An Event View Model and DSL for Engineering an Event-based SOA Monitoring Infrastructure

Emmanuel Mulo Uwe Zdun Schahram Dustdar

Distributed Systems Group, Institute of Information Systems
Vienna University of Technology, Vienna, Austria
{ *lastname*} @ infosys.tuwien.ac.at

ABSTRACT

An event-based solution that uses events to convey information to a monitoring tool is well suited to implementing a non-intrusive monitoring infrastructure. This enables an SOA system's stakeholders to observe the system aspects of interest to them. However, implementation of SOA today, let alone the monitoring infrastructure, is a complex task due to the heterogeneous environment consisting of multiple technologies, platforms and components. We propose an approach for implementing such an event-based SOA monitoring infrastructure, that introduces a dedicated event view model and an eventing domain-specific language in a model-driven framework. The event view model captures SOA artifacts and links them with the event domain, while the eventing domain-specific language enables a system developer to specify instances of the event view model. With our model-driven approach, most of the runtime monitoring infrastructure is generated. These two ingredients (view model and domain-specific language) focus implementation efforts on the concern of eventing, thereby helping to manage complexity. We apply and evaluate our approach in the context of a case study.

Keywords

Event-based, service-oriented architecture, monitoring, domain-specific language, model-driven software development

1. INTRODUCTION

Service-oriented architecture (SOA) is defined as a paradigm or architectural style for organizing and utilizing distributed capabilities that may be under the control of different ownership domains [14]. Like any other large-scale distributed system, those implemented with the SOA paradigm require monitoring and management mechanisms [16, 11] that enable system developers, administrators, enterprise managers and other system stakeholders to constantly observe aspects of the system like its operating status, functional correctness and the business value generated. Such monitoring and management mechanisms should not, however, interfere with the functioning of the system. An *event-based* mode of interaction has been proposed as well-suited to implementing monitoring logic and components in a distributed system because system components that communicate with each other in an event-based fashion are inherently decoupled and communicate asynchronously [13, 2]. This implies that the monitoring and monitored components do not have to be dependent on each other. Moreover an asynchronous mode of communication allows for monitoring activity to be carried out as and when system resources are available. Therefore, engineering eventing concepts into a SOA shall enable the monitoring and management tasks.

However, a number of challenges in developing systems that interact in an event-based mode still exist, among them, the characterizing, specifying and implementing (the representation of) events in such a system [9, 5]. Moreover the specification of these events is targeted to different domains like intrusion detection, accounting, and performance, therefore the notion of an event has so many different meanings to different stakeholders within and across organizations. In the specific case of distributed event-based systems, their complex, heterogeneous architecture that comprises a collection of technologies, platforms, and components, adds to the challenge of developing such systems [3]. One type of distributed system, where these development challenges are clearly evident, is SOA implemented as Web services through use of Internet technologies and protocols.

We propose an approach for implementing the event-based monitoring infrastructure in an SOA. Our approach constitutes the application of a dedicated event view model and an eventing domain-specific language (DSL). A view model is a representation of the SOA system from the perspective of a related set of concerns [18], in this case the SOA artifacts and eventing concepts. The eventing DSL provides a tool for system developers to specify an instance of this event view model, that is, the eventing infrastructure for a particular system. Having these two ingredients enables us to use a model-driven approach to automatically generate most of the eventing infrastructure (code). With this approach, we believe that a system developer is able to better manage the complexity of implementing eventing infrastructure in an SOA, by focusing on the particular concerns of eventing. Among the resulting benefits, we expect that such an approach shall help in improving on the maintainabil-

DEBS '10, July 12–15, 2010, Cambridge, UK

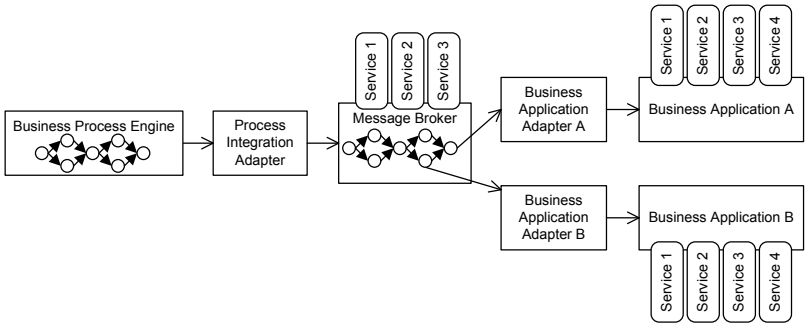

Figure 1: Illustrative small-scale SOA

ity, reusability and understandability of eventing concepts in SOA systems. We evaluate our approach in the context of a case study in the domain of advanced telecommunications services, in which we demonstrate the implementation of an eventing infrastructure for the purpose of monitoring an SOA.

In the rest of this paper, we give a background and scope to the type of SOA around which we design our approach (Section 2) and we discuss the development of such an SOA using views to manage complexity. We then present our approach (Section 3), followed by a case study to demonstrate our approach (Section 4). We discuss some lessons learnt from this exercise together with the related works (Section 5) and finally conclusions (Section 6).

2. VIEW-BASED DEVELOPMENT OF PROCESS-DRIVEN SOA SYSTEMS

SOAs are typically realized as layered architectures [8], comprising, for example, a communication layer that abstracts from platform and communication protocol details, and a remoting layer that provides the typical functionalities of distributed object frameworks, such as request handling, request adaptation, and invocation. The layers realize fundamental service orientation principles like loose coupling, abstraction, composability, autonomy and interoperability. A *process-driven* SOA, in addition to the above, comprises a service composition layer. This layer provides a process engine (or workflow engine) that executes business processes through invoking services to realize individual activities in the process. The main goal of such process-driven SOAs is to increase the productivity, efficiency, and flexibility of an organization via process management.

Figure 1 shows a small scale process-driven SOA as an illustrative example. A single business process engine is used, which uses service-based integration adapters to access a service-based message broker, e.g., offered by an Enterprise Service Bus (ESB) infrastructure. Service-based business application adapters are used to access backends, such as databases or legacy systems. The service adapter interface is hence used to unify the interfaces of different kinds of backends. A typical SOA in enterprise organizations today is much larger than this illustrative example. That is, multiple process engines – e.g., one per department – are deployed, in addition to multiple instances of other components. These are integrated using service-based interfaces, i.e., a process engine can invoke business processes (as sub-processes) in other engines via the service-oriented invocation interface of the engines.

When taking into account the different technologies and concepts required to realize such a SOA, the complexity of the entire system becomes evident. The View-based approach [18] tackles this complexity by separately addressing the different concerns in development of SOAs. Each concern is handled as a view, i.e., a (model) representation of the system from the perspective of a related set of concerns, thus focusing attention on one distinct set of concerns at a time. Four model (view) manipulation activities are identified in this approach: *definition/design* of a new architectural view on a system, *extension* of existing views with more elements, *integration* of multiple views to provide a richer view, and *transformation* of existing views in order to generate executable code.

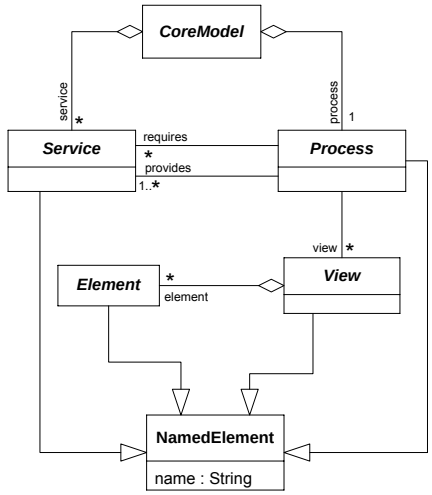

Figure 2: VbMF Core Model

For engineering process driven SOAs, the View-based Modeling Framework (VbMF) [17] defines and realizes some basic concerns (views) as: a *Flow* view to model the control flow of a business process, a *Collaboration* view that models how a business process and its partners (other business processes or services) interact, and an *Information* view that models data objects that are passed around during the flow of a business process. These views are related to each other through a core view, shown in Fig. 2, that defines basic constructs needed in such a view-based approach for

engineering process-driven SOAs.

The core view consists of the abstract classes `View`, `Process`, and `Service`. Each of these needs to be extended depending on the requirements of the view that one is designing. At the heart of the core model is the `View` class that captures the central view concept. An extension of this class represents a single perspective on a process-driven SOA. The `Service` class represents functionality that a corresponding `Process` provides or requires. A `View` is also a container for several `Element` instances, which represent objects that describe a process interior.

A view model for a specific concern of a business process is mostly defined by extending the concepts of the core model. As such, different view models are independent of each other but are related to each other through the core model. We capitalize on the concept of having separate views for different concerns, and in this work we focus on the concern of event-based monitoring. We propose an approach for designing and implementing the monitoring infrastructure for a process-driven SOA. We present details of our approach in Section 3.

3. SUPPORTING SOA MONITORING

3.1 Approach Overview

Our model-driven approach to support SOA monitoring is realized through an event view and an eventing DSL. The event view is a model that captures the concepts that, from our perspective, link an SOA environment and the eventing domain. This enables us to implement the event-based infrastructure to support monitoring in the SOA. Whereas this event view model captures these concepts in an abstract manner, we additionally develop an eventing DSL that provides a concrete syntax with which a system developer is able to specify instances of the event view model, i.e., the eventing infrastructure that supports the SOA monitoring.

In Figure 3, we illustrate our proposal to apply this approach to implement an eventing infrastructure in a SOA.

Before implementing any eventing infrastructure we need to *determine the information we are interested in monitoring*. The developer gets this information from the stakeholders or derives it from business goals. With this information we can use the eventing DSL to *specify an event view model instance*; this includes specifying event types and monitored artifacts along with their attributes. Concerning the event types, we specify those that capture and convey the information we would like to monitor, to a monitoring tool. Ultimately, our goal is to generate executable system components. Therefore, we need to *specify code generation templates*. These templates are specific to the technology of the monitored SOA artifacts. In order to define them, a developer would require some experience with implementing that particular artifact for which he/she wants to create an eventing infrastructure. The final step is run the *code generation* to create the necessary eventing infrastructure that would support monitoring of the SOA artifacts. We are thus able to deploy SOA artifacts (services, processes) that are augmented with eventing code to emit events to a monitoring tool.

In this overview we present these activities sequentially but in practice one probably performs a number of iterations (and in no fixed order) in order to develop a satisfactory solution. Moreover, the monitoring needs change or are upgraded regularly so the resulting eventing infrastructure is expected to also change to match these needs.

In terms of technical implementations, our event view model and the DSL are realized using Frag [19]. Frag is a dynamic programming language designed, among other things, to build (meta-)models and DSLs. These features imply that Frag could also be dubbed a language workbench. The language contains a modeling framework (libraries) that enables specifying (meta-)models that capture concepts in a domain. We use this modeling framework to specify our event view model. The Frag syntax can be tailored to build new language features or adapt existing ones to build a concrete DSL syntax based on an abstract model; this is how we realize our eventing DSL. Our eventing DSL is classified as an embedded one, that is, its syntax and underlying libraries are based on that of Frag, which is its host language.

The event view model and eventing DSL focus only on the concerns of eventing in the context of an SOA and in this way, isolate a system developer from having to deal with any other development concerns in parallel. Since we follow a model-driven approach, the model and DSL provide a basis from which we are able to generate an eventing infrastructure that would enable the SOA monitoring objective. In the next sections we provide more details about the model, DSL and generated eventing infrastructure.

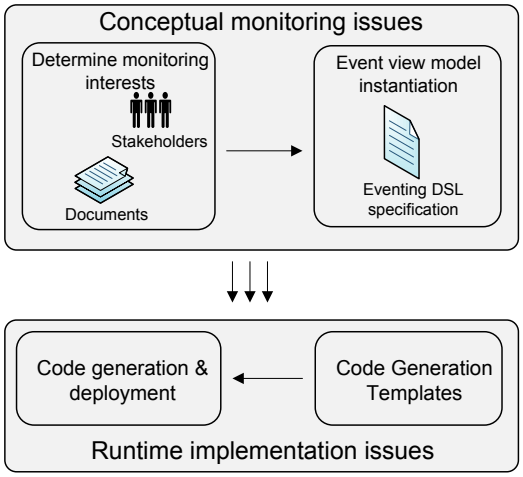

Figure 3: Approach Overview

3.2 Event View Model

The event view model captures and links the concepts of the SOA and eventing domains, in an abstract manner. Implementation of an SOA, requires one to have knowledge of many different technologies, platforms and components. However, the abstract concepts are relatively stable, therefore, representing them in an abstract manner enables us to specify instances of such systems (or components) that are not dependent on technology and can be easier adapted even with rapidly changing technology [10]. Our event view model is illustrated in Figure 4. The shaded classes indicate a relationship between our view and the VbMF framework discussed in Sect. 2.

The model elements are aggregated under the `EventViewModel` class. We define two main kinds of elements, `SOAArtifact` elements and `SOAEventType` elements. We can consider

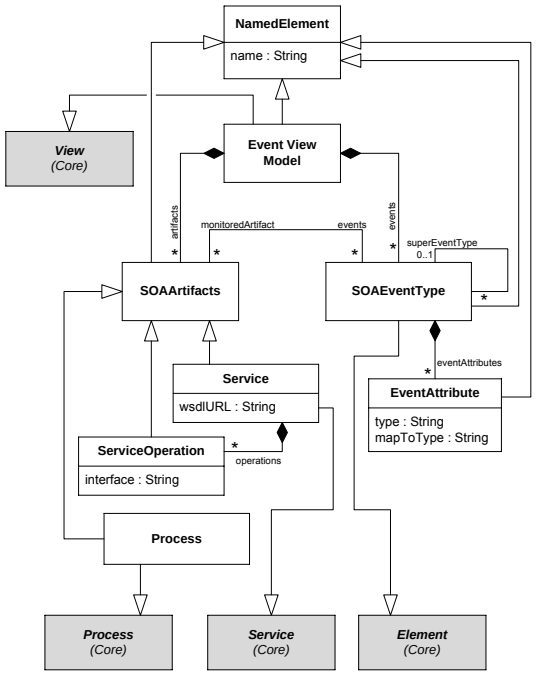

Figure 4: Event View Model

these two elements in light of their role in monitoring. The `SOAArtifacts` elements are the ones we are interested in observing, and the `SOAEventType` convey the information that we use to perform this observation.

`SOAArtifacts` are further specialized into `Service`, `ServiceOperation` and `Process` elements, which are found in the SOA domain. A `Service` element represents a web service for which we want to generate the eventing infrastructure. `Service` elements have a `wsdl` attribute that relates an instance of these elements to a WSDL specification (either a file or a URL). Having access to the WSDL specification presents the possibility to automatically generate eventing infrastructure, or to perform validation before the eventing infrastructure is generated. A `Service` element is composed of (contains) a number of `ServiceOperation` elements. Each `ServiceOperation` element specifies a distinct operation that is found within the interface (portType) definition of its containing service's WSDL specification. The `interface` attribute in each element instance should correspond to an existing and correct WSDL interface definition. `Process` elements represent a business process that orchestrates services to derive some business value.

The `SOAEventType` class provides a template for defining event types that enable monitoring in the SOA domain. Event type definitions are related to a particular artifact, say a `Service` element. Each type definition may comprise a number of `EventAttribute` elements, some of which carry data about the `SOAArtifact` that an event type definition is related to. The `EventAttribute` elements have either a `type` attribute or a `mapToType` attribute; the former defines a data type whereas the latter maps the attribute to a type in the monitored artifact. Event type specifications are setup in a hierarchical manner, that is, each event type may have a super (class) event type associated with it (through its `superEventType` attribute). Event type definitions (instances), therefore, inherit attributes from their super event type. Having this hierarchical setup enables a developer to extend event definitions and specialize them as needed, while still benefiting from the features of the super event types.

The event view model presented here captures the SOA domain elements in an abstract manner. It (the model) acts as a framework that we are able to incrementally extend with other concepts depending on the monitoring context. As an example, if we discover a special need to have process activities as a prominent feature in SOA monitoring, we can represent them in our event view model without disrupting our development activities. In the next section we present our eventing DSL.

3.3 Eventing DSL

Whereas the event view model captures the SOA and eventing concepts in an abstract manner, we now present an eventing DSL that provides a system developer with the ability to concretely specify eventing infrastructure to enable monitoring. This specification of the eventing infrastructure is actually an instantiation of the event view model. The DSL syntax can be modified depending on what the targeted technology is. We design our DSL with the assumption that a system developer using it is familiar with the basic concepts of web services based implementations of SOA and perhaps has some experience implementing them.

The DSL concrete syntax consists of the elements defined in the abstract model, that is, `Service, ServiceOperation, Process, SOAEventType, EventAttribute`. We use a sample of the DSL syntax, shown in Figure 5, to explain the DSL semantics.

```
Service create watchMeService\
       -wsdl "resources/watchMeService.wsdl"\
       -events ServiceInvocationEvent


ServiceOperation create searchContent\
       -name "SearchContent"\
       -interface "watchMeService"\
       -service watchMeService

SOAEventType create ServiceInvocationEvent\
       -name "ServiceInvocationEvent"\
       -superEventType ServiceEvent\
       -eventAttributes\
              [EventAttribute create\
                     -name "operation"\
                     -type "String" ]\
       -eventViewModel WatchMeEventViewModel
```

Figure 5: Eventing DSL

A `create` keyword, placed after an element, instantiates the element. An instance name follows to identify a particular instance. The attributes or relationships defined for an element are expressed like switches in a command line syntax.

In order to define the eventing infrastructure we need to make the definitions in two parts. In the first we specify the instances of elements we are interested in observing. In Figure 5, we create `Service` and `ServiceOperation` instances.

The `watchMeService` instance also identifies the WSDL specification file. The service instance name (`watchMeService`) has to correspond to an existing service definition in the WSDL; the same goes for the `ServiceOperation` instance. Each `SOAArtifact` is able to define a relationship to a set of events through the `events` switch. This means that the event infrastructure that shall be generated for that artifact emits that type of event; so in Figure 5 the watchMeService event infrastructure emits `ServiceInvocationEvent` types.

After defining the elements we are interested in observing, we define a series of `SOAEventType` elements that are related to each monitored element through the `monitoredArtifact` switch. Each event type definition has the possibility to extend the definition of a another event type using the `superEventType` switch. The event type definition inherits all the attributes from the super event type and can, in addition, define extra attributes that one is interested in observing. In Figure 6, we model a hierarchy of events that we propose for monitoring a SOA. We have generic `SOAEvent` event down to specific events like `ProcessActivityStartEvent`. This model demonstrates the possibility for a user to extend the event type definitions to create a more specific event that monitors attributes not present in the more generic version of the event. For example, we can see that the `SOAEvent` has attributes name, type, and timestamp, which are expected to be present in all event types, whereas the `ProcessInstanceEvent` type is a new type definition with an attribute `instanceID` that is specific only to process events. The `ProcessInstanceEvent` inherits all the attributes from the former in addition to defining its own attribute. Therefore, `ProcessInstanceEvent` instances would enable monitoring of processes including instance level identification information. Note that Figure 6 is actually a graphical representation of the instantiation of parts of our event view model. The specification of this and other event view model instances is done using the eventing DSL.

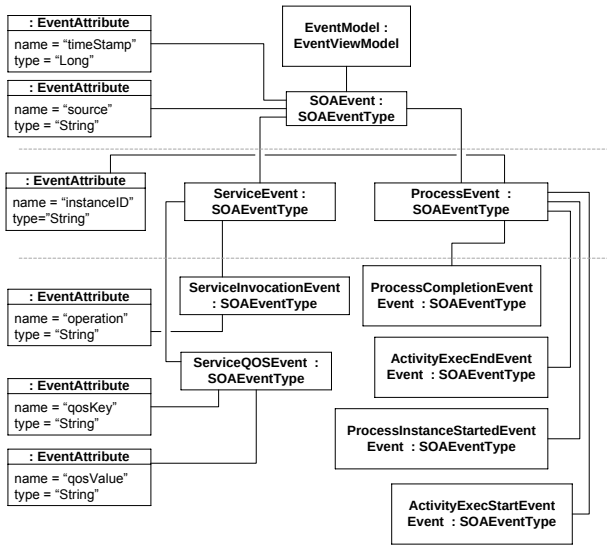

Figure 6: Event View Model Instantiation

As the eventing DSL is embedded into the Frag language, any extensions to the event view model can be quickly reflected in the eventing DSL through the modeling framework from Frag [19]. Hence, we would not require extra effort to extend tools like parsers to understand the DSL extensions.

Ultimately, we would like to generate the eventing infrastructure code. In the next section we discuss how this is achieved in a web services based SOA implementation.

3.4 Eventing in Web Services based SOA

In this section we describe how the event view model and the eventing DSL are applied, using a model-driven approach, to generate the eventing infrastructure that enables monitoring. In order to generate the eventing infrastructure we need to define, in addition to the event view model and the eventing DSL, the necessary code generation templates that realize this infrastructure. The code generation templates address different aspects of the process driven SOA that we wish to monitor, specifically, service invocations, and process executions.

A service invocation, in the case of SOAP based web services, constitutes processing a SOAP message that contains all the necessary information regarding, for example, the service operation invoked and the parameters passed. Web service frameworks like Apache Axis2[1] and Apache CXF[2] realize this SOAP processing model through a number of processing steps known as *Handlers* (Axis2) or *Interceptors* (CXF), each handler performing a specific processing action on a SOAP message. It is possible to extend this processing model with a customized processing step (handler). We realize our service invocation eventing infrastructure by generating code for a customized eventing handler embedded in the web services framework.

Workflow engines like ApacheODE[3], jBPM[4] and Windows Workflow Foundation[5] already emit events concerning the different occurrences of interest during process executions. They also provide the possibility to create a customized event handling component which allows for handling events that we might be interested in monitoring. Therefore, as far as such workflow engines, we realize the eventing infrastructure through generating a customized event handler. In both the case of service invocations and process executions, the role of generated code is to extract information relevant for an external monitoring tool, and forward this information to such a tool.

Since we are dealing with implementations of different technologies, we assume that a developer using our approach is familiar with implementing and deploying web services. This knowledge would be a prerequisite in order to create the code generation templates we have discussed in this section. For example, for our situation where we implement the technologies described, we propose to generate an eventing infrastructure as illustrated in Figure 7. The `EventingInInterceptor` and the `EventListener` both implement a AbstractEventMonitor interface. The `sendEvent()` method, when implemented, sends the monitoring events to a monitoring tool.

In Section 4, we present our case study and provide more detailed scenarios of how we realize the eventing infrastructure. In our scenarios, we use the Apache CXF web services

[1]http://ws.apache.org/axis2/
[2]http://cxf.apache.org/
[3]http://ode.apache.org
[4]http://www.jboss.org/jbpm
[5]http://msdn.microsoft.com/en-us/netframework/aa663328.aspx

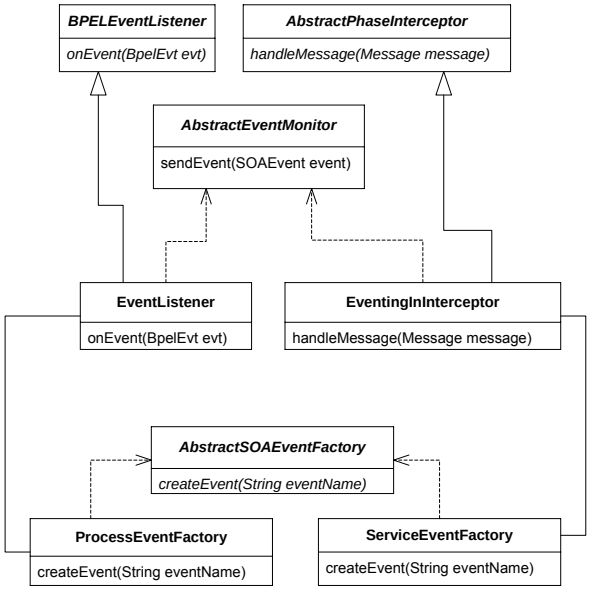

Figure 7: Eventing Infrastructure Class Diagram

framework and the ApacheODE workflow engine.

4. CASE STUDY

In this section, we apply our approach in the context of a case study. Our case study addresses development of an eventing infrastructure that enables monitoring of a process-driven SOA. The SOA infrastructure for our case realizes a business process, the multimedia streaming business process, that is a scenario of advanced telecom services offered by a Mobile Virtual Network Operator (MVNO). MVNOs provide value-added services to users by accessing and aggregating various facilities from other content providers. In our scenario, multimedia streams from third-party providers are offered to mobile phone users through a web service, the WatchMe service. Through this service, a user is able to perform searches for multimedia content they are interested in. The user's access rights to different media vary based on licensing options (pay per use or time-based) they choose when paying for stream access. After paying, a user has access to a media stream for as long as their access rights are in accordance with the license selected. The multimedia streaming business process is modeled in Figure 8. The detailed flow of the business process is as follows;

1. The user contacts the WatchMe service to access a media stream of interest.
2. The user submits search criteria to the WatchMe service, which in turn contacts providers with whom the MVNO has agreements – the service searches for streams matching the user's search criteria.
3. Search results from different providers are aggregated and provided to the user.
4. The user selects a stream and is provided with more details, e.g., a description and a preview, costs/price and relevant license plan of accessing the stream.

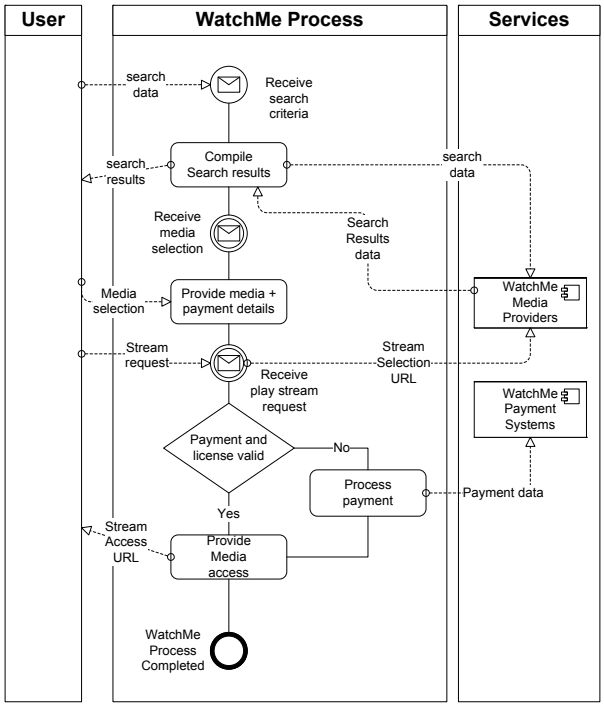

Figure 8: Multimedia stream access business process

5. The user selects a license plan and confirms payment.
6. Once payment is confirmed, the stream is available to the user. Whenever the user accesses the stream, the WatchMe service provides an access URL from a particular provider.
7. Each time the user accesses the stream, the combination of the user identification and the stream URL is checked to ensure that license agreements are not violated.

In the scenario, an MVNO is interested in monitoring the multimedia streaming process to identify the kind of content users are searching for and accessing, adherence to the user licenses, as well as detection (and reporting) of violations and anomalous events. The test implementation for the business process is known as the WatchMe service. The service provides an interface to the workflow engine that executes the business process; it exposes two operations, `searchContent` and `streamContent`.

With our approach we create an eventing infrastructure to support monitoring of the WatchMe service. Our eventing infrastructure plays the role of capturing the data that the MVNO is interested in monitoring, viz. the types of user searches that are invoked on the service, the content they select for viewing, and the details concerning the process execution when a search is performed, or when content is accessed for viewing. After capturing this data our infrastructure it to a monitoring tool that would perform the necessary processing to identify events of interest. In the following sections we describe how we realize this monitoring infrastructure with our event view model and eventing DSL.

4.1 Event Infrastructure for Services

In the first part of the scenario we would like to realize the eventing infrastructure for individual web services, in this case the WatchMe service. For our approach we assume that all the necessary service business logic has already been implemented. We are only interested in generating the eventing infrastructure to support monitoring. The infrastructure emits events at runtime, containing information conveyed to a monitoring tool. From the use case scenario, we have identified user searches, and accesses to stream content as candidates for monitoring. Therefore, we monitor the operations `searchContent` and `streamContent` every time they are invoked on the WatchMeService.

Using the eventing DSL, we specify services, service operations, and event types (cf. Fig. 5). Only services whose operations are to be monitored are specified. The relationships between the different events and services is implicit through service operations. For each event attribute we map the attribute to a type in the service description where possible. At the very least, we should be able to derive this information from an existing concrete type. From the hierarchy of events presented in Figure 6, we have a `ServiceInvocationEvent` type that has an `operation` attribute. This particular attribute represents name of the operation to be monitored and is represented here simply as a string.

Besides expressing the desired event infrastructure in the DSL, we also need to create the code generation templates for the event emitting code. Defining these templates would require a developer to have some experience with implementing and deploying web services. The developer would have to make an exact choice concerning how he/she would like to implement and emit the events. Typically events are implemented as an immutable object, i.e., capturing state of the monitored component at a point in time. Therefore, the template for generating events would require that the values for all the monitored attributes are initialised at the time the event is created. As far as emitting events in the case of services that we are monitoring, one of the possible solutions, that we chose for our scenario, is to generate an interceptor that extends handler chains as discussed in section 3.4. In Figure 7 we already showed the structure for the event infrastructure that we generate.

We illustrate the code generation templates for our event factories in Figure 9. The template generates an event creation factory based on the events specified. The interceptor code shown in Figure 10 requests a factory for the event based on the operation that was invoked, and then sends the event to a monitoring tool.

We run the code generation to generate the necessary eventing infrastructure that would support monitoring of services. Ideally, the generated code should be integrated with the rest of the services infrastructure during development time, such that the service code can simply be deployed after generating and compiling all the sources including the monitoring infrastructure. However, we do not do this at the moment and so we manually deploy the monitoring code for the services with the event-based monitoring infrastructure.

4.2 Event Infrastructure for Processes

In the second part of our scenario, we realize the eventing infrastructure that shall enable observing of events related to a business process execution. Again we assume that the process execution logic is implemented separately. We only

```
public class <~ $factoryName ~>Factory implements
SOAEventFactory {
 public enum EventType {
       <~ self applyForeach event $events {
             <~ $event name ~>Event,
       }
       ~>
 }

 public SOAEvent createEvent(String qName) {
       EventType eventType = getEventType(qName);

       switch(eventType){

       <~ self applyForeach event $events {
             case <~ $event name ~>Event :
               return new <~ $event name ~>Event();
       }
       ~>

       }
 }

 public EventType getEventType(String qName){
       EventType type = EventType.ServiceInvocationEvent;

       <~ self applyForeach event $events {
             if(qName.equals("<~ $event name ~>"))
               type = EventType.<~ $event name ~>Event;
       }
       ~>
       return type;
 }
}
```

Figure 9: Event Factory Code Generation Templates

```
public class EventingInInterceptor extends
AbstractPhaseInterceptor<Message> implements
AbstractEventMonitor {

 public EventingInInterceptor() {
       super(Phase.USER_LOGICAL);
 }

 public void handleMessage(Message message)
throws Fault {
   QName name = (QName)
       message.get(Message.WSDL_OPERATION);
   SOAEvent event = (new
       ServiceEventFactory()).
        createEvent(name.getLocalPart());
   this.sendEvent(event);
 }

 public void sendEvent(SOAEvent event) {}

}
```

Figure 10: Eventing Interceptor

aim to generate code related to the monitoring. In our scenario, we use the sample implementation for the WatchMe search content business process, a sub-process from our multimedia streaming business process. This process consists of the three steps involved in searching for user content, i.e., verify the user identify, contact the service providers and search for the content requested, and lastly provide the user with feedback.

The implementation of the event infrastructure for workflow execution engines is considerably different from how we implement the services' event infrastructure, in that they (engines) typically have a facility to emit events, and moreover they already have a number of predefined event types. In the case of the event infrastructure for processes, therefore, we specify which of the engine's predefined events we are interested in transmitting to a monitoring tool and what information we wish to extract from these events.

```
Process create searchContentProcess\
       -events [list build\
              ProcessInstanceStartedEvent\
              ActivityExecStartEvent\
              ProcessCompletionEvent ]

SOAEventType create ProcessInstanceStartedEvent\
       -name "ProcessInstanceStartedEvent"\
       -superEventType ProcessInstanceEvent\
       -eventViewModel WatchMeEventViewModel

SOAEventType create ActivityExecStartEvent\
       -name "ActivityExecStartEvent"\
       -superEventType ProcessInstanceEvent\
       -eventViewModel WatchMeEventViewModel

SOAEventType create ProcessCompletionEvent\
       -name "ProcessCompletionEvent"\
       -superEventType ProcessInstanceEvent\
       -eventViewModel WatchMeEventViewModel
```

Figure 11: Process Eventing Specification

We use the eventing DSL to specify the processes and the process related event types for the content search business process. The attributes that we specify in the process related event types should exist in the event types that are offered by the workflow engine. For our implementation we use the Apache ODE workflow engine. We show the specification of the processes and process related event types in Figure 11.

For the code generation, we take as a base the `BPELEventListener` from the ApacheODE implementation. This class is designed to be extended for customized event processing duties. The code for the customized event listener is shown in Figure 12.

4.3 Extending the Event Infrastructure

In the last part of our scenario, we consider a situation where we already implemented our event infrastructure and discover the need to monitor new information; for example, we might want to observe the search terms that users submit whenever searching for media content from our WatchMe service. In order to do this we need to extend our existing event view model *instance* using the eventing DSL, to reflect this new monitoring information. We would need to identify the source of this information in order to make a decision

```
public class EventingListener implements
BpelEventListener, AbstractEventMonitor{

 public void onEvent(BpelEvent bpelEvt) {
  String name =
       bpelEvt.getType().getClass().
       toString();

  SOAEvent event = (new
       ProcessEventFactory()).
        createEvent(name);
  this.sendEvent(event);
 }

 public void sendEvent(SOAEvent event) {}
}
```

Figure 12: Process Eventing Specification

on the extensions to make to the event view model.

From our WatchMe service, we observe that we can extract this information (detailed user search terms) from the `searchContent` operation, therefore, we create a more specialised event (extend our model instance) that is specifically dedicated to capturing search terms. In our event view model instance shown in Figure 6, the most generic event is the `SOAEvent`. Therefore, in the case where we are creating a more specialised event it should at the very least have the `SOAEvent` type as its parent. For this case, however, we are interested in monitoring a particular aspect (search terms) of a specific (`searchContent`) service invocation, so we create a specialised event type definition `SearchContentEvent` that has as its super event type the `ServiceInvocationEvent`.

The `SearchContentEvent` type captures information that could not be previously observed by the existing monitoring infrastructure, therefore, we need to specify its attributes that shall capture that information. We define an attribute, `searchTerms`, which stores the information we want to monitor. We illustrate the extension of the event view model instance, using our eventing DSL, in Figure 13.

```
SOAEventType create SearchContentEvent\
       -name "SearchContentEvent"\
       -superEventType ServiceInvocationEvent\
       -monitoredArtifact searchContent\
       -eventAttributes\
              [ EventAttribute create\
                     -name "searchTerms"\
                     -type "String"]\
       -eventViewModel WatchMeEventViewModel
```

Figure 13: DSL Event Specifications for `SearchContentEvent`

Once we have extended the model instance, we need to update our code generation templates. For example, we may need to update the templates to reflect how the new information to be monitored shall be extracted from the monitored artifacts and inserted into the event. Finally we regenerate and compile the executable code.

5. DISCUSSION AND RELATED WORK

We argue in our paper that implementing an event-based monitoring infrastructure in an SOA is a difficult task with challenges like handling the complexity of a typical SOA implementation that requires knowledge of a number of technologies, platforms and components. In addition development of an event-based monitoring component faces a number of challenges including specification and implementation of the representation of events. In this section we present a discussion concerning some of the advantages and limitations of our approach while attempting to address these challenges. We also present related works.

5.1 Discussion

In terms of complexity while developing an SOA, there are a number of issues that arise that we believe are addressed by our approach. In the first instance, during development of an SOA, there are a number of requirements, functional and non-functional, that need to be addressed. By using a *focused development approach* that addresses a single related set of concerns at a time, we are better able to manage that complexity of implementing the monitoring infrastructure. With projects where deadline pressure results in focusing on the core functional requirements, an approach that eases the development of non-core requirements like monitoring could help ensure that these requirements are also fulfilled.

For the case of an SOA implementation, multiple technologies, platforms and components are required. If we consider a process-driven SOA that we address in our work, we have web service frameworks, web service interface specifications, workflow execution engines, configuration files, legacy applications, and middleware components to contend with. However, we have to implement a monitoring infrastructure for the *entire* system. Having an *abstract, technology-agnostic specification* of the eventing infrastructure frees us from dealing with details of the different technologies while specifying our eventing infrastructure. We are able to focus attention on the eventing concepts without thinking about the different implementation technologies.

We see that the specification of the event infrastructure using our approach enables *integration of development efforts and assets for different kinds of event-based system implementations*. Consider the case where we had to implement infrastructure for a process versus for a service. In the case of processes, we found that the workflow execution engines already provide the facilities to emit events. In this case, instead of emitting the events, our infrastructure forwards the events to the monitoring tool. However, can also perform a simple type-based filtering and selection of only the events we are interested in. Although this is also possible using the configuration of the workflow engine. Another use that we did not implement could be to (re)generate the parts of configuration files that pertain to events.

Having an extensible model for our eventing infrastructure implies that we are able to *incrementally implement the eventing concepts* within the SOA. As we specify the necessary concepts in an abstract manner, we are not obliged to immediately instantiate them or generate code for them, therefore, we can iteratively represent concepts and generate their code. This also fits naturally into the nature of a monitoring tool where monitoring needs evolve and new requirements arise all the time.

In terms of the resulting runtime components, an advantage in this approach is that the executable code can be (re)generated. This implies that for a large scale system, there are *productivity and efficiency gains*, in terms of writing and reusability of code, as more and more components are added, because the code generation templates might not need to be changed a lot. Such an approach can be thought of as a step in the direction of generalizing and reusing event-based components in different systems.

Regarding some of the drawbacks that we can identify, the first issue would be concerning the fact that the developer has to *learn the syntax of such an eventing DSL*. We feel that a developer would not have such great difficulty because the concepts are kept to a minimum. However, we make the assumption that the developer using our approach has some experience in implementing and deploying web services and workflow engines. This knowledge would be needed in order to define the code generation templates for creating the runtime code. Nevertheless, aside from the code-generation templates, we believe a less experienced developer would be able to specify, generate and deploy the eventing infrastructure because of the small number of concepts he/she would need to learn. The developer would need to understand the syntax statements that define event types, their attributes and their relationships to SOA artifacts. In addition we have to spend some effort on the maintenance of the eventing DSL.

In Sect. 3.4, we refer to the execution model of services, comprising handler chains. We propose extending these handler chains, in order to intercept data and emit it as an event to our monitors. In this work, we considered the case where we are monitoring service invocations. However, a monitoring solution may be required to concurrently monitor many more concerns, e.g., quality of service, license agreements, and network performance. Using our approach would require that we emit an event for each of these concerns, or provide an option where a single event encapsulates multiple events representing the different concerns. Either way, *the amount of processing at the point where events are emitted could increase with the number of monitoring concerns.* Additionally, as far as the proposed approach to intercept events from process execution engines, we are limited by how the event emission facilities of the process execution engine are implemented. If the engine does not incorporate a mechanism to extend its event model, one would have to do (possibly a lot of) adaptation work to emit events / data that is relevant for one's monitoring concerns. We are not sure how these issues could affect the performance of such solutions or the feasibility of this approach.

Although event-based systems evaluations typically consist of a runtime aspect, in this particular case, we do not do any runtime evaluation because it is difficult to prove that the code that we generate is the most optimal implementation of an event-based monitoring architecture. We do, however, believe that our approach allows for the generation of whatever code could be considered optimal and therefore, the efficiencies gained in terms of reusability of the code generation templates for system development and extension is in itself a bonus.

5.2 Related Work

We now discuss some related works and contrast them with our approach.

The WS-Eventing [1] and WS-BaseNotification [15] stan-

dards address issues concerning event-based interaction in web services based SOA implementations. WS-Eventing defines a protocol that enables a (sink) web service to subscribe to another (source) web service such that the former is informed of any particular events of interest that occur in the latter. This specification, however, focuses on the subscription mechanism. The event-based infrastructure we propose to generate can be said to create SOA components that fulfill the roles of *NotificationProducer* in the WS-BaseNotification standard.

Fenkam et al. [7] discuss some requirements for a methodology for developing correct event-based systems. Among the general requirements are that the methodology should be scalable to large systems and should quickly produce high-quality, maintainable software at a low cost. In our discussion we do not address the scalability of our approach, however, we believe that having automated code generation does address the cost, quality and time issues. Delgado et al. [4] perform a survey of up to 30 runtime software fault monitoring tools. They present a taxonomy and catalog of these tools. Included in the taxonomy, is the concept of a *specification language* that is used to define monitoring directives. For the majority of the tools, the specification language did not provide constructs to support expression of domain-based system properties in the monitoring tool. The authors argue that as systems become more complex, this ability to capture domain-based properties becomes more critical. Although our DSL falls in the category of technical DSLs [10], it does capture domain-based knowledge for a developer who works in a technical domain.

Edwards et al. [6], Chowdhary et al. [3], and more recent work like Momm et al. [12] propose a model-driven approach in developing SOA related components. Edwards et al. [6] use a model driven approach to automate middleware service configuration and deployment tasks. They create a language EQAL that models configurations for CORBA–based publish/subscribe services. They are then able to validate instances of the models and generate the corresponding configuration code. They also generate metadata necessary for service deployment. In our approach, we model and generate the eventing infrastructure that transmits data to a monitoring tool. Chowdhary et al. [3] present a model-driven approach to implement business performance monitoring (BPM). The authors abstract a BPM solution to the level of models and using an MDD framework to generate a monitoring and control component. The generated monitoring component gathers monitoring information from real-time business events, aggregates information to calculate metrics, recognizes situations warranting business actions, and invokes those actions. Their solutioon differs from ours in that they focus on the role of processing events from other sources. Their framework does not generates the infrastructure to emit the events that they process. Momm et al. [12] use a model-driven approach to generate monitorable web service compositions. They provide meta-models that enable specification of elements to be monitored, as well as organization-specific indicators that should be monitored, and they generate instrumented BPEL code along with configurations for the types of events to be monitored for. This approach is very similar to ours except that the authors focus their work on generating an infrastructure for process monitoring, that is, workflow engine monitoring. We also include web services monitoring. In all cases where model-driven approach is applied, there is consensus on the code reusability and developer productivity gains from using a model-driven approach.

6. CONCLUSIONS

In this paper we have presented an approach for implementing the event-based monitoring infrastructure in an SOA. Our approach constitutes the application of a dedicated event view model and an eventing DSL. The view model captures the SOA artifacts and eventing concepts while the eventing DSL provides a tool for system developers to specify an instance of this event view model. Having these two ingredients enables us to use a model-driven approach to automatically generate most of the eventing infrastructure (code). With this approach, we believe that a system developer is able to better manage the complexity of implementing eventing infrastructure in an SOA, by focusing on the particular concerns of eventing. Consequently we expect that such an approach shall help in improving on the maintainability, reusability and understandability of eventing concepts in SOA systems. We evaluated our approach in the context of a case study where we implement an SOA monitoring infrastructure for a telecommunications services provider. We conclude that our approach has a number of advantages; it helps have a focused development approach, abstract away from specific technologies, allow for incremental implementation/evolution of an eventing infrastructure, and last but not least demonstrates productivity and efficiency gains in terms of the development efforts.

In addition to the case study presented in this work, we have also done preliminary work in two other cases, an ICT security case study and a business process adaptation case study. A part of the ICT security case study implements a loan process for a bank that involves customer verification and credit ranking before they are awarded loans. We attempt to use event-based monitoring for this business process. One of the major issues that comes up in this case study is the fact that the event-based monitoring can pose a security threat itself, depending on how much information gets included in events. In the business process adaptation case study, we attempt to combine our approach with complex event processing for monitoring and possibly adaptation of a business process as it executes. For this case, we are considering the necessity of combining the eventing view with a mechanism that specifies such complex event processing rules, and the actions to take when complex events are detected.

Overall, this work provides an implementation of a domain-specific language that enables engineering event-based infrastructure. In the case we have presented, we apply the problem to engineering monitoring infrastructure, however, event-based infrastructure can be applied to many more problems. We intend to extend our domain-specific language. As a follow up, we would like to incrementally extend the event view model and eventing DSL to address other aspects of engineering event-based SOAs and not just the monitoring aspects.

Acknowledgments

This work was supported by funds from the European Commission (contract No. 215175 for the FP7-ICT-2007-1

project COMPAS). The authors would also like to thank Anton Michlmayr for his reviews.

7. REFERENCES

[1] D. Box, L. F. Cabrera, C. Critchley, F. Curbera, D. Ferguson, S. Graham, D. Hull, G. Kakivaya, A. Lewis, B. Lovering, P. Niblett, D. Orchard, S. Samdarshi, J. Schlimmer, I. Sedukhin, J. Shewchuk, S. Weerawarana, and D. Wortendyke. Web Services Eventing (WS-Eventing). `http://www.w3.org/Submission/WS-Eventing/`. [accessed in Feb 2010].

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[3] P. Chowdhary, K. Bhaskaran, N. S. Caswell, H. Chang, T. Chao, S.-K. Chen, M. J. Dikun, H. Lei, J.-J. Jeng, S. Kapoor, C. A. Lang, G. A. Mihaila, I. Stanoi, and L. Zeng. Model driven development for business performance management. *IBM Systems Journal*, 45(3):587–606, 2006.

[4] N. Delgado, A. Q. Gates, and S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Trans. Software Eng.*, 30(12):859–872, 2004.

[5] P. Dini. Industrial Challenges in Working with Events. In *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*, pages 1–2, Edinburgh, Scotland, UK, 2004.

[6] G. T. Edwards, G. Deng, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. Model-driven configuration and deployment of component middleware publish/subscribe services. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 337–360. Springer, 2004.

[7] P. Fenkam, M. Jazayeri, and G. Reif. On methodologies for constructing correct event-based applications. In *DEBS '04: Proceedings of the 3rd International workshop on Distributed event-based systems*, pages 38–43. IEEE Computer Society, 2004.

[8] C. Hentrich and U. Zdun. Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, Irsee, Germany, July 2006.

[9] A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–15, New York, NY, USA, 2009. ACM.

[10] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, Reading, MA, 2008.

[11] G. A. Lewis, D. B. Smith, K. Kontogiannis, S. R. Tilley, M. Kajko-Mattsson, and N. Chapin. A research agenda for maintenance & evolution of soa-based systems. In *ICSM*, pages 481–484. IEEE, 2007.

[12] C. Momm, M. Gebhart, and S. Abeck. A model-driven approach for monitoring business performance in web service compositions. In M. Perry, H. Sasaki, M. Ehmann, G. O. Bellot, and O. Dini, editors, *ICIW*, pages 343–350. IEEE Computer Society, 2009.

[13] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, July 2006.

[14] Organization for the Advancement of Structured Information Standards. Reference model for service oriented architecture 1.0. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`. [accessed in Jan 2010].

[15] Organization for the Advancement of Structured Information Standards. Web Services Base Notification 1.3 (WS-BaseNotification). `http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf`. [accessed in Feb 2010].

[16] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[17] H. Tran. *View-based and Model-driven Approach for Process-driven, Service-Oriented Architectures*. PhD in software engineering, Vienna University of Technology, Distributed Systems Group, Information Systems Institute, Argentinier Str. 8/184-1, Vienna A-1040, Austria, Dec. 2009.

[18] H. Tran, U. Zdun, and S. Dustdar. View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. In W. Abramowicz and L. A. Maciaszek, editors, *International Conference on Business Process and Services Computing (BPSC)*, volume 116 of *LNI*, pages 105–124. GI, 2007.

[19] U. Zdun. A DSL toolkit for deferring architectural decisions in DSL-based software design. *Information and Software Technology*, 52(7):733 – 748, 2010.