View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

Sensoria Patterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity

Citation for published version:
Wirsing, M, Hölzl, M, Acciai, L, Banti, F, Clark, A, Fantechi, A, Gilmore, S, Gnesi, S, Gönczy, L, Koch, N, Lapadula, A, Mayer, P, Mazzanti, F, Pugliese, R, Schroeder, A, Tiezzi, F, Tribastone, M & Varró, D 2008, Sensoria Patterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. in T Margaria & B Steffen (eds), Leveraging Applications of Formal Methods, Verification and Validation. Communications in Computer and Information Science, vol. 17, Springer-Verlag GmbH, pp. 170-190. DOI: 10.1007/978-3-540-88479-8_13

Digital Object Identifier (DOI):
10.1007/978-3-540-88479-8_13

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Leveraging Applications of Formal Methods, Verification and Validation

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Apr. 2019

Wirsing, M., Hölzl, M., Acciai, L., Banti, F., Clark, A., Fantechi, A., Gilmore, S., Gnesi, S., Gönczy, L., Koch, N., Lapadula, A., Mayer, P., Mazzanti, F., Pugliese, R., Schroeder, A., Tiezzi, F., Tribastone, M., & Varró, D. (2008). Sensoria Patterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. In T. Margaria, & B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. (pp. 170-190). (Communications in Computer and Information Science). Springer Berlin / Heidelberg. 10.1007/978-3-540-88479-8_13

SENSORIA Patterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity*

Martin Wirsing[1], Matthias Hölzl[1], Lucia Acciai[2], Federico Banti[2], Allan Clark[3], Alessandro Fantechi[2], Stephen Gilmore[3], Stefania Gnesi[4], László Gönczy[5], Nora Koch[1], Alessandro Lapadula[2], Philip Mayer[1], Franco Mazzanti[4], Rosario Pugliese[2], Andreas Schroeder[1], Francesco Tiezzi[2], Mirco Tribastone[3], and Dániel Varró[5]

[1] Ludwig-Maximilians-Universität München, Germany
[2] Università degli Studi di Firenze
[3] University of Edinburgh, Scotland
[4] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" of CNR
[5] Budapest University of Technology and Economics

Abstract. The IST-FET Integrated Project SENSORIA is developing a novel comprehensive approach to the engineering of service-oriented software systems where foundational theories, techniques and methods are fully integrated into pragmatic software engineering processes. The techniques and tools of SENSORIA encompass the whole software development cycle, from business and architectural design, to quantitative and qualitative analysis of system properties, and to transformation and code generation. The SENSORIA approach takes also into account reconfiguration of service-oriented architectures (SOAs) and re-engineering of legacy systems.

In this paper we give first a short overview of SENSORIA and then present a pattern language for augmenting service engineering with formal analysis, transformation and dynamicity. The patterns are designed to help software developers choose appropriate tools and techniques to develop service-oriented systems with support from formal methods. They support the whole development process, from the modelling stage to deployment activities and give an overview of many of the research areas pursued in the SENSORIA project.

1 Introduction

Service-oriented computing is a paradigm where services are understood as autonomous, platform-independent computational entities that can be described, published, categorised, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. These characteristics have been responsible for the widespread success that service-oriented computing enjoys nowadays: many large companies invest efforts and resources in promoting service delivery on a variety of computing platforms, mostly through the Internet in the form of Web services. Soon there will be a plethora of new services as required for e-government, e-business, and e-science, and other areas within the rapidly evolving Information Society.

* This work has been partially sponsored by the project SENSORIA, IST-2 005-016004.

However, service-oriented computing and development today is mostly done in a non-systematic, ad-hoc way. Full-fledged theoretical foundations are missing, but they are badly needed for trusted interoperability, predictable compositionality, and for guaranteeing security, correctness, and appropriate resource usage.

The IST-FET Integrated Project SENSORIA addresses the problems of service-oriented computing by building, from first-principles, novel theories, methods and tools supporting the *engineering of software systems for service-oriented overlay computers*. Its aim is to develop a novel comprehensive approach to the engineering of service-oriented software systems where foundational theories, techniques and methods are fully integrated into pragmatic software engineering processes. The SENSORIA approach to service-oriented software development encompasses the whole development process, from systems in high-level languages, to deployment and re-engineering, with a particular focus on qualitative and quantitative analysis techniques, and automatic transformation between different development artifacts.

However, the broad range and the depth of the methods developed as part of the SENSORIA project means that it may be difficult for developers to identify the technique or tool that solves a particular problem arising in the development process, unless the developers are familiar with the whole range of scientific results of the project. To ameliorate this problem we are developing a catalogue of *patterns* that can serve as an index to our results and that illustrates, in a concise manner, the advantages and disadvantages of the individual techniques.

The structure of the paper is as follows: after a short overview of the SENSORIA project we explain the reasons for using patterns to present the SENSORIA results. Patterns are referenced in the usual format, with the pattern name followed by the page number of the pattern in parenthesis, e.g., *Service Modelling* describes the pattern named "Service Modelling" on page 6.

We then introduce several patterns ranging from the early design stage to deployment: *Service Modelling*, *Service Specification and Analysis*, *Functional Service Verification*, *Sensitivity Analysis*, *Scalability Analysis*, *Declarative Orchestration*, *Declarative Service Selection*, and *Model-Driven Deployment*. The last section summarises other results of the SENSORIA project and concludes.

2 The SENSORIA Project

SENSORIA is one of the three Integrated Projects of the Global Computing Initiative of FET-IST, the Future and Emerging Technologies action of the European Commission. The SENSORIA Consortium consists of 12 universities, three research institutes and four companies (two SMEs) from seven countries[6].

[6] LMU München (coordinator), Germany; TU Denmark at Lyngby, Denmark; Cirquent GmbH München, S&N AG, Paderborn (both Germany); Budapest University of Technology and Economics, Hungary; Università di Bologna, Università di Firenze, Università di Pisa, Università di Trento, ISTI Pisa, Telecom Italia Lab Torino, School of Management Politecnico di Milano (all Italy); Warsaw University, Poland; ATX Software SA, Lisboa, Universidade de Lisboa (both Portugal); Imperial College London, University College London, University of Edinburgh, University of Leicester (all United Kingdom).

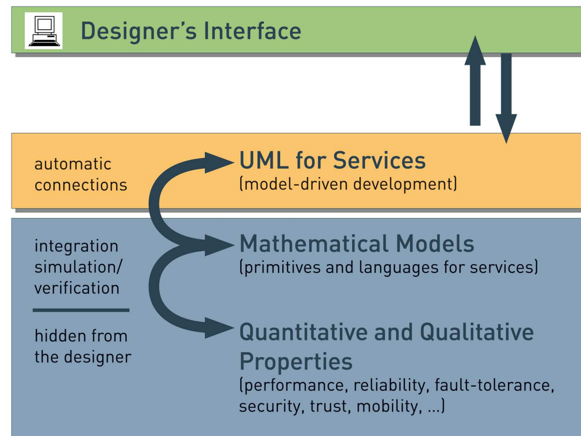

Fig. 1. SENSORIA approach: high-level models in UML4SOA are transformed into mathematical models based on the foundational calculi; qualitative and quantitative analsys can then be performed on these models.

2.1 The SENSORIA Approach

SENSORIA is focusing on global services that are context adaptive, personalisable, and may require hard and soft constraints on resources and performance, and takes into account the fact that services have to be deployed on different, possibly interoperating, platforms, to provide novel and reusable service-oriented systems.

To this end, SENSORIA is generalising the concept of service in such a way that

- it is independent from the particular global computer and from any programming language;
- it can be described in a modular way, so that security issues, quality of service measures and behavioural guarantees are preserved under composition of services;
- it supports dynamic, ad-hoc, "just-in-time" composition;
- it can be made part of an integrated service-oriented approach to business modelling.

The results of SENSORIA include a comprehensive service ontology, and modelling languages for service-oriented systems based on UML [32] and SCA [21,40]. We have also defined a number of process calculi for service-oriented computing, such as SCC [6], a session-oriented general purpose calculus for service description; *Sock* [26], a three layered calculus inspired by the Web Services protocol stack; and COWS [30], the Calculus for the Orchestration of Web Services.

These foundational process calculi serve as a base for higher-level formalisms to specify and analyse service-oriented systems, such as process calculi and languages for coordination, quality of service and service-level agreements [10,13], or type systems for services, e.g., for data exchange [31] or resource usage [3].

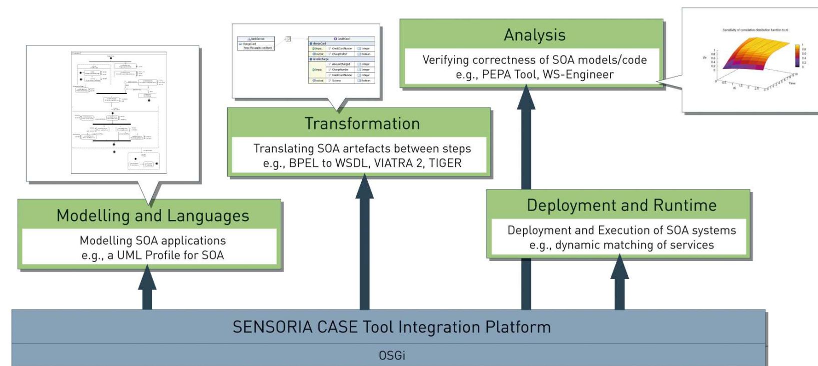

Fig. 2. SENSORIA tools, see [39] for the current list.

SENSORIA is also addressing the important areas of languages, frameworks, tools and techniques for qualitative and quantitative analysis. Qualitative analysis methods are successfully applied, e.g., to the areas of cryptography, security and trust [4,35,37], whereas calculi, logics and methods for quantitative analysis such as *StoKlaim* [16], *MoSL* [15], and *PEPA* [27] can be used in areas such as scalability or performance analysis [7].

Further work of SENSORIA concerns service contracts for checking the compliance of protocols and for automatic discovery and composition [8,9], new techniques for specifying and verifying the dynamic behaviour of services, including spatial logics and the verification of fault-tolerant systems [11,25], and programming- and modelling-level approaches to software architectures [22].

Moreover, SENSORIA is proposing a model-driven approach for service-oriented software engineering [41,32] that starts from high-level specifications in languages like SRML or UML4SOA and uses model transformation techniques [17,2] to generate both suitable input for the analysis tools, and executable services.

The development of mathematical foundations and mathematically well-founded engineering techniques for service-oriented computing constitutes a main research part of SENSORIA. Another important research direction focuses on making these foundations available for designers and developers by creating systematic and scientifically well-founded methods of service-oriented software development (cf. Fig. 1). The proposed approach is to build high-level models, e.g., in UML4SOA which can then be transformed into mathematical models based on the foundational calculi. Because of the precise definition of these calculi it is then possible to perform qualitative and quantitative analysis on the transformed models in order to gain valuable information about the quality, security, and performance of the system in the early stages of system development. Since the results of static analysis are transformed into annotations for the original high-level model, the designer does not have to be concerned with the formalisms used in the analysis process.

To facilitate the practical application of the results, SENSORIA is developing a service-based suite of tools (cf. Fig. 2) that support the new language primitives, the

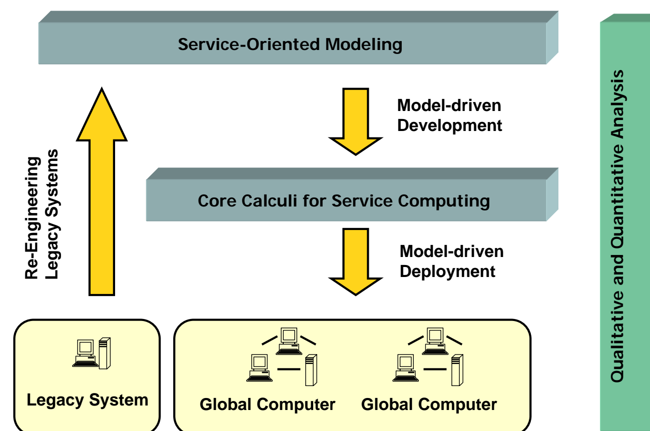

Fig. 3. SENSORIA research themes

analysis techniques, re-engineering of legacy software into services [14] and other aspects of service development and deployment [23,34]. The tool suite gives continuous feedback on the usefulness and applicability of the research results; it is also a starting point for the design of new industrial support tools for service-oriented development.

Another main element of SENSORIA is the set of realistic case studies for different important application areas including telecommunications, automotive, e-university, and e-business. Most of the case studies are defined by the industrial SENSORIA partners to provide continuous practical challenges for the new techniques of Services Engineering and demonstrate the research results.

The interplay of the different research themes and activities of SENSORIA are illustrated in Fig. 3: Service-oriented modelling provides specifications and models which are transformed by model-driven development into the core calculi for service computing. Model-driven deployment is used for implementing services on different platforms. Legacy systems can be transformed into services using systematic re-engineering techniques. Qualitative and quantitative analysis back the service development process and provide the means to guarantee functional and non-functional properties of services and service aggregates.

The impact of SENSORIA on the development of services will be to bring mathematically well-founded modelling technology within the reach of service-oriented software designers and developers. By offering these techniques and tools, we hope to allow adopters to move to a higher and more mature level of SOA software development. In particular, we hope to contribute to an increased quality of service of SOA applications, measured both in qualitative and quantitative terms. As SENSORIA methods are portable to existing platforms, application of these methods is possible while keeping existing investments.

2.2 A Pattern-Based Approach to Service Engineering

The SENSORIA project is investigating many issues of engineering SOAs. One of the challenges is to make the research results available in a way that is useful not only as the basis for future research but also for software developers seeking to apply the research results. To this end we are developing a pattern language that describes which problems are addressed by the various SENSORIA tools and techniques, how they solve the problems they address, and which forces determine whether a technique is appropriate for a given situation or not.

The SENSORIA patterns are not limited to implementation issues, they encompass a wide range of abstraction levels, from implementation-oriented patterns in the spirit of [24] to architectural or process patterns. We structure the patterns in a way that approximately follows the "Pattern Language for Pattern Writing" presented in [33], but add some pattern elements that seem to be helpful for describing patterns specifically related to service-oriented software engineering. For readers familiar with the pattern community, it should be noted that we use the pattern format as an expository tool; our patterns are not necessarily obtained by "mining" existing applications for patterns.

Several elements have to be contained in each pattern: a pattern name; a context in which the pattern is applicable; a concise description of the problem solved by the pattern; the forces that determine the benefits and drawbacks of using the pattern; the solution proposed and the consequences resulting from the use of the solution. Furthermore each pattern has to be accompanied by examples. Several optional sections can be used to clarify the pattern, e.g., related patterns, code or model samples, or tools to support the pattern. For space reasons we have omitted some of the mandatory elements from some of the patterns in this paper.

3 Service Modelling

Context. Systems built on SOAs add new layers of complexity to software engineering, as many different artifacts must work together to create the sort of loosely coupled, adaptive, fault-tolerant systems envisioned in the service domain. It is therefore important to apply best practices already in use for older programming paradigms to services as well; in particular, modelling of systems on a higher level of abstraction should be used to get a general idea of the solution space. Modelling services should be possible in a language which is both familiar to software architects and thus easy to use, but also contains the necessary elements for describing SOA systems in a straightforward way.

You are designing a system which is based on a SOA. The system is intended to offer services to multiple platforms and makes use of existing services and artifacts on multiple hosts which must be integrated to work together in order to realise the functionality of the system.

Problem. When designing SOA systems, it is easy to get lost in the detail of technical specifications and implementations. Providing an overview of the service oriented architecture to realise is therefore crucial for effective task identification, separation, and communication in large projects. In this context, using a familiar, easy-to-understand, and descriptive language is a key success factor.

Forces.

- The amount of specifications and platforms in the SOA environment makes it difficult to get a general idea of the solution space.
- Modelling the whole system in an abstract way gives a good overview of the tasks to be done, but does not directly yield tangible results. For small systems and projects, it is necessary to tailor this modelling task or even to skip it altogether.
- The model must be updated to reflect the architecture if it changes during implementation, or new requirements appear.
- The model is platform independent, and may be used to generate significant parts of the system. In case the system's target platform is not fixed or may experience changes, the workload involved in system re-implementation can be reduced considerably.
- Having a global architectural view eases the task of understanding the SOA environment considerably. This fact is of major significance if the SOA environment is to be extended by another team of software engineers or at a later date.
- The envisioned target platform(s) and language(s) should be supported by the modelling approach such that code generation may be used.

Solution. Use a specialised (graphical) modelling language to model the system and employ these models as far as possible for generating the system implementation. There are several languages which might be employed for this kind of task. One of the most widespread languages in the software engineering domain for modelling tasks is the Unified Modelling Language (UML). As UML itself however does not offer specific constructs for modelling service-oriented artifacts, it needs to be extended using its built-in profile mechanism. One profile for service oriented architectures is the UML4SOA profile [32], which enables modelling of both the static and the dynamic aspects of service-oriented systems. UML4SOA features specialised constructs for services, service providers and descriptions in its static part, as well as service interactions, long-running transactions, and event handling in its dynamic part. UML4SOA is also part of a model driven development approach for SOA, MDD4SOA, which in turn offers tools for generating code from UML4SOA models.

Consequences. A positive result of modelling a service-oriented system in a high-level way is that it gives a better idea of how the individual artifacts fit together. This is of particular importance in larger projects and for communication between developers and/or the customer. By using transformations, the models can also be employed for generating skeletons to fill with the actual implementation. However, the effort involved in creating readable models should not be underestimated. Also, care should be taken to only model aspects relevant on the design level instead of implementing the complete system on the modelling level.

A problem arising when specifying systems by models and applying model transformations to generate implementation fragments is the problem of model/implementation divergence. Therefore, special care must be taken that models are kept consistent with the implementation.

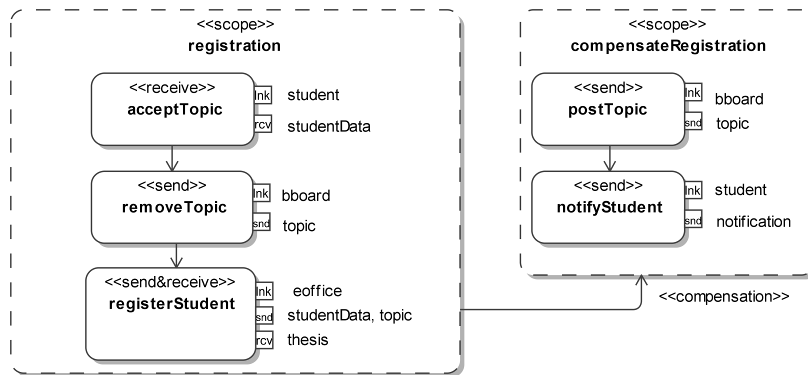

Fig. 4. UML4SOA activity diagram example

Tools. The use of a UML profile has the advantage that all UML CASE tools that support the extension mechanisms of the UML can be used, i.e. there is no need for the development of specific and proprietary tools. The UML4SOA profile may be provided already for the UML tool of choice, or may be defined by the means provided by the platform. In the SENSORIA project, the UML4SOA profile was defined for the Rational Software Modeler (RSM) and MagicDraw. MDD4SOA provides executable transformations for models from both UML tools to code skeletons of various target platforms, including the Web service platform and the Java platform. The transformers are integrated into the Eclipse environment.

Example. We illustrate the process by modelling an excerpt of a service-oriented eUniversity system: the management process of a student thesis, which is specified from the announcement of a thesis topic by a tutor to the final assessment and student notification. Figure 4 shows part of the orchestration process, namely the registration of the thesis and the compensation in case of cancellation.

The UML2 activity diagram shows several stereotypes from the UML4SOA profile:

– A scope is a UML StructuredActivityNode that contains arbitrary ActivityNodes, and may have an associated compensation handler.
– Specialised actions have been defined for sending and receiving data. In particular, a send is an UML CallBehaviourAction that sends a message; it does not block. A receive is a UML AcceptCallAction, receiving a message, which blocks until a message is received.
– Service interactions may have interaction pins for sending or receiving data. In particular, lnk is an UML Pin that holds a reference to the service involved in the interaction, snd is a Pin that holds a container with data to be sent, and rcv is a Pin that holds a container for data to be received.
– Finally, specialised edges connect scopes with handlers. For example, compensation is a UML ActivityEdge to add compensation handlers to actions and scopes.

Our profile also contains elements for event- and exception handling; they are not included here for lack of space. For a complete overview see [32].

4 Service Specification and Analysis

Context. You are designing a service-oriented system that has to operate in an open-ended computational environment. The system is supposed to rely on autonomous and possibly heterogeneous services, hence different services may be implemented by different languages. Information about actual implementation of some services may be not accessible and only the services interactive behavior is known.

Problem. Specify a service-oriented system and verify that it guarantees some desirable behavioural properties.

Forces.

- Process calculi have been proved able to define clean semantic models and lay rigorous methodological foundations for service-based applications and their composition.
- Process calculi enjoy a rich and elegant meta-theory and are equipped with a large set of analytical tools, such as e.g. typing systems, behavioural equivalences and temporal logics, that support reasoning on process specification.
- The additional cost and development effort incurred by using process calculi is only justified for systems with particularly high quality or security requirements.
- The use of process calculi requires highly trained personnel.

Solution. Use a service-oriented process calculus for formally specifying the system under consideration. Analyse the formal specification of the system by using suitable analytical tools.

Consequences. Process calculi, being defined algebraically, are inherently compositional and, therefore, convey in a distilled form the paradigm at the heart of service-oriented computing. On the other hand, a formal specification of service-oriented systems based on process calculi permits using powerful analysis tools to guarantee relevant properties.

Various kinds of typing systems, behavioural equivalences and temporal logics can be defined in order to deal with specific aspects of service-oriented systems. Thus, from time to time, the appropriate kind of reasoning mechanisms to work with should be chosen/defined depending on the property one intends to guarantee. As an example, to ensure that a system respects the expected behaviours, type systems can work in a complete statical manner or combine static and dynamic checks.

On the negative side, an analytical tool designed for a process calculus, in general, cannot be directly applied to a different one but has to be properly tailored.

Example. Two examples of process calculi suitable for modelling service-oriented systems are CaSPiS [5] and COWS [28]. Two classes of properties that, for example, can be verified on top of specifications defined by using the above calculi are *progress* (e.g. a client does not get stuck because of inadequate service communication capabilities) and *confidentiality* (e.g. critical data can be accessed only by authorised partners). Both properties can be verified by using the type systems introduced in [1], for CaSPiS, and in [29], for COWS, respectively.

Consider now a bank account service scenario where a client can ask for his balance. Specifically, upon receiving a balance request, the bank account service waits for the client's credentials and sends either the requested balance or an error message, depending on the validity of the credentials.

Code Example. Consider the following CaSPiS specification of the scenario:

$$
\begin{aligned}
BA &= bank_account.(c : \mathsf{credts}).\mathbf{if}\ \mathbf{is_valid}(c)\ \mathbf{then}\ \langle balance\rangle\ \mathbf{else}\ \langle \mathbf{err}\rangle \\
C &= \overline{bank_account}.\langle cred\rangle.\big(\,(b : \mathsf{int}).\uparrow \langle \mathbf{true}, b\rangle\ +\ (e : \mathsf{err}).\uparrow \langle \mathbf{false}, 0\rangle\big) \\
Sys &= C \mid BA
\end{aligned}
$$

where $\mathbf{err}$ is a message with associated type err and the validity of credentials is checked by means of an auxiliary function, $\mathbf{is_valid}$, that is private to the service $bank_account$.

According to the safety result in [1], client progress is guaranteed in Sys. Indeed, supposing that $bank_account$ has associated type $?\mathsf{credts}.(\,\tau.\,!\mathsf{int}\ +\ \tau.\,!\mathsf{err}\,)$, it can be inferred that client C is well-typed. More precisely, C's protocol has associated type $!\mathsf{credts}.(\,?\mathsf{int} + ?\mathsf{err}\,)$, which is compliant with $bank_account$'s type. Therefore, the whole system Sys is well-typed.

Consider now the new system Sys' defined below, where client C' does not comply with $bank_account$ communication protocol:

$$
\begin{aligned}
C' &= \overline{bank_account}.\,\langle cred\rangle.\,(b : \mathsf{int}).\uparrow \langle \mathbf{true}, b\rangle \\
Sys' &= C' \mid BA\ .
\end{aligned}
$$

C''s protocol has associated type $!\mathsf{credts}.?\mathsf{int}$, which clearly does not comply with $bank_account$'s type. Therefore, client progress is not guaranteed in Sys'. Actually, Sys' can reduce to $[(b : \mathsf{int}).\uparrow \langle \mathbf{true}, b\rangle |\!|\!| \langle \mathbf{err}\rangle]$, where client protocol $(b : \mathsf{int}).\uparrow \langle \mathbf{true}, b\rangle$ is stuck.

The same scenario can be specified by using COWS as follows:

$$
\begin{aligned}
BA = {} & *\,[x_{client}, x_{credts}]\, bank_account \bullet balance_req?\langle x_{client}, x_{credts}\rangle. \\
& \qquad [p, o]\,(\,p \bullet o!\langle \mathbf{is_valid}(x_{credts})\rangle \\
& \qquad\qquad \mid p \bullet o?\langle \mathbf{true}\rangle.\, x_{client} \bullet balance_resp!\langle balance\rangle \\
& \qquad\qquad\ +\ p \bullet o?\langle \mathbf{false}\rangle.\, x_{client} \bullet balance_resp!\langle \mathbf{err}\rangle\,) \\
C = {} & bank_account \bullet balance_req!\langle client, cred\rangle \mid [x]\, client \bullet balance_resp?\langle x\rangle \\
Sys = {} & C \mid BA\ .
\end{aligned}
$$

The type system for COWS introduced in [29] permits expressing and forcing policies regulating the exchange of data among interacting services and ensuring that, in

that respect, services do not manifest unexpected behaviours. This permits checking confidentiality properties, e.g., that client credentials are shared only with the bank account service. The types can be attached to each single datum and express the policies for data exchange in terms of sets of partners that are authorised to access the data. Thus, the credentials $cred$, communicated by C to BA, gets annotated with the policy $\{bank_account\}$, that allows BA to receive the datum but prevents it from transmitting the datum to other services. The typed version of C is defined as follows

$$\begin{array}{l} bank_account \bullet balance_req!\langle client, \{cred\}_{\{bank_account\}}\rangle \\ \mid [x]\, client \bullet balance_resp?\langle x\rangle \end{array}$$

Once the static type inference phase ends, the BA's variable x_{credts} gets annotated with the policy $\{bank_account\}$, which means that the datum that dynamically will replace x_{credts} will be used only by the partner $bank_account$. In this way, the communication can safely take place.

Suppose instead that service BA (accidentally or maliciously) attempts to reveal the credentials through some "internal" operation such as $p_{int} \bullet o!\langle\{x_{credts}\}_r\rangle$, for some set r such that $p_{int} \in r$. Then, as result of the inference, we would get declaration of variable x_{credts} annotated with r', for some set r' such that $r \subseteq r'$. Now, the communication would be blocked by a runtime check because the datum sent by C would be annotated as $\{cred\}_{\{bank_account\}}$ while the set r' of the receiving variable x_{credts} is such that $p_{int} \in r \subseteq r' \nsubseteq \{bank_account\}$.

Related Patterns. The *Functional Service Verification* pattern is often useful to verify services specified according to this pattern.

5 Functional Service Verification

Context. You are designing a service-oriented system that has to operate in an open-ended computational environment. The system should perform its tasks and should not manifest unexpected behaviours in each state of the environment.

Problem. Current software engineering technologies for service-oriented systems remain at the descriptive level and do not support formal reasoning mechanisms and analytical tools for checking that systems enjoy desirable properties.

Forces.

- The functionalities required of a service must be verified at design time.
- Properties to be insured by services should be expressed at a higher level of abstraction and therefore be independent from the technical details of the implementation.
- Logics have been since long proved able to reason about complex software systems as service-oriented applications are. In particular temporal logics have been proposed in the last twenty years, as suitable means for specifying properties of complex systems owing to their ability of expressing notions of necessity, possibility, eventuality, etc.
- The additional cost and development effort incurred by verification may only be justified for systems with particularly high quality or security requirements.
- Logic-based verification can only be performed by highly qualified developers.

Solution. Use a logical verification framework for checking functional properties of services by abstracting away from the environments in which they are operating. In particular, specify the properties of interest by using a temporal logic capable of capturing specific aspects of services, e.g. the logic SocL [20]. Define a formal specification of the system under consideration by using a process calculus, e.g. COWS [28], and, on top of this specification, define more abstract views by appropriately classifying system actions. Finally, verify the formulae over the abstract views of the system by using a model checker, e.g. the on-the-fly model checker CMC [20].

Consequences. The fact that the verification of properties is done over the abstract views of the system has many important advantages. On the one hand, it enables defining and working with many different abstract views of a system, thus reducing the complexity of the model of the system to be analysed. On the other hand, it enables defining service properties in terms of typical service actions (request, response, cancel, ...) and in a way that is independent of the actual specification of the service, both with regards to the process calculus used and with regards to the actual actions' names used in the specification. As a further consequence, it permits to identify classes of functional properties that services with similar functionalities must enjoy.

Example. Consider the following general properties that express two desirable attributes of services:

- *responsiveness*: the service under analysis always guarantees a response to each received request;
- *availability*: the service under analysis is always capable to accept a request.

Consider now a bank service scenario where a client can charge its credit card with some amount. Specifically, consider a client that tries to charge his credit card `1234` with two different amounts, Euros `100` and `200`, by performing two requests in parallel. An abstract view of the above system can be obtained by properly identifying the system actions corresponding to requests, responses and failure notifications of the interaction between the bank service and the client, and by specifying the system states where the service is able to accept a request. This way, the two general properties can be verified over the abstract system specification.

Code Example. The two properties presented in the previous section can be expressed as SocL formulae as follows:

- *responsiveness*: `AG(accepting_request(charge))`
- *availability*:
```
AG[request(charge,$id)]
    AF{response(charge,%id)
       or fail(charge,%id)} true
```

where `charge` indicates the interaction between the bank service and the client, while the variable `id` is used to correlate responses and failure notifications to the proper accepted requests.

A COWS specification of the scenario is

```
let
  Bank = * [CUST] [CC] [AMOUNT] [ID]
        bank.charge?<CUST,CC,AMOUNT,ID>.
        [p#][o#] (p.o!<>
              | p.o?<>. CUST.chargeOK!<ID>
            + p.o?<>. CUST.chargeFail!<ID>)

  Client = bank.charge!<client,1234,100,id1>
         | (client.chargeOK?<id1>.nil
            + client.chargeFail?<id1>.nil)
       | bank.charge!<client,1234,200,id2>
          | (client.chargeOK?<id2>.nil
             + client.chargeFail?<id2>.nil)
in
  Bank() | Client()
end
```

Once prompted by a request, the service `Bank` creates one specific instance to serve that request and is immediately ready to concurrently serve other requests. Two different correlation values, `id1` and `id2`, are used to correlate the response messages to the corresponding requests. Notably, for the sake of simplicity, the choice between approving or not a request for charging the credit card is here completely non-deterministic. An abstract view of the system can be obtained by applying the following rules:

```
Abstractions {
  Action charge<*,*,*,$1> -> request(charge,$1)
  Action chargeOK<$1>     -> response(charge,$1)
  Action chargeFail<$1>   -> fail(charge,$1)
  State  charge           -> accepting_request(charge)
}
```

The first rule prescribes that whenever the concrete actions `bank.charge!<client,1234,100,id1>` and `bank.charge!<client,1234,200,id2>` are executed, then they are replaced by the abstract actions `request(charge,id1)` and `request(charge,id2)`, respectively. Variables "`$n`" (with `n` natural number) can be used to defined generic (templates of) abstraction rules. Also the wildcard "`*`" can be used for increasing flexibility. The other rules act similarly. Notably, communications internal to the bank service are not transformed and, thus, become unobservable.

Related Patterns. The *Service Specification and Analysis* pattern is often useful to specify services that should be verified.

Tools. The tool CMC can be used to prove that the bank service specified above exhibits the desired characteristics to be available and responsive. A prototypical version of CMC can be experimented via a web interface available at the address `http://fmt.isti.cnr.it/cmc/`.

6 Sensitivity analysis

Context. You are analysing a service-oriented system in order to identify areas where the system performance can be improved with relatively little effort. There are many potential ways in which the system can be modified including optimising software components, purchasing new hardware or infrastructure, re-deploying existing hardware resources for other purposes, and many other possibilities.

Problem. Identify a low-cost method of improving system performance.

Forces.

- The impact of changes on system performance can be hard to predict. Improving the efficiency of one component will not necessarily lead to an improvement overall. Optimisations which are applied in the wrong place may even lead to the overall performance being reduced.
- Some changes are expensive, others cheap. One change might require replacing a large part of the communication network, another might require rewrites of complex software, whereas one might require only reducing a delay such as a timeout.
- Given the many possible changes one could make it is infeasible to try each of them and compare the relative increase (or decrease) in performance.

Solution. Develop a high-level quantitative model of the service and experiment on the model in order to determine the changes which have the greatest positive impact. Of these, identify those which can be implemented with lowest cost, and carry out this implementation. The quantitative model can be evaluated using a modelling tool such as a simulator or a Markov chain solver computing the transient and passage-time measures which relate to user-perceived performance, together with the use of parameter sweep across the model to vary activity rates.

Consequences. The analysis has the potential to identify useful areas where optimisations can be applied. The numerical evaluation may be long-running but it is entirely automatic. The quantitative evaluation has the potential to generate many analysis results which need to be considered and assessed by a domain expert.

Example. This pattern is applied in [13] to investigate an automotive accident assistance service. A framework for experimentation and analysis allows many instances of a Markov chain solver to be executed and the results combined to identify how the service can most easily meet its required service-level agreement.

Related Patterns. The *Service Specification and Analysis* pattern is complementary in the sense that it uses similar methods to analyse behaviour.

Tools. The SENSORIA Development Environment hosts formal analysis tools which allow service engineers to perform parameter sweep across models of services expressed in the PEPA process algebra [27]. The PEPA model is automatically compiled into a continuous-time Markov chain and passage-time analysis is performed using the `ipclib` analysis tools [12].

7 Scalability analysis

Context. You are a large-scale service provider using replication to scale your service provision to support large user populations. You need to understand the impact on your service provision of changes in the number of servers which you have available or changes in the number of users subscribed to your service.

Problem. Understanding the impact of changes on a large-scale system.

Forces.

- Large user populations represent success: this service is considered by many people to be important or even vital. Scale of use is a tangible and quantifiable measure of value and being able to support large-scale use is an indicator of quality in planning, execution and deployment in service provision. Maintaining a large-scale system attracts prestige, attention and acclaim.
- Large user populations represent heavy demand. The service must be replicated in order to serve many clients. Replication represents cost in terms of hosting provision, hardware and electricity bills. Service providers would like to reduce service provision while continuing to serve large user populations.
- Modelling would help with understanding the system but large-scale systems are difficult to model. Conventional discrete-state quantitative analysis methods are limited by the the size of the probability distribution vector across all of the states of the system. Discrete-state models are subject to the well-known state-space explosion problem. It is not possible simply to use a very large Markov chain model to analyse this problem.

Solution. Develop a high-level model of the system and apply continuous-space analysis to the model. A continuous-space model can make predictions about a large-scale system where a discrete-state model cannot.

Consequences. TO DO

Related Patterns. The *Sensitivity Analysis* pattern is closely related in that it is possible to use the parameter sweep employed there to perform *dimensioning* for large-scale systems (i.e. determining whether a planned system has enough capacity to serve an estimated user population).

Tools. The SENSORIA Development Environment hosts analysis tools which allow service engineers to perform continuous-space analysis on models expressed in the PEPA process algebra [27]. The PEPA model is automatically compiled into a set of coupled ordinary differential equations and the initial value problem is evaluated using numerical integration. This predicts the number of users in different states of using the service at all future time points. Static analysis, compilation and integration are performed using the PEPA Eclipse Plug-in Project [36].

8 Declarative Orchestration

Context. You are designing a service-oriented system that has to operate in an open-ended, changing environment in which the presence of certain services cannot be guaranteed. The system should perform its tasks to the maximum extent possible in each state of the environment, possibly by utilising features of the environment that were not present when the system was designed.

Problem. Design a service-oriented system that can operate in an open-ended, changing environment.

Forces.

- A pre-determined orchestration of services cannot adapt to significant, unforeseen changes in the environment.
- Specifying orchestrations for all possible changes is not feasible in some environments.
- Not having a pre-determined orchestration makes it more difficult to reason about the system.
- If the environment is too different from the one for which a system was originally designed it may no longer be possible to fulfil the system's function.
- Services have to provide a rich semantic description to be usable for declarative orchestrations.

Solution. Define an ontology for the problem domain that is rich enough to capture the capabilities of services. Specify the results of combining several services in a declarative manner, e.g., as plan components or logical implications. Use a reasoning component such as an planner, model checker, or a theorem prover to create orchestrations from these specifications and a description of the current environment.

Consequences. Declarative orchestrations can adapt to large changes in the environment without manual reconfiguration. They can easily incorporate information about new kinds of services and use them to fulfil their tasks.

On the negative side, declarative orchestration depends on an expressive domain model for which the reasoning process is often computationally expensive and time consuming, and also on the availability of rich semantic descriptions of unknown services. It is often difficult to control the behaviour of systems built on top of reasoning components and to ensure their correctness.

Related Patterns. Unless the environment is extremely unpredictable, a system designed according to *Declarative Service Selection* can often satisfy similar requirements while remaining easier to understand and analyse.

9 Declarative Service Selection

Context. You have designed an orchestration for a service-oriented system. During runtime, a number of services with similar functionality but different cost, reliability and quality trade-offs are available that can fulfil the requirements of the orchestration.

Problem. Find an optimal combination of services, taking into account the current situation and user preferences.

Forces.

- The functionality required of the services is determined by the orchestration.
- The services available at run-time are not known during design-time.
- Different services with the same functionality can be differentiated according to other Quality of Service metrics.

Solution. Define a context-aware soft-constraint system that ranks solution according to their quality. Model user preferences using a partial order between the criteria described by individual soft constraints when possible, otherwise build a more complex mapping from the values of individual constraints to an overall result that describes the user preferences. A soft-constraint solver can the compute the optimal combination of services or a "good enough" combination of services computable in a certain time frame.

Consequences. The specification of the problem can be given without reference to a solution algorithm, thus the communication with domain experts and users is simplified. The computation of the quality of different combinations of services and the preference given to each individual characteristic are decoupled from each other. A soft-constraint solver provides a general mechanism to compute the desired combination of services.

On the other hand, the choice of evaluation functions is restricted by the theories that the soft-constraint solver can process. A general-purpose mechanism such as a solver is often less efficient than a well-tuned specialised implementation.

10 Model-Driven Deployment

Context. You are designing a service configuration where non-functional requirements (security, reliable messaging, etc.) play an important role. Models are designed in UML while the underlying standards-compliant platform have to be parametrised at a very low abstraction level (e.g. using specific APIs or XML formats).

Problem. There is a big semantical gap between the modelling and deployment concepts. Platforms and concepts are changing rapidly, interoperability is not guaranteed between low level models.

Forces. A service configuration is typically designed in high level modelling languages such as UML. The configuration of the underlying implementation platforms, however, needs the deep technical knowledge of related standards and product specific know-how. Services have to be redeployed, refactored and moved between runtime environments. Moreover, non-functional properties should be handled differently for different classes of users. It should be avoided to have the service designer specify the detailed technical requirements, he should rather work with predefined profiles.

Solution. We propose a multiple-step model driven workflow where separate model transformations implement the PIM2PSM and PSM2code mappings, as defined in the MDA approach. Services have to be modelled either in a specialised UML dialect or in a Domain Specific Editor. First, relevant parts of the model are filtered out and stored in a simplified representation format (neglecting e.g. tool-specific information). Then different Platform Independent Models are created for the different aspects of non-functional requirements, e.g. security, reliable messaging, component deployment, etc. Up to this step, platform requirements do not affect the process. Platform Specific Models contain implementation-specific attributes, taken from the PIM and predefined parameter libraries. Finally, structured textual code (e.g. XML descriptors) is generated.

Consequences. The method has the potential to connect high level models to low level runtime representations. Transformation chain targets server configurations with extensions for reliable messaging and security.

Example. Examples are the UML4SOA for modelling, VIATRA2 framework for transformation and Apache Axis (using Rampart and Sandesha modules) and IBM WebSphere as relevant industrial platforms. The method is used in different scenarios of the project.

Tools. The input of transformation is UML2 models in UML4SOA (designed e.g. in Rational Software Architect (RSA)). The transformation is integrated in the SENSORIA Development Environment while the output consist of descriptor files and client stubs.

11 Related Work

The idea of using patterns to describe common problems in software design and development was popularised by the so-called "Gang of Four" book [24]. Since its publication a wide range of patterns and pattern languages for many areas of software development has been published, see, e.g, the Pattern Languages of Programs (PLoP) conferences and the associated Pattern Languages of Program Design volumes, or the LNCS Transactions on Pattern Languages of Programming.

The area of patterns for SOA has recently gained a lot of attention, and several collections of design patterns for SOA have been recently published or announced [19,38]. The article [18] provides a short introduction. However these patterns address more general problems of SOA, while our patterns are focused on the formally supported techniques provided by SENSORIA. Therefore, our patterns can serve as an extension of, rather than as a replacement for, other pattern catalogues.

12 Conclusions and Further Work

In this paper, we have presented some results of the IST-FET EU project SENSORIA, in the form of a pattern language. The patterns address a broad range of issues, such

as modelling, specification, analysis, verification, orchestration, and deployment of services. We are currently working on systematising and extending the collection of patterns in these areas, and we will also be developing patterns for areas which are not currently addressed, e.g., business process analysis and modelling.

This pattern catalogue is a useful guide to the research results of the SENSORIA project: as already mentioned in Section 2.1, we are investigating a broad range of subjects and without some guidance it may not be easy for software developers to find the appropriate tools or techniques.

However, the patterns presented in this paper only present a very brief glimpse at the research of the SENSORIA project. Important research areas include a new generalised concept of service, modelling languages for services based on UML and SCA, new semantically well-defined modelling and programming primitives for services, new powerful mathematical analysis and verification techniques and tools for system behaviour and quality of service properties, and novel model-based transformation and development techniques. The innovative methods of SENSORIA are being demonstrated by applying them to case studies in the service-intensive areas of e-business, automotive systems, and telecommunications.

By integrating and further developing these results SENSORIA will achieve its overall aim: a comprehensive and pragmatic but theoretically well founded approach to software engineering for service-oriented systems.

References

1. L. Acciai and M. Boreale. A Type System for Client Progress in a Service-Oriented Calculus. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 642–658. Springer, 2008.
2. András Balogh and Dániel Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pages 1280–1287, Dijon, France, April 2006. ACM Press.
3. Massimo Bartoletti, Pierpaolo Degano, Gianluigi Ferrari, and Roberto Zunino. Types and effects for Resouce Usage Analysis. *Foundations of Software Science and Computation Structures, FOSSACS'07*, 4423, 2007.
4. Maurice H. ter Beek, Corrado Moiso, and Marinella Petrocchi. Towards Security Analyses of an Identity Federation Protocol for Web Services in Convergent Networks. In *Proceedings of the 3rd Advanced International Conference on Telecommunications (AICT '07)*. IEEE Computer Society, Los Alamitos, CA, 2007.
5. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *Proc. of Formal Methods for Open Object-Based Distributed Systems (FMOODS2008)*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2008.
6. Michele Boreale, Roberto Bruni, Luis Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, Antonio Ravara, Davide Sangiorgi, Vasco Vasconcelos, and Gianluigi Zavattaro. SCC: a Service Centered Calculus. In M. Bravetti and G. Zavattaro, editors, *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer Verlag, 2006.

7. Mario Bravetti, Stephen Gilmore, Claudio Guidi, and Mirco Tribastone. Replicating web services for scalability. In G. Barthe and C. Fournet, editors, *Proceedings of the Third International Conference on Trustworthy Global Computing (TGC'07)*, volume 4912 of *LNCS*, pages 204–221. Springer-Verlag, 2008.
8. Mario Bravetti and Gianluigi Zavattaro. A Theory for Strong Service Compliance. In Amy L. Murphy andJan Vitek, editor, *Proceedings of COORDINATION 2007*, volume 4467 of *Lecture Notes in Computer Science*, pages 96–112, Paphos, Cyprus, 2007. Springer.
9. Mario Bravetti and Gianluigi Zavattaro. Contract based Multi-party Service Composition. In Farhad Arbab and Marjan Sirjani, editors, *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings*, volume 4767 of *Lecture Notes in Computer Science*, Iran, Tehran, 2007. Springer.
10. Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In R. De Nicola, editor, *Proc. of the 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2007.
11. Vincenzo Ciancia and Gianluigi Ferrari. Co-Algebraic Models for Quantitative Spatial Logics. In *Quantitative Aspects of Programming Languages (QAPL'07)*, 2007.
12. Allan Clark. The ipclib PEPA Library. In Mor Harchol-Balter, Marta Kwiatkowska, and Miklos Telek, editors, *Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 55–56. IEEE, September 2007.
13. Allan Clark and Stephen Gilmore. Evaluating quality of service for service level agreements. In Luboš Brim and Martin Leucker, editors, *Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems*, pages 172–185, Bonn, Germany, August 2006.
14. Rui Correia, Carlos Matos, Reiko Heckel, and Mohammad El-Ramly. Architecture migration driven by code categorization. In Flavio Oquendo, editor, *ECSA'07*, volume 4758 of *LNCS*, pages 115–122. Springer, 2007.
15. Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, Michele Loreti, and Mieke Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, 2007.
16. Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. STOKLAIM: A Stochastic Extension of KLAIM. Technical Report 2006-TR-01, ISTI, 2006.
17. Karsten Ehrig, Gabriele Taentzer, and Dániel Varró. Tool Integration by Model Transformations based on the Eclipse Modeling Framework. *EASST Newsletter*, 12, June 2006.
18. Thomas Erl. Introducing soa design patterns. *SOA World Magazine*, 8(6), June 2008.
19. Thomas Erl. *SOA Design Patterns*. Prentice Hall/Pearson PTR, 2008. To appear.
20. Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese, and Francesco Tiezzi. A model checking approach for verifying COWS specifications. In J. L. Fiadeiro and P. Inverardi, editors, *Proc. of Fundamental Approaches to Software Engineering (FASE'08)*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.
21. José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. *Web Services and Formal Methods*, 4184:193–213, 2006.
22. Howard Foster, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Towards Self-Management in Service-oriented Computing with Modes. In *Proceedings of Workshop on Engineering Service-Oriented Applications (WESOA07), Vienna, Austria*, Vienna, September 2007. Imperial College London.
23. Howard Foster and Philip Mayer. Leveraging integrated tools for model-based analysis of service compositions. In *In Proceedings of the Third International Conference on Internet and Web Applications and Services (ICIW 2008)*, Athens, Greece, 2008. IEEE Computer Society Press.

24. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
25. László Gönczy and Dániel Varró. Modeling of Reliable Messaging in Service Oriented Architectures. In *Proc. of the International Workshop on Web Services - Modeling and Testing*, 2006.
26. Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proceedings of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.
27. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
28. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.
29. A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *Proc. of IPM International Symposium on Fundamentals of Software Engineering (FSEN'07)*, volume 4767 of *Lecture Notes in Computer Science*, pages 223–239. Springer, 2007.
30. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Calculus for Orchestration of Web Services. In R. De Nicola, editor, *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
31. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Regulating data exchange in service oriented applications. In F. Arbab and M. Sirjani, editors, *Proc. of IPM International Symposium on Fundamentals of Software Engineering (FSEN'07)*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.
32. Philip Mayer, Andreas Schroeder, and Nora Koch. A Model-Driven Approach to Service Orchestration. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2008)*, IEEE. IEEE, 2008.
33. G. Meszaros and J. Doble. Metapatterns: A pattern language for pattern writing, 1996.
34. Arun Mukhija, Andrew Dingwall-Smith, and David S. Rosenblum. QoS-Aware Service Composition in Dino. In *Proceedings of the 5th European Conference on Web Services (ECOWS 2007), Halle, Germany*, Halle, Germany, 2007. IEEE Computer Society.
35. Flemming Nielson and Hanne Riis Nielson. A flow-sensitive analysis of privacy properties. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 249–264. IEEE Computer Society, 2007.
36. Web site for the pepa eclipse plugin. `http://homepages.inf.ed.ac.uk/mtribast/plugin/download.html`, last accessed 2008-06-24.
37. Christian W. Probst, Flemming Nielson, and René Rydhof Hansen. Sandboxing in myKlaim. In *The First International Conference on Availability, Reliability and Security, ARES 2006*, 2006.
38. Arnon Rotem-Gal-Oz. *SOA Patterns*. Manning, 2009. To appear.
39. Tools integrated into the SENSORIA Development Environment. `http://svn.pst.ifi.lmu.de/trac/sct/wiki/SensoriaTools`.
40. Martin Wirsing, Laura Bocchi, Allan Clark, José Luiz Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, and Rosario Pugliese. *SENSORIA: Engineering for Service-Oriented Overlay Computers*, chapter 7. MIT, June 2007. submitted.
41. Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. Semantic-Based Development of Service-Oriented Systems. In E. Najn et al., editor, *Proc. 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems(FORTE'06), Paris, France*, LNCS 4229, pages 24–45. Springer-Verlag, 2006.