TUM

INSTITUT FÜR INFORMATIK

Towards a Formal Engineering Approach for SOA

Manfred Broy, Christian Leuxner, Daniel Méndez Fernández, Lars Heinemann, Bernd Spanfelner, Wolfgang Mai, Rainer Schlör

TUM-I1024
Dezember 10

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-12-I1024-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2010

Druck: Institut für Informatik der
Technischen Universität München

Towards a Formal Engineering Approach for SOA

Manfred Broy[1], Christian Leuxner[1], Daniel Méndez Fernández[1], Lars Heinemann[1], Bernd Spanfelner[1], Wolfgang Mai[2], Rainer Schlör[2]

SOA Innovation Lab e.V.
Workstream Formal Methods for SOA
c/o Deutsche Post AG Charles-de-Gaulle-Straße 20
53113 Bonn

[1]Technische Universität München, Institut für Informatik
D-80290 München Germany, {broy | leuxner | mendezfe | heineman | spanfeln}@in.tum.de

[2] Deutsche Post DHL, IT BRIEF, Business Architecture
D-53250 Bonn Germany, {Wolfgang.Mai | Rainer.Schloer}@Deutschepost.de

Abstract. Service-oriented architectures (SOA) have received much attention for providing specification principles in order to develop flexible and interoperable software systems. This is achieved by concentrating on “non-technical” concepts of the application domain in order to structure software systems in a functional, business process-oriented manner - thereby enabling efficient reactions to changing requirements. Although there exists a common understanding and agreement on the core essentials of an SOA, available architecture frameworks still differ in the incorporation of those. In particular, there still is no common agreement on (1) the properties of underlying modelling concepts, and (2) how these concepts can be related consistently to each other.

We develop a seamless, model-based engineering approach for SOA that relies on a semantically sound theory of its basic concepts and relations. In this report, we present actual results of on-going research in cooperation with the SOA Innovation Labs. We first discuss the basic SOA concepts of a *service*, a *process,* and a *component* (*architecture)* inferred from a previously developed mathematical model: the FOCUS theory for modelling reactive systems (see [6, 7]). We show that (SOA) processes and services are dual notions for capturing system behaviour at different levels of abstraction. We argue for the use of this formalisation as a basis for the development of a tool-supported engineering approach and conclude by illustrating a case study.

1 INTRODUCTION 3

1.1 Problem Statement 3

1.2 Contribution 4

1.3 Outline 4

2 OBJECTIVE: IDEALISED SOA – ENGINEERING PROCESS 4

3 BASIC CONCEPTS OF A (FORMAL) FOUNDATION 5

3.1 Basic Notions 5

3.2 Service, Process, and Component 6
3.2.1 Service (1) 6
3.2.2 Process (2) 7
3.2.3 Components (3) 7
3.2.4 Interrelation of the Concepts 7
3.2.5 Operationalisation 8

4 CASE STUDY 8

5 RELATED WORK 10

6 CONCLUSION AND FUTURE WORK 11

7 REFERENCES 12

1 Introduction

Services and service-oriented architectures (SOA) have received much attention in recent years for good reasons. When dealing with large, complex systems, it is generally recognized that we need appropriate abstractions and structuring principles. Service-oriented architectures aim, in particular, at structuring software systems. The structure thereby is governed by concepts of the application domain with domain-specific terms and notions rather than by technical concepts of implementation.

However, SOA comes with a variety of concepts with different meanings and interpretations. We see the main differences in the various interpretations of SOA in their different levels of abstraction and their emphasis on different design aspects. The main concerns of SOA may be characterised as follows:

- An application domain structuring principle
- A system architecture principle with general goals
- A system design methodology
- A system implementation concept

SOA has various levels of abstraction - similar to object orientation where we distinguish OO-analysis, OO-design, and OO-programming. These levels address quite different aspects like business process modelling, system architecture and implementation. Application domain structuring and system design principles are usually quite general and not very technical. Design methodologies and system implementation concepts are, in contrast, much more technical and suggest programming concepts for the specification and implementation of services.

Ideally, the different levels of abstraction do not exist independently but are related to each other. so that they can be employed in a seamless development process. If the SOA-related concepts of the different layers of abstraction are related, the information they carry can be passed to the respective subsequent level of abstraction. For our work, we envision an idealised process of system design. The process is outlined in Section 2.

1.1 Problem Statement

The success of any SOA approach relies on its appropriateness for a certain application domain and the usability for the system designers. Only if the concepts of SOA are used according to their specific properties and within a clearly defined purpose, a consistent application can be ensured, thus providing additional benefits with respect to domain analysis, domain understanding and problem specification.

So far, available approaches to SOA - including architecture frameworks - offer a roughly accepted agreement and understanding of the principles of a SOA. Still, the transfer of these principles into concrete modelling concepts is not clear since most modelling languages emphasize single notions like the ones of business processes and services rather than their underlying concepts.

Available frameworks do not relate available techniques to a basic, comprehensive semantic model. Terms like *service, process* and *component* are ubiquitous, but their exact meaning and mutual delimitation, respectively their mutual relation is poorly defined. As a consequence, they only provide marginal benefit for the understanding and modelling of relevant application domains across a development process. The benefit of SOA as a holistic approach is lost and existing frameworks evolve to a collection of specific-purpose description techniques. They fail to:

- capture every necessary concept of the application domain in a *consistent* manner,
- (syntactically) integrate concepts, since the primary purpose of the techniques lies in the transfer of basic information rather than an integrated, seamless modelling thus causing methodological disruptions when taking these techniques as a backbone of development activities,
- use a commonly accepted terminology rather then a framework specific in which the technique is situated.

The reason for this development is that many of the available modelling techniques are developed with the purpose of:

- Offering a pragmatic and easy-to-use tool for specification
- Capturing basic information in a self-explanatory notion that is understandable by different stakeholders with different backgrounds.

Although, the problem of available techniques providing island solutions is recognized, yet missing is the harmonisation of the like. In fact, the properties of and dependencies in-between services, processes, and components are not clear from a theoretical point of view, yet.

However, in order to establish an engineering approach for SOA, such a theoretical foundation of the basic concepts is crucial.

1.2 Contribution

We contribute a formal engineering approach for SOA. Our main goal is to support the engineering of a SOA in a seamless manner with consistent artefacts. For this, we transfer existing contributions in the area of formal methods to the application domain of SOA. The resulting research agenda is depicted in Figure 1.

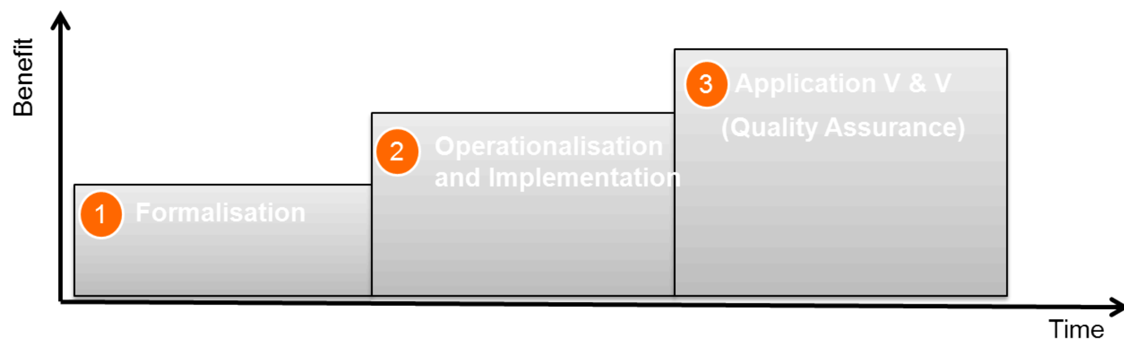

Figure 1: Layered Development Approach

We first provide a precise definition of the terms is needed such that they can be related to each other in order to gain a consensual view on SOA that is useful for the development across all levels of abstraction,. Hence, a mapping to a general purpose system model is given that allows a rigorous definition of the basic concepts.

Once we established a common understanding of concepts, we operationalize the formalisation. This operationalisation is purpose-specific. Typical purposes are, e.g., simulation, verification, methodological support, tools etc. and can be mapped to the existing methodologies, tools, and frameworks as needed. We envision in particular the establishment of formal means for the purpose of validation and verification.

In this report, we show some results of our operationalisation of SOA, developed in a research cooperation between the TU München and the SOA Innovation Labs. This work relies on our previous contributions in the area of embedded systems, such as the comprehensive system modeling theory FOCUS [6,7], and ProBaMa [12].

1.3 Outline

'The reminder of the report is as follows. In section 2, we outline the idealised SOA-engineering process at which we aim with our contributions. Section 3 presents, in a nutshell, the formalisation of the SOA concepts and the operationalization (see step 1 and 2 in Fig. 1). Section 4 then comprehends a case study, before discussing related in section 5. Finally, we give in section 6 some concluding remarks.

2 Objective: Idealised SOA – Engineering Process

Capturing the peculiarities of an application domain is a key factor for the success of any system development process. In business process modelling, the flow of information between any participating party including sequences of their activities and services they offer are of interest. SOA provides means to capture this information.

Initially, the properties of the application domain together with the business process and use cases have to be captured. In such a model only parts of the processes are computer-supported. One

rather is interested in a holistic approach that allows to describe the complete business process - deferring decisions about the actual extent of computerisation to later phases.

Subsequently, the software parts of the system are chosen. The information from the business process model is used to derive the requirements for the aspired applications in a constructive and consistent manner. The requirements are used to build up an architecture for the applications and design their interactions. Since the requirements are derived from the original business process model, their compliance with the initial specification is supported. Furthermore, a common semantic model ensures that the satisfaction of requirements by the applications even can be verified using (semi-) automatic proof systems.

Finally, from the architecture and design of the applications an implementation can be inferred (in some cases even automatically) that realises the requirements that were imposed by the original business process. This results in an implementation that is compliant with the envisioned use cases. Additionally, the original requirements can be traced such that changes arising in the system's life-cycle can easily be mapped to the implementation. Again the compliance with the initial process and requirements can be verified.

3 Basic Concepts of a (Formal) Foundation

Information hardly can be utilised in sequel development steps if concepts appearing at different levels of abstraction in the development process are not related. Using a common understanding and a common formal framework on each layer of abstraction together with appropriate mappings between concepts allows for relating the artefacts of a system development process. Before we relate the notions of *Process*, *Service* and *Component*, we briefly recall the basic notions of the formalisation, which underlies the concepts.

3.1 Basic Notions

We first define *a basic terminology* that can be used to describe a functional perspective of SOA, not only in the sense of a conceptual framework, but also in the sense of *an algebraic theory*. The notions consider elementary notions and algebraic operations, which will be informally introduced in the following.

Message. A *message*, in our understanding, is representing the transmission of information. A message has a sender and a receiver. The content of a message is a value, which has a defined type (a strong typing discipline is assumed here). A message is transmitted at a certain point in time. Message types can vary from simple types to very complex data records, involving a significant set of business objects.

Channel. A *channel* is the carrier of information, being transmitted in the form of messages. We assume that a channel represents a directed flow of information. Channels are connected to ports. A channel is a connection between an "out" port – being the sender – and an "in" port – which is the receiving part of the communication. Again, the concept of a channel can be used to represent a communication medium as simple as a hardware wire, or represent a medium as complex as an enterprise service bus (ESB).

Interface. An *interface* consists of a set of input- and output ports. It conceptually describes the boundary of an object (service, process or component). If we look at an agent and its interface without knowing more about its internal structure (nor its internal behaviour), we call this a "black box". As the ports of the interface are strongly typed, the interface expresses an important information about compatibility, for example relevant for reuse.

Denotational Semantics (external behavior view). Even without knowing anything about the internals of an agent, we can tell its *semantics* by making observations at its interface. Formally, we are looking at histories of values observed at an agent's interface. The history may be finite or infinite. We assume a simple, discrete model of time, without loss of generality. More complex timing models as, e.g., needed to model embedded control systems, can be handled in the same framework.

Operational Specification. A denotational semantics, which has a descriptive nature, is appropriate to explain and relate concepts without constraints to their forming or implementation. In contrast a operational semantics facilitates certain constructs in order to establish an execution model. Therefore the denotational semantics is more appropriate as a common basis for the definition of concepts independent of their use in the development process, whereas some operational semantics is chosen purpose specific and allows to carry out concrete tasks like simulation, etc.

Composition. Once we have a set of agents with compatible interfaces, we may connect matching out- and in- ports by *composition*. The connections are made with channels. During the connection, a network of internal communication channels is established.

3.2 Service, Process, and Component

In order to allow for a seamless integration of concepts, the theory of SOA is mapped on the formerly explained, formal framework. Figure 2 illustrates the three typical perspectives of SOA, whose ingredients and interrelations have to be discussed from the perspective of the theory.

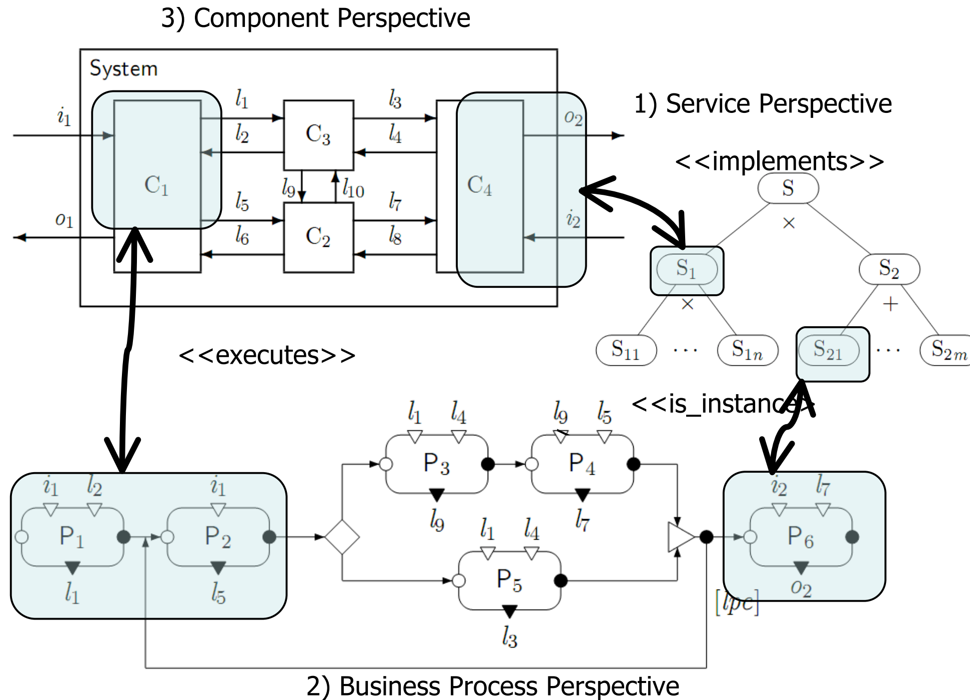

Figure 2: Perspectives taken by the Theory

3.2.1 Service (1)

Each service can be characterised syntactically by its type of input and its type of output messages, i.e., its syntactic interface. The behaviour of a service is characterised by the relation of input- and output messages. As compositions usually parallel and (sometimes) alternative composition are used in order to construct a comprehensive behaviour out of simpler services. Thereby, usually no communication between the services is allowed. Services are just projections on the interface. Only mode information may be passed in order to antagonise feature interactions which describe mutual influences.

As an extension, so called service provider networks are considered. This way, a complex service is decomposed in simpler ones that interact in a producer consumer manner in order to provide the aspired service. However, this is methodologically motivated and already marks a transition to the component view.

The service perspective is the most abstract perspective within the SOA framework, which focuses on observable functionality of the object of consideration. The structure within this perspective defines which services are provided at the interface (black-box-view).

A service is a general purpose specification to the behaviour of an object of consideration. Starting from the (software)system as the object of consideration (with the system user in the role of a service user and the software system in the role of a service provider) iterations in a development process redefine the roles and map them components that are part of the system under development.

By changing the scope, i.e., the object of consideration, the principles of services can be applied to any object that may occur as a part of the system under development. This retains the flexibility to change the scope from the whole software system to its sub-system on demand, thus delivering a black-box *component specification* rather than a system specification..

3.2.2 Process (2)

If we are interested in a more dynamic view onto the system we use processes. Processes describe the execution order of services. A process is a description of the interaction between the service providers to provide a particular superordinate service.

Again a process is characterised by its interface and the relation between input and output messages as well. Only the composition operations are different in order to map the peculiarities of the view to the system. In contrast to the service view a sequential composition that is available for processes allows a sequentialisation of behaviours. Together with parallel and alternative composition services that are provided by some object leading to the duality between services and processes: A system is a service that is provided by means of providing sub-services in a particular timely manner (process view). Any of the sub-services may be again a subject to a decomposition into timely correlated sub-services etc.

3.2.3 Components (3)

A component is again characterised by its interface and I/O-behaviour. According to the associated view as an architecture, only parallel composition is available. In contrast to the service view, the component view puts emphasis on the interaction between entities. Therefore internal communication is allowed and particularly welcomed. The component view is closer to the implementation than any of the other two views because it already fixes an architecture. This decision intentionally deferred at the process and service view. Only methodological aspects allow to enhance both (services and processes) with additional information in order to ease the step over to the component view. Service provider networks are one such enhancement for services, the assignment of roles to processes is another for the process view.

3.2.4 Interrelation of the Concepts

In Figure 2, the relation between the service and component perspective is illustrated by the `<<implements>>` stereotype, which indicates that the interface behaviour of component C4 among other things implements the service S1. The process perspective on the other hand structures the executions of the system into sequentially executed behavioural modules, called processes. This perspective supplements the service view that is used to decompose interface behaviours (black-box) by an activity view that is used to fix an order of execution of services. It concentrates on the activities that are executed during the usage of the system and how these processes *interact* in order to provide a certain functionality of the system (glass-box-view). The process perspective is usually taken by the business representatives and defines responsibility both for data and functions incorporating the workflow performed in the business. In the end, each process within this perspective must be executed by some component within the system or some actor in the real world, which is indicated by the `<<executes>>` stereotype within Figure 2, e.g., process P1 and P2 are both executed sequentially by component C1. Note, that the corresponding interfaces of the process and the component coincide (i1, l1, l2, l5). The component perspective (3) defines how the system is decomposed into simultaneously executing subsystems, called components, and how these components interact with each other to provide the corresponding services - thereby executing the software part of the underlying business process. In this view, both processes and services define some part of the system's interface behaviour within the component perspective, whereby services may only refer to the system's external interface (i1, i2, o1, o2). The relation between services and processes is a little bit more subtle: during the execution of the business process, also called workflow, each service may appear one or several times. In this sense, each such service appearance within the workflow can be understood as an *instance* (usage, invocation) of that service, which is reflected by the `<<is_instance>>` stereotype within Figure 2. Clearly, the interface of every "service instance" (e.g. process P6) and the service itself (e.g. service (S21) must coincide. However, since the individual processes within the workflow may also interact over internal communication (glass-box view), not every process within this workflow corresponds to a service at the same level of abstraction but may refer to services with respect to a certain sub-system - remember that services follow a black-box view onto the (sub-)system.

The major benefit of this common understanding, the relations of concepts, and finally a suitable formal framework is to allow for relating artefacts by means of validation and verification in order to

ensure the compliance of the solution with the initial business requirements and, thus, raising the quality of the solutions.

3.2.5 Operationalisation

As one goal of an operationalisation of the relations between the concepts the field of verification and validation (V&V) is outlined. Figure 3 summarizes the verification steps, which are possible in an appropriate formal environment.

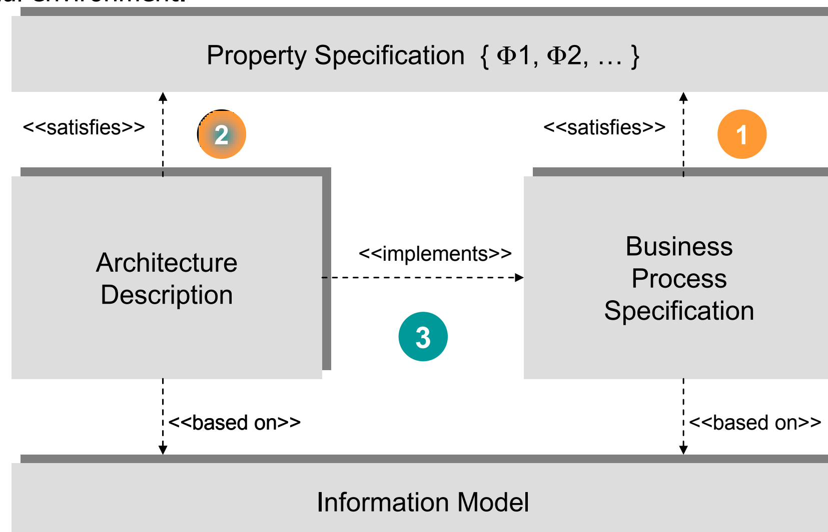

Figure 3: Verification Steps

The starting point are business process specifications that can be verified against general purpose or domain specific properties (1). Similarly, models of the architecture design must satisfy the same set of properties (otherwise, a behaviour could exist in the implemented IT-system network, which is not entailed in the original business process specification). More advanced topics are imposed by the question, how the architecture description can be verified directly against a given business process specification (3), or how an initial architecture description can be derived through a sequence of ("correct by construction") transformation steps from a validated business process specification, to mention just a few of the questions analysed by on-going research activities.

In this section, we introduce the basic concepts while we intentionally omit the introduction of a complex, formal modelling theory. Our goal is a uniform understanding of the concepts and a seamless integration of these concepts into a development process. Only if concepts are clearly defined, their exact relations can be developed. Without these relations the vision of a highly integrated development process based on SOA remains out of reach.

We first describe the main SOA ingredients, i.e. a set of concepts that we used in the algebraic theory, before inferring the concepts of services and processes and conclude with a small example.

Based on the notions introduced in the previous section, we formulate our model of a *service* and of a *process*. The model assumes a very restricted, purely functional viewpoint; since we are interested in precise assertions to be made about a model (e.g., in order to perform property verification), the model must be very rigid, avoiding all unnecessary connotations at this stage. The benefit is, in turn, that intuitively very different notions – like the notions of a SOA service and that of a business process – suddenly exhibit a similarity that is far from obvious. In our model, we will see that the notions become closely related; in fact, one notion generates objects of the other type.

4 Case Study

We performed a case study, in which we applied the operationalised theory in the context of SOA. We aimed with the case study at giving evidence on the suitability of our approach to describe SOA-typical scenarios while benefiting from the advantages offered by formal methods. The purpose of the case study is, however, not a benchmark of our approach with respect to related ones, whereby we describe in the following a pure action research study without concluding the performed approach with an assessment.

For modelling an excerpt of a SOA, we referred to a scenario, which is described in detail in the upcoming textbook [16]. One of the co-authors directly participated in the case study and was available to clarify open questions about the scenarios.

The content of the scenario is a purchase order process (namely “BANF”, German: “Bestellanforderung”). For modelling the case study, we referred to our prototypical eclipse-plugin, called AutoFOCUS 3 (AF3) including an extension for modelling services (ProBaMa). The tool provides means to formally specify behaviour by means of I/O-automata. A snapshot of the tool is provided by

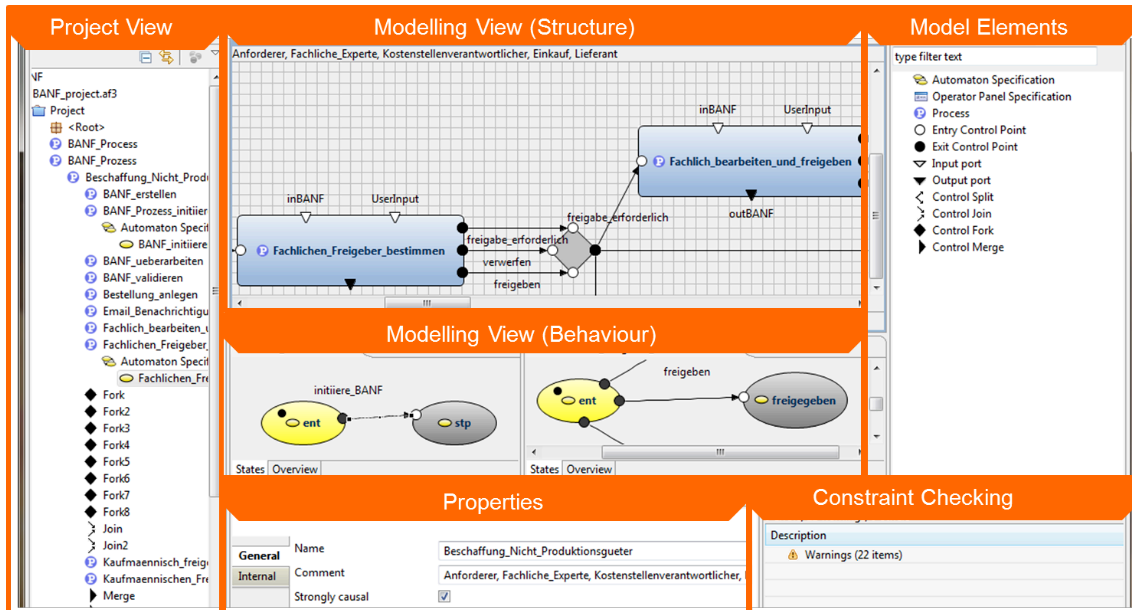

Figure 4 Overview over used Tool

Fig. 4 introducing the different views of the ProBaMA-plugin.

During the process of transferring the information given by the BANF from the textbook to the tool, a lagre number of questions about the BANF arose that mostly were a consequence of blurred, imprecise or missing information about the BANF, which could not easily be discovered in the originally used Business Process Modelling Notation, but immediately became obvious upon the formalisation. In the sequel, we shortly present clippings from the case study.

The tool provides a number of views appropriate for the modelling or processes. Most notably, a structural view allows to specify and relate processes. This is close to the BPMM notation and allows to sequentialised behaviour. The behavioural view additionally associates with each process from the structure view a behaviour by means of automata, respectively state machines. A constraint checker ensures the syntactic validity of the models. Additionally project views allow a easy handling of the

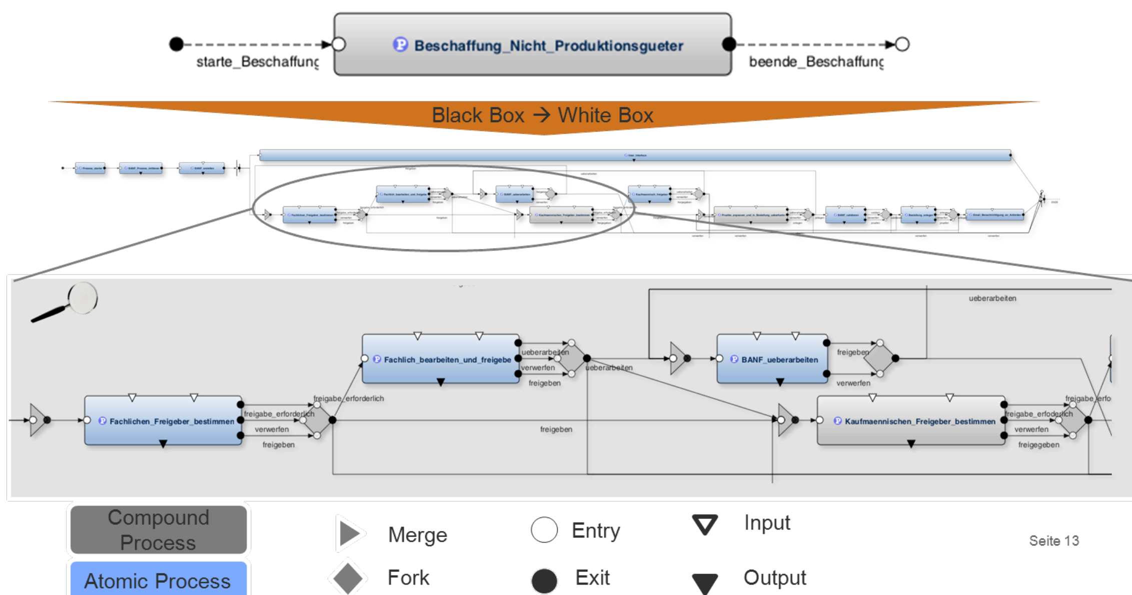

Figure 5 From Services to Processes

model.

During the modelling procedure, we followed a step-wise refinement from an abstract black-box view towards a more and more detailed glass-box (see Fig. 5). The BANF has been decomposed into a number of steps that were regarded as services. They were equipped with an appropriate interface and related via the process view. All of the services were either atomic and directly associated with an appropriate automaton (Fig. 6) or they were compound and consequently were iteratively decomposed in the same manner. By data type modelling, all ports of the respective interfaces were associates with the kind of massages they are able to communicate.

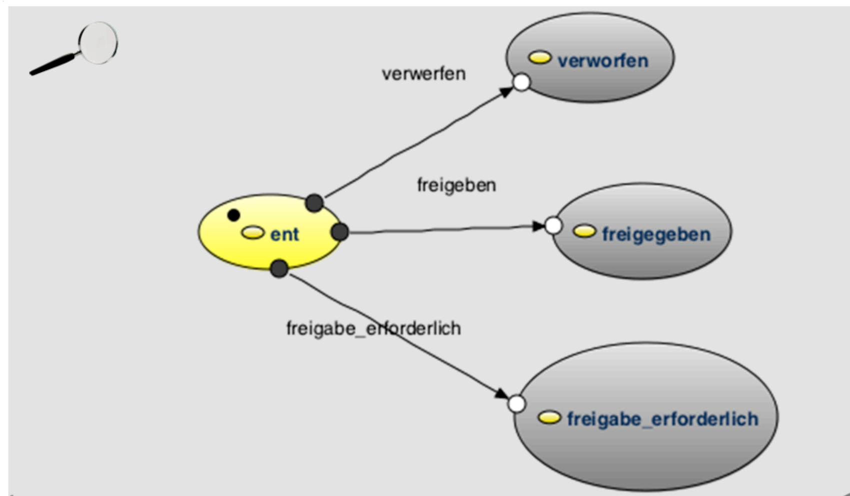

Figure 6 Modelling Behaviour via State Machines

In a final step, the BANF specification, as represented by the process model, has been mapped to a component architecture that implements the specification. By means of the tool, we can ensure that this step is carried out in a consistent manner and (as an outlook) bears the potential to verify this transition by means of model checking or (again as an outlook) carry out the transition on a formal deductive manner that guaranties the correctness of this step (correct by construction).

5 Related Work

In the following, we discuss some related approaches for modelling the concepts typically found in the context of SOA, whereby we concentrate on the concept of (business) processes or workflows. A variety of description techniques and formalism for describing these concepts already exists, which differ in many aspects such as communication, formal foundations, separation between control and data flow, process composition, refinement concepts, hierarchical structuring, and so on. We mention just a few formal approaches which influenced the definition of our description technique the most. Clearly, the following list of related approaches is necessarily incomplete due to space limitations.

The concept of (de)activating processes via control points goes back to the control tokens introduced in *Petri Nets* [13] and variants thereof [17]. *Activity Diagrams* as used in the Unified Modeling Language [5] constitute an actually informal description technique, which also supports the specification of control flow in terms of choice, iteration, and concurrency - in version 2.0 also data flow is supported. Approaches like [8,15] emerged since their introduction, which formalize the Activity Diagram semantics in terms of existing formalisms such as (Colored) Petri Nets or by introducing new formalisms. Owing to the Petri Net formalism, the specifications can also be simulated. Typically, these approaches do not cover the entire language of Activity Diagrams and neglect more complicated concepts such as procedure calls and exception handling.

Another Petri Net-based approach focusing on the control flow aspect of work flows is YAWL [19]. Similar to the approach presented here, YAWL formalizes work flows via labeled transition systems. YAWL differs from the presented approach in the sense that we support the data flow aspect of the language in our graphical notation, and - even without taking data flow into account - YAWL exhibits some rather tricky semantic constructs. Another drawback of YAWL is, that the usage of certain composition operators makes the specification inaccessible for compositional verification.

BPEL [3,1] constitutes a dominant language for the definition and execution of business processes using Web services, which concerning its level of abstraction

resembles a basic programming language: When writing BPEL specifications one is compelled to use involved language constructs for writing abstract and input-disabled specifications. The language lacks a graphical notation - Activity Diagrams are often used as a surrogate. Approaches such as [2] emerged since BPEL's introduction to define a formal semantics for its analysis, but do not refer to Activity Diagrams.

A couple of approaches exist, which use process algebras [4] like ACP, CCS, CSP, and variants thereof in order to formalize work flows. We argue that process terms do not provide the high level of abstraction which is useful in the early phases of system development, in which only activities and their causal ordering are taken into account and communication aspects should be hidden. Process terms typically require the specification of inter-process communication - details that may be unknown in the early development process. Moreover, algebraic specifications quickly become illegible for more sophisticated systems, thus complicating their validation via domain experts and users. Graphical notations and corresponding tools for process algebras like, e.g., [10, 11] have been introduced to overcome this issue. Clearly, these notations and tools adhere to the paradigms dictated by their underlying process algebra - such as rendezvous communication and bi-simulation as the preferred equivalence relation. Since we aim at translating process specification into asynchronously communicating data flow nets [6], these paradigms run a bit contrary to our model of computation and our notion of refinement.

The idea of defining process behaviour mathematically and structuring processes hierarchically is inspired by the *business process nets* introduced in [18]. However, the specification of behaviour in business process nets is restricted to `conventional' mathematical functions. This appears to be very limiting when describing complex system behaviour, since this approach imposes some kind of `*one step per process*' semantics. In addition, we wanted a specification technique that explicitly captures the notion of control flow for enabling a proper composition of processes - a concept not supported in business process nets.

An explicit separation of data and control flow enables a proper composition by partitioning the overall system behaviour according to control points. More precisely, control points structure a complex labeled transition system into modular behavioural parts. Indeed, our approach is essentially inspired by Henzinger's *components* [9] and Schätz's *functions* [14]. Both approaches focus on the construction of reactive systems, thereby providing designers with disjunctive and conjunctive composition of behavioural modules, which cover sequential executions by handing over activation and simultaneous executions by exchanging messages. However, in order to cope with the specification of work flows, some design decisions within the above approaches had to be reconsidered. As opposed to the strict black-box view of functions, processes support the internal communication via shared variables which emphasise their concentration on (system) executions. In order to reflect their exemplary nature, the processes in our approach are *input disabled* as opposed to the components in [9] and the functions in [14]. Accordingly, the model of computation, the definition of behavioural composition, and the corresponding appearance of processes differ w.r.t. functions and components.

6 Conclusion and Future Work

In this report, we presented our approach towards a semantically founded seamless engineering approach for SOA.

We first introduced a set of basic notions, derived from a proven, widely applicable theoretical framework (FOCUS). We have shown how a basic, coherent semantic model evolves into an understanding of the core notions characterising SOA and BPM approaches, viz. the notions of *process* and that of *service*. The main finding of this work is to understand that "service" and "process" are truly dual notions – leading to important consequences for advanced methodology approaches, which develop BPM further towards an engineering approach, with the ultimate vision and perspective to achieve a better alignment between the business side and IT departments.

Based on this formalisation via a descriptive semantic, we introduced the operationalisation of the concepts, which is based, in turn, on a constructive semantic foundation. We showed an excerpt of a prototypical case tool, which implements the concepts used in ProBaMa. We are able to specify business processes, services, and component architectures in a consistent manner, since the concepts are related to each other and captured in a unified data model. We gave evidence by performing a case study, in which we directly applied our approach to a real-life scenario.

We firmly believe that the current tendency towards semantically "enriched" business process models and -notations can provoke a similar evolution of formal method support as we have seen in

the automotive industry. Still, we are aware of the challenges given in the application of formal methods that yet are not solved but remain an indispensible ingredient in the description of SOA: non-functional properties.

For instance, given the fact that we unify the notions of behaviour (services, processes, states and data dictionaries) and the one of a component architecture, non-functional properties, such as business rules or service level agreements:

- can be by now exclusively expressed by our models, if they directly relate to behaviour, such as response time
- remain, if captured in our models, as one irreproducible motivation of a certain design decisions, such as a set of states being motivated by a business rule.

Hence, although the application of formal methods is promising in the context of SOA we still see the need of further fundamental research, especially in the context of non-functional properties.

7 References

[1] Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, et al. Web services business process execution language version 2.0. OASIS Standard, 11, 2007.

[2] M. Butler, C. Ferreira, and M.Y. Ng. Precise modelling of compensating business transactions and its application to BPEL. Journal of Universal Computer Science, 11(5):712–743, 2005.

[3] Business process execution language for web services (bpel), version 1.1, May 2003.

[4] J.A. Bergstra, A. Ponse, and S.A. Smolka. Handbook of process algebra. Elsevier Science, 2001.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Reading Mass, 1999.

[6] M. Broy and K. Stølen. Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, 2001.

[7] M. Broy, I. Krüger, M. Meisinger: A Formal Model of Services. TOSEM - ACM Trans. Softw. Eng. Methodol. 16, 1 (Feb. 2007), article no. 5

[8] H. Eshuis. Semantics and Verification of UML Activity Diagrams for Workflow Modelling. PhD thesis, Univ. of Twente, November 2002.

[9] T. A. Henzinger. Masaccio: A formal model for embedded components. In TCS ’00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, pages 549–563, London, UK, 2000. Springer-Verlag.

[10] G.H. Hilderink. Graphical modelling language for specifying concurrency based on CSP. IEE proceedings-software, 150(2):108–120, 2003.

[11] D.S. Jovanovic, B. Orlic, G.K. Liet, and J.F. Broenink. gCSP: a graphical tool for designing CSP systems. 2004.

[12] Leuxner, W. Sitou, B. Spanfelner. A Formal Model for Workflows. In SEFM2010: Proceedings of the 8th International Conference on Software Engineering and Formal Methods, 2010.

[13] W. Reisig. Petri nets: an introduction. Springer-Verlag, New York, NY, USA, 1985.

[14] Schätz. Modular functional descriptions. In Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2007), 2007.

[15] H. Störrle. Semantics and verification of data flow in UML 2.0 activities. Electronic Notes in Theoretical Computer Science, 127(4):35–52, 2005.

[16] Slama,R. Nelius, D. Breitkreuz. Enterprise BPM: Erfolgsrezepte für unternehmensweites Prozessmanagement. DPunkt Verlag, 2011.

[17] [K. Salimifard and M. Wright. Petri net-based modelling of workflow systems: An overview. European Journal of Operational Research, 134(3):664 – 676, 2001.

[18] V. Thurner. A Formally Founded Description Technique for Business Processes. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, Software Engineering for Parallel and Distributed Systems, PDSE98, pages 254 – 261. IEEE Computer Society, 1998.

[19] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. Information Systems, 30(4):245 – 275, 2005.

[20] M. Wirsing. Algebraic specification languages: An overview. In COMPASS/ADT, pages 81–115, 1994.