From Scenarios to Aspects: Exploring Product Lines

Ingolf H. Krüger, Reena Mathew
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093-0114, USA
{ikrueger,rmathew}@cs.ucsd.edu

Michael Meisinger
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
meisinge@in.tum.de

ABSTRACT

Software product lines are gaining importance because they allow improvements in time to market, cost, productivity and quality of software products. Architecture evaluation is one important aspect in the development of product lines for large-scale distributed systems. It is desirable to evaluate and compare architectures for functionality and quality attributes before implementing or changing the whole system. Often, the effort required for the thorough evaluation of alternatives using prototypes is prohibitive. In this paper, we present an approach for cost-efficient software architecture evaluation, based on scenario-oriented software specifications, modeling the system services. We show how to map the same set of services to several possible target architectures and give a procedure to generate evaluation prototypes using aspect-oriented programming techniques. This significantly reduces the effort required to explore architectural alternatives. We explain our approach using the Center TRACON Automation System as an example.

1. INTRODUCTION

As stated by the SEI, software product lines are rapidly emerging as a viable and important software development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers. A *software product line (SPL)* is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [20].

Designing product line software architectures for complex distributed systems is a difficult task. One of its difficulties, the exploration of different architectural alternatives often falls victim to time pressure and lack of resources: it typically involves writing large parts of the code for each product line alternative upfront to support the evaluation – this binds people, time and financial resources.

1.1 Problem Definition

Building prototypes and running simulations complements and provides input for architecture evaluation techniques based on reviews and estimation [2]. However, building multiple prototypes to explore different architectural alternatives is costly.

One way to address this problem is to separate an overall software architecture into abstract models (sometimes called *domain models* [3]) and implementation models; approaches advocating this separation are architecture-centric software development [22] and model-driven architecture [15].

This separation is often difficult to achieve – especially in situations where requirements including performance and other Quality-of-Service properties suggest a tight coupling between the two types of models. Then, exploring multiple architectural alternatives is often not an option, because writing "throw-away" prototypes is too costly: large parts of the deployment infrastructure have to be written and rewritten for each prototype. Besides that, the mapping between abstract and implementation model is non-trivial in general; there is likely more than one mapping that makes sense. Multiple explorative hand-crafted mappings between the different models quickly also become costly.

The core of the problem is that the scenarios supported by the system typically involve a multitude of collaborating entities within each of the two types of models. The interplay of these entities in the abstract model needs to be mapped to a corresponding setup within the implementation model; for multiple architecture alternatives of a product line this leads to re-implementing the same interplay over and over again. Our goal is to provide a solution where one unchanged specification can be reused across all implementation models (or target-architectures) to be evaluated.

1.2 Service-Oriented Development

In this paper, we propose an approach to architecture evaluation and exploration that establishes a clean separation between the services provided by the system under consideration, and the architecture – comprised of components and their relationships – implementing the services.

We use the notion of *service* to decouple abstract behavior from implementation architectures supporting it. The term "service" is used in multiple different meanings and on multiple different levels of abstraction throughout the Software Engineering community [21]. We view services as specializations of use cases to specify scenarios; they "orchestrate" the interaction among certain entities of the system under consideration to achieve a certain goal [3]. In contrast to "use cases", which describe functionality typically in prose and on a coarse level of detail, we define a service via the interaction pattern among a set of collaborators required to deliver the functionality.

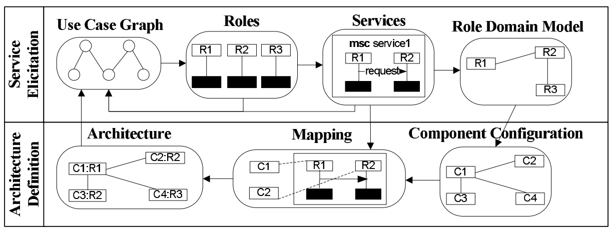


Figure 1: Service-Oriented Development Process

We employ a two-phase, iterative development process as shown in Fig. 1. Phase (1), *Service Elicitation*, consists of defining the set of services of interest – we call this set the service repository. Phase (2), *Architecture Definition*, consists of mapping the services to component configurations to define deployments of the architecture.

In phase (1) we identify the relevant use cases and their relationships in the form of a use case graph. This gives us a relatively large scale scenario-based view on the system. From the use cases, we derive sets of *roles* and *services* as interaction patterns among roles. This decouples from interaction details, because roles abstract from components or objects. Roles describe the contribution of an entity to a particular service independently of what concrete implementation component will deliver this contribution. An object or component of the implementation typically will play multiple roles at the same time. The relationships between the roles, including aggregations and multiplicities, develop into the *role domain model*.

In phase (2) the role domain model is refined into a *component configuration*, onto which the set of services is *mapped* to yield an *architectural configuration*. These architectural configurations can be readily implemented and evaluated as target architectures for the system under consideration.

The process is iterative both within the two phases, and across: Role and service elicitation feeds back into the definition of the use case graph; architectures can be refined and refactored to yield new architectural configurations, which may lead to further refinement of the use cases.

1.3 Architecture Exploration and Aspects

Our service notion is based on interaction patterns that cross-cut the entities implementing them; services emerge as cross-cutting aspects of both abstract and concrete models. This observation motivates our use of aspect-oriented programming to capture services as aspects and to use weaving techniques [8] to establish the mapping between one abstract model and multiple concrete models.

This ability to effectively and efficiently generate multiple candidate architectures for execution is critical to render architecture evaluation practical; it allows us to support architectural review and analysis techniques [2] with data gathered from prototypic implementations and simulation runs.

AspectJ [7] provides the infrastructure we need to translate services into aspects. In Sect. 3 we explain this mapping in detail, and show how the combination of services and aspects helps us to solve the problems explained above for architecture implementation, evaluation and improvement of software product lines.

1.4 Contributions and Outline

The major contribution of this work is to introduce an approach for exploring software architectures based on the notions of services and aspects. We give a translation procedure from service models to AspectJ implementations. Because we disentangle the specification of functionality (the services) from the infrastructure on which they are implemented we can build and evaluate different target architectures without having to rewrite large portions of the code, resulting in reduced effort and time for performing an architecture evaluation.

In Section 2 we introduce the Center Tracon Automation System (CTAS) as our running example and show how it is modeled in terms of services. In Section 3, we explain how to translate the architecture definition to aspects for implementing the system. In Section 4 we present experiences with applying our approach to evaluate various architectures for CTAS and in section 5 we discuss it in the context of related work. Section 6 contains conclusions and an outlook.

2. SERVICE-ORIENTED MODEL OF CTAS

To demonstrate our approach, we use the Center TRACON Automation System (CTAS), a case study from the air-traffic control domain, as an example of a realistic distributed system [17]. CTAS is a set of tools and processes designed to help air traffic controllers manage the increasingly complex air traffic flows at large airports. An important part of this system is the distribution of weather updates to interested clients; this is the part we concentrate on in our case study.

The main component of the CTAS weather update system is the communications manager CM; other processes, including route analysis (RA), and the plan-view GUI (PGUI), are clients to CM. Clients are distinguished as *aware* or *unaware* depending on whether they participate in the weather update process. The CTAS requirements [17] explain how the clients initialize with CM, and how CM subsequently relays the latest weather information to all aware clients. For this paper, we design and implement various architectures for the weather update functionality of the CTAS system. CM continuously checks if a new weather report is available. If so, the CM sends a message to all aware clients. Each client responds, indicating whether it can process the weather update successfully. If all clients indicate success, the CM asks all the clients to use the latest weather information. If at least one of the clients indicates failure, the CM informs all clients to use the old weather information, to prevent inconsistent use of weather data across the different clients.

In the following we given an overview how to define, implement and evaluate architectures for supporting this process using services and aspects. For more detailed information refer to [11].

2.1 Scenario-Oriented Specification: Services

Analyzing the requirements leads to a number of use cases and roles. The roles relevant for our example are *Aware-Client* (weather-aware clients), *Manager* (drives the update process), *Broadcaster* (broadcasts messages to a group of clients) and *Arbiter* (collects responses from groups of clients).

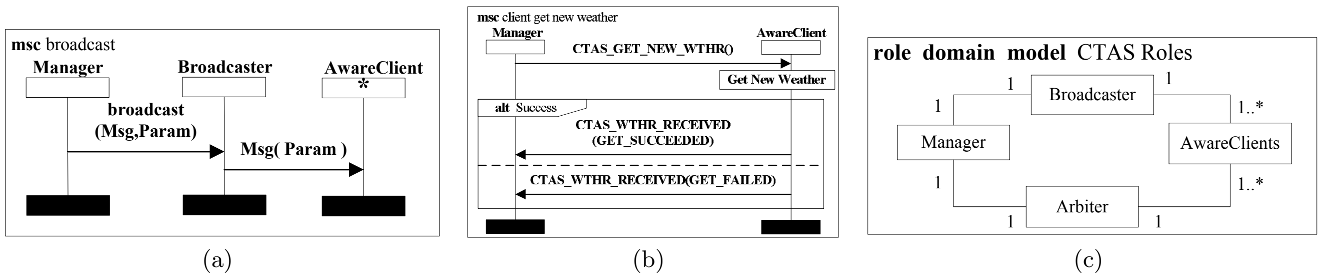


Figure 2: Services, and Role Domain Model

We specify the services of the CTAS weather update system using a notation based on Message Sequence Charts (MSC) [5, 9, 22]. An MSC defines the relevant sequences of *messages* (represented by labeled arrows) among the interacting *roles*. Roles are represented as vertical axes in our MSC notation. Fig. 2(a) and 2(b) show the specification of several services as interaction patterns.

In the following, we briefly describe a subset of the relevant services for the CTAS weather update cycle without giving their precise MSC specifications – the full set of service specifications is available at [14].

- **Broadcast:** The Manager commands the Broadcaster role to broadcast a message to all AwareClients. The Broadcaster relays this message to all AwareClients.
- **Client Get New Weather:** The Manager role notifies an AwareClient role that new weather information is available. The AwareClient reacts by performing a local action to get the new information. The outcome of this local action (success or failure) is submitted back to the Manager.
- **Client Use Weather:** The Manager role informs an AwareClient role whether the new weather information should be used subsequently, or whether the AwareClient should continue to use the old data.
- **Arbiter Collect:** The Arbiter role collects replies from all AwareClients and determines an overall response. If at least one of the AwareClients replies with a failure flag, the Manager will get a failure response; otherwise the Manager receives a successful response.
- **Check For Weather Update:** When the Manager role detects new weather information locally, it triggers the weather update process: it informs the AwareClients by using the Broadcast service.

System behavior can be specified in terms of the services listed above. Using high-level MSCs that reference and combine the separate system services allows to specify the full system behavior. We make use of powerful techniques to combine separate service specifications, for instance by sequential and parallel composition, and interleaving [9].

2.2 Architecture Definition

The next step after eliciting the services is to define a suitable component architecture onto which these services can be mapped. Our goal is to explore multiple such architectures for their adequacy in supporting the elicited services. For our example, we define four architectures that differ in the component configurations and the roles played by the components; see Fig. 3, rows 1–4.

Arch	CTASMgr	CTASBroadcaster	CTASArbiter	CTASClient
1	Manager, Arbiter, Broadcaster			AwareClient
2	Manager, Broadcaster		Arbiter	AwareClient
3	Manager, Arbiter	Broadcaster		AwareClient
4	Manager	Broadcaster	Arbiter	AwareClient
5	Manager, Forwarder	Broadcaster	Arbiter	AwareClient

Figure 3: Component Role Mappings

The table shows the roles played by the components in the various architectures. A blank cell means that the corresponding component does not exist for that architecture. CTASClient plays the role of an AwareClient in all architectures. CTASMgr also exists in all architectures, but the roles it plays differ. The components CTASArbiter and CTASBroadcaster are present depending on whether CTASMgr will play the role of Arbiter or Broadcaster, respectively.

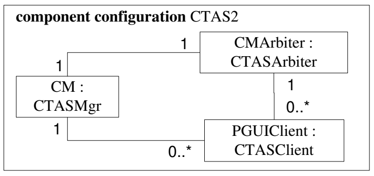


Figure 4: CTAS Architecture 2

To define an architecture we first capture the different types of components that occur in the system under consideration. For each component type we identify what roles it plays in the system. In Architecture 2 of Fig. 4, for instance, the CTASMgr component type plays two roles: Manager and Broadcaster. We also identify the component types CTASArbiter and CTASClient, playing the Arbiter and AwareClient role, respectively.

In this section we have shown how to specify a system in terms of services and roles, and how to model a system architecture with a concrete component configuration. Our models can be used to create executable prototypes for architecture exploration. In the next section, we provide a translation procedure using aspect-oriented programming techniques.

3. TRANSLATING ARCHITECTURES TO ASPECTS

We now show how to translate architecture definitions into aspect-oriented programs. The basic idea is that we translate the interaction patterns defining the services into aspects such that they can be weaved into any given component configuration using AspectJ's weaving capability. Using AspectJ, we are able to implement structure (roles), behavior (services) and their mapping to a specific component architecture separately and combine them very easily as needed for exploring multiple different architectures implementing the same functionality.

3.1 Aspect-Oriented Programming: AspectJ

AspectJ [7, 1] is a general-purpose aspect-oriented extension to the Java programming language; its language constructs facilitate clean modularization of crosscutting concerns. Commonly cited examples for crosscutting concerns are logging, tracing, error handling, synchronization and performance monitoring. AspectJ provides a compiler that weaves aspects at well-specified locations into Java classes. We make use of AspectJ to translate services into executable code. In the following, we briefly introduce the AspectJ concepts we will use in our translation procedure that is explained in detail in the subsequent sections.

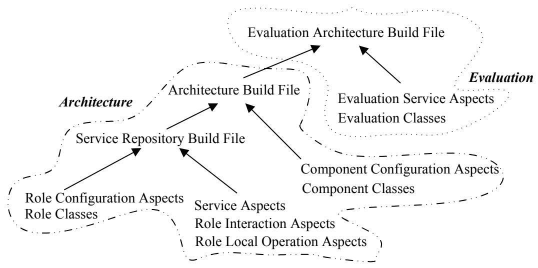
We translate services into executable code using AspectJ's *join points*, *pointcuts*, *advice*, *aspects*, and *intertype declarations* [1]. Examples of join points are method calls, method executions, object instantiations, constructor executions, field references and handler executions. Pointcuts are used for selecting these join points; an example of a pointcut is "all invocations of method xyz". Advice defines code that executes before, after or around a pointcut. An aspect can be the combination of a pointcut and the corresponding advice. In other words, using pointcuts, an aspect can specify at what points in the execution – or under what circumstances – a particular piece of code, represented as an advice, should be called. An intertype declaration, another form of aspect, can be used to specify a set of members (attributes, methods) that should be present in multiple classes. We use pointcuts and advice to translate patterns of interactions defining a service as an aspect; we use aspects describing intertype declarations to implement associations between roles and components.

3.2 Translation Process and Artifacts

The translation process has two phases (cf. Fig. 5, within the dashed boundary): (1) implementing a common service repository based on a set of identified roles, and (2) implementing multiple architectures for the service repository. In (1) we use a build file to weave together the following artifacts: classes for roles, aspects implementing the associations in the role domain model, aspects introducing the methods and local operations that each role needs to support, and aspects that implement the interaction pattern of each service. In (2) we weave together the output of (1) with classes for the components and aspects implementing the associations of the component configuration. These steps are explained, in detail, below.

3.3 Translating Roles

Define classes for roles: For each role appearing in a service definition we create a class. All specific role classes



Evaluation Architecture Build File
Evaluation
Architecture
Architecture Build File
Evaluation Service Aspects
Evaluation Classes
Service Repository Build File
Component Configuration Aspects
Component Classes
Role Configuration Aspects
Role Classes
Service Aspects
Role Interaction Aspects
Role Local Operation Aspects

Figure 5: Implementation Process Artifacts

are derived from a base "role" class, which has attributes representing the role's state and parent component name. The parent component will capture the name of the component for which the role instance is playing the role.

Define aspects for the role domain model: We capture the interconnection of roles as specified in the role domain model in form of attributes of the created role classes. Fig. 2(c) for instance indicates that the Manager role needs to have a reference to a Broadcaster role. Thus, a new attribute of type Broadcaster and corresponding accessor methods are introduced to the Manager role class with the help of an intertype declaration. All these relationships are captured in one aspect representing the configuration for the roles. Using an intertype declaration to weave the role associations into the role classes instead of directly adding this information there allows for much easier refactoring of the role model later. Role dependencies are only captured within the role configuration aspect.

3.4 Service Repository

For each service to be supported by the architecture, we follow the steps described below.

Define aspects for role interactions and local operations: For each possible interaction of a role we introduce a method for that role using an intertype declaration. For the service shown in Fig. 2(b), we introduce the method CTAS_GET_NEW_WTHR for the role AwareClient. We do this for all interactions of the service and for all local operations of a role.

```
public aspect ServiceClientGetNewWeather {

  pointcut Interaction1(AwareClient ac):
   target(ac) && (call(void CTAS_GET_NEW_WTHR()));

  after(AwareClient ac) :Interaction1(ac){
          ac.GetNewWeather();
  }

  pointcut LocalOperation1(AwareClient ac):
   target(ac) && (call(boolean GetNewWeather()));

  after(AwareClient ac) returning(boolean flag): LocalOperation1(ac)
 {
   ac.getManager().CTAS_WTHR_RECEIVED(ac,flag);
  }
}
```

Figure 6: Service to Aspect

Define aspects for services: So far we have defined classes for all roles and connected them as specified in the role domain model. We have provided methods within each role class for all possible role interactions and local actions. Now, we make use of these role classes and define each service as an aspect and use pointcuts and advice to connect interactions and local actions. A series of these definitions enables us to coordinate the interactions in the implementation. In essence, this corresponds to capturing a "global" state machine for each service in terms of an aspect definition. Consider, for instance, the service shown in Fig. 2(b). We have to define a pointcut for the receipt of a CTAS_GET_NEW_WTHR message by an AwareClient role and an advice for the pointcut just defined for executing the local operation of Get New Weather. The aspect defined for this service is shown in Fig. 6. This step projects the specified service behavior onto a specific role. The role implementation basically provides a state machine. The projection step can be automated by applying a synthesis algorithm, for instance the one described in [10].

Define build file for service repository: Based on the roles and services we created, we now define a build file that selects all classes and aspects for building a role implementation of the service repository. Note that at this point of time, there are still no concrete components involved. The build file for the service repository will be reused by multiple architecture configurations as we will show in the following sections.

3.5 Translating Components

We can now easily define multiple configurations for the system under consideration. We do this by creating different components based on the roles they play in an architecture.

Define classes for component types: We define one class for each component type in a specific configuration. We can reuse these classes across multiple different configurations that make use of the same component types.

Define aspects for each component configuration: We establish a specific mapping of roles to a component type by introducing attributes into the component type classes. Wo do this again with the help of intertype declarations. We define one aspect for each component configuration to reflect the roles the components play in that configuration.

3.6 Defining the Architecture

To finally establish a specific architecture, we create a build file that selects the build file for the service repository and the classes and aspects for a specific component configuration. As a consequence, we can create multiple configurations by defining multiple architecture build files which differ *only* in the classes and aspects selected for a component configuration. The code for the services remains unchanged.

In this section we have defined a procedure for translating a service-based system model into executable prototypes for specific architecture configurations. We used AspectJ because it nicely allows to decouple the common service definitions from the mapping to specific architectures. Fig. 7 depicts in an overview the translation of the different elements of the service model into classes and aspects. We will present experiences with applying our approach for architecture exploration in the next section.

4. EXPERIENCES

We have applied our approach extensively by evaluating

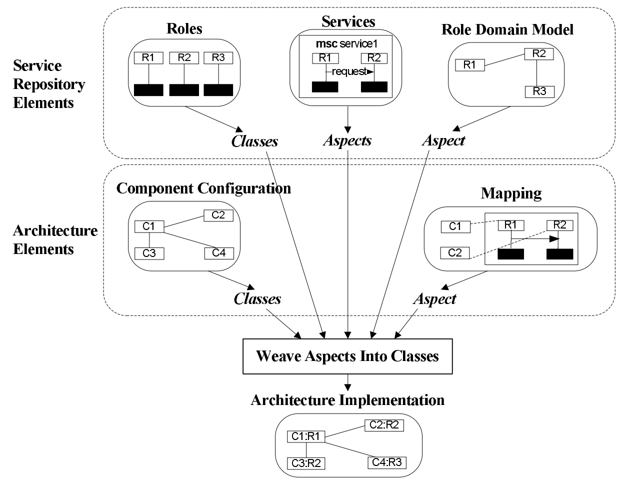


Figure 7: Service Model to Implementation

and optimizing different architectures for the CTAS case study, as documented in [13]. Our architecture exploration included several steps. First, we've evaluated the first four architecture alternatives shown in Fig. 3 by applying our scenario-to-aspects approach to create executable prototypes. We have defined new services to perform performance measurements of the running prototypes. Our approach was very effective in weaving these services as aspects into the existing functionality without changing any of the existing services.

The previous exploration lead to the assumption that a different role set, incorporating a Forwarder role between Manager and Broadcaster, is more effective in reducing the time required to perform a weather update cycle – a critical part of the system. We followed our process of Fig. 1 and iteratively evolved the architecture, role and service specifications. This was very simple and fast given using our service model and AspectJ implementation artifacts. This architecture variant turned out to be most efficient. We finally explored systematically multiple instantiations of this architecture with different numbers of Forwarder components. By measuring the performance of different prototypes we could optimize the component configuration to find the "sweet spot".

For performance evaluation, we measured absolute elapsed time as well as logical communication latency using the notion of logical clocks [12]. Evaluating absolute times as well as relative latency values allowed us to abstract from the used communication infrastructures. We could select architecture configurations that are optimized in terms of communications overhead and that perform similarly well in concrete deployments on specific messaging infrastructures. Details, statistics and performance charts are documented in [13].

The beforementioned findings show how our approach improves architecture exploration for product lines. First, it is easy to generate prototypes for different archtecture variants. Second, a selection of different services for different products of a product line is very easy to achieve with our approach. Certainly, the full thinkable genericity that might occur with software product lines is not addressed by our approach. Our approach requires certain fixed elements, such as the selection of roles or the definition of the services within the service repository. However, because we provide a high degree of decoupling between logical structure, behavior and architecture mapping, exploring relatively similar product line alternatives in many different configurations and environments is very cost and time-effective. The same is true for service repository refactorings and architecture reexploration. In this paper, we put the focus on architecture exploration of software product line alternatives. There are many more related problems and open research areas surrounding software product lines that we cannot address here, such as configuration management, requirements tracing and software evolution, to just name a few.

5. DISCUSSION AND RELATED WORK

Our approach is related to the Model-Driven Architecture (MDA) [15] and architecture-centric software development (ACD) [22]; similar to MDA and ACD we also separate the software architecture into abstract and concrete models. In constrast to these two, however, we consider services and their defining interaction patterns as first-class modeling elements of *both* the abstract and the concrete models. We consider services as aspects in the sense of AOP [8] at the modeling level, by focusing on cross-cutting interaction patterns. In Aspect-Oriented Modeling [4], the cross-cutting concerns are captured as *design* aspects, while our approach models these concerns as services. In the tradition of [23] the role concept is also adopted in [4] to define aspects abstractly; both of these approaches, however, lack the "join operator" we use to describe *overlapping* services sharing messages in an interaction pattern.

Often the notion of service-oriented architectures is identified with technical infrastructures for *implementing* services, including the popular web-services infrastructure [19]. Our work, in contrast, supports *finding* the services that can later be exposed either as web-services, or implemented as "internal" services of the system under consideration.

The approach we present here improves upon our earlier work in [11], by using pointcuts and advice to represent interaction patterns. This provides better decoupling between the roles and the interaction patterns they participate in as compared to the class-based approaches in [16] and [6].

6. CONCLUSIONS AND OUTLOOK

Thorough exploration of architectural alternatives is particularly important to support product lines for complex distributed and reactive systems. However, tight coupling between the domain logic and the implementation infrastructure, as well as prohibitive costs for building prototypes needed to evaluate *multiple* architectures often are stumbling blocks for architecture exploration.

In this paper we have shown how to define software architectures and explore architecture alternatives using the notion of services and their embodiment as aspects in AspectJ. We have decoupled the services provided by the system from the many target architectures that can implement the same set of services. We have introduced a translation process turning service-oriented architecture specifications into AspectJ aspects. This process exploits AspectJ's weaving capability to map service specifications to target architectures.

Resulting experience with our approach shows how easy it is to iteratively change a given architecture; only a subset of the service repository needed to be modified to fundamentally change an architecture for the CTAS case study. Performing changes and subsequent exploration was a matter of minutes in the given system.

Future work will include automating the translation from MSCs to aspects, and investigation of the relationship between our technique for architecture exploration with runtime verification techniques, such as [18].

7. ACKNOWLEDGMENTS

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). Further funds were provided by the Deutsche Forschungsgemeinschaft (DFG) within the project *InServe*. We are grateful to the anonymous reviewers for insightful comments.

8. REFERENCES

[1] AspectJ Team. The AspectJ programming guide. `http://eclipse.org/aspectj/`, 2004.

[2] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures-Methods and Case Studies*. Addison-Wesley, 2002.

[3] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Pub Co; 1st edition, 2003.

[4] R. France, G. Georg, and I. Ray. Supporting multi-dimensional separation of design concerns. In *OSD Workshop on AOM: Aspect-Oriented Modeling with UML*, 2003.

[5] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.

[6] E. Kendall. Aspect oriented programming for role models. *International Workshop on Aspect Oriented Programming at ECOOP*, 1999.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming, number 2072 in Lecture Notes in Computer Science, pages 327-353*, 2001.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. 1997.

[9] I. Krüger. Distributed system design with message sequence charts. *PhD Thesis, Technische Universität*, 2000.

[10] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[11] I. Krüger and R. Mathew. Systematic development and exploration of service-oriented software architectures. *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, 2004.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM, Vol. 27, No. 7, July 1978, pp. 558-565*, 1978.

[13] R. Mathew. Systematic definition, implementation and evaluation of service-oriented software

architectures. Master thesis, University of California, San Diego, 2004.

[14] R. Mathew and I. H. Krüger. Full service specification for CTAS system, 2004. http://sosa.ucsd.edu/publications/icse2005/CTASServiceSpecification.pdf.

[15] OMG Model Driven Architecture. http://www.omg.org/mda.

[16] B. Paech. A framework for interaction description with roles. Technical Report TUM-I0731, Technische Universität München, München, 1997.

[17] SCSEM 2003 Case Study. 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. CTAS Casestudy Overview, Requirements. http://www.doc.ic.ac.uk/~su2/SCESM/CS/requirements.pdf, 2002.

[18] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[19] J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[20] Software product lines. Software Engineering Institute. http://www.sei.cmu.edu/productlines/index.html.

[21] D. Trowbridge, U. Roxburgh, G. Hohpe, D. Manolescu, and E. Nadhan. Integration Patterns. Patterns & Practices. Available at http://www.microsoft.com.

[22] UML 2.0. http://www.omg.org/uml.

[23] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of OOPSLA'96*. ACM, 1996.