Model-Driven Development of Adaptive Applications with Self-Adaptive Mobile Processes

Holger Schmidt[1], Chi Tai Dang[2], Sascha Gessler[1], and Franz J. Hauck[1]

[1] Institute of Distributed Systems, Ulm University, Germany
{holger.schmidt,franz.hauck}@uni-ulm.de
[2] Multimedia Concepts and Applications, University of Augsburg, Germany
dang@informatik.uni-augsburg.de

Abstract. Writing adaptive applications is complex and thus error-prone. Our self-adaptive migratable Web services (*SAM-WS*s) already provide adaptation support in terms of location, available state, provided functionality and implementation in use. Yet, *SAM-WS*s still require developers implementing the adaptation logic themselves.

In this work, we present an approach to ease the implementation of adaptive applications with *SAM-WS*s. We introduce our concept of a self-adaptive mobile process (*SAMProc*), an abstraction for adaptive applications, and *SAMPEL*, an XML application to describe a *SAMProc*. We show a tool that automatically generates *SAM-WS*s adaptation code on the basis of the *SAMPEL* description. Then, we go even one step further by providing an Eclipse plug-in that allows automatic generation of the *SAMPEL* description on the basis of a graphic model. This enables generating a *SAM-WS* implementation with few clicks; developers have to write pure application logic only.

1 Introduction

Mobile and ubiquitous computing (UbiComp) [1] scenarios are characterised by a high heterogeneity of devices, such as personal digital assistents (PDAs), mobile phones and desktop machines. They face a very dynamic environment due to the fact that devices, users and even applications can potentially be mobile. For tapping the full potential of the environment, such scenarios require adaptive applications. For instance, such applications should be able to use as much resources as possible on powerful devices and only few resources on resource-limited devices. Additionally, they should be mobile in terms of migration to enable applications running on the best-fitting devices in the surroundings (e.g., to run on a specific user's device or on the most powerful device).

Our approach to tackle such scenarios is self-adaptive migratable Web services (*SAM-WS*s) [2]. We advocate that Web services will become a standard mechanism for communication in mobile and UbiComp scenarios due to the fact that Web services have already gained acceptance in standard environments to provide a heterogeneous communication model. This is supported by the fact that there is already work on Web services providing reasonable communication between heterogeneous sensors [3]. Our *SAM-WS*s provide means to adapt themselves in terms of their location (i.e., weak service migration [4]), available state, provided functionality and implementation in use. At the same time, *SAM-WS*s maintain a unique service identity that allows addressing the Web service independent of its current location and adaptation (i.e., required to foster the collaboration between different *SAM-WS*–based applications). Although *SAM-WS*s provide a great flexibility for adaptive applications, developers have to manually implement the actual adaptation logic on their own.

In this work, we present a model-driven approach to ease the development of adaptive applications on the basis of *SAM-WS*s. Therefore, we build on our concept of a self-adaptive mobile process (*SAMProc*), which provides a novel abstraction for an adaptive application [5]. The basic idea is to describe the application as a *SAMProc* and to use this information to automatically generate the *SAM-WS* adaptation logic. As a novel description language, we present the self-adaptive mobile process execution language (*SAMPEL*), our new XML application to describe a *SAMProc*. Due to the fact that the business process execution language (BPEL) already provides means for orchestration of *standard* Web services, we implemented *SAMPEL* as a BPEL extension, which additionally supports describing *SAM-WS* behaviour regarding adaptation. We provide a tool, which automatically generates the adaptation logic of the corresponding *SAM-WS*; developers have to implement the pure application logic only. In comparison to related work [6,7], our approach is more lightweight because we generate node-tailored code which is not interpreted but executed at runtime. Additionally, we present an Eclipse plug-in that allows describing adaptive applications with a graphical notation. Modelling leads to an automatic generation of an appropriate *SAMPEL* description. In the overall process, our approach allows generating the adaptation logic of an adaptive application with only few clicks.

The rest of the paper is structured as follows. First, we introduce *SAM-WS* and an appropriate example application. In Section 3, we present our model-driven approach to develop adaptive applications with our *SAMProc* abstraction. After a discussion of related work in Section 4, we conclude and show future work.

2 Preliminaries

In the following, we introduce Web service basics and our *SAM-WS* extension. Then, we present a novel *mobile report* application, which acts as an exemplary adaptive application for the rest of the paper.

2.1 Web Services and Adaptive Web Service Migration

Web services are a common XML-based communication technology built upon standard Internet protocols [8]. They implement a service-oriented architecure (SOA) approach, in which functionality is provided by services only [9]. Web services are uniquely identified by uniform resource identifiers (URIs). The service interface and its protocol bindings are specified with the *Web services description language* (WSDL) [10]. WSDL binds the interface to a message protocol that is used to access the Web service. Therefore, Web services commonly use *SOAP* [11]. Due to the fact that Web services are built on top of XML technologies they are independent of platform and programming language. Thus, they perfectly suit heterogeneous environments.

We advocate that Web services will become a standard for communication in UbiComp. They have already gained acceptance in standard environments to provide heterogeneous communication and there is already work on Web services providing reasonable communication between heterogeneous sensors [3]. Yet, for tapping the full potential of UbiComp environments applications require adaptivity. In recent work [2], we proposed the concept of a *self-adaptive migratable Web service* (*SAM-WS*), which provides means for dynamic adaptation in terms of location (i.e., migration[1]), state, functionality and implementation.

For making Web services adaptive we introduce a *facet* concept. A facet represents a particular characteristic of the *SAM-WS* running on a particular *node* and comprising a particular *interface*, *implementation* and *state*. A *SAM-WS* adaptation can dynamically be applied at runtime by changing the location, interface, implementation or state of the *SAM-WS* (multiple concurrent changes are supported as well). A unique *SAM-WS* identity, which is used for continuously addressing the application, is maintained while adapting the *SAM-WS*.

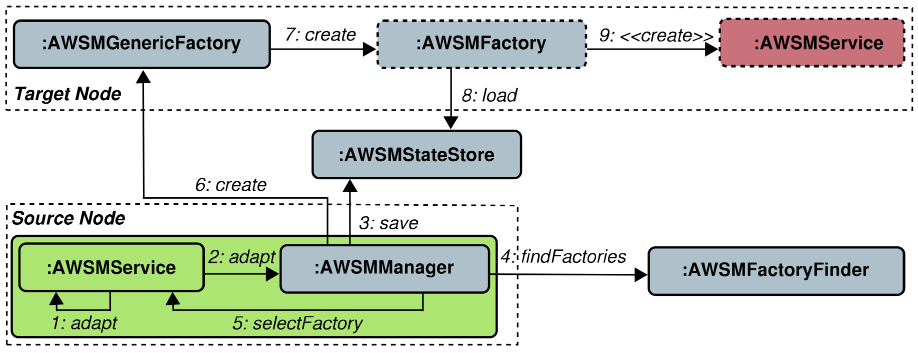


Fig. 1. Collaboration of adaptive Web service migration (*AWSM*) entities

Figure 1 shows our infrastructure services and their collaboration for the adaptation of a *SAM-WS*. For providing adaptation methods *SAM-WS*s have to implement our `AWSMService` interface. For easing application development the *SAM-WS* is able to use a generic service-internal `AWSMManager` entity[2], which manages the remaining adaptation steps (see Figure 1, step 2).

[1] *SAM-WS*s support weak migration [4]. Here, only application state is transferred but no execution-dependent state, such as values on the stack and CPU registers.

[2] Our *AWSM* prototype provides an `AWSMManager`. It is generic in the sense that it can be used within any *SAM-WS*.

The `AWSMManager` stores the active state[3] into an `AWSMStateStore` (step 3). The target location is determined by our `AWSMFactoryFinder` service[4]. The URI of an appropriate `AWSMGenericFactory` service is returned, which provides means to create *SAM-WS*s on a remote node. For allowing an application-specific selection of the best-fitting factory out of the list of appropriate factories returned from the factory finder we use a basic call-back mechanism (step 5). Then, the `AWSMGenericFactory` creates the desired target facet with the needed active state being loaded from the state store. If the code for the facet to create is unavailable at the target location, our infrastructure tries to load the code for a specific `AWSMFactory`, which is able to deploy the required *SAM-WS* facet (more details about our dynamic code loading feature can be found in [2]). In a last step, the old *SAM-WS* facet is removed.

Currently, application development support regarding *SAM-WS*s is quite limited. Our system provides generic adaptation code in terms of an abstract `AWSMServiceImpl` class for Java but the actual adaptation logic has to be implemented by the developer. Due to the fact that this is a non-trivial issue, there is still a need for further development support.

2.2 Example Application

Our example application supports crisis management by providing means for *spontaneous reporters* in the crisis surroundings to document the current situation (see Figure 2). Such a system is able to support rescue coordination centres with up-to-date information about the scene. Additionally, it can help afterwards with the investigation of causalities.

First, spontaneous reporters enter text, audio and video messages into a report application on their mobile device. The report is sent to *virtual first-aiders*, which undertake the task of reviewing the report. They prove the documentation and reject meaningless reports to disburden the *rescue coordination centre* where the accepted report is eventually presented.

The report application is implemented as a *SAM-WS*. With respect to this, unique reports are self-contained *SAM-WS* instances, which are adapted in terms of location to implement the mobile workflow. Additionally, our report application provides adaptivity in terms of programming language (due to heterogeneous environment with different hardware), functionality (each step in the mobile workflow needs different functionality) and state (e.g., anonymous reviewing: information about reporters should not be available at the virtual first-aiders but at the rescue coordination centre).

[3] Only implementation-independent state is externalised. It comprises only variables being interpretable by any possible implementation of a particular functionality. Moreover, we differentiate the overall *SAM-WS* state into active and passive state [2]. Thereby, active state is used within a particular *SAM-WS* facet while passive state is not. Passive state is stored in the `AWSMStateStore` and can be activated within another facet again.

[4] Due to the available universal description, discovery and integration (UDDI) mechanism for Web services [12], the `AWSMFactoryFinder` is implemented as a UDDI extension.

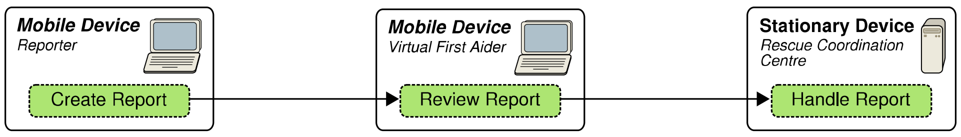


Fig. 2. Basic report application workflow

3 Building Adaptive Applications with SAMProc

The following sections present our model-driven approach to ease the implementation of adaptive applications with *SAM-WS*s.

3.1 Self-Adaptive Mobile Processes

In previous work [5], we introduced *SAMProc*:

> *A SAMProc can be seen as an ordered execution of services. It is able to adapt itself in terms of state, functionality and implementation to the current context and to migrate either for locally executing services or for accessing a particular context, while maintaining its unique identity.*

Thus, a *SAMProc* provides a high-level abstraction for adaptive applications. The idea is that developers should be able to model the application with its interactions and deployment aspects as a *SAMProc*. After a two-stage process (i.e., graphical model and its textual representation), a code generator automatically generates the application adaptation code, which is implemented as *SAM-WS*. Thus, only the pure application logic has to be provided by the developer.

In the following, we present *SAMPEL*, our textual *SAMProc* representation.

3.2 Self-Adaptive Mobile Process Execution Language

We introduce the *self-adaptive mobile process execution language* (*SAMPEL*) as our novel XML application, which is an extension of BPEL [13]. BPEL is an XML application being commonly used for describing business processes that are realised by Web services. Such a business process consists of the involved Web service interfaces, the interaction between them and a process state. Thus, BPEL is commonly characterised as a means for orchestration of Web services to build a business process. Such as Web services, BPEL uses WSDL to describe the involved Web service interfaces. A BPEL process itself is offered as a Web service. It is interpreted by BPEL engines, such as *ActiveBPEL* [14].

The way to describe processes with BPEL is suitable for describing *SAMProc*s as well. Yet, there are some issues why BPEL does not meet all requirements for *SAMProc*s. First, BPEL was particularly designed for business processes with focus on orchestration of Web services whereas *SAMProc*s rely on advanced concepts such as Web service facets and active process state. BPEL lacks support for these concepts. Additionally, BPEL processes are designed to be executed at a static location. Hence, BPEL does not provide the indispensable support for distribution aspects of *SAMProc*s. For instance, before migrating, a *SAMProc* has to select an appropriate location. Therefore, it needs context information about possible targets, such as available resources, being matched with its own context requirements. BPEL has to be extended for describing required context. Furthermore, current devices being used in UbiComp environments are highly resource-limited. Thus, it is in general not feasible to run a BPEL engine on these devices due to their high resource usage. Unlike BPEL, the *SAMPEL* process is not executed by a particular *SAMPEL* engine but used for node-tailored code generation. Additionally, *SAMPEL* provides support for the concepts of *SAMProc*s, in which the process is able to adapt itself according to the current execution platform. In Section 3.3, we present a code generator, which is able to automatically create code skeletons for all required implementations (i.e., Web service facets) of a *SAMPEL* process description. These code skeletons have already built-in support for process adaptation. Application developers have to implement the pure application logic only.

In the following subsections, we present *SAMPEL* in more detail. We attach particular importance to our extensions of BPEL.

Description Language. Like BPEL, a *SAMPEL* description is always paired with at least one WSDL description, which declares the *SAMProc* interfaces (i.e., interfaces of the Web service facets being implemented by the *SAMProc*).

Processes and Instances. A crucial difference between BPEL and *SAMPEL* is the conceptual view on a process. A BPEL process is an instance within a BPEL-engine and always has similar behaviour (e.g., starting with accepting a purchase order then communicating with involved Web services and eventually informing the purchaser about the shipping date). Unlike this, a *SAMPEL* process (i.e., *SAMProc*) is characterised by a highly dynamic behaviour since it can get adapted and migrated to another location at runtime, where it exists as an instance and handles user interactions (e.g., report application). Additionally, *SAMProc*s provide means to change the process state at runtime. Due to the inherent dynamics, a *SAMPEL* process is more functional oriented as opposed to a BPEL process (i.e., *SAMPEL* focuses on process functions instead of offering predefined process behaviour, such as in the purchase order example). This difference is reflected in the BPEL description by the placement of activities.

Figure 3 shows the process definition of a BPEL process with its activities. Activities specify the process behaviour, such as invoking a Web service and assigning a value to a variable. A BPEL process has activities in the main scope, whereas a *SAMPEL* process has not. Due to the functional oriented design, the activities of a *SAMPEL* process are basically determined by the activities within the method definitions (i.e., `eventHandler`, see below). Figure 4 shows the basic layout of a *SAMPEL* description. The `process` element contains all remaining parts of a *SAMProc*, which are explained in more detail in the following.

```
<process ...>
  ...
  ACTIVITIES
</process>
```

Fig. 3. Basic BPEL process description

```
<process ...>
  <partnerLinks>+
  ...
  </partnerLinks>
  <variables>?
  ...
  </variables>
  <correlationSets>
  ...
  </correlationSets>
  <eventHandlers>
  ...
  </eventHandlers>
</process>
```

Fig. 4. Basic *SAMPEL* process description

Scopes. A scope is a container for activities. As such, it is a structuring element that forms the control sequence of other elements. There are two kinds of scopes: the main scope and its sub-scopes. The main scope is implicitly defined by the `process` element and contains global variables, correlation sets and methods as shown in Figure 4. It must contain at least one method definition. Otherwise, the process has no activities. Sub-scopes (i.e., local scopes) can be defined by the `scope` element. As shown in Figure 5, sub-scopes must contain the activities to be executed and can contain elements that are used by the activities within the scope or its sub-scopes, such as local variables.

```
<scope>
  <partnerLinks>?
  ..
  </partnerLinks>
  <variables>?
  ..
  </variables>

  ACTIVITIES+
</scope>
```

Fig. 5. Basic *SAMPEL* scope description

PartnerLinks. Partner links are a *SAMPEL* concept inherited from BPEL. They allow declaring the communication endpoints of the *SAMPEL* process and its partner services. *SAMPEL* allows declaring partner links in the main scope as well as in sub-scopes. Figure 6 shows the concept of partner links illustrated for the report application. There, both processes—the reporter and the supervisor[5]—are represented by their *SAMPEL* description and WSDL description. Each process describes its communication endpoint by means of a partner link (i.e., `RLink` for the reporter process and `SLink` for the supervisor process). Each partner link relates to a partner link type of its WSDL description, which works as a bridge between partner links and a specific WSDL port type (i.e., contains the available operations). The same applies to the endpoints of other Web services (see Figure 6 for the partner link `RLink1` which links to the partner link type `SType` of the supervisor process). These definitions allow referring to communication partners within the process description.

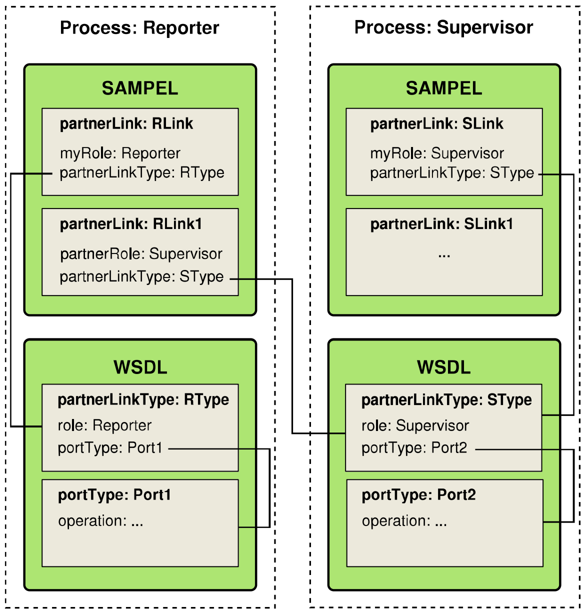


Fig. 6. Partner links

Figure 7 outlines the corresponding definition of partner links in a process description and a WSDL description. The upper part belongs to the *SAMPEL* description and the lower part to the corresponding WSDL description. A partner link must be defined inside a `partnerLinks` container element and is composed of a role, a name and the partner link type. The name of the partner link is referred within the process description to specify a communication endpoint for an activity. The role and partner link type refer to the WSDL description to select the WSDL port type for that communication endpoint. As shown in Figure 7, a partner link type has a name and must have at least one `role` element inside. A partner link from the process description points to a specific `role` element in the WSDL description by following the role and the name of the partner link type. Eventually, the `role` element refers to a WSDL port type.

[5] Here, we extended our report application with a particular supervisor entity that provides information about currently required information. This can highly increase the report quality.

```
<!-- SAMPEL -->
<process name="Report" xmlns:rpt="e1.wsdl">
  <partnerLinks>
    <partnerLink myRole="Reporter"
                 name="RLink1"
                 partnerLinkType="rpt:RType" />
  </partnerLinks>
</process>

<!-- WSDL -->
<definitions targetNamespace="e1.wsdl"
             xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"...>
  ...
  <plnk:partnerLinkType name="ReportPLT">
    <plnk:role name="Reporter"
               portType="tns:ReportPortType" />
  </plnk:partnerLinkType>
</definitions>
```

Fig. 7. Partner link example

Variables. An important means for storing temporary values or maintaining the process state are variables. *SAMPEL* allows variable declarations in the main scope and sub-scopes with the `variable` element. The variables in the main scope are global variables and are treated as the implementation-independent process state (i.e., considered for migration; see Section 2.1). Variables within sub-scopes are local variables used by activities in the sub-scope and its nested sub-scopes. Figure 8 shows an example for two variable declarations with respect to the report application. A variable must be declared within the `variables` container and is composed of a variable name and a data type for the value. The data type can be declared either using an XML schema type (see Figure 8, line 2) or as a reference to a WSDL message type (line 3).

```
1  <variables>
2    <variable name="report" type="xsd:string" />
3    <variable name="recID" messageType="rpt:ID"/>
4    ...
5  </variables>
```

Fig. 8. Variable description example

*Correlation Sets. SAMProc*s are created at a particular location. Then, they are able to migrate to other locations. Thus, there can be multiple instances of the same *SAMProc* at the same location. For distinguishing between them, *SAMPEL* inherits the concept of correlation sets from BPEL to create a unique identifier (i.e., composed of two parts in the form `process-id.instance-id`). The process identifier identifies the *SAMPEL* description and the instance identifier identifies an actual process instance. The instance identifier is derived from one correlation set only. Therefore, it must be defined within the `process` element.

Figure 9 shows how to define a correlation set for identifying a report application instance within *SAMPEL* and WSDL. Correlation sets have to be defined

```
<!-- SAMPEL -->
<process ...>
  <correlationSets>
    <correlationSet name="ID"
     properties="rep:ReporterID rep:ReportNr" />
    ...
  </correlationSets>
  ...
</process>

<!-- WSDL -->
<definitions xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop" ...>
  ...
  <vprop:property name="ReporterID" type="xsd:int" />
  <vprop:propertyAlias propertyName="ReporterID" messageType="inMsg" part="ID" />
  ...
</definitions>
```

Fig. 9. Correlation set to identify report instance

in the process description inside a `correlationSets` element and consist of a name and properties. The name of a correlation set is referenced from within the process description, whereas the properties have to be defined in the corresponding WSDL description (properties must be mapped to message types defined in the WSDL description). This way, correlation to an instance can be determined from incoming messages. All properties have a name, are of particular XML schema type and can be assigned to a WSDL message type by means of a `propertyAlias` element. The property alias refers to a property and to a WSDL message type. If the WSDL message type has more than one part, the `part` attribute addresses the appropriate part.

Methods. The behaviour of *SAMPEL* processes is basically described by the activities in methods, which are specified with the `onEvent` element. In Figure 10, we specify a `setReport` method as part of our report application. An `onEvent` element requires several mandatory attributes: a reference to a partner link that declares the communication endpoint, the port type of the partner link, a name for the method and a message type that declares the input to the method. Beside these mandatory attributes, there is an optional `variable` attribute that implicitly creates a local variable in the scope of the method and fills that variable with the values submitted at method invocation. Each method must have a scope for activities and a reference to the correlation set to use (i.e., to address the right instance on the basis of the received message).

Distribution Aspects. Distribution aspects are specified with the `requires` element, which can be placed as part of an `onEvent` element (see Figure 10, line 11–13). It can also be placed before an activity and thereby effect only the following activity. Figure 10 shows a basic example where the scope is restricted to the reporter role. A `property` element needs a key and a value. We allow multiple properties inside the `requires` element, which are interpreted as follows. If two properties have different keys, then the values are linked in a logical

```
<eventHandlers>
  <onEvent partnerLink="ReporterPL"
          portType="ReporterPT"
          operation="setReport"
          messageType="ReportMsg"
          variable="Message" >
    <correlations>
      <correlation set="ReporterCS" />
    </correlations>

    <requires>
      <property key="role" value="Reporter" />
    </requires>

    <scope>
    ...
    </scope>
  </onEvent>
  ...
</eventHandlers>
```

Fig. 10. *SAMPEL*: method description for reporter

'and' manner. They are linked in a logical 'or' manner in case of the same keys. This forms a property set with key/value-pairs that restricts an activity. It is a flexible means to describe requirements regarding distribution. Unlike BPEL, this feature is unique to *SAMPEL*.

Basic Activities. Activities determine the behaviour of a process. Therefore, *SAMPEL* inherits most of the BPEL activities. Basic activities actually contribute to a process step and are essential elements, such as a variable copy operation and waiting for an answer.

Communication with other Web services is covered by the `invoke` activity (see Figure 11 for an invocation at the supervisor within our report application). For invoking a Web service, a partner link to the desired Web service has to be specified. Furthermore, the name of the operation must be provided and an optional port type can be specified. Parameters of the Web service invocation have to be specified with the `toPart` element. *Waiting for an invocation* (i.e., at server-side) can be established with a `receive` activity, which is similar to an `invoke` activity for the partner link, operation and port type.

Sending a reply to an invocation is an important activity for communication with other services. A `reply` element either corresponds to an `onEvent` or a `receive` element (there has to be a partner link to identify the corresponding element). Additionally, a variable containing the return value has to be provided.

Assigning a value to a variable is done by the `assign` activity copying a value to a destination variable. Within the `assign` element, multiple `copy` elements are allowed. Each `copy` element performs a copy operation to a declared variable.

An *explicitly waiting* activity is possible with the `wait` element. It allows specifying a blocking wait state either for a particular duration or until a given date and time. This can be used as part of polling sequences.

Extensible activities allow extending *SAMPEL* with custom activities. Figure 12 shows how to define an extensible activity by the `activity` element.

```
<invoke partnerLink="SupervisorPL" operation="getRequiredInfo" portType="SupervisorPT" >
  <toParts>
    <toPart part="ID" fromVariable="varId" />
  </toParts>
  <fromParts>
    <fromPart part="RequiredInfo" toVariable="varRequiredInfo" />
  </fromParts>
</invoke>
```

Fig. 11. *SAMPEL*: invocation at supervisor

```
<activity name="spellCheckReport" />
```

Fig. 12. *SAMPEL*: custom reporter activity

Here, we define an activity supporting reporters with spell-checker functionality. There is only one attribute allowed, which denotes the activity name. In a corresponding *SAM-WS* implementation, such an activity is mapped to abstract methods (supported by our code generator; see Section 3.3), which have to be implemented by application developers. Thus, developers are able to use advanced programming language features that cannot be specified with pure *SAMPEL*. In contrast to *SAMPEL*, this feature is not supported by BPEL.

Explicit middleware support for adaptation is realised by the `copy` and `adapt` elements. These elements are not supported by BPEL. The `copy` activity creates a copy of the instance and assigns it a new instance identifier. The `adapt` element contains a property set. According to the given adaptation properties the process is able to adapt in terms of location, state, functionality and implementation. Figure 13 shows an example that requests an adaptation of the report application. A corresponding *SAM-WS* implementation (see Section 3.3) is able to pass the property set to our *AWSM* platform, which automatically handles the required steps to implement the *SAMPEL* description.

```
<adapt>
  <property key="role" value="Reviewer" />
</adapt>
```

Fig. 13. *SAMPEL*: adaptation to reviewer

Structuring Activities. Structuring activities form the control sequence for basic activities and can be arbitrarily nested in order to build complex control sequences. The first sub-scope of a method represents the top-level structuring element for starting a control sequence. It contains basic activities and structuring activities. Any structuring activity can contain further sub-scopes. Basic activities can be executed in sequence by surrounding them with a `sequence` element. Execution in parallel can be performed with the `flow` element, which starts each containing activity at the same time and ends when the last activity has finished. *SAMPEL* also offers constructs for conditional execution as known from traditional programming languages. Figure 14 outlines the usage of an if/elseif/else-construct for our report application. The condition has to be an XPath expression that evaluates to a Boolean value. Other conditional execution constructs are loops described with the `while` and `repeatUntil` elements. Both evaluate an XPath expression to repeat the containing activities. The difference is that the `while` element stops as soon as the condition evaluates to a Boolean `false`, whereas the `repeatUntil` stops if the condition evaluates to a Boolean `true`. For a corresponding *SAM-WS* implementation, sequential and conditional activities can be mapped to corresponding programming language constructs, whereas parallel activities should use threads.

```
<if>
  <condition>string-length($report)&lt;=100</condition>
  <adapt><property key="Mem" value="1MB"/></adapt>
  <elseif>
    <condition>string-length($report)&lt;=1000</condition>
    <adapt><property key="Mem" value="5MB"/></adapt>
  </elseif>
  <else>
    <adapt><property key="Mem" value="10MB"/></adapt>
  </else>
</if>
```

Fig. 14. *SAMPEL*: conditional adaptation

3.3 Automatic Code Generation

For implementing the model-driven approach with our *SAMProc* concept, code skeletons for the *SAM-WS* implementations have to be automatically generated from the *SAMPEL* description (see Section 3.1). We provide a Java code generator for this task. It keeps pure application logic written by application developers separated from generated implementation skeletons by using abstract classes and inheritance. This allows developers extending and customising the implementations with the pure application logic.

Figure 15 shows the overall code generation process. The code generator uses the *SAMPEL* description and all referenced WSDL documents to generate skeletons of appropriate *SAM-WS* facets (see Section 2.1). In general, code generation for any programming language is possible but this paper focuses on Java.

First, the code generator determines the interface (i.e., a set of methods) and available state for each *SAM-WS* facet. Therefore, it analyses the property sets of the method definitions within the *SAMPEL* description (a method describes its property set with the `requires` element). The distinct sets out of all property sets of the method definitions determine the required facets. Thus, each facet provides a particular set of properties, which determine the methods building up the overall interface. For completing a specific facet, the respective active state is determined by identifying the global variables being used within the methods. Finally, an XML file is created that holds meta data about the facet and its implementation. This allows our *AWSM* platform to register implementations and take meta data into account for adaptation decisions.

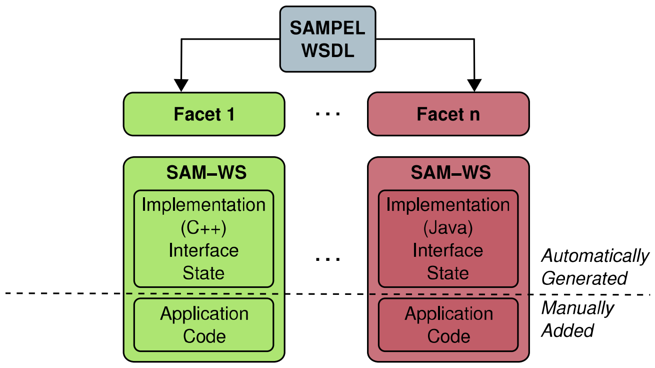
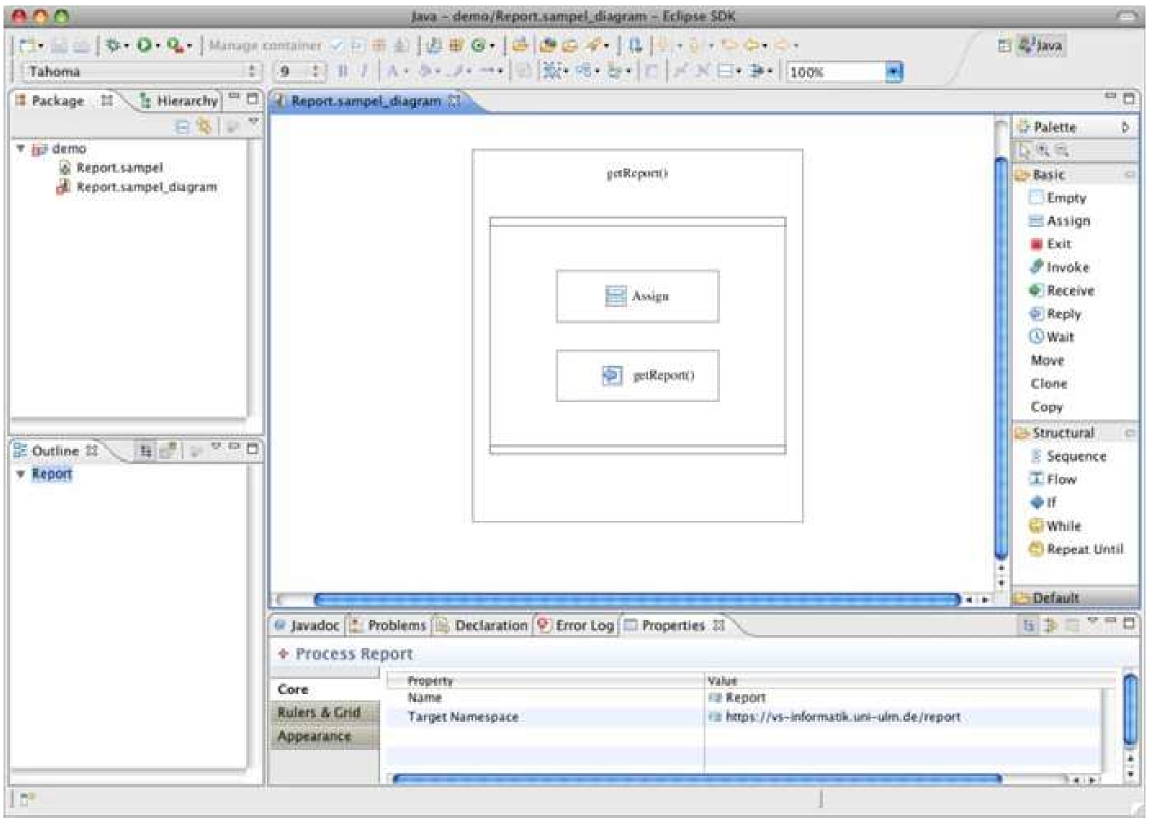


Fig. 15. *SAMPEL* code generator



Process Report

Property	Value
Name	Report
Target Namespace	https://vs-informatik.uni-ulm.de/report

Fig. 16. *SAMPEL* diagram editor with a part of the report application model

For each facet, code generation starts with the main scope and recursively processes sub-elements, such as methods, sub-scopes and activities. Methods are implemented with conventional methods as provided by programming languages with the following exception. Due to the fact that *SAMPEL* methods allow activities after replying to a request, such as a `return` instruction, the instructions of each method are wrapped into an own Java thread, which continues with activities while the requested method can apply its `return` instruction.

Most basic activities are implemented using their direct programming language counterparts or with extensions, such as Apache Axis for Web service invocations. Extensible activities result in abstract methods that have to be implemented by application developers. For mapping the adaptation logic to a particular *SAM-WS* implementation, the property set of an adaptation request is passed through to our *AWSM* platform, which automatically manages the needed steps as described in Section 2.1. Structuring activities, such as conditional execution, are mapped to their direct programming language counterparts (e.g., `flow` elements are implemented as threads to achieve parallel execution).

Finally, the generated *SAM-WS* facet skeletons (i.e., abstract classes) contain basic adaptation support and programming-language–dependent realisations of the activities being specified with *SAMPEL*. This includes the adaptation logic as well. Thus, developers only have to add the pure application logic to implement their adaptive application on the basis of a *SAM-WS*.

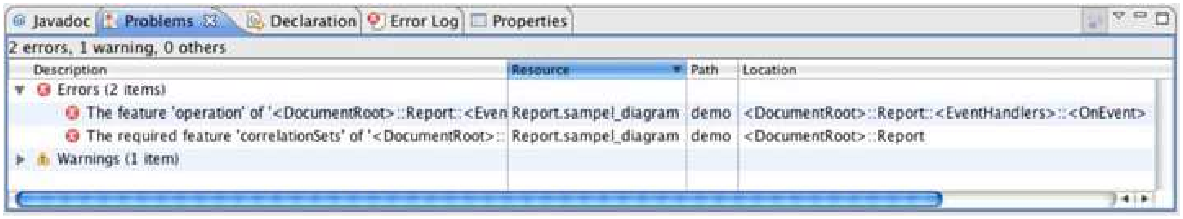
To ease addressing of a specific *SAM-WS* with our *AWSM* platform, we implemented an addressing schema, in which the process name and the target location is sufficient. For this purpose, our code generator creates a *proxy Web service*, which should be deployed within the Web service container. The proxy receives all messages for a particular application (corresponds to a *SAMPEL* description) and routes them to the appropriate *SAM-WS* instance specified by the correlation set. Therefore, the generated *SAM-WS*s contain programme logic that automatically registers the particular instance at the proxy.

3.4 Modelling Self-Adaptive Mobile Processes

Overall, generating an adaptive application requires various XML documents. Paired with at least one WSDL description, an appropriate *SAMPEL* description provides the adaptation logic. For simplifying the generation of these XML documents, we developed an Eclipse plug-in to model adaptive applications with a graphical notation. Since Eclipse already provides a WSDL editor [15], our plug-in delivers a novel diagram editor that assists application developers in building *SAMPEL* descriptions by allowing them to drop and to combine activities onto a drawing canvas. During modelling, the editor provides further assistance by validating the structural and semantic correctness of the document. For initially creating a *SAMPEL* description we provide an Eclipse wizard. This wizard allows selecting different templates, which serve as scaffolds for common use cases.

The diagram editor is realised with the graphical modeling framework (GMF) [16], which provides a generative component and runtime infrastructure for graphical editors based on a structured data model (i.e., an Eclipse modeling framework (EMF) [17] *ecore model* derived from the *SAMPEL* XML schema). Since GMF separately manages data model, graphical representation and tooling definition, the editor is highly customisable and easy to extend.

Figure 16 shows the user interface of the diagram editor. The graphic notation is similar to the *business process modelling notation* (BPMN) [18] that allows modelling BPEL processes. All *SAMPEL* elements inherited from BPEL are represented with the direct BPMN counterpart. We only introduce new activities for explicit adaptation support (i.e., `move`, `copy`, `clone` and `adapt`). The diagram canvas represents the *SAMProc*. The palette on the right allows selecting differ-



2 errors, 1 warning, 0 others

Description	Resource	Path	Location
Errors (2 items)			
The feature 'operation' of '<DocumentRoot>::Report::<Even	Report.sampel_diagram	demo	<DocumentRoot>::Report::<EventHandlers>::<OnEvent>
The required feature 'correlationSets' of '<DocumentRoot>::	Report.sampel_diagram	demo	<DocumentRoot>::Report
Warnings (1 item)			

Fig. 17. *SAMPEL* editor validation

ent tools to edit the process. In particular, there are tools for basic activities and structuring activities (see Section 3.2). Basic activities are essential elements of a process and are represented by a rectangular shape with a distinct icon and title. Since structuring activities form the control sequence for basic activities they are represented as titled rectangular containers for nested activities. Due to the fact that every change on the data model is performed using the GMF command framework, undo behaviour is seamlessly integrated.

The diagram itself does not display all required information to create a valid document. Otherwise, it would be too cluttered to be readable. Therefore, additional information can be captured and edited in the *Eclipse Properties View* by selecting the respective diagram element (see Figure 16).

The *Eclipse Problem View* displays errors and warnings being detected during validation (see Figure 17). Beside structural flaws, such as the fact that a scope has to contain at least one method definition, there are also various semantic correctness constraints. For instance, each process describes its communication endpoints by means of a partner link that has to relate to a predefined partner link type of its WSDL description. Those constraints are specified using the openArchitectureWare Check (oAW-Check) language [19], which is straightforward to use. Consequently, constraints can be extended with low efforts.

4 Related Work

There is related work in the area of mobile processes. For instance, *Ishikawa et al.* present a framework for mobile Web services, i.e., a synthesis of Web services and mobile agents [6]. Each mobile Web service has a process description on the basis of BPEL, which is used for interaction with other processes and for supporting migration decisions at runtime. This approach with its BPEL extension has similarities with *SAMProc*s and *SAMPEL*. Unlike our approach, it does not support adaptivity. Additionally, while the process description of mobile Web services is interpreted at runtime, we use *SAMPEL* for generating code. *Kunze et al.* follow a similar approach with DEMAC [7]. Here, the process description is a proprietary XML application being executed by the DEMAC process engine at runtime. Instead of using Web service and mobile agent concepts, plain process descriptions are transferred between process engines for achieving mobility. Unlike the DEMAC approach, we do not require a process execution engine on each device the platform is running. Additionally, we generate node-tailored code, which makes our approach more lightweight at runtime.

There is a lot of research regarding model-driven development of adaptive applications on the basis of distributed component infrastructures. Most of these systems, such as proposed by *MADAM* [20] and *Phung-Khac et al.* [21], allow modelling adaptation (i.e., dynamic component reconfiguration) decisions being executed at runtime. Unlike our approach, these frameworks are restricted to a custom component framework and do not support application migration.

Notations such as BPMN [18] define a business process diagram, which is typically a flowchart incorporating constructs suitable for process analysts and designers. Yet, the graphical notation for *SAMPEL* reflects the underlying structure of the language regarding its specific functional oriented design. This particularly suits application developers by emphasising the control-flow of an adaptive application.

5 Conclusion

In this work, we introduced *SAMPEL*, a novel XML application to describe adaptive applications on the basis of our *SAMProc* abstraction. Unlike related work, *SAMPEL* is not interpreted at runtime but used for generating the adaptation logic for our *SAM-WS*s. Application developers do not have to implement adaptation logic; they can focus on pure application logic. Furthermore, we allow modelling *SAMProc* adaptation with a graphic notation. Modelling with our Eclipse plug-in results in an automatic generation of an appropriate *SAMPEL* description. Thus, this work is a first step towards a model-driven development of tailored adaptive applications on the basis of *SAMProc*s.

If an application has many adaptation cases, our tool generates a lot of facets. For supporting the developer by generating as much code as possible we investigate adding Java code to custom *SAMPEL* activities. Such an approach is similar to *BPELJ* [22], which allows using Java code within BPEL descriptions.

References

1. Weiser, M.: The computer for the 21st century. Scientific American 265(3) (1991)
2. Schmidt, H., Kapitza, R., Hauck, F.J., Reiser, H.P.: Adaptive Web service migration. In: Meier, R., Terzis, S. (eds.) DAIS 2008. LNCS, vol. 5053, pp. 182–195. Springer, Heidelberg (2008)
3. Luo, L., Kansal, A., Nath, S., Zhao, F.: Sharing and exploring sensor streams over geocentric interfaces. In: GIS, pp. 1–10. ACM, New York (2008)
4. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding code mobility. IEEE TSE 24(5), 342–361 (1998)
5. Schmidt, H., Hauck, F.J.: SAMProc: middleware for self-adaptive mobile processes in heterogeneous ubiquitous environments. In: MDS, pp. 1–6. ACM, New York (2007)
6. Ishikawa, F., Tahara, Y., Yoshioka, N., Honiden, S.: Formal model of mobile BPEL4WS process. IJBPIM 1(3), 192–209 (2006)
7. Kunze, C.P., Zaplata, S., Lamersdorf, W.: Mobile process description and execution. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 32–47. Springer, Heidelberg (2006)

8. W3C: Web services architecture (2004), `http://www.w3.org/TR/ws-arch/`
9. Barry, D.K.: Web Services and Service-Oriented Architectures. Elsevier, Amsterdam (2003)
10. W3C: Web services description language (WSDL) version 2.0 part 1: Core language (2007), `http://www.w3.org/TR/wsdl20/`
11. W3C: SOAP version 1.2 part 1: Messaging framework (2007), `http://www.w3.org/TR/soap12-part1/`
12. OASIS: Introduction to UDDI: Important features and functional concepts. Whitepaper, OASIS (2004)
13. OASIS: Web services business process execution language version 2.0 (2007)
14. Active Endpoints: ActiveBPEL open source engine project (2009), `http://www.active-endpoints.com`
15. Eclipse Foundation: Eclipse web tools platform (2009), `http://www.eclipse.org/wtp`
16. Eclipse Foundation: Graphical modeling framework (2009), `http://www.eclipse.org/gmf`
17. Eclipse Foundation: Eclipse modeling framework (2009), `http://www.eclipse.org/emf`
18. OMG: Business process modeling notation (BPMN), version 1.2. OMG Document formal/2009-01-03 (January 2009)
19. openArchitectureWare.org: openarchitectureware (2009), `http://www.openarchitectureware.org`
20. Geihs, K., Barone, P., Eliassena, F., Floch, J., Fricke, R., Gjorven, E., Hallsteinsen, S., Horn, G., Khan, M., Mamelli, A., Papadopoulos, G., Paspallis, N., Reichle, R., Stav, E.: A comprehensive solution for application-level adaptation. Software: Practice and Experience 39(4), 385–422 (2009)
21. Phung-Khac, A., Beugnard, A., Gilliot, J.-M., Segarra, M.-T.: Model-driven development of component-based adaptive distributed applications. In: SAC, pp. 2186–2191. ACM, New York (2008)
22. Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., Rowley, M.: BPELJ: BPEL for Java. Whitepaper, BEA (2004)