Automating Software Architecture Exploration with M2Aspects

Ingolf H. Krüger, Gunny Lee
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093-0404, USA
{ikrueger,gulee}@ucsd.edu

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

ABSTRACT

An important step in the development of large-scale distributed reactive systems is the design of effective system *architectures*. The early availability of *prototypes* facilitates the selection of the most effective architecture for a given situation. Although it is very beneficial to *evaluate and compare* architectures for functionality and quality attributes before implementing or changing the entire system, this step is often skipped due to the required time and effort. In this paper we present on the status of our *tool chain* to automate our approach of *efficient* prototype creation for scenario-based software specifications using aspect-oriented programming techniques [10]. It transforms interaction-based software specifications (scenarios) into AspectJ programs. Central part of this tool chain is *M2Aspects*, which implements the methodological transition from scenarios to aspect implementations. It also handles architectural configurations; M2Aspects maps of the same set of scenarios to different candidate architectures. This significantly reduces the effort required to explore architectural alternatives. We explain our tool-chain using the Center TRACON Automation System as a running example.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Algorithms, Design

Keywords

Scenarios, Services, Distributed Reactive Systems, Roles, Components, Software Architecture Exploration, Architecture Comparison, Aspects, Aspect-Oriented Programming, AspectJ

SCESM'06, May 27, 2006, Shanghai, China.

1. INTRODUCTION

1.1 Architecture Exploration

One of the difficulties in designing complex distributed systems is finding a suitable system architecture that supports the system's goals while being effective in fulfilling non-functional requirements, such as Quality-of-Service or performance concerns. Commonly, different candidate architectures need to be considered and compared to find the most fitting one. Architecture exploration needs to happen early in the development of a system, without delaying model refinement and implementation too much. Furthermore, lack of time and financial resources require effective prototyping and architecture exploration strategies.

Important questions to be addressed when designing architectures include, for instance, how to design the components and their interfaces, how to connect and distribute them i.e. how to design the communication topology, and how to replicate components for most efficient operation on a given middleware. Answering these questions requires consideration and exploration of different alternative architecture candidates. Building prototypes and running simulations complements and provides input for architecture evaluation techniques based on reviews and estimation [1].

Despite the importance of prototyping and architecture exploration, it often does not get the attention and resources that would be necessary. The reason often lies in the absence of effective prototyping and architecture exploration solutions.

1.2 From Scenarios to Prototypes

In this paper, we present an automated approach and tool solution to generate executable prototypes of reactive software systems; we use these prototypes to explore and evaluate architectures. One important feature of our approach is the possibility to generate prototypes for different candidate architectures based on the same specification model. This has a negligible overhead for different but comparable architectures and thus makes the exploration step very efficient and quick.

Our solution is based on software specification models in form of scenarios. Each scenario captures a different function or service of the system. Scenarios capture the interactions between independent interacting entities. We call these entities *roles*, because they fulfill certain logical functions in the system. The concept of capturing scenarios as interactions between roles allows us to abstract from target architectures.

Additionally, we capture target architectures composed out of independent *components*. These architectures realize the functionality that is specified within the scenarios and also take into account non-functional requirements and deployment infrastructures. Components have defined interfaces and behavioral models that are described by state machines.

In order to keep the logical scenario model and the target architecture model synchronized, we also capture the mapping in between. We map roles to components in the target architecture – we say a component of a certain type "implements" a number of roles. Doing so will result in a component that exposes an interface that is a combination of the different role interfaces it implements. Thereby, our architecture model is a true refinement of our scenario model. Fig. 1 depicts our approach.

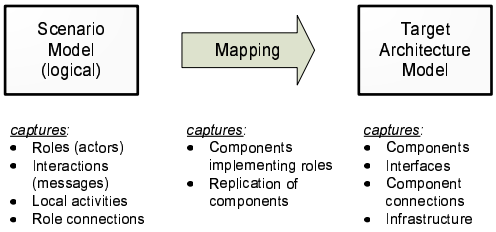


Figure 1: Scenarios and Architectures

For generating prototypes, we use aspect-oriented programming (AOP) languages and techniques [6]. We make use of the "separation of concerns" feature that characterizes AOP. Different concerns are separated into different *aspects*. Such aspects usually span multiple entities (objects, classes) in the source code. The task of an aspect weaver is to collect all the different aspects and weave them at the correct locations into the base source code.

We interpret scenarios as independent concerns and thus express them in the form of separate aspects. As mentioned, aspects need a base source could where they can be applied to (or woven into). Our base source code are classes for the roles and components in our model. The elegance of our approach is that the aspect weaver automatically handles the tedious task of combining all the different aspects with the role classes. The weaver creates implementations for each of the roles, which thereby form state machine implementations. Build files contain the information about the actual mapping of roles to components and thereby also determine the different architecture variants of the prototypes. Fig. 2 depicts the translation step.

The advantage of this approach over a generation of state machines (see [8, 12]) lies in the generation of aspect-oriented source code: Developers can freely modify the generated code, for instance by inserting logging and performance measurement statements. Different architectures can be explored by simply changing build files without regenerating the code. This keeps any modified source code intact.

1.3 Contributions and Outline

The contribution of this paper is the presentation of requirements and an architecture of tool support for the automation of the translation procedure from scenarios to aspect-oriented implementations of prototypes. In Sect. 2, we introduce the Center Tracon Automation System (CTAS) as our running example and show its scenario and architecture model. In Sect. 3, we explain the steps that are necessary to bridge the gap between model and implementation; we list requirements that we identified as important for our tool solution. We describe our tool solution and the current status of the implementation. We discuss it in Sect. 4 and show what is left to be done. In Sect. 5, we present related work. Sect. 6 contains conclusions and an outlook.

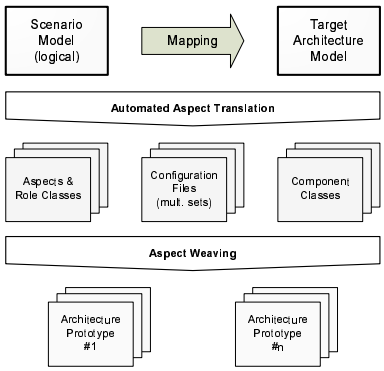


Figure 2: Translation to Aspects

This paper explains the automation of our scenario to aspects approach using our tool chain with the tool *M2Aspects*. The underlying methodology and process is explained in detail in our paper [10]. Please refer to [7, 11, 9, 14, 13] for more details and applications of our service-oriented model and its formal foundations.

2. CTAS SCENARIOS & ARCHITECTURES

To demonstrate our automated procedure to generate architecture exploration prototypes, we use the Center TRACON Automation System (CTAS), a case study from the air-traffic control domain, as an example of a large-scale distributed system [18]. 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 requirements [18] identify the communications manager CM as the main component of the CTAS weather update system. 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 [18] explain how the clients initialize with CM, and how CM subsequently relays the latest weather information to all aware clients.

For this paper, we model and explore various architectures and communication setups for the weather update functionality of the CTAS system as a refinement of the structure given in the CTAS requirements [18]. We model the relevant scenarios as MSCs, as described in [15, 10].

2.1 CTAS Scenarios as Interaction Specifications

Analyzing the requirements leads to a number of scenarios which share a number of roles. The roles that are relevant for our example are *AwareClient* (weather-aware clients),

Manager (drives the update process), *Broadcaster* (broadcasts messages to a group of clients) and *Arbiter* (collects responses from groups of clients).

We specify the scenarios of the CTAS weather update system using a notation based on Message Sequence Charts (MSC) [3, 7, 19]. 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. 3 shows the specification of scenarios as interaction patterns. The MSC syntax we use should be fairly self-explanatory, especially to readers familiar with UML2 [19]. In particular, we support labeled boxes in our MSCs indicating alternatives and conditional repetitions (bounded and unbounded loops). Labeled boxes *on* an axis indicate actions, such as local computations.

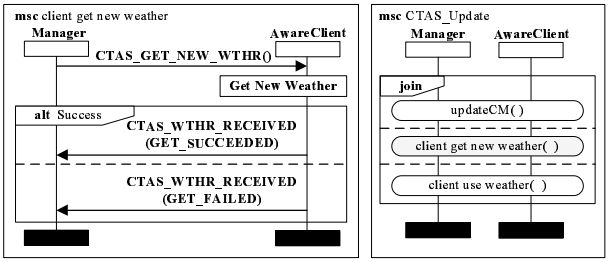


Figure 3: CTAS Scenarios as MSCs

We deviate from the standard MSCs in a few points, see [11, 13]. First, we represent *roles* rather than a class, object, or component with an axis in the MSC. As explained above, we perform the mapping of roles to components in a later stage in our development process, as an explicit design step. Second, we use an operator called *join* [7, 11] to easily compose *overlapping* scenarios. The join operator will *synchronize* two scenarios on their shared messages, but otherwise interleave the scenarios. Join is a powerful operator for scenario composition and isolation of common behavior.

For reasons of brevity, we omit the specification and description of all scenarios for the CTAS weather update cycle in this paper. The full set of specifications is available at [15].

2.2 Architecture Definition

The next step after collecting the scenarios is to define a suitable component architecture that supports them. The architecture must observe the dependencies of the roles, and take further constraints given by the requirements, for instance about technical infrastructures or non-functional properties, into account. Our goal is to explore multiple such architecture alternatives to find out the one that support the scenarios most optimally in the given situation. For our architecture explorations, we compare architectures with different component configurations. We compare alternatives where components differ by the roles they implement, by their replication and their mutual connection.

We specify component architectures precisely in an Architecture Definition Language – the Service-ADL – that we introduced in [13, 14]. It captures services, roles and components and builds the input for the automated creation of AspectJ architecture exploration prototypes. Fig. 4 shows a shortened example of such an architecture definition. We omitted the declaration and descriptive parts. In the diagram, we see three components implementing the four roles that we identified above. Component *CTASMgr* in this case actually implements two roles, namely *Manager* and *Broadcaster*.

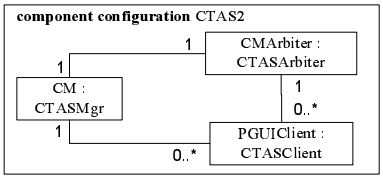


Figure 4: CTAS Architecture Candidate

2.3 From Architectures to Prototypes

In [10] we explained in detail our approach to translate scenario-based system specification into aspect-oriented programs [6]. Here, we only give a condensed summary.

We make use of AOP's separation of concerns capabilities on a programming language level and of the available tools to create executable programs. We translate the elements of our scenario-based model into programming language constructs and exploit the strong similarities in both model and implementation. There is, however, an unavoidable conceptual gap between a specification model and the implementation of a system, no matter how well both sides fit together. We explain some of the challenges and problems caused thereby in [10] and in Sect. 3, below.

We chose AspectJ [5, 20] as target language for our generated prototypes. AspectJ is a general-purpose aspect-oriented extension to the Java programming language; its language constructs facilitate clean modularization of separate concerns. AspectJ provides a compiler that weaves aspect code at well-specified locations into Java classes.

We translate our scenario model into executable code using AspectJ's *join points*, *pointcuts*, *advice*, *aspects*, and *intertype declarations* [20]. 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 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 that make up a scenario into an aspect and we use intertype declarations to implement associations between roles and components.

Fig. 5 gives an overview over the translation scheme. Roles, interaction scenarios (services), components, together with connections and mappings, are translated into their AspectJ counterparts according to [10]. Our algorithm creates a number of result files which are input for the AspectJ compiler and eventually will result in executable prototypes, see also Fig. 2. For performance evaluation, we measure absolute elapsed time as well as logical communication latency using the notion of logical clocks. Evaluating absolute times as well as relative latency values helps to abstract from the communication infrastructures used. We select architecture

configurations that are optimized in terms of communication overhead and that perform similarly well in concrete deployments on specific messaging infrastructures.

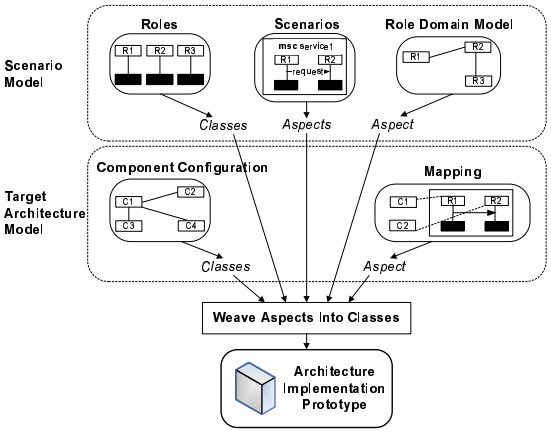


Figure 5: Translation artifacts

3. AUTOMATING ASPECT TRANSLATION

In this section, we will describe a tool chain we are developing for automatically generating customized AspectJ implementations from scenario specifications. First, we list the main requirements that we identified as important to support our methodological approach. We will then describe the tool chain and M2Aspects as a piece in it to generate executables from scenarios.

3.1 Requirements

We identified five main requirements for tool support to translate service-oriented specifications into executable AspectJ implementations, which we explain and discuss in detail afterwards:

(R1) Efficient modeling support
(R2) Flexible target architecture configuration
(R3) Directly executable generated code
(R4) Robust and flexible interfaces
(R5) Leverage AspectJ features

3.1.1 (R1) Efficient modeling support

The implementation architecture generated by the tool chain should efficiently and accurately reflect the model described in the specification. In particular, MSC operators such as *alt, loop, par,* and *join* should be accurately represented in the resulting Java implementation.

3.1.2 (R2) Flexible target architecture configuration

The tool chain should allow system designers to selectively specify architecture configurations to translate into implementation. Generating multiple prototypes of different architecture configurations this way opens the door for comparative performance evaluation. By constructing multiple concrete architectures, system designers can evaluate the performance of each of these architectures and choose one that best fit their needs. The primary aim for the tool chain is to allow this process to be relatively quick and simple.

3.1.3 (R3) Directly executable generated code

Requirement R2 leads directly into R3: the tool chain should automatically generate *directly executable* code. This requires the model to be deterministic and causal. Errors in consistency should therefore be checked and reported. It also requires precision at the specification level – message and variable names must be consistent throughout the specification, for instance. Unintentional inconsistencies are difficult if not impossible to detect, but obvious inconsistencies should be detected and alerted to the user. Lastly, where multiple implementation options arise due to ambiguity in the model or a lack of a clear-cut translation, the tool may choose one at default or at random (depending on the situation).

An alternative to generating directly executable code is to produce code skeletons, leaving out parts where multiple implementation options arise for the user to manually implement. Although this would increase flexibility in the translation, we opted not to choose this route due to the likelihood of requiring redundant manual effort and thus increasing turnaround time. For example, if the designer chooses to generate a number of different prototypes using different architecture configurations, it is likely that parts of these configurations will need the same ambiguity resolutions. We conjecture that it is more important for designers to be able to quickly produce a number of prototypes for comparative evaluation without being encumbered by such resolution efforts.

3.1.4 (R4) Robust and flexible interfaces

Robustness and flexibility are important concerns in our tool chain. Representation formats for scenarios and architecture configurations should be adequately expressive, while at the same time being manageable to systematically process. Also, parsing and processing these specifications should be abstracted away as much as possible from the rest of the tool chain in order to facilitate future changes to the specification format. This is crucial given the experimental stage of the tool chain. It is likely that modifications to specification format will be required as unforeseen requirements arise. The tool chain should be designed such that these changes do not cause a "ripple-effect" in the rest of the implementation of the tool chain.

3.1.5 (R5) Leverage AspectJ features

Since scenarios emerge as system-wide aspects in both the logical and implementation models, M2Aspects should leverage AspectJ features when generating implementation of services. The tool chain should also support generating AspectJ build files in order to support the composition of services with architecture configurations.

3.2 A tool chain for prototype generation

We designed a tool chain as a solution to fulfill the above listed requirements. We hereby make use of parts of the tool chain for generating executable RT CORBA components [12]. The starting point for both approaches is *M2Code*, a modeling tool for interaction specifications. M2Code uses Microsoft Visio as graphical front-end for editing MSCs and HMSCs to model service specifications. M2Code captures the modeled MSCs from Visio in form of an integrated interaction-based data model. M2Code saves this MSC model as

an XML file, which in turn is the input for the M2Aspects generator for AspectJ code.

M2Aspects is the component implementing the translation procedure. In addition to the scenario specifications in form of MSCs, it also requires the specification of target architecture configurations in form of Service-ADL files. Once invoked, M2Aspects creates all Java classes, aspects and AspectJ build files required for a compilable and runnable prototype. M2Aspects can be invoked for different architecture configurations using the same service repository, by using different architecture configuration files as input. This supports a comparative exploration of alternative component architectures that provide the same set of services to the environment. Fig. 6 shows the aspect generation tool chain and an overview over necessary input and created output artifacts.

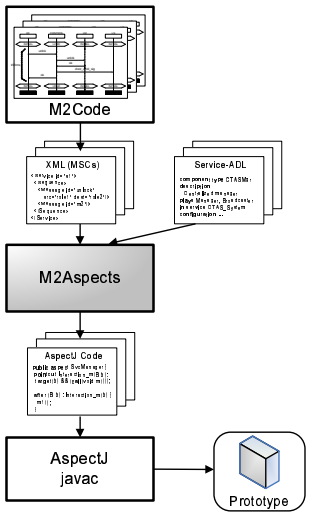


Figure 6: Aspect Generation Tool Chain

The results of M2Aspect's model translation, Java classes, aspect definitions and build files, are the input for the AspectJ compiler that produces the executable prototype by aspect weaving and compilation.

3.3 M2Aspects Architecture and Design

The functioning of M2Aspects can be described in five steps. In step 1, XML documents representing scenario specifications and architecture configurations are passed as input to M2Aspects. These XML documents are then parsed and translated into Java classes and interfaces in step 2 using the Java Architecture for XML Binding (JAXB) technology [4]. The architecture configuration handler (in step 3) processes these generated Java files and interfaces and generates the appropriate classes, aspects and build files to implement the architecture configuration. Finally, the Aspect Generator translates the scenarios into AspectJ implementations to be weaved into the architecture configuration to form a complete, runnable prototype. Fig. 7 shows an overview of M2Aspects, the necessary input files, and the generated output artifacts.

We designed and developed M2Aspects with the aforementioned requirements in mind. We now discuss how our implementation aims to address these requirements.

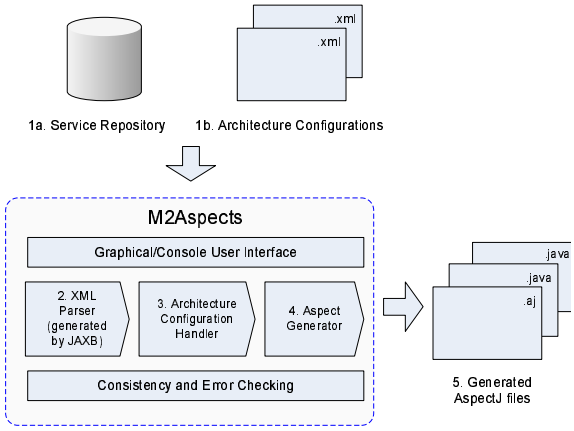


Figure 7: Overview of M2Aspects

3.3.1 *(R1) Efficient modeling support*

All variables and constants from the specification are assigned a type and stored in a symbol table. This allows the tool to check for type consistency as well as for proper use of variables. Message equivalence checking for the *join* operator is implemented using the unification algorithm [17], leveraging the type system to bind arguments to their correct types and check for compatibility. MSC operators are translated using native Java programming constructs such as *while, if,* and simple booleans. For example, the *alt* operator is implemented as an *if-else* block and the *loop* operator is implemented as a *while* block. The *join* operator, which is a little more complicated, uses boolean guards for synchronization of messages. See [10] for a detailed explanation on translating MSC operators to Java implementations.

One area where we make shortcuts in the translation of MSC operators is with the *par* operator. Although technically, the *par* operator indicates that its elements should be run in parallel, in most cases sequentially executing the elements is adequate and within the semantic correctness of the scenario. We therefore implement *par* this way, randomly choosing an order, and require the user to manually translate *par* in the case that true parallel execution is necessary.

3.3.2 *(R2) Flexible target architecture configuration*

The feature of selecting architecture configurations for code generation is achieved through the Architecture Configuration Handler component (shown in Fig. 7). This component allows system designers to selectively specify different architecture configurations to process and generate prototype implementations for. After specifying an architecture configuration, the Architecture Configuration Handler will generate all role classes, build files, and component classes necessary for the implementation of the architecture. The Aspect Generator component will then generate AspectJ code implementing the services to be weaved in to the generated architecture to form a complete runnable system.

3.3.3 *(R3) Directly executable generated code*

M2Aspects checks for both *syntactic* and *semantic* errors in the model at every step of its execution, from the parsing of service and architecture configuration specifications to the generation of implementation infrastructures. Examples of *syntactic* errors include incorrect XML encodings of specifi-

cations or using undefined variables and methods. Examples of *semantic* errors include joining services that do not have a common interaction or using the *alt* operator with only one element. All errors are logged and reported to the user

3.3.4 (R4) Robust and flexible interfaces

M2Aspects leverages JAXB to parse XML representations for service and architecture configuration specifications. A schema is first used to specify the format of XML documents encoding these specifications. The schema is then given as input to the JAXB binding compiler, which generates a JAXB binding framework consisting of a package of Java classes and interfaces that reflect the rules defined in the schema. XML documents encoding services and architecture configurations can then be unmarshalled into Java content trees using this framework.

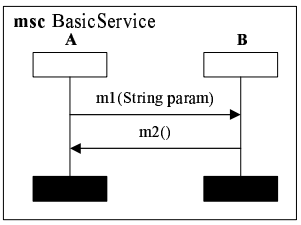
This approach has two key benefits. First, it allows the Architecture Configuration Handler and Aspect Generator components (shown in Fig. 7) to work with schema-compliant XML data represented as Java objects, achieving a clear separation of concerns from parsing the XML documents with the code generator logic. Second, it allows one to change the way service and architecture configuration specifications are represented in XML without modifying the parser, since the parser is automatically generated.

M2Aspects also employs symbol tables and scope systems to produce correct implementation code.

3.3.5 (R5) Leverage AspectJ features

M2Aspects implements scenarios in terms of an aspect with the help of pointcuts and advice. The next operation or interaction within the service is implemented as an advice for the pointcut defined for the current interaction. A series of these definitions allows for the coordination of the interactions in the implementation.

In Fig. 8, we show how an MSC with simple sequential ordering of interactions is translated to an aspect. We identify pointcuts for the occurrences of the method (interaction) and define advice for this pointcut. For this example, we define a pointcut called Interaction m1(B b) which captures the method call m1(String) for the targets of type role B. We define an after advice for this pointcut which executes after the method call. The advice defines the call of the method m2() of role A. Thus, the coordination of the interactions for this MSC is achieved with the help of the aspects defined using pointcuts and advice.




```
public aspect BasicService{

  pointcut Interaction_m1(B b):
    target(b) && (call( void m1(String) ));

  after (B b) : Interaction_m1(b) {
          A a = getARole();
          a.m2();
    }
}
```

Figure 8: Implementation of a basic interaction

4. STATUS AND EXPERIENCES

The current state of M2Aspects supports the generation of AspectJ code to implement scenarios, but not the generation of architecture configuration entities such as role and component classes and build files. These files currently require manual effort to create. The tool is executed in the Eclipse environment using an Ant build file. We are currently working on creating a user interface to make the tool more interactive.

The aspect generator supports the translation of all MSC operators including *alt, par, ref, loop* and *join*. It employs the composite pattern to support nested operators, such as an *alt* embedded in another *alt*. The translation of all operators has been tested using an automotive case study, and preliminary evaluation shows that it can handle most uses of MSC operators with an exception for the more sophisticated edge cases. A particular weak point with the tool at its current stage is the translation of the join operator. Although it can translate simple uses of *join*, more sophisticated uses such as nested *join*s are beyond what M2Aspects can currently support. Improving the *join* translation is also in our future work plan.

As an example of a generated service implementation, we ran M2Aspects on the CTAS *Client_Get_New_Weather* scenario shown previously in Fig. 3. The generated output for this service is shown in Fig. 9.

```
/*
 * Generated Aspect
 */
public aspect CTASGetNewWeather{

  pointcut Interaction_CTAS_Get_New_Weather()(AwareClient
       awareclient): target(awareclient) && (call( void
       AwareClient.CTAS_Get_New_Weather() ));

  after(AwareClient awareclient): Interaction_CTAS_Get_New_Weather()
      (awareclient) {
      awareclient.GetNewWeather();
  }

  pointcut Interaction_GetNewWeather(AwareClient awareclient):
      target(awareclient) && (call( void
      AwareClient.GetNewWeather() ));

  after(AwareClient awareclient): Interaction_GetNewWeather(
      awareclient) {
      if(success){
        Manager manager = awareclient.getManagerRole();
        manager.CTAS_Weather_Received(manager.getGet_sucess());
      }
      else {
        Manager manager = awareclient.getManagerRole();
        manager.CTAS_Weather_Received(manager.getGet_fail());
      }
  }
}
```

Figure 9: Translated CTAS Service

5. DISCUSSION AND RELATED WORK

In our approach, we separate a system architecture into logical and implementation models. The logical model contains the scenario definitions based on interactions. The implementation model contains the target component architecture. Thus, our approach is related to the Model-Driven Architecture (MDA) [16] and architecture-centric software development (ACD) [19]. In contrast to MDA and ACD, however, we consider scenarios (services) and their defining interaction patterns as first-class modeling elements of *both* the abstract and the concrete models. Furthermore, we see the implementation model as strict refinement of the logical model and require consistency of the mapping. Our models make use of MSCs as notation and are independent from any programming language constructs.

We see scenarios as aspects in the sense of AOP [6] at the modeling level, by focusing on inter-component interaction patterns. In Aspect-Oriented Modeling [2], the cross-cutting concerns are captured as *design* aspects, while our approach models these concerns as scenarios.

We systematically translate our partial interaction specifications in form of MSCs into AspectJ code and thereby create an aspect for each scenario. At first sight this indicates a major limitation of this approach: the final code emerges only from the weaving of all classes containing structural information and all aspects capturing role configurations, interactions, and local actions; the complete picture of the behaviors of each individual component is only contained in the resulting Java bytecode. Clearly, this prevents easy refinement on the *component* level. Recall, however, that we set out to develop our approach for efficient and effective *architecture* exploration, where a lot depends on the cross-cutting interaction rather than on the detailed, fine-grained behavior of individual components, which can be developed after settling on one component configuration. We see our approach most effective for exploring different choices on the architectural level; the scenarios as captured for the architecture exploration can, of course, still inform the implementation of the final deployment architecture.

6. CONCLUSIONS AND OUTLOOK

Thorough exploration of architectural alternatives is particularly important 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 presented a tool solution to automate our approach to define software architectures and explore architecture alternatives using interaction-based scenario models and their translation into AspectJ aspects. Scenarios are partial interaction specifications of systems. We have decoupled the system's scenarios from the many target architectures that can implement them. We presented an AspectJ code generation tool chain around the tool M2Aspects that implements the translation algorithm and generates executable prototypes. Providing automation and tool support is an important step in further evaluating our approach.

We presented on the current status of the implementation, listed requirements that influenced our solution and described some implementation challenges. Future work will include finishing automation of the architecture configuration, closing the gaps in the tool chain to eliminate any manual interaction, and providing translations for further and more expressive model elements and operators.

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

[2] 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.

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

[4] JAXB. `http://java.sun.com/webservices/jaxb/`.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer Verlag, 2001.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. Technical report, Xerox Corporation, 1997.

[7] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[8] 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.

[9] I. Krüger, R. Mathew, and M. Meisinger. From Scenarios to Aspects: Exploring Product Lines. In *Proceedings of the ICSE 2005 Workshop on Scenarios and State Machines (SCESM)*, 2005.

[10] I. Krüger, R. Mathew, and M. Meisinger. Efficient Exploration of Service-Oriented Architectures using Aspects. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[11] I. H. Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In M. Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer Verlag, 2003.

[12] I. H. Krüger, J. Ahluwalia, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, and S. Rittmann. Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.

[13] I. H. Krüger and R. Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 177–187. IEEE, 2004.

[14] R. Mathew. Systematic Definition, Implementation and Evaluation of Service-Oriented Software. Master's thesis, University of California, San Diego, 2004.

[15] R. Mathew and I. H. Krüger. Full Service Specification for CTAS System, 2006. `http://sosa.ucsd.edu/publications/icse2006/CTASServiceSpecification.pdf`.

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

[17] J. A. Robinson. Computational Logic: The Unification Computation. In *Proceedings of the 6th International Conference on Machine Intelligence*, 1971.

[18] SCSEM 2006 Case Study. 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. CTAS Case study Overview, Requirements, 2006. `http://ise.gmu.edu/scesm06/case-study-2/requirements.pdf`.

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

[20] Xerox Corp., Palo Alto Research Center Inc. The AspectJ Programming Guide. `http://www.eclipse.org/aspectj/doc/released/progguide`, 2004.