Model of the Story Component.

Figure 18: Three components of the dating service.

customer can publish a success or failure story about its date. Component  $C_{Match}$  can be configured and monitored by help of interface  $i_{M-Base}$ .

Component  $C_{Check1}$ , which is presented in Fig. 18(b), is invoked by component  $C_{Match}$  (interface  $i_{Ch1-M}$ ). It checks whether a customer is a known marriage swindler. This component has also a configuration interface ( $i_{Ch1-Base}$ ) which is used to update the data base storing all the marriage swindlers. There is also a second check component  $C_{Check2}$ . However, this component is not depicted in Fig. 18, because it has the same interface as component  $C_{Check1}$ .

Figure 18(c) depicts the component model of the story component  $C_{Story}$ . As already

mentioned, this component interacts with component  $C_{Match}$  (interface  $i_{St-M}$ ). It receives from the customer the story about the date (interface  $i_{St-Cust}$ ). This story is then published.

Figure 19 presents the detailed component model of the dating service. Component  $C_{DS}$  contains three components: the two check components  $C_{Check1}$  and  $C_{Check2}$  and component  $C_{MainFunc}$ . The latter is a composite component that contains the two basic components  $C_{Match}$  and  $C_{Story}$ . The customer interfaces  $i_{M-Cust}$  and  $i_{St-Cust}$  of these two components are merged to interface  $i_{MF-Cust}$  in component  $C_{MainFunc}$ . It is important to mention that the concept of wiring one interface with multiple interfaces is used in the dating service model: Interface  $i_{MF-Ch}$  is wired with interfaces  $i_{Ch1-M}$  and  $i_{Ch2-M}$ ; that is, the matching component is wired with both check components. For example, if the check of the customer is subject to fee, the dating service might always choose the check component with the lowest fee. In our example, however, this conflict is nondeterministically solved at runtime.

## 5.2 Process Model of the Dating Service

In this section, we introduce the process models of the three components *Match*, *Story*, and *Check*.

The process model of component *Match* is depicted in Fig. 20. Its interface is known from Fig. 18(a). First of all, let us have a look at the case process (which is to the left of the dashed vertical line). The case process consists of seven case activities  $ca_1 - ca_7$  and four case entities  $c_{name}$ ,  $c_{profile}$ ,  $c_{address}$ , and  $c_{account}$ . These data entities store the name, the dating profile, the address, and the bank account of the customer. With address we mean here the customer's contact data; that is, its component address and case id. For purposes of readability, the spheres of the case activities are not depicted in Fig. 20. They are defined as follows:

$$\begin{aligned}
 sphere(ca_1) &= \{c_{name}, c_{profile}, c_{address}\}, \\
 sphere(ca_2) &= \{c_{name}, c_{profile}, c_{address}, b_{name}, b_{profile}, b_{address}\}, \\
 sphere(ca_3) &= \{c_{account}, c_{address}, b_{price}\}, \\
 sphere(ca_4) &= \emptyset, \\
 sphere(ca_5) &= \{b_{name}, b_{profile}, b_{address}\}, \\
 sphere(ca_6) &= \{c_{address}\}, \\
 sphere(ca_7) &= \{c_{name}, c_{profile}, c_{address}, b_{name}, b_{profile}, b_{address}\}.
 \end{aligned}$$

The process model works as follows: If the login information of a customer is received, start activity  $ca_1$  creates an instance of the case process. The customer's name, profile, and address is stored in the respective case data entities. Next, to check the customer, the check component is called (activity  $ca_2$ ). If the customer is identified to be a marriage swindler, activity  $ca_7$  is executed and it sends a goodbye message to the customer and deletes the customer's data. Otherwise, the customer's data are saved in the corresponding base data entities (and are thus available for the matching process). Then,

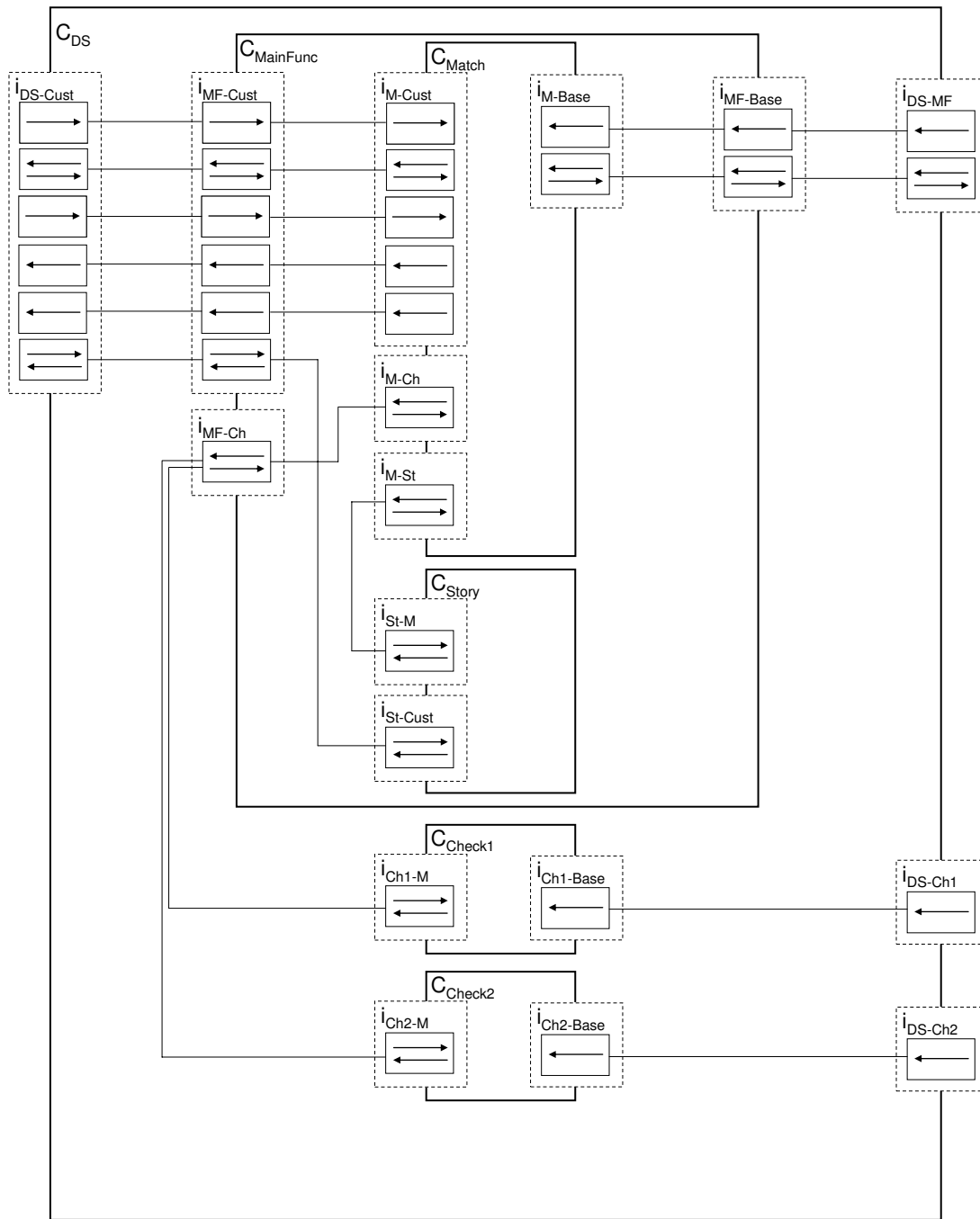


Figure 19: Detailed component model of the dating service.

activity  $ca_3$  sends the bill to the customer and receives the customer's bank information (which are saved in  $c_{account}$ ). Afterwards, the customer may ask for a matching partner or leave the dating service (activity  $ca_4$ ). In case that he leaves, activity  $ca_7$  is executed.

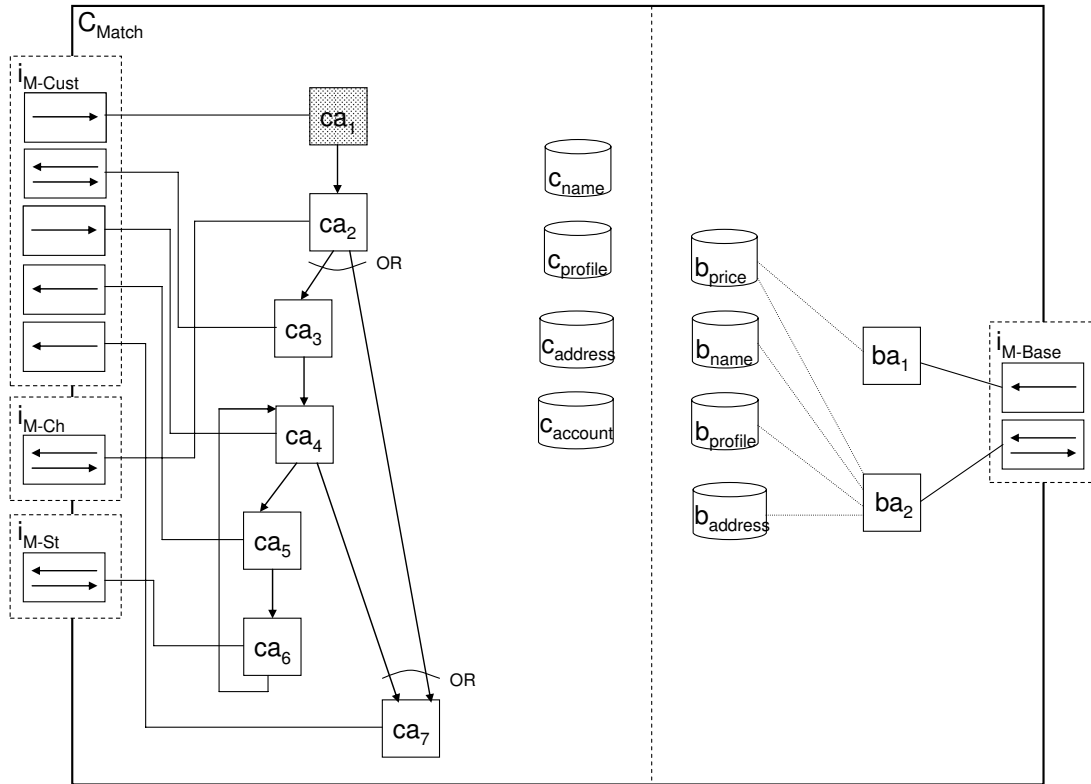


Figure 20: Process model of component  $C_{Match}$  – without spheres of the case activities.

Otherwise, if he asks for a partner, the component looks for a matching partner and sends him the contact data of this partner (activity  $ca_5$ ). Next, the matching component sends information to the story component such that this component can identify the customer. That way, the customer can publish the story of its date. Finally, by help of the loop back to activity  $ca_4$ , the customer may continue the dating process or leave the dating service. The customer's name, profile, and address is stored in the case and in the base data, because this information can only be stored in the base data if the customer is checked to be not known as a marriage swindler.

The base process of component *Match* is simpler than its case process. The base process only consists of two activities,  $ba_1$  and  $ba_2$ . The former is used to configure the matching component; that is, to change the fee customers have to pay. The latter is connected to the monitor interface and thus collects all base data and sends it to the requester.

Figure 21 depicts the process model of component *Check1*. Its case process consists of two case activities and one data entity which stores the name of the customer to be checked. The base process has one base activity and one base data entity. The latter stores the names of all known marriage swindlers. When the check component is called by the matching component, case activity  $ca_1$  creates an instance of the case process.

The customer's name sent by the matching component is stored in  $c_{name}$  and it is checked whether this name matches with an entry in  $b_{name}$ . The result of this check is replied to the matching component. Activity  $ca_2$  then deletes the process instance. To update the base data, the configuration interface  $i_{Ch1-Base}$  and thus activity  $ba_1$  is used. As component *Check2* has the same process model, we do not show it.

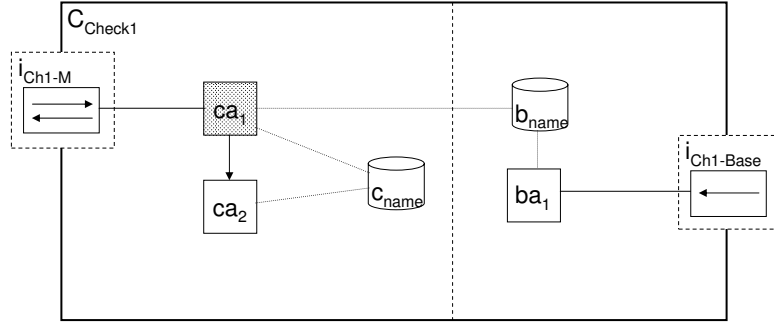


Figure 21: Process model of component Check1.

Finally, the process model of the story component is shown in Fig. 22. The process has four case activities and two base data entities that store the customer's name ( $b_{name}$ ) and its story ( $b_{story}$ ). This component works as follows: When a customer, say  $A$ , receives information about a possible partner, say  $B$ , both,  $A$  and  $B$ , have to write a story about their date. For this purpose, the instances of the matching component from  $A$  and  $B$  send the correlation information about their respective customer to the story component (activity  $ca_1$ ). Thereby, the first message creates an instance of the case process. Afterwards,  $A$  and  $B$  have to send their part of the dating story and the story component replies a message to both ( $ca_2$  and  $ca_3$ , respectively). Then, the two instances of the matching component are informed that the stories have been published ( $ca_4$ ).

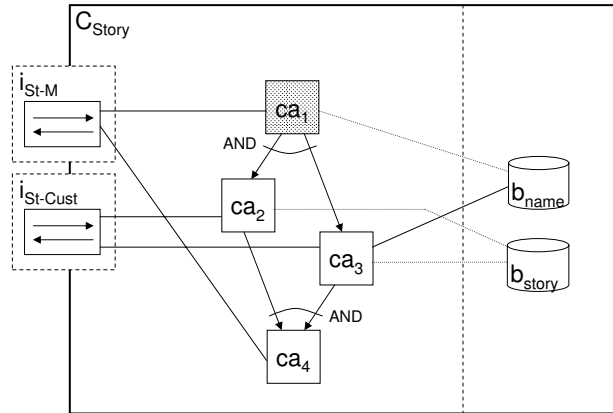


Figure 22: Process model of component Story.

### 5.3 Data Model of the Dating Service

In the following, we present the data models of the matching, check, and story component using UML class diagrams.

Figure 23 visualizes the data model of the matching component. Every customer has a bank account, a profile, and a unique address (component address and case id). From entity “person”, it inherits a name. Furthermore, also the fee, the customer has to pay, is modeled by entity “price”.

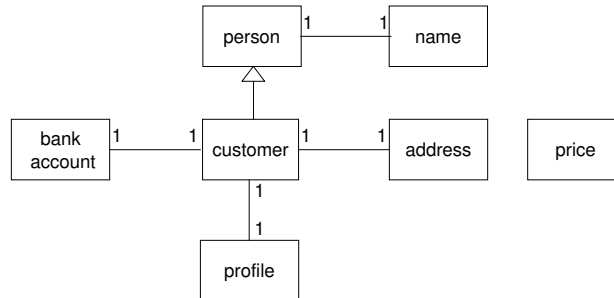


Figure 23: Data model of component Match.

The data model of the check component, depicted in Fig. 24, is similar to the data model of the matching component. By help of a directed association, we express that a person might be a marriage swindler.

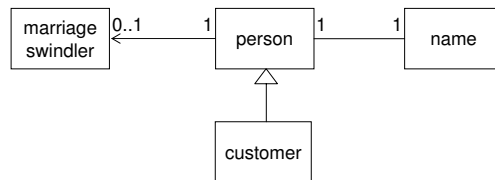


Figure 24: Data model of component Check1.

Figure 25 presents the data model of the story component. A customer is an author and every author can be author of several parts of a story (because of having more than one date). Two parts make a story.

### 5.4 Colored Petri Net Model of the Dating Service

A Colored Petri net (CPN) is an extension of usual low-level Petri nets. As a main difference, tokens in a CPN are not restricted to be black tokens, but they can be of any type (i.e., color). Therefore, places have a type and only contain tokens of this type. In addition, it is possible to define functions that can change the value of tokens. These functions are depicted as arc inscriptions. Further, a transition guard, a Boolean expression, can be added to a transition. This transition is then only activated if its

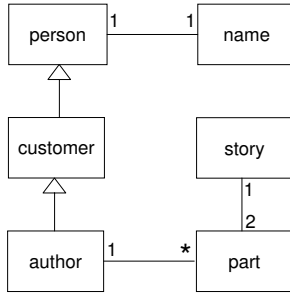


Figure 25: Data model of component Story.

guard is evaluated to true. For a more detailed introduction into CPNs we refer the interested reader to Jensen [Jen92].

Before we present the CPN models of the dating service, we explain the general concepts of the transformation process.

#### 5.4.1 General Transformation Rules

To understand the CPN models, we have to explain, how interfaces, activities, and data entities are transformed into CPNs. Furthermore, on the level of CPNs we also have to take the concept of message correlation into account.

In our proposed architecture framework, an interface contains one or more operations (see Fig. 16 for details). Every operation is transformed into one or two Petri net places. More detailed, every operation of type notification or one-way is transformed into one place and every request-response or solicit-response operation is transformed into two places (one place for the incoming message and the other for the outgoing message). Such a place can be seen as a *message channel*. A message is modelled by a Petri net token of type message; that is, the six tuple  $(address_{snd}, address_{rec}, caseID_{snd}, caseID_{rec}, corrInfo, msgCnt)$  introduced in Sect. 4.2.

As already explained in Sect. 4.2 (see Fig. 11), an activity is modelled by a Petri net transition. A place in the CPN model models a data store. We use one place to model all case data entities and another place to model all base data entities. Both places are of type list of  $(id, data\ entity\ name, data\ value)$ . In case the place models case data,  $id$  is the case id. Otherwise, if the place models the base data,  $id$  represent a primary key. For purposes of simplification, we model a case id by a natural number.

#### 5.4.2 Transformation of Component Match

We start with the transformation of the component model of the matching component presented in Fig. 18(a). Figure 26(a) depicts a modified version of this component model. In addition to Fig. 18(a), we have labeled each operation. For example, the notification operation in interface  $i_{M-Base}$  has the label  $c\_config$  and the incoming and outgoing part of the solicit-response operation has the label  $c\_monitor$  and  $c\_info$ , respectively. The model in Fig. 26(a) is transformed into the CPN model in Fig. 26(b). As operations

are transformed into places, we use these labels in the CPN model to label the channel places and can thus identify the corresponding operation. For example, the notification operation in interface  $i_{M-Base}$  is transformed into a single place  $c\_config$  and the solicit-response operation into two places  $c\_monitor$  and  $c\_info$ . Additionally, each place is labeled with *In* or *Out*. The label shows if it is an incoming or outgoing channel. All places are of type *MSG*; that is, the type representing a message, and connected with the hierarchy transition *Group 1*. A hierarchy transition represents a (transition-bordered) subnet.

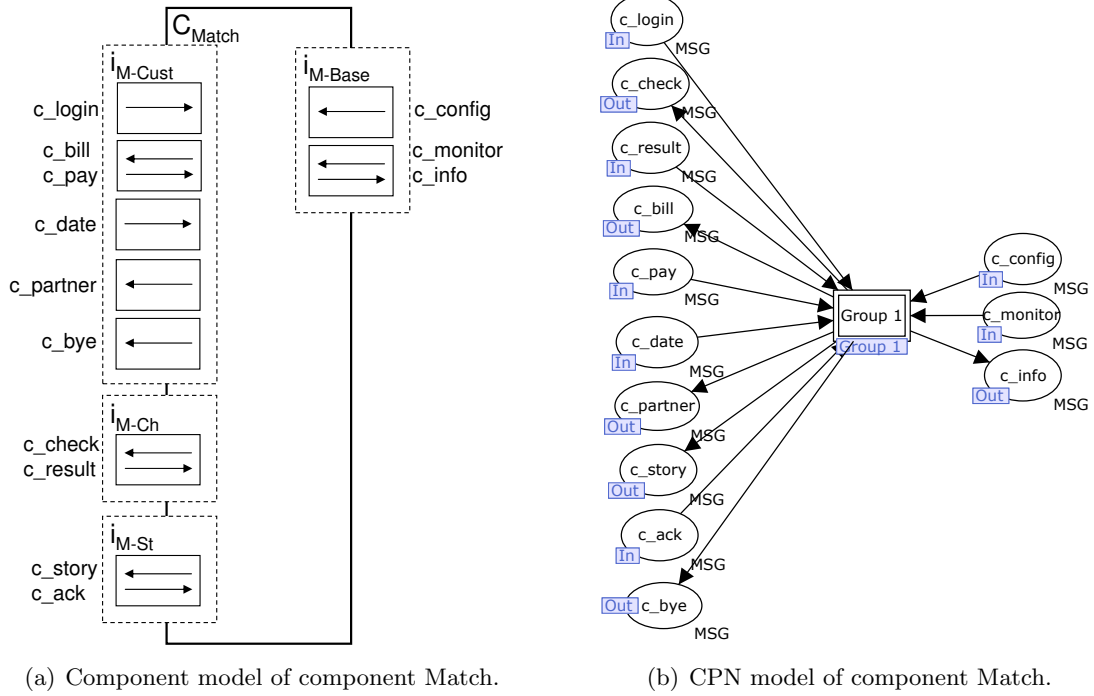


Figure 26: Transforming the component model of component Match into a CPN model.

Next, Fig. 27 presents a detailed model of the case process. For a better readability, data entities are not shown. All places, except the channels, are of type case id (*CaseID*). The arc inscription  $id$  is a variable of type *CaseID*.

We will now shortly describe the semantics of this net. When the customer sends its login information, there is a token of type *MSG* on place  $c\_login$ . Thus, transition *receive login* is activated (the value of variable  $m$ , which is of type *MSG*, is validated with the message content) and can fire. An instance is created and the case id is produced on place  $p1$  (by help of function  $getCaseID$ ). Then, transition *check* is activated (the case id of  $p1$  is assigned to variable  $id$ ) and fires. This yields the case id on place  $p2$ . Furthermore, function  $msgToCheckComp$  generates the message which is sent to the check component. It can be seen that every arc from a transition to an *Out* channel has as an arc inscription a function that generates a message of the above mentioned six tuple containing all the correlation information. Next, when the check component sends its reply message, a

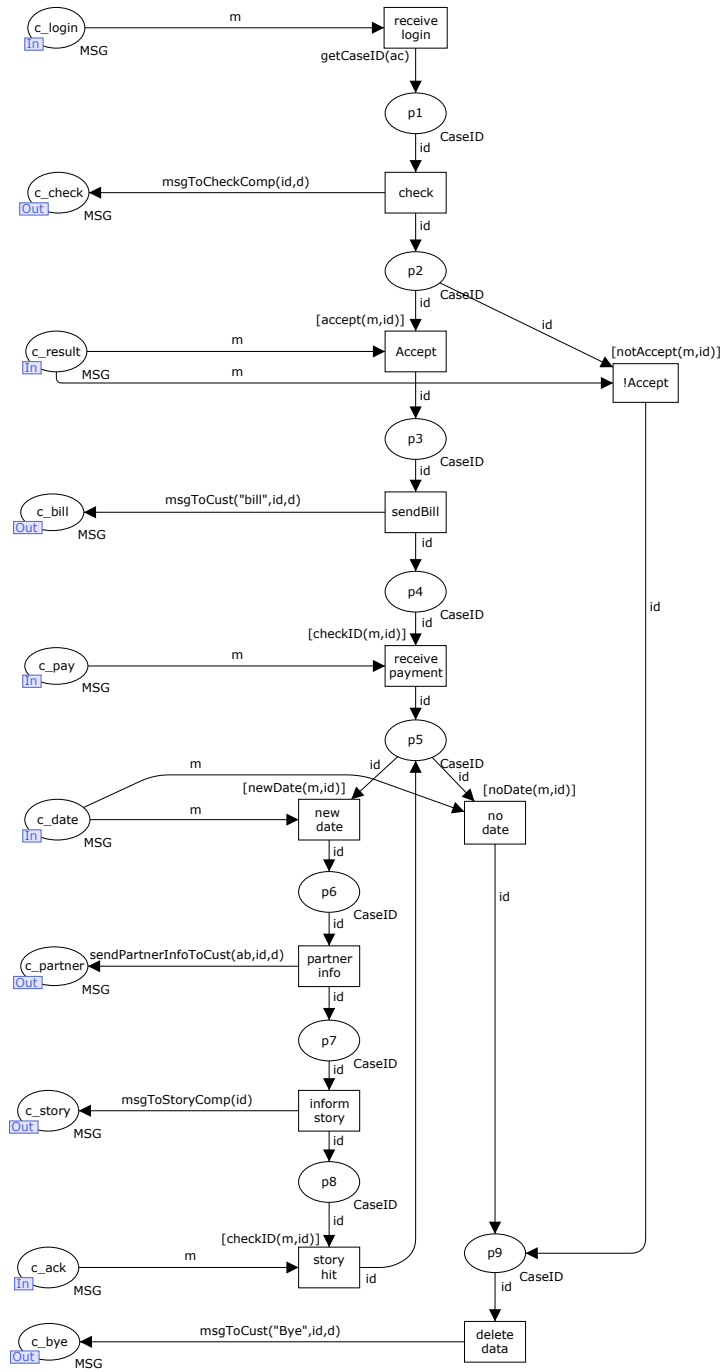


Figure 27: CPN model of the Match process without data entities.

message token is placed on *c\_result*. Transition guard  $[accept(m, id)]$  holds if the message is correlated to the process instance and the customer was not identified to be a marriage swindler. In contrast, transition guard  $[notAccept(m, id)]$  holds if the message correlates to the process instance and the customer was identified to be a marriage swindler. If  $[accept(m, id)]$  is evaluated to true, transition *Accept* is activated (the case id is assigned to variable *id* and the message content to variable *m*) and can fire yielding the case id on *p3*. Otherwise, if  $[notAccept(m, id)]$  holds, *!Accept* is activated and produces the case id on *p9*. Every transition connected to an *In* channel, for example *Accept*, has a transition guard that checks whether the message is correlated to the process instance. Furthermore, in the CPN model conflicts in the control flow are solved deterministically (e.g., the conflict between transitions *Accept* and *!Accept*). The rest of the control flow can be sketched as follows: Having fired *Accept*, the component sends the bill to the customer (*sendBill*) and receives the customer's bank information (*receive payment*). Then, the customer can either leave the dating service (*no date*) or ask for a date (*new date*). In case the customer wants a new date, the process initiates a matchmaking and sends the contact data of the resulting match to the customer (*partner info*). Next, the process sends the correlation information about the customer to component story (*inform story*) and waits for the reply message. The story component sends this reply message as soon as the customer has sent its story (*story hit*). Transition *delete data* sends a goodbye message to the customer and deletes the data of the customer from the data base.

The complete model of the case process, which also visualizes how the activities are connected to the data entities, is presented in Fig. 28. With it, three places are added in Fig. 27: *CS*, *Case Var*, and *Base Var*. Place *CS* stores the case id for all running cases. *Case Var* and *Base Var* model the case data entities and base data entities, respectively. We do not want to explain this net in every detail, because most of its functionality has been already explained in this section. Thus, we restrict us to some interesting facts. When transition *receive login* is activated, the message content is assigned to variable *m*, the set of case ids for all running cases is assigned to variable *ac*, and all case data (i.e., the list of (*CaseID*, *data entity name*, *data value*)) are assigned to variable *d*. If the transition fires, a new case id is created and added to the set of existing case ids (function *createCase*). Further, the id of this new created case is produced on *p1* and the login information of the customer are added in the respective case data entities (function *setLoginData*). We have already mentioned that in case of the firing of *Accept* the customer's login information are stored in the base data. For this purpose, all base data are assigned to variable *ab*. Using function *writeProfileToBase*, the customer's data are added to the base data and the respective token is produced on *Base Var*. Similar to this, the information about the customer's bank account is added to the case data (transition *receive payment*). In detail, the case data is assigned to variable *d* and function *setPayData* adds the account data to the case data and produces the respective token on *Case Var*. The process ends if transition *delete data* is activated. Then, functions *deleteCaseData* and *deleteBaseData* delete the information about the customer, stored in the data entities, and finally, function *deleteCase* deletes the process instance.

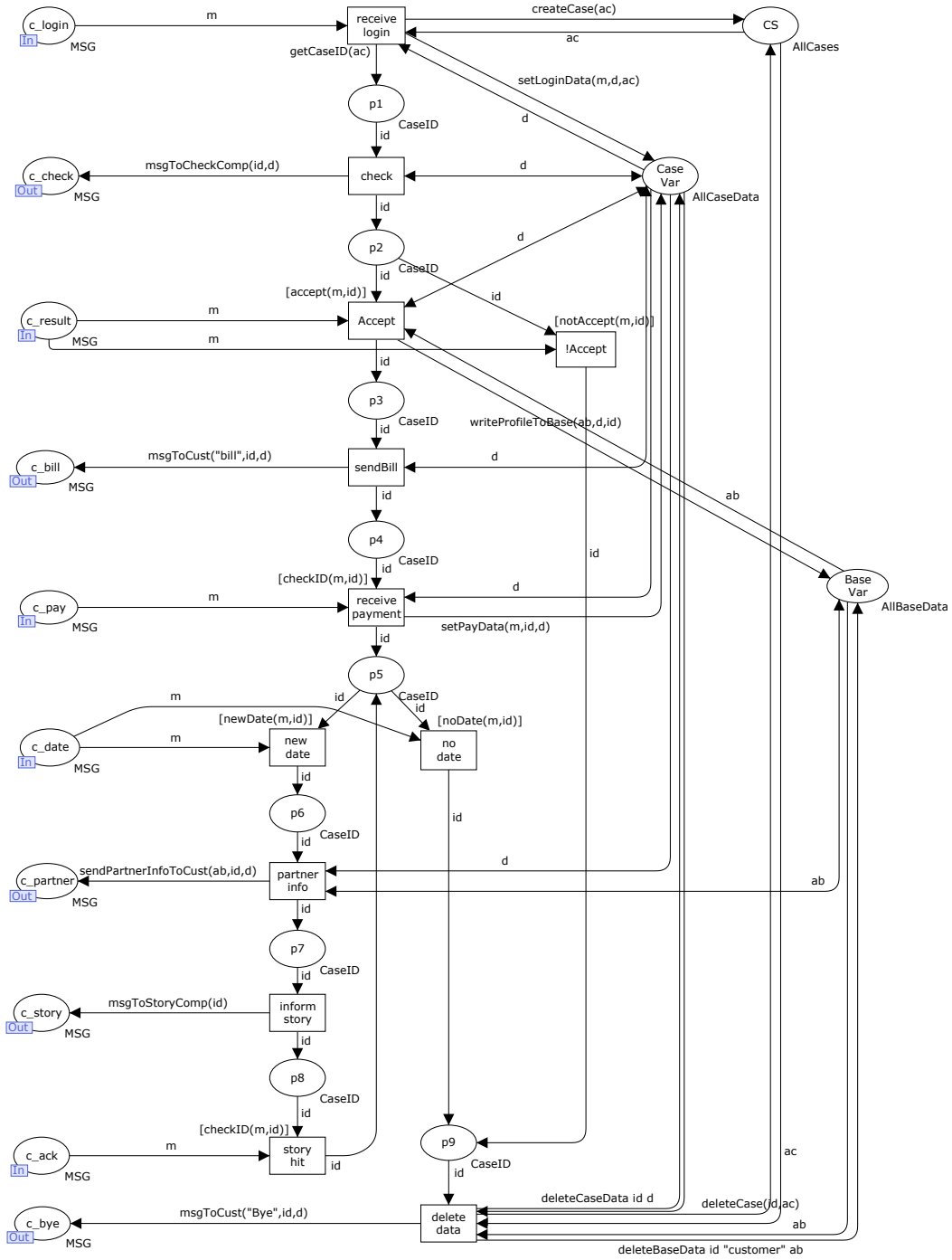


Figure 28: CPN model of the case process of Match.

The base process is shown in Fig. 29. It is worthwhile to mention that the whole process can be derived by merging the place *Base Var* in Fig. 28 with place *Base Var* in Fig. 29.

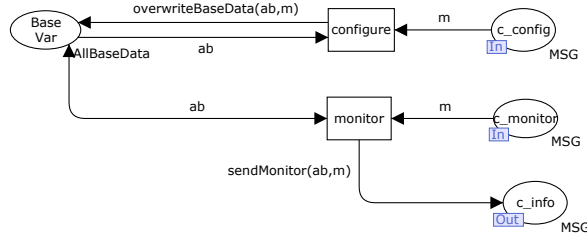


Figure 29: CPN model of the base process of Match.

The configuration interface is used to change the current fee customers have to pay. this is modeled by a token on *c\_config*. Firing transition *configure* (the message content is assigned to variable *m* and the content of *Base Var* to variable *ab*) overwrites the current fee stored in the base data (function *overwriteBaseData*). In case of monitoring the matching process, a request is sent (i.e., there is a token on *c\_monitor*). Transition *monitor* is activated (message content and content of *Base Var* are assigned to variables *m* and *ab*, respectively). Firing *monitor* yields a message on *c\_info*. The message content contains a copy of the base data (function *sendMonitor*). Note that the arrow connecting *monitor* and *Base Var* is only a read arc; that is, the value of *ab* is just read and then put back on *Base Var*.

### 5.4.3 Transformation of Component Check1

In Fig. 30, it is shown, how component *Check1* is transformed into a CPN model. Figure 30(a) depicts the component model of component *Check1*. As in the matching component in the last section, all operations are labeled to identify the corresponding channel places in the CPN model presented in Fig. 30(b).

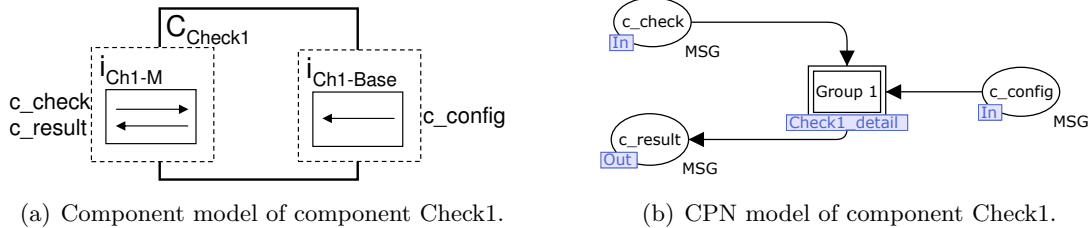


Figure 30: Transforming the component model of component Check1 into a CPN model.

The CPN, which models the process of the check component, is depicted in Fig. 31. If there is a token in channel *c\_check*, transition *customer info* is activated. The semantics of firing this transition is equivalent to transition *receive login* in Fig. 28. To add the

name of the customer to the case data, function *setCheckData* is used. Afterwards, transition *result* is activated. If this transitions fires, it is checked whether the customer is stored in the base data and thus the customer is a known marriage swindler. The result of this check is send back to the matching component (function *msgPS2DS*). Firing *delete* deletes the process instance and thus the case data.

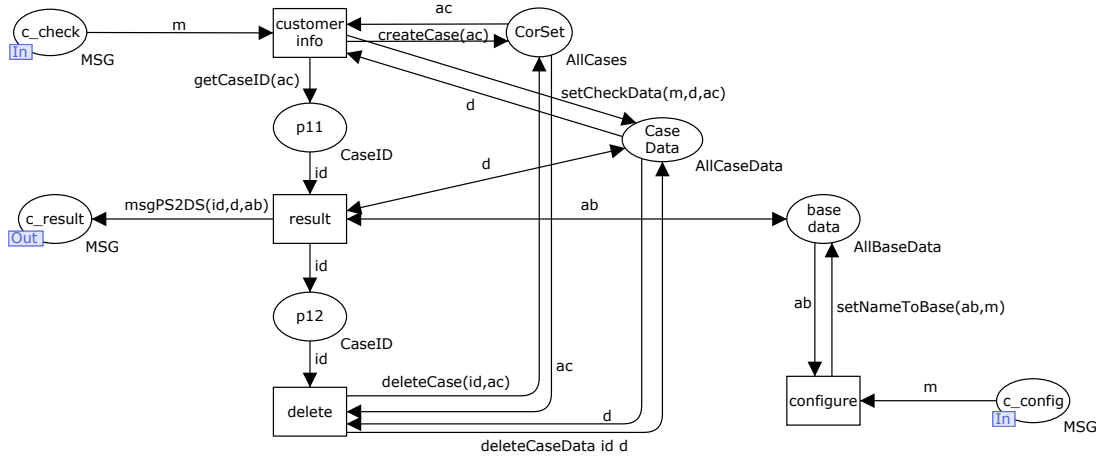


Figure 31: CPN model of component Check1.

#### 5.4.4 Transformation of Component Story

Next, we show the transformation of the story component into a CPN model. As we have done for the other two components, we start in Fig. 32 with the transformation of the component model into a CPN model. The labeled component model is shown in Fig. 32(a) and its corresponding CPN model in Fig. 32(b).

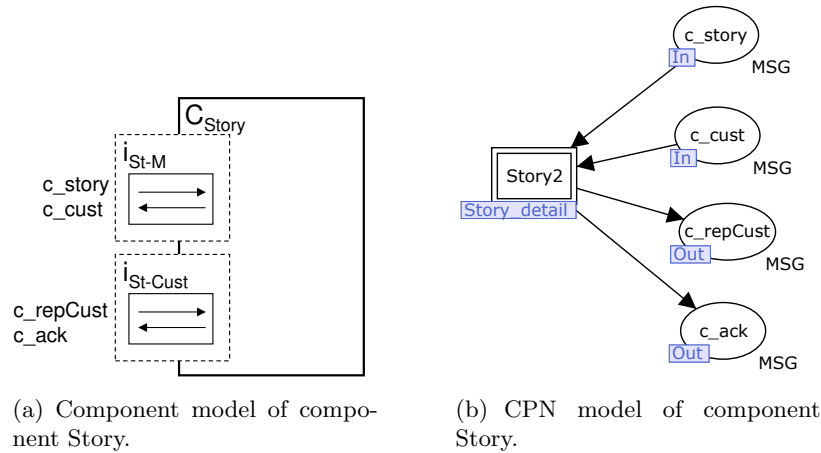


Figure 32: Transforming the component model of component Story into a CPN model.

Figure 33 depicts the CPN model of the case process without data. We shortly explain the control flow of this process model. Let  $A$  and  $B$  be the two customers chosen by the matchmaker. Then, the case of the matching component serving  $A$  and the one serving  $B$  send the correlation information about their customers to the story component. These messages will reach channel  $c\_story$ .

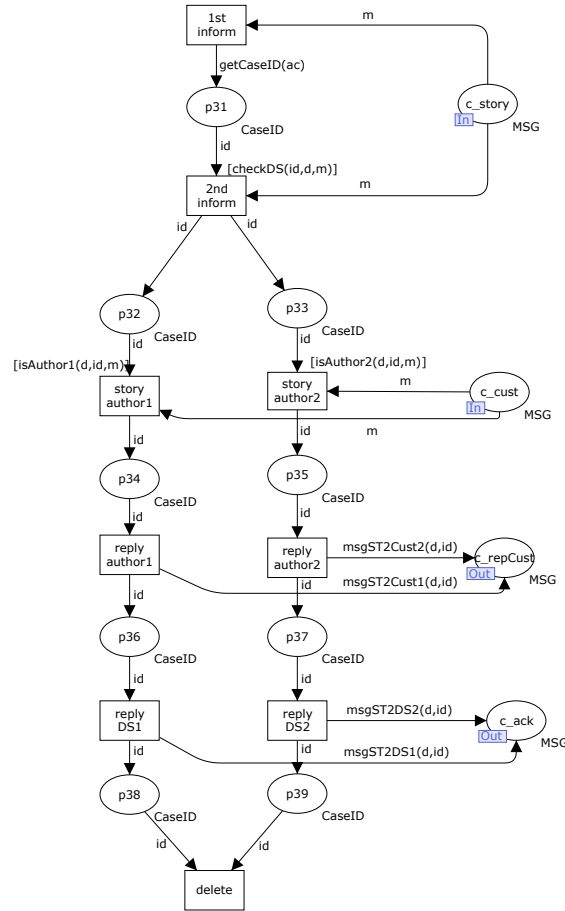


Figure 33: CPN model of the process without data of component Story.

The first message, say from the case serving  $A$ , is consumed by *1st inform*. When this transition fires, the process is instantiated. The second messages is consumed by *2nd inform*. Guard  $[checkDS]$  holds if this message is from the case of the matching component serving  $B$ . If this transition fires, it yields the case id on  $p32$  and  $p33$ . Then, the two branches are executed concurrently. Let us restrict to the left branch. When customer  $A$  sends its story, a token is on  $c\_cust$ . Guard  $[isAuthor1]$  holds if the message is correlated to the correct case (i.e., it is checked that this message is definitely from  $A$ ). Then, *story author1* is activated and can fire yielding the case id on  $p34$ . Firing of *reply*

*author1* and *replyDS1* response to *A* (function *msgST2Cust1* generates the message) and the case of matching component serving *A* (function *msgST2DS1* generates the message), respectively. Finally, *delete* synchronizes both branches and deletes the case.

Next, in Fig. 34 the complete data model of the story component is presented as a CPN model. In this model, also the data entities (see places *CS*, *Case Data*, *BaseData*) are shown. From this model it can be seen that transitions *1st inform* and *2nd inform* write the case data (i.e., the correlation information of *A* and *B*) by help of functions *setInfoData* and *setInfo1Data*. Transitions *story author1* and *story author2*, however, add the story of customers *A* and *B* to the base data (see functions *addStory1* and *addStory2*).

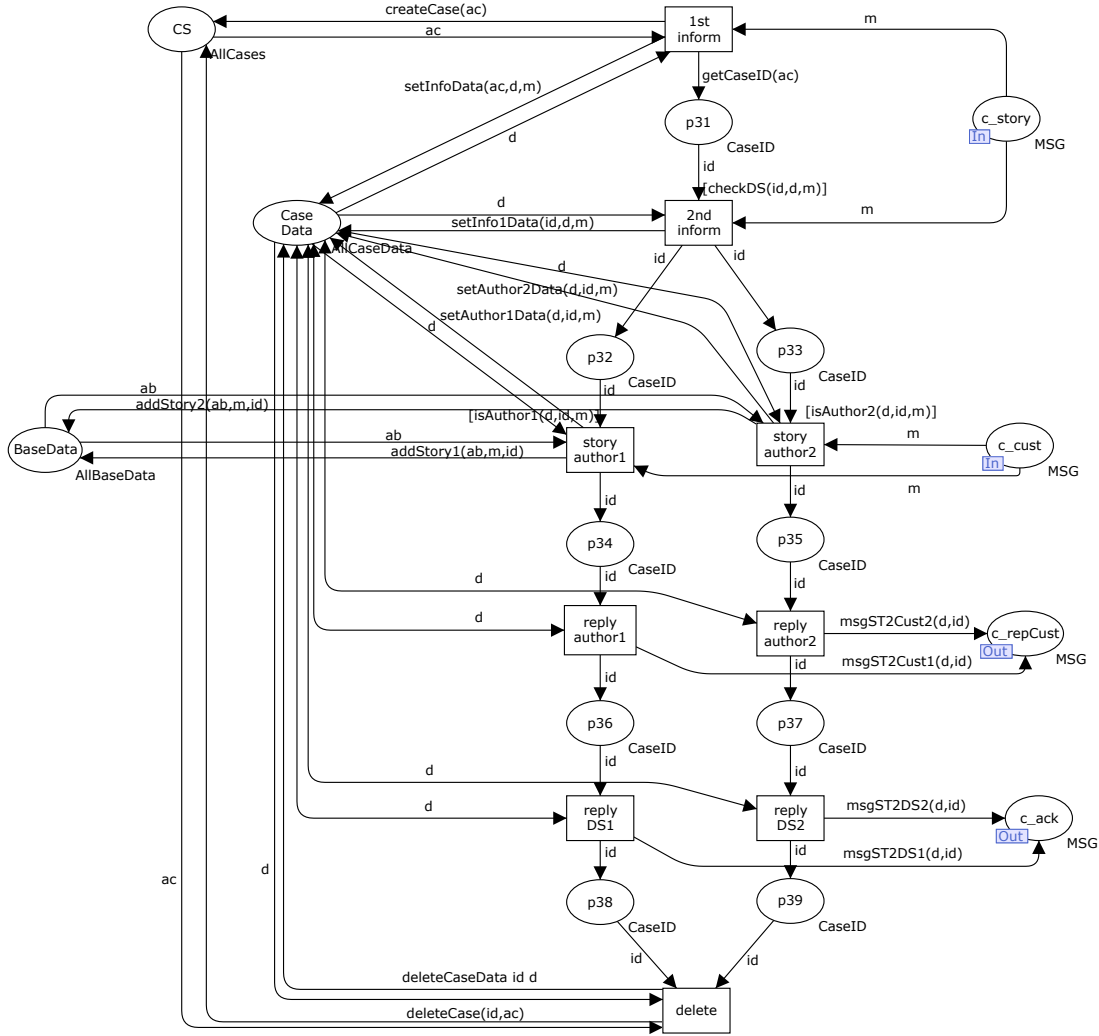


Figure 34: CPN model of component Story.



illustrates, how components *Story* and *Match* are connected. In general, every component has one or more incoming channels but at most one outgoing channel (*c\_outStory* and *c\_outDS*). This place results from merging all outgoing channels. By help of transition guards the outgoing messages are forwarded to their receiving component. For example, a message on place *c\_outStory* is either forwarded to the customer component (transition *to\_Cust*) or to the matching component (transition *to\_DS*). For this purpose, function *isRcv* checks whether the receiver's address is the customer component or the matching component. For each component there is one incoming place, for instance *DS\_in*. All messages, which are sent to the matching component, reach this place. By help of the five transitions and the transition guards (with function *isOP*), each message can be forwarded to the operation it is sent to. As a result, on every incoming channel such as *c\_pay* there are only tokens that belong exactly to this operation of the matching component. Then, the message has only to be correlated to the correct case of the matching component.

## 6 Comparing the Architecture Framework with SCA

### 6.1 Introduction to SCA

The *Service Component Architecture* (SCA) [BBE<sup>+</sup>06] 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

- *services* – business functions offered to other components
- *references* – dependencies on business functions needed from other components
- *properties* – values that influence the component implementation
- *implementation* – concrete realization 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. Fig. 36 illustrates an example SCA composite. Note that the wires (as in Fig. 16) describe a dependency relationship and not control flow.

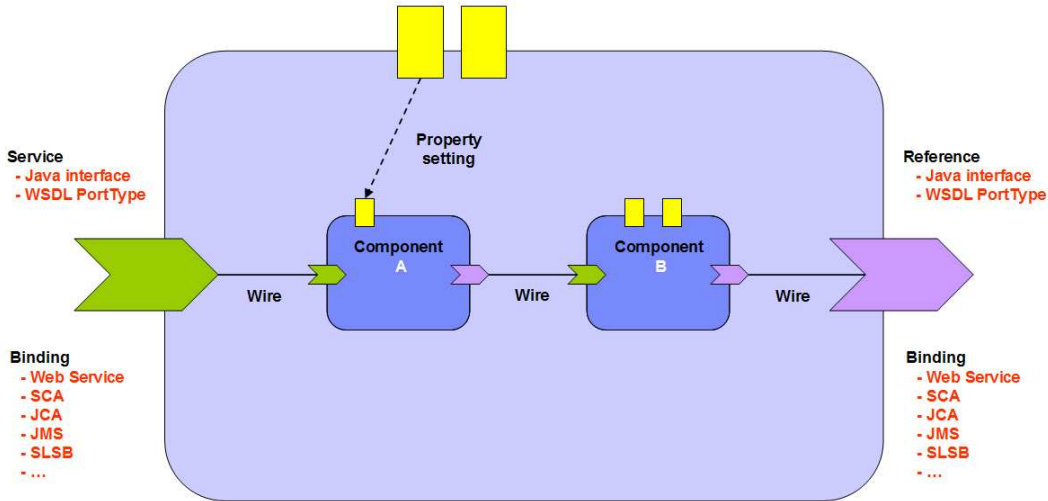


Figure 36: An example SCA composite.

An SCA *system* represents the configuration of an SCA runtime 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 runtime 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 data base stored procedure, or an EIS service binding.

A policy is a declaration of a specific set of behaviors, 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 meta model for SCA in UML notation is presented in Fig. 37.

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.

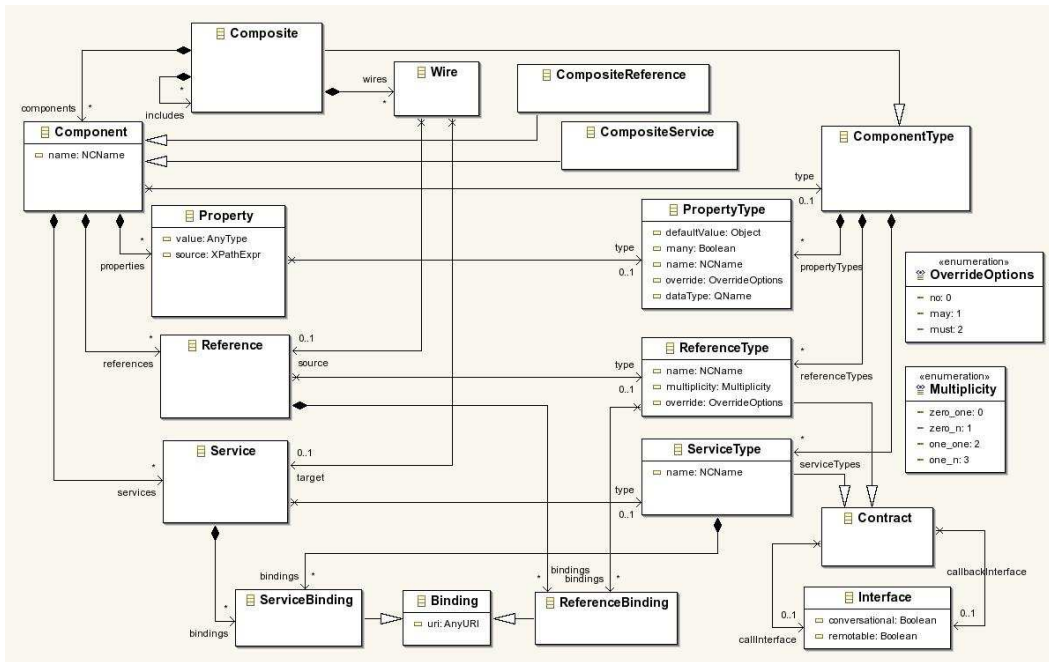


Figure 37: Meta model of the Service Component Architecture.

## 6.2 Comparison to the Architecture Framework

To compare SCA with our architecture framework, we first of all present in Table 2 a comparison of the terms used. Then, we step into the details of both frameworks.

Table 2: Comparing the terms of SCA and the architecture framework.

SCA	Framework as depicted in Fig. 16
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 component, wire, and process. 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 Sect. 4.2.6, 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 Fig. 16 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 by help of properties. In our framework, we use base activities for the configuration of components (cf. Sect. 4.2.2).

To summarize, 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 Sect. 4.3.

## 7 Outlook

In this paper, we addressed our efforts in developing an architecture framework for Service-Oriented Architectures (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. We also presented rules to translate the architecture framework into Colored Petri nets (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. [AKR06] present a meta model for business processes modelling (AMFIBIA) which captures the aspects control flow, data, and organization. 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 summarize, 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

- [AAA<sup>+</sup>06] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web Services Business Process Execution Language Version 2.0. Public Review Draft 02, 20 November, 2006, OASIS, November 2006.
- [Aal98] Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
- [Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [AKR06] Björn Axenath, Ekkart Kindler, and Vladimir Rubin. AMFIBIA: A Meta-Model for the Integration of Business Process Modelling Aspects. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, nov 2006.
- [ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. 1980.
- [BBE<sup>+</sup>06] Michael Beisiegel, Dave Booz, Mike Edwards, Oisin Hurley, Sabin Ielceanu, Anish Karmarkar, Ashok Malhotra, Jim Marino, Martin Nally, Eric Newcomer, Sanjay Patil, Greg Pavlik, Michael Rowley, Ken Tam, Scott Vorthmann, and Lance Waterman. 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, August 2006.
- [BCH<sup>+</sup>03] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael

- Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In Wil M. P. van der Aalst and Eike Best, editors, *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*, pages 483–505. Springer, 2003.
- [BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley Professional, 2 edition, 2003.
- [BGK<sup>+</sup>04] Michael Blow, Yaron Goland, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller, and Michael Rowley. BPELJ: BPEL for Java. Whitepaper, BEA, IBM, March 2004.
- [BS03] Egon Börger and Robert Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [BS05] Bruno Bouyssounouse and Joseph Sifakis, editors. *Embedded Systems Design – The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.
- [CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Discription Language (WSDL) 1.1. W3C Note 15 March 2001, Ariba, International Business Machines Corporation, Microsoft, March 2001.
- [CMRW06] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Candidate Recommendation 27 March 2006, W3C, 2006.
- [DE95] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*. Number 40 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1995.
- [DMV<sup>+</sup>05] Boudewijn F. van Dongen, Ana Karla A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and Wil M. P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005.
- [FLB06] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A formal approach to service component architecture. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.

- [HKG05] Rob High, Stephen Kinder, and Steve Graham. IBM's SOA Foundation – An Architectural Introduction and Overview. Technical Report 1.0, IBM, November 2005.
- [Jen92] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.
- [Jon90] Cliff Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.
- [LMW06] Niels Lohmann, Peter Massuthe, and Karsten Wolf. Operating Guidelines for Finite-State Services. Techn. Report 210, Humboldt-Universität zu Berlin, December 2006.
- [Mäk02] Marko Mäkelä. Maria: Modular Reachability Analyser for Algebraic System Nets. In Javier Esparza and Charles Lakos, editors, *Applications and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002, Adelaide, Australia, June 24-30, 2002, Proceedings*, volume 2360 of *Lecture Notes in Computer Science*, pages 434–444. Springer, 2002.
- [Mar04] Axel Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services*. PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, 2004.
- [McI68] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, volume 1, pages 138–150, Garmisch, Germany, October 1968.
- [McM93] Kenneth L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1993.
- [MS03] David Messerschmitt and Clemens Szyperski. *Software Ecosystem—Understanding an Indispensable Technology and Industry*. MIT Press, 2003.
- [Mur89] Tadao Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, April 1989.
- [OLKM00] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.
- [OMG03] Object Management Group. UML2.0 Object Constraint Language (OCL) Specification. Specification, Object Management Group (OMG), October 2003.

- [Omm02] Rob C. van Ommering. Building product populations with software components. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 255–265. ACM, 2002.
- [OVA<sup>+</sup>05] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephen Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical report (revised version), Queensland University of Technology, October 2005.
- [Pap01] Mike P. Papazoglou. Agent-oriented technology in support of e-business. *Commun. ACM*, 44(4):71–77, 2001.
- [Rei85] W. Reisig. *Petri Nets*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, EATCS Monographs on Theoretical Computer Science edition, 1985.
- [RWL<sup>+</sup>03] Anne V. Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In Wil M. P. van der Aalst and Eike Best, editors, *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*, pages 450–462. Springer, 2003.
- [SCCS05] Natasha Sharygina, Sagar Chaki, Edmund M. Clarke, and Nishant Sinha. Dynamic Component Substitutability Analysis. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 512–528. Springer, 2005.
- [Sch00] Karsten Schmidt. LoLA: A Low Level Analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, number 1825 in *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, June 2000.
- [Sch05] Karsten Schmidt. Controllability of Open Workflow Nets. In Jörg Desel and Ulrich Frank, editors, *Enterprise Modelling and Information Systems Architectures*, number P-75 in *Lecture Notes in Informatics (LNI)*, pages 236–249. Entwicklungsmethoden für Informationssysteme und deren Anwendung (EMISA, RWTH Aachen), Bonner Köllen Verlag, 2005.
- [SHS05] Karsten Schmidt Sebastian Hinz and Christian Stahl. Transforming BPEL to Petri Nets. In Wil M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Third International Conference on Business Process Management (BPM 2005), 6–8 September 2005 Nancy, France*, volume 3649

of *Lecture Notes in Computer Science*, pages 220–235. Springer, September 2005.

- [SR00] Peter H. Starke and Stephan Roch. Ina et al. In Kjeld H. Mortensen, editor, *Tool Demonstrations 21st International Conference on Application and Theory of Petri Nets*, pages 51–56. Department of Computer Science, University of Aarhus, june 2000.
- [Szy98] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, 1998.
- [VBA01] H. M. W. (Eric) Verbeek, Twan Basten, and Wil M. P. van der Aalst. Diagnosing Workflow Processes using Woflan. *Comput. J.*, 44(4):246–279, 2001.
- [Whi04] Stephen A. White. Business Process Modelling Notation Version 1.0. Technical report, Business Process Management Initiative (BPMI), May 2004.