Sensoria Patterns

Matthias Hölzl, Nora Koch, Philip Mayer, and Martin Wirsing[⋆]
{hoelzl,koch,mayer,wirsing}@pst.ifi.lmu.de

Ludwig-Maximilians-Universität, München, Germany

Abstract. We describe the SENSORIA development approach using a pattern language for augmenting service engineering with formal analysis, transformation and dynamicity. The pattern language is designed to help software developers choose appropriate tools and techniques to develop service-oriented systems with support from formal methods; the full pattern catalog spans the whole development process from the modeling stage to deployment activities. Some of the patterns are specific to SENSORIA; other patterns are extensions or adaptations of patterns presented by other authors.

1 Introduction

The SENSORIA project is investigating a broad range of issues related to engineering service-oriented architectures, ranging from foundational research to practically usable tools. SENSORIA proposes a model-driven approach in which services are first modelled in a platform-independent notation such as UML4SOA [14]; these designs are then transformed into formal models that can be analysed using tools based on mathematical methods. Afterwards, the design models can be used to generate code for different service platforms.

The research results of SENSORIA are widely disseminated and well known in the broader scientific community. However, scientific publications generally contain little guidance for the practical software developer who seeks to apply them. To make research results available in a way that is useful beyond the scientific community we have developed a pattern catalog that enables software engineers to quickly determine whether SENSORIA tools and techniques exist to address a particular development problem and, in the case of a positive answer, the recommended approach for employing them.

Pattern-based approaches to presenting guidance for software developers has a well-established history in computer science; patterns have been used to describe problems and possible solutions in areas ranging from business processes to software design and low-level implementation methods. Given the wide scope of SENSORIA it is not surprising that the full catalog of SENSORIA patterns also encompasses a wide range of abstraction levels, from implementation-oriented patterns in the spirit of [10] to architectural or process patterns. The pattern language is inspired by the "Pattern Language for Pattern Writing" presented in [15], the pattern language used in [8] and the guidance in [5]. For readers familiar with the pattern community, it should be noted that we use

[⋆] This work has been partially sponsored by the EU project SENSORIA (IST-2 005-016004).

the pattern format as an expository tool; our patterns are not necessarily obtained by "mining" existing applications for patterns.

1.1 Overview of SENSORIA Patterns

The patterns presented in this section can be classified into three different categories:

- Patterns for new activities in the software development process that allow SENSORIA methods to be introduced. For example, the pattern *Functional Service Verification* presented in this chapter is of this kind: it introduces the formal verification of services and details the benefits and costs of performing such analysis.
- Patterns that show how SENSORIA techniques change an activity that is commonly performed when developing and deploying service-oriented architectures. The patterns *Service Modeling* and *Generate Implementation* in this chapter are examples of this type of pattern: while all model-driven approaches generate parts of the implementation from models, the benefits and costs of doing so vary significantly when tools based on formal models are employed in the course of software development.
- Adaptations of patterns presented by other authors, in those cases where the tools and techniques developed by SENSORIA contribute novel aspects to the proposed solution or offer new solution possibilities. In these cases we include an extended version of the existing pattern in the SENSORIA pattern language to document the influence of our techniques on the proposed solution and to point developers to the tools that can be employed when realizing the pattern. Tailoring patterns to specific tools or circumstances has a long tradition in the pattern community, see, e.g., [1], which shows how the patterns in [10] can be adapted to a particular programming language.

1.2 The SENSORIA Pattern Language

The pattern language used in this chapter is a slight revision of the one first introduced in [25]; it is based on principles described in [15,5] and similar to the one used in [8].

As usual, each pattern consists of mandatory and optional elements which are presented in a fairly rigid structure to simplify the process of selecting and applying suitable patterns. The elements that have to be present in each pattern are:

- A pattern **name** that provides a short and descriptive way to refer to the pattern.
- A **context** in which the pattern is applicable: most patterns are not "universal" solutions but only apply in certain circumstances which are described by the context.
- A concise **description** of the problem solved by the pattern. This is different from the context in that the problem is the design or implementation challenge that is directly addressed by the pattern whereas the context describes conditions in which the pattern is applicable but which are generally not influenced by an application of the pattern.
- The **forces** that determine whether using the pattern is appropriate.

– The **solution** proposed and the **consequences** resulting from the use of the solution. The solution is the concrete description how the pattern is applied; the consequences show how this influences the resulting system. There is a certain amount of overlap between consequences and forces, but generally the forces are more abstract, concise and less precise than the detailed consequences.
– Furthermore each pattern has to be accompanied by **examples**. In this chapter we mostly refer the reader to other chapters that demonstrate the relevant pattern.

Several optional sections can be used to clarify the pattern, e.g., **related patterns**, example **code** or **models**, or **tools** to support the pattern. For space reasons we have omitted some elements appearing in the full pattern catalog from some of the patterns in this chapter.

This chapter is structured as follows: the following section contains patterns that outline a development process that allows SENSORIA techniques to be used to maximum effect. Afterwards, section 3 shows an example how patterns for developing service-oriented architectures from Erl's pattern catalog [8] can be enhanced by making use of SENSORIA results. The final sections present related work and conclude.

2 SENSORIA Development Patterns

This section uses patterns to describe a development process that makes maximum use of SENSORIA results: *Service Modeling* introduces a modeling process using SoaML and UML4SOA; this pattern is a slight revision of the one presented in [25]. Having models in UML is the basis for two more patterns presented in this chapter: *Extract Formal Models* and *Generate Implementation*. Formal models are mainly useful as input for analysis tools; this is described by pattern *Analyze with Formal Methods*. Most development projects extend or replace existing legacy systems. The pattern *Extract Service Model* enables the developer to use SENSORIA tools and techniques for these systems as well.

2.1 Service Modeling

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, modeling of systems on a higher level of abstraction should be used to get a general idea of the solution space. Modeling 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.

Context. 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. Visualizing the planned service oriented architecture is therefore crucial for effective task identification, separation, and communication. Using a familiar, easy-to-understand, and descriptive language is a key success factor in this context.

Forces.

– The amount of specifications and platforms in the SOA domain makes it difficult to get a general idea of the solution space.
– Modeling 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 modeling task or even to skip it altogether.
– The model must be updated to reflect the architecture if it changes during implementation, or if 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. 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 modeling approach such that code generation may be used.

Solution. Use a specialised (graphical) modeling 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 modeling languages in the software engineering domain is the Unified Modeling Language (UML). As UML itself does not offer specific constructs for modeling service-oriented artifacts, it needs to be extended using its built-in profile mechanism. SoaML [18] and UML4SOA [14] are two such profiles, which together enable modeling of both static and dynamic aspects of service-oriented systems. SoaML allows modelling the static part of SOA systems and features specialised constructs for services, service providers, and message types. UML4SOA complements SoaML with support for the dynamic parts of SOA systems, featuring service interactions, long-running transactions, and event handling. Models designed using SoaML and UML4SOA can be used in a model driven development approach for SOA, MDD4SOA [14], which offers tools for generating code.

Consequences. *Pros:* A positive result of modeling 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 modeling level.

Cons: Often the fully automated generation of implementations is not feasible; instead only implementation fragments can be generated and their implementation has to be completed manually. In this scenario model/implementation divergence may pose a significant problem and special care has to be taken that models are kept consistent with the implementation. This increases the cost of modeling and reduces the benefit of model-driven development.

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 SoaML and UML4SOA profiles 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; SoaML is available for these platforms as well. 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.

Examples. More detailed descriptions of languages for service modeling and their applications are given in Chapter 1-1 (UML Extensions for Service-Oriented Systems) and Chapter 1-2 (The SENSORIA Reference Modelling Language); a model-driven approach to business processes is introduced in Chapter 1-3 (Model-Driven Development of Adaptable Service-Oriented Business Processes). More detailed examples for models can be found in the chapters on case studies, in particular Chapter 7-4 (The SENSORIA Approach Applied to the Finance Case Study) and Chapter 7-2 (SENSORIA Results Applied to the Case Studies).

Related Patterns. This pattern forms the basis for the approach described in this chapter; it is a requirement for *Extract Formal Models* and *Generate Implementation*.

2.2 Extract Formal Models

Context. You have modeled a part of the system using UML4SOA and want to ensure that the model satisfies certain properties.

Problem. Many properties of models in UML4SOA cannot be directly analyzed. Manually building models for formal analysis has several disadvantages: (1) The manually created models may not faithfully represent the UML4SOA model. (2) Manually building models is a time-intensive process. (3) The model has to be manually kept in sync with changes made to the UML4SOA model.

Forces.

– Extraction of formal models allows the UML4SOA model to be analyzed without manually creating additional models.

– The extracted model faithfully represents the UML4SOA model if the extractor is correct.
– The extracted model may be more complicated than a manually created model and contain details that are unnecessary for the desired analysis. This may significantly increase the complexity of the analysis step.
– The UML4SOA model has to be elaborated in detail to contain enough information for model extraction.

Solution. Use tools to automatically extract formal models from the UML4SOA models. The kind of models that should be extracted depend on the analysis that is to be performed.

Consequences. *Pros:* Manually building formal models is expensive and not economically feasible for most large systems. Additionally, a manual extraction process may introduce errors not present in the original model, or fail to correctly specify all subtleties of the original model. If a tool that automatically extracts the required models exists, analysis with formal methods becomes more reliable and significantly cheaper. SENSORIA provides a number of model transformations from UML4SOA into process calculi and orchestration languages – for example, the process calculi COWS and PEPA, or the language BPEL. These tools can be integrated into the build process of the system such that the availability of up-to-date formal models is ensured.

Cons: Automatically generated models often contain details that are not relevant to the performed analyses. Since many tools based on formal methods suffer from "state explosion" problems, the increased size of the extracted model can impede analysis efforts.

Tools. SENSORIA provides various tools for transforming UML models (and in particular, SoaML and UML4SOA models). The model transformer Hugo/RT [12] translates UML specifications into input languages for the well-known model checkers UPPAAL and SPIN. The SRMC/UML bridge translates UML4SOA activities into the process calculus PEPA [23]. The VENUS tool allows converting UML4SOA activities into the process calculus COWS [13]. Furthermore, the MDD4SOA transformer suite [14] translates UML4SOA diagrams to Java, the orchestration language Jolie, and WS-BPEL; the latter can be used as input for the verification tool WS-Engineer [9].

Examples. A detailed example of how the pattern *Extract Formal Models* can be used in the software development process can be found in Chapter 6-1 (Methodologies for Model-Driven Development and Deployment: an Overview); more details about transformations is contained in Chapter 6-2 (Advances in Model Transformations by Graph Transformation: Specification, Analysis and Execution).

Related Patterns. The pattern *Extract Formal Models* is closely related to *Analyze with Formal Methods* since model extraction often precedes formal analysis. It is also related to *Generate Implementation* since the additional modeling effort required for formal analysis can better be recouped if the model also serves as input to code generation.

2.3 Analyze with Formal Methods

While models are often at a higher level of abstraction than code, they nevertheless are susceptible to the same problems: they may not satisfy certain properties that the modeler expects them to have; different models may specify the same part of the system in contradictory ways, etc. Unless the models are executable and therefore relatively low-level, these defects may remain undetected until the system is actually implemented. This negates one of the main benefits that modeling is supposed to provide.

Context. You have either extracted or manually specified formal models of the system.

Problem. You want to verify that the formal models satisfy certain properties, e.g., a given service should always be available to accept new requests or the overall system should be free from deadlocks.

Forces.

- Formal models of the components under consideration exist or can be extracted.
- The desired properties can be formally specified.
- Developers have to be qualified to decide which analysis tools are adequate for the given problem, be able to use the tools, and in some cases interpret their output.

Solution. Use tools based on formal methods such as model checkers or the performance analysis tools for Pepa [6,11] to analyze whether the desired properties hold.

Consequences. *Pros:* Tools based on formal methods can verify that the system satisfies certain properties that are difficult to check otherwise, or that the system exhibits certain performance characteristics. If the tools can be applied at an early stage of system development it is possible to find design and modeling errors long before the system is implemented and therefore reduce the development cost. Furthermore, certain kinds of errors that are well-suited to formal analysis, such as deadlocks or unintended interactions that divulge secret information to third parties, are notoriously difficult to find using traditional approaches.

Cons: On the other hand the use of analysis tools based on formal methods requires a lot of experience on the part of the users: even when using hidden formal methods the user has to be able to determine which properties of the system are amenable to formal analysis, which tools are appropriate, and how the desired properties can be encoded. This can, to a certain extent, be ameliorated by new developments such as Venus [22], but it is unlikely that the use of formal methods will be completely transparent to the developer in the foreseeable future. Furthermore, many tools based on formal methods require a detailed specification of the complete system behavior and therefore necessitate comprehensive models of all system components, even ones that are not directly involved in the behavior under consideration. Related to this last point is another weakness of some formal methods: they cannot work on open systems and results can therefore only be obtained for "closed approximations" of the specified system. This is, in general, not problematic when the existence of undesirable behavior is demonstrated by the formal analysis, e.g., when deadlocks or traces which validate system invariants are found. But it is often not clear that positive results, e.g., the absence of invalid traces, can be transferred from a closed approximation to the open system.

Tools. SENSORIA provides several tools for formal analysis and verification. WS-Engineer [9] is a verification tool for performing model-based verification of web service compositions. The SRMC/PEPA tool [23] covers steady-state analysis of the underlying Markov chain of SRMC descriptions. CMC and UMC are model checkers and analysers for systems defined by interacting UML statecharts [21]. The sCOWS Model Checker [20] allows to perform statistical model checking on sCOWS, a stochastic extension of COWS. Finally, the LySA tool is a static analyzer for security protocols defined in the LYSA process calculus [4].

Examples. Parts 2, 4 and 5 of this book contain many examples for formal analysis methods; in particular, examples for qualitative analysis techniques can be found in Chapter 2-3 (Static Analysis Techniques for Session-Oriented Calculi), Chapter 4-1 (Analysing the Protocol Stack for Services), Chapter 4-2 (An Abstract, On-The-Fly Framework for the Verification of Service Oriented Systems), Chapter 4-3 (Tools and Verification), and Chapter 4-4 (Specification and Analysis of Dynamically-Reconfigurable Service Architectures); examples for quantitative analysis techniques are given in Chapter 5-1 (SoSL: Service Oriented Stochastic logics), Chapter 5-2 (Evaluating Service Level Agreements using Observational Probes), Chapter 5-3 (Scaling Performance Analysis using Fluid-Flow Approximation), Chapter 5-4 (Passage-End Analysis for Analysing Robot Movement) and Chapter 5-5 (Quantitative Analysis of Services).

Related Patterns. The formal models for analysis can often be extracted as described in the pattern *Extract Formal Models*. The detailed system model needed for formal analysis often contains many of the same model refinements that are needed to employ the *Generate Implementation* pattern.

2.4 Generate Implementation

Generating implementations from models is the key characteristic of model-driven development. The SENSORIA approach for formally supported software development supports such generation with multiple tools.

Context. You are deciding which development approach to apply to a software system, or you have already developed (UML/UML4SOA) models for the system. You want to implement the system on one or more platforms.

Problem. While UML models are a useful development tool, many models do not specify executable behavior, and even for behavioral models there is no widely used execution platform that can directly operate on UML models.

Forces.

- UML models can be specified at various levels of abstraction ranging from very abstract structural views of a system to detailed behavioral descriptions.
- Even behavioral specifications are often not detailed enough to completely describe the intended behavior of the system.

– The amount of work to fully specify all system behaviors is significant when compared to the commonly used level of abstraction for models.
– Implementations can be obtained in various ways: manual implementation of the model, partial code-generation by a CASE tool, or generation of the complete application.
– Some parts of an application are not easily specified using UML, e.g., user interfaces.

Solution. Fully specify the important behavior of the system (that implements the business process) in the model, generate code from this implementation, and manually implement parts of the code that are difficult to model and verify.

Consequences. *Pros:* By generating the implementation from models, the correctness of the implementation relative to the model depends only on the quality of the transformation from models to code. Once a mature code generator has been developed, the consistency of model and implementation can be assumed. Changes to the model can immediately be reflected in the implementation without incurring additional implementation costs.

Generating implementations for different execution platforms is easy if model transformations into all platforms exist. Only the manually written parts of the code have to be rewritten when supporting an new platform or transitioning to a new platform. Necessary deployment artifacts can automatically generated.

Neutral: By manually implementing those parts of the system which are difficult to express in UML and for which no formal verification is necessary, the modeling effort can be reduced, at the cost of an increase in platform dependencies and implementation costs.

Cons: The required detail of the models increases significantly, thereby making the modeling step more time consuming and increasing the difficulty of changes to the models. If no model transformation into the desired target platform is available it has to be developed, often at significant cost. In some cases, manually generated code can be smaller and more efficient than automatically generated code; in particular if the code generator is not sophisticated enough to perform static analysis and optimization. This may be particularly significant when developing software for embedded or otherwise resource-constrained systems. Furthermore, debugging of generated implementations often has to be performed at the source-code level since no back-translation from code to model elements is available in the debugger. This can make debugging of generated implementations difficult.

Tools. SENSORIA provides both a generic model transformation tool for writing and executing arbitrary transformations and specific tools tailored towards a single use case. The former tool is Viatra2 [24], a framework which provides general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains. The latter are provided by multiple tools. The first two are written in VIATRA2: The SOA2WSDL transformation takes high level UML4SOA models and produces WSDL output, whereas the

UML2Axis transformations take high level UML4OA models and produce WSDL, WS-ReliableMessaging, WS-Security and Apache Axis-specific configuration files as output. The tool suite MDD4SOA [14] already mentioned above transforms SoaML and UML4SOA models to BPEL, WSDL, and XSD as well as Java and Jolie. Finally, the Modes Parser and Browser [9] generates broker requirements for Dino from UML2 Modes models.

Examples. The chapters on the SENSORIA case studies contain examples for the generation of implementations, see in particular Chapter 7-4 (The SENSORIA Approach Applied to the Finance Case Study) and Chapter 7-2 (SENSORIA Results Applied to the Case Studies).

Related Patterns. This pattern enjoys a synergistic relationship with *Extract Formal Models*, since a more detailed UML model can enable both extraction of formal models and generating the implementation. This can significantly alter the cost/benefit balance of detailed modeling.

2.5 Extract Service Model

Service-oriented architectures are generally not developed from scratch; in most cases the functionality of existing business-critical legacy systems has to be integrated or replaced. A number of patterns for integrating legacy software into a service-oriented architecture exist, with different trade-offs. For example, in [8] the *Legacy Wrapper* pattern is introduced which wraps the legacy system with a service façade. While this is a relatively quick and cheap solution it often poses difficulties for long-term maintenance and deployment. As SENSORIA has developed the powerful, model-based re-engineering tool *CareStudio* the *Extract Service Model* pattern is a viable alternative with higher up-front investments but better long-term maintainability.

Context. You have a legacy application that performs a vital business function, possibly with complex business logic integrated into the code. You want to integrate the legacy application into a service-oriented infrastructure.

Problem. Often service-oriented systems are introduced to supersede existing legacy technologies. In these cases it is usually not feasible to re-implement the functionality of the legacy system; therefore, its functionality is wrapped by a service binding. If the legacy system is hidden behind a thin service layer, the interface of the wrapper is largely pre-determined by the capabilities and interfaces of the existing solution which are often at the wrong level of abstraction or granularity for a service-oriented architecture. This leads to non-standard service contracts that expose details of the legacy system's implementation and technology. Writing a thick wrapper that exposes a clean service-oriented interface is often difficult as many legacy systems do not expose a clean separation between user interface, domain logic and storage backend.

Forces.

- A legacy application performing vital functionality exists.
- It is not economically feasible or desirable to develop replacements for the legacy systems from scratch. Therefore the legacy application should be integrated into a service-oriented architecture.
- The resulting software is not only supposed to facilitate transition to another system.
- The new system should exhibit clean service contracts between its components to be maintainable and extensible for future requirements.

Solution. Use the SENSORIA re-engineering approach which consists in annotating the original source code, extracting a service model from the annotated source code, and generating a new, service-oriented implementation from the annotated code.

Consequences. *Pros:* By creating a service model and a truly service-oriented implementation the long-term maintainability and extensibility is ensured. Often integration into a service-oriented architecture can be more seamless than wrapper-based solutions; the extracted service-oriented implementation can make use of standard infrastructure services provided by the environment and therefore profit from enhancements made to the overall system.

Cons: For re-engineering purposes the source code of the legacy system has to be available, which is often not the case. The legacy code has to be annotated which makes it necessary that developers that understand the old code base are available or that developers familiarize themselves with the old code base.

Neutral: The up-front cost of re-engineering can be significantly higher than the cost of wrapping a legacy system, althogh this will often be offset by reduced deployment and operational costs and better future extensibility of the system.

Tools. This pattern is supported by a set of SENSORIA tools grouped around *CareStudio* [2] which allow transformations to be applied to source code, with a focus on achieving SOA-compliant code. The tool uses a model transformation approach to the migration and includes emitters for source code.

Examples. A comprehensive description of the *Extract Service Model* pattern is contained in Chapter 6-4 (Legacy Transformations for Extracting Service Components).

Related Patterns. When following this pattern, a service model is extracted and then used for code generation. The pattern is therefore closely related to *Service Modeling* and *Generate Implementation*. The extracted model can be used to formally validate properties of the system using patterns *Extract Formal Models* and *Analyze with Formal Methods*.

3 Enhancing SOA Patterns

This section describes how existing patterns for SOA development can be extended with SENSORIA tools and methods, contributing novel aspects to the proposed solution and offering new solution possibilities. We present extensions of two patterns from Thomas Erl's "SOA Design Patterns" [8], *Concurrent Contracts* and *Trusted Subsystem*.

3.1 Concurrent Contracts

Contracts between client and service and interfaces offered by services play an important role in the development of service-oriented architectures. The following pattern shows how techniques developed as part of SENSORIA can help service providers to offer suitable interfaces for different classes of clients, and clients to find and utilize the most appropriate contract offered by a provider. It is an extension of the *Concurrent Contracts* pattern from Thomas Erl's "SOA Design Patterns" [8] with SENSORIA-specific material. The pattern as originally described deals only with contracts; we have extended the pattern to also take into account services that offer multiple interfaces backed by the same implementation.

Context. Services often have to serve different customers which have slightly different needs and permissions, and which may be of unknown provenience and therefore trusted to different degrees. Exposing the same interface and service contract to all clients may therefore not be feasible; on the other hand having different services for closely related and largely overlapping functionality is not desirable.

Problem. Often services are described as exposing a single interface or contract that the service fulfills. This simple view does not adequately reflect the situation encountered when building service-oriented architectures: often different clients have many overlapping requirements but also significant differences. For example, several clients may request personnel data from a company's "personnel service," but there may be differences in that

- they may be trusted to different degrees, e.g., services operated by the company itself may enjoy higher trust than services operated by clients or partners of the company;
- some clients may be allowed to see protected data, e.g. services operated by the accounting department may have access to salary information which is not available to other services;
- some clients may be allowed to issue more powerful queries, e.g., the statistics department may be allowed to issue queries that aggregate data whereas other clients may only be able to query individual employees.

To avoid undue multiplication of services it seems desirable to have a single service that handles all clients; on the other hand the differences in the clients may make it difficult or even impossible to define a single service interface or contract that satisfies the needs of all clients. Furthermore such an interface will expose unneeded complexity to clients that do not need advanced capabilities and the definition of a single policy that covers all the different clients is often difficult and poses governance and administration problems.

Forces.

- The service has to accommodate different types of consumers with important similarities but significant differences. For example, some customers may be less trusted than others.
- It is desirable to limit the number of deployed services and to avoid duplicated functionality in several services.
- Defining a single interface and a single contract that satisfy the needs of all clients is difficult or impossible.
- Exposing several contracts and interfaces for a service may increase the complexity of the system and make it more difficult for clients to choose an appropriate service.

Solution. The same underlying service implementation may expose several different interfaces or contracts. Each exposed interface or contract can be optimized for the needs of one customer or several customers with similar needs and trustworthiness. Each contract can be versioned and governed individually, thereby simplifying deployment and governance of individual contracts; interfaces are generally more closely tied to the service implementation than contracts, but by using a model-driven approach and carefully separating interfaces and implementation during design time a certain degree of independent versioning and governance can be ensured for interfaces as well; in particular it is often possible to maintain backward-compatible interfaces when the implementation of a service is upgraded.

On the other hand, having to provide the functionality for several contracts places additional burden on the implementation and evolution of the service itself. Care has to be taken that changes to the implementation do not violate the guarantees of any exposed interface or contract. This effect can be ameliorated by employing the formal methods developed as part of the SENSORIA project to verify that the implementation is faithful to the guarantees of each exposed interface or contract.

Similar situations exists for consumption and provisioning of the service: By providing multiple interfaces each client has to choose the most appropriate one; this increases the time a developer needs to understand the system and diminishes the positive effect of having specialized interfaces for the needs of several clients. For the service provider, multiplying the number of contracts and interfaces may increase the governance effort and deployment complexity of the whole system, even though they are reduced for each individual service interface. In both cases techniques developed by SENSORIA, in particular "call-by-contract" as provided by λ^{req} and the dynamic selection of available service interfaces and contracts by Dino [16], can help ameliorate these problems.

Consequences. *Pros:* Introducing new interfaces and contracts that are closely matched to the requirements of a group of clients can greatly simplify the development of clients as well as governance and deployment. Providing several interfaces can reduce the need for different services that provide closely related functionality.

Cons: Adding new interfaces to a service has similar governance and management overhead to adding completely new services. Indeterminate application of the *Concurrent Contracts* Pattern can therefore lead to a overly large service inventory that is difficult to use, maintain and develop.

Tools. By using UML4SOA in an early development stage, as for example with the application of the *Service Modeling* pattern, you can simplify the application of *Concurrent Contracts*. Service providers can also use UML4SOA during deployment to formulate the capabilities and potential consumers of each service contract.

When formal methods are used to verify the contracts with respect to their implementation, the model transformations and tools corresponding to the chosen verification method can be used. Particularly applicable tools for this pattern are PEPA or SMRC to analyze whether the performance characteristics of the provided contracts match the needs of the clients, see Chapter 5-3 (Scaling Performance Analysis using Fluid-Flow Approximation) and Chapter 5-5 (Quantitative Analysis of Services). Furthermore, λ^{req} enables requirements-based selection of service contracts, see Chapter 2-4 (Call-by-Contract for Service Discovery, Orchestration and Recovery). Dino can be used to provide semantic matching of services and interfaces at run time, see Chapter 6-3 (Runtime Support for Dynamic and Adaptive Service Composition). Process-calculus based static analysis methods can be used to verify the correctness of the provided contracts with respect to their implementation; see pattern *Analyze with Formal Methods* for more details.

Examples. Examples for the application of the *Concurrent Contracts* pattern can be found in Erl [8]; more detailed examples of the application of the SENSORIA methods are described in the chapters of this volume mentioned in the previous section. In addition Chapter 3-2 (Advanced Mechanisms for Service Composition, Query and Discovery) and Chapter 6-3 (Runtime Support for Dynamic and Adaptive Service Composition) provide information about dynamic discovery of appropriate service contracts.

Related Patterns. For *Concurrent Contracts* to be applied, the service contract itself should ideally be fully decoupled from the underlying service implementation; often a façade that supports multiple contracts without the need for redundant service logic can be used to implement this. See the patterns *Decoupled Contract* and *Service Façade* described in Erl [8] for further information about this topic. Other patterns that support different clients for a service are *Contract Denormalization* and *Validation Abstraction*, also described in Erl [8]. However, when using *Concurrent Contracts* the need for contract denormalization may be reduced since the capabilities required by different clients could be exposed by separate contracts.

Application of the *Concurrent Contracts* pattern can often be simplified by using the *Service Modeling* pattern to model the contracts and the shared implementation artefacts. After the *Concurrent Contracts* pattern has been applied, the *Analyze with Formal Methods* pattern can be used to ensure that the functional and non-functional properties of the resulting service satisfy the requirements of the contracts and interfaces.

3.2 Trusted Subsystem

As more and more critical data is stored in and processed by service-oriented systems, ensuring their availability while securing them against unauthorized access and malicious attacks has become a priority. A large number of tools and techniques have been developed to address these issues. Here we focus on one possible design pattern, the

Trusted Subsystem. Our description is an adaptation and extension of some important points presented in Thomas Erl's "SOA Design Patterns"; the full description of the pattern with examples and discussion of useful technologies can be found there.

Context. You are designing a service-oriented system that processes critical or confidential data. In this system, some services are exposed to clients that do not have access rights. You want to protect the data and make it easy and transparent to grant and revoke authorizations.

Problem. Granting clients direct access to services containing important data poses many security problem and complicates the management of authorizations. Furthermore it poses problems of *transitive trust:* if service A calls service B on behalf of client C, who is responsible for checking that the call is authorized?

Forces.

– Services should be protected from unauthorized access.
– Management of authorizations should be easy and transparent.

Solution. The services containing critical or confidential data can only be accessed via another service that is responsible for verifying the client's authorizations. This trusted front-end service always uses its own credentials to access the protected resources. Client authorizations are not passed on by the front-end to the protected resources, but a client identifier may be included in the calls to the protected resources. The trusted subsystem is responsible for verifying that all accesses to the resources are performed only by authorized clients and that clients cannot pass counterfeit identifiers to services. The front-end service thus establishes a trust boundary. When applying this pattern to several front-end services acting on the same resources it is possible to establish nested or overlapping trust boundaries.

Consequences. *Pros:* The front-end service is responsible for enforcing the trust boundary for the protected subsystems. Therefore there is a single point where access policy can be implemented, monitored and authored. Since credentials are established by the client for complete transactions there is no problem with transitive trust relationships. Services inside the trusted boundary can have very simple security mechanisms since they only have to authenticate the trusted subsystem.

Cons: The trusted subsystem is a single point of failure and also a potential performance bottleneck since it must process every interaction with the protected resources. Security breaches of the front-end can have devastating consequences for the whole system as a compromise of this subsystem can be used to exploit all downstream resources in its trust boundaries. It is therefore a prime target for attackers.

Neutral: SENSORIA methods can achieve a particularly good relationship between cost and effectiveness when they are applied to the trusted subsystem: by validating the security properties of this service using qualitative methods a high degree of trust in the security of the whole system inside the trust boundary can be established; by using qualitative analysis to analyze the performance characteristics of the system bottlenecks can be discovered and prevented during early design stages.

Tools. Essentially, the whole range of SENSORIA modeling and analysis methods can be gainfully employed to model and analyze the trusted subsystem. In particular, λ^{req} can often be used to validate the contracts of the trusted subsystems, and Lysa [3], the corresponding LysaTool [26,4] and CryptoKlaim [17] can be employed to establish the security of the protocols between clients and the trusted service as well as inside the trust boundary. Qualitative analysis of arbitrary properties including security is also supported by the SENSORIA model checkers WS-Engineer [9], CMC and UMC [21], and sCOWS [20]. Finally, the SRMC/PEPA tool [23] covers the performance side of the analysis with steady-state analysis of the underlying Markov chain of SRMC descriptions.

Examples. Examples for trusted subsystems can be found in Erl [8]. Qualitative analysis including verification of security properties is discussed in Chapter 4-2 (An Abstract, On-The-Fly Framework for the Verification of Service Oriented Systems), and Chapter 4-3 (Tools and Verification). See Chapter 4-1 (Analysing the Protocol Stack for Services) for an example of applying the LysaTool. Quantitative, and in particular performance analysis is discussed in Chapter 5-5 (Quantitative Analysis of Services).

Related Patterns. Since the *Trusted Subsystem* pattern identifies services which are particularly worthwhile targets for the SENSORIA tools and methods, it is related to most of the other patterns presented in this section: *Service Modeling* of the trusted subsystem can enable the use of other patterns, such as *Extract Formal Models*, *Analyze with Formal Methods*, or *Generate Implementation*.

4 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 [10]. 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 published or announced [8,19]. The article [7] 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.

5 Conclusions and Further Work

In this chapter, 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

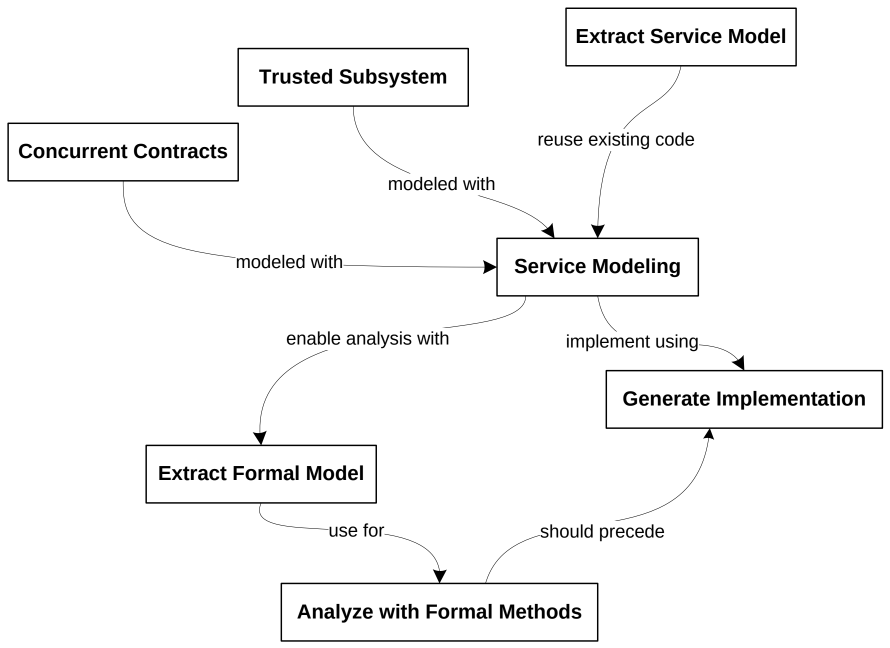

Fig. 1. SENSORIA Pattern Relationships

as modelling, specification, analysis, verification, orchestration, and deployment of services. As a final treat, the relationships between the patterns introduced in this chapter is shown in the above figure.

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 the introduction, 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.

References

1. S. Alpert, K. Brown, and B. Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley Professional, 1998.
2. ATX Technologies. *Modernizing Software and Increasing Business Values*. `http://www.atxtechnologies.co.uk/`.
3. C. Bodei, P. Degano, H. Gao, and H. Nielson. Detecting Replay Attacks by Freshness Annotations. In *Proceedings of WITS'07*, Informatics and Mathematical Modelling, Technical University, April 2007. Dipartimento di Informatica.
4. M. Buchholtz and H. R. Nielson. *LySaTool*. `http://www.imm.dtu.dk/English/Research/Language-Based_Technology/Software/LySaTool/`.

5. F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley, June 2007.
6. A. Clark, S. Gilmore, J. Hillston, and M. Tribastone. *Formal Methods for Performance Evaluation: the 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007*, volume 4486, chapter Stochastic Process Algebras, pages 132–179. Springer-Verlag, Bertinoro, Italy, May–June 2007.
7. T. Erl. Introducing soa design patterns. *SOA World Magazine*, 8(6), June 2008.
8. T. Erl. *SOA Design Patterns*. Prentice Hall/Pearson PTR, 2008.
9. H. Foster, S. Uchitel, J. Kramer, and J. Magee. WS-Engineer: A Tool for Model-Based Verification of Web Service Compositions and Choreography. In *IEEE International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 2006.*, 2006.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Co., Inc., Boston, MA, USA, 1995.
11. J. Hillston. Fluid Flow Approximation of PEPA models. In *Proc. 2nd Int. Conf. Quantitative Evaluation of Systems (QEST 2005)*, IEEE, 2005.
12. A. Knapp. A formal approach to object-oriented software engineering. *Softwaretechnik-Trends*, 21(3), 2001.
13. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In R. D. Nicola, editor, *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
14. P. Mayer, A. Schroeder, and N. Koch. A Model-Driven Approach to Service Orchestration. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2008)*, IEEE. IEEE, 2008.
15. G. Meszaros and J. Doble. Metapatterns: A pattern language for pattern writing, 1996.
16. A. Mukhija, A. Dingwall-Smith, and D. 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.
17. C. Nielsen, F. Nielson, and H. Nielson. CryptoKlaim. Work in progress., 2006.
18. OMG. *Service Oriented Architecture Modelling Language Beta 1*. `http://www.soaml.org`.
19. A. Rotem-Gal-Oz. *SOA Patterns*. Manning, 2009. To appear.
20. S. Schivo. *sCOWS Model Checker*. `http://sites.google.com/site/sschivo/scows-model-checker`.
21. M. H. ter Beek, F. Mazzanti, and S. Gnesi. Cmc-umc: a framework for the verification of abstract service-oriented properties. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 2111–2117. ACM, 2009.
22. F. Tiezzi. *Venus: A Verification ENvironment for UML models of Services*. `http://rap.dsi.unifi.it/cows/`.
23. M. Tribastone. The PEPA Plug-in Project. In *Fourth International Conference on the Quantitative Evaluation of Systems*, pages 53–54, UK, September 2007. IEEE Computer Society.
24. VIATRA2 Project. *VIATRA2 (VIsual Automated model TRAnsformations)*. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html`.
25. M. Wirsing, M. M. Hölzl, L. Acciai, F. Banti, A. Clark, A. Fantechi, S. Gilmore, S. Gnesi, L. Gönczy, N. Koch, A. Lapadula, P. Mayer, F. Mazzanti, R. Pugliese, A. Schroeder, F. Tiezzi, M. Tribastone, and D. Varró. SENSORIA Patterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 170–190. Springer, 2008.
26. E. Yüksel, H. Nielson, C. Nielsen, and M. Örencik. A Secure Simplification of the PKMv2 Protocol in IEEE 802.16e-2005. *FCS-ARSPA'07*, 2007. Informal Proceedings.