A Lightweight Model-Driven Orchestration Engine for e-Services

Johann Oberleitner, Florian Rosenberg, and Schahram Dustdar

Distributed Systems Group, Institute of Information Systems,
Vienna University of Technology
{joe, rosenberg, dustdar}@infosys.tuwien.ac.at

Abstract. Service-oriented Computing (SoC) in general, and e-service orchestrations in particular have the potential to increase reuse and to ease maintainability. Typically, interoperating e-services require orchestration efforts, which should be accomplished outside the application logic itself. In this paper we present a novel MDA-based approach for generating orchestrations of e-services, enabling the automatic generation of e-service orchestrations based on UML models. Secondly, such orchestrations may include GUIs. Thirdly, we discuss our execution environment supporting heterogeneous e-service orchestrations, including Web services, COM, CORBA, and .NET objects. Such heterogeneous software system landscapes are very common today, where many (legacy) applications still exist and are not wrapped as e-services, nor BPEL process descriptions are available.

Keywords: Model-Driven Approach, Service Orchestration, e-Services.

1 Introduction

Today, there is a growing recognition that the Service-oriented Computing (SoC) paradigm [1], including its property of loose-coupling, facilitates higher flexibility of interoperable information systems. To increase reuse and to ease maintainability, interoperating e-services require orchestration efforts, which should be accomplished outside the application logic itself.

In this vein, recent Model-driven Development [2] efforts, provide a viable conceptual framework allowing software or e-service generation, with (ideally) minimal extra coding efforts. Current ambitions in research and industry are, therefore, aimed at moving e-service development towards a higher level of abstraction, where models of e-services and their orchestrations are modeled and the code is generated thereafter. On the one hand, Service-oriented Architectures (SOA) gain wider acceptance as a paradigm for loose coupling of software services distributed on the Internet. On the other hand, activities carried out by humans increasingly require higher flexibility and new ways of supporting loosely-coupled work teams and its involved team members. Today, both users of e-services, i.e., humans or other e-services, are not integrated well enough to leverage the full potential of SoC.

The contribution of this paper is threefold. We provide: a) an MDA-based approach for generating orchestrations of e-services, enabling the automatic generation of e-service orchestrations based on UML models; b) the integration of GUIs in such orchestrations; and c) an execution environment supporting heterogeneous e-service orchestrations, including Web services, COM, CORBA (by using [3]), and .NET objects. Such heterogeneous software system landscapes are very common today, where many (legacy) applications still exist and are not wrapped as e-services, nor BPEL process descriptions are available. Nevertheless, such heterogeneous systems require integration into larger orchestrated systems.

The remainder of this paper is structured as follows. Section 2 provides a motivating example from an application domain where we evaluated the viability of our implementation. Section 3 outlines the modeling support provided in our implementation. Section 4 discusses the execution engine which processes UML state- and activity-diagrams and generates required orchestrations. Section 5 presents related work. Section 6 concludes the paper and outlines our future work.

2 Motivating Example

We motivate our model-driven orchestration approach by considering the following example from the hospital domain. When a patient arrives at the hospital several departments and different specialists are involved in the diagnosis. Each department has different, heterogeneous software systems — ranging from applications using COM or CORBA components, and systems offering Web services — which have to be used to make a diagnosis. Some parts of the workflow are performed manually, by entering data using a GUI, other steps are performed automatically, e.g., storing blood-specific data or x-ray results in the database. In addition, it might be necessary to include data from previous examinations that are not stored in-house. Therefore, external services need to be queried. Our scenario, a routine meniscus surgery, has the following workflow: (1) When the patient arrives at the hospital, her personal data is collected and entered in the hospital information system by using a GUI application. On finishing the registration, the health insurance data (containing information from previous diagnoses, etc.) is collected from the insurance company by using their Web services, and stored in the patient record. (2) After the registration process, a doctor is briefing the patient where previous diagnoses and illnesses are discussed and upcoming examinations are clarified. The doctor directly enters important notes to the hospital information system by using the provided GUI application. (3) The standard procedure for a meniscus surgery is to take blood and make x-rays of the necessary parts. The order of these examinations is not important. Due to the high amount of data, the x-rays result is not stored in the hospital information system directly, it is archived in a special database in the radiology department and linked with the patient record. (4) Then, the patient has to go back to the doctor to do the final medical examinations and to discuss the

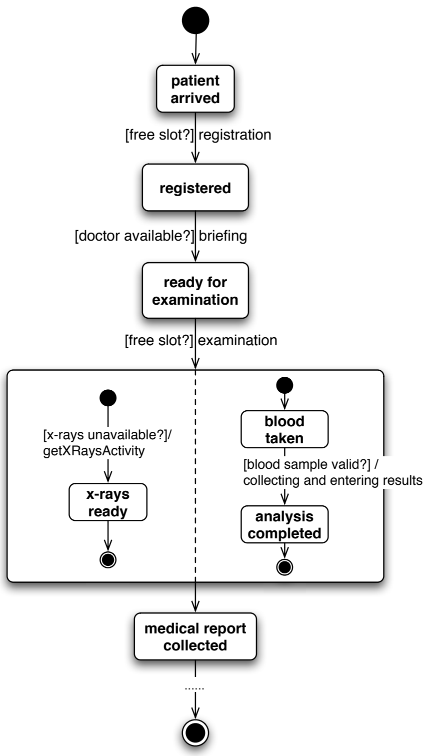
patient arrived
[free slot?] registration
registered
[doctor available?] briefing
ready for examination
[free slot?] examination
[x-rays unavailable?]/ getXRaysActivity
x-rays ready
blood taken
[blood sample valid?] / collecting and entering results
analysis completed
medical report collected
......

Fig. 1. Hospital Workflow State Diagram

results of the blood sample and the x-rays. The notes from the doctor are again directly entered into the system.

A simplified version of this workflow is shown in Figure 1 as a UML statechart diagram. We use this model throughout the paper to explain the concepts of our model-driven orchestration approach.

3 Model Driven Approach

In the next sections we describe our orchestration models and how these models are transformed to our internal representation so that our execution engine can process it.

3.1 Modeling Support

We support UML state machines as well as activity diagrams to build a model based orchestration for services. Both diagram types are important for building complex business processes. While state machines are suitable when explicit states can be identified and activities infer transitions between these states, activity diagrams are preferred when no states can be identified or each activity would require the introduction of pseudo-states. Furthermore, in state diagrams, the state of an orchestration is always explicit. This explicit state offers support for long-running transactions, which require such explicit points to wait for other activities (e.g., user input).

For state-charts, we support most constructs provided by the UML 1.5 standard [4]. In particular, simple states, composite states to structure an orchestration, history states, concurrent states, initial (start) and final (end) states and transitions, eventually restricted by guards, are supported. In the example depicted in Figure 1, we have used an initial state, multiple simple states, one concurrent state with two nested child states and multiple transitions. To execute application logic associated with states or transitions, references to an action may be linked to state entry, state exit or the firing of transitions. An action itself may be modeled by an activity diagram or another state machine, or we refer to an invocation. Most UML modeling tools support these links directly in the model.

According to Figure 1, when a new patient arrives, the first state is entered. When a free registration slot is available, the registration process is initiated, in which the nurse enters the patient data by using a GUI application which will be invoked by the `registration` action. Such actions can be modeled by activity diagrams which may include service invocations, but also processing steps of a GUI. Actions included in a model can refer to GUIs to fill and retrieve data and to steer the control flow. For instance, user decisions (e.g., pressing buttons) in a GUI may directly be reflected in the control flow of the orchestration.

After several state transitions, the patient is waiting for a free examination slot, which leads to states running concurrently: the right part models the blood examination, whereas the left one models how to deal with required x-rays images which is handled by the `getXRaysActivity` in case the x-rays have not already been available in the system. After both results are available, the concurrent states are left and the medical report is prepared.

Actions invoked on state entry, state exit and on state transitions, are either service invocations, activity diagrams or new state machines. The highest flexibility among these possibilities is provided by executing activity diagrams, as can be seen in Figure 2. Actions in this diagram are activated when no recent x-ray images are available. It models the possibilities of providing a recent x-ray image, (1) either by scanning the one brought by the patient, or (2) by requesting it via Web service from a medical specialist or (3) create one in-house by doing an x-ray.

Unlike standard design processes, we require that the models are complete, since the various models are the single source of input. Due to space restrictions, our illustrations are just simplifications of real models. There must not be any informal actions or guards. For our model in Figure 1, this means that the guard `[doctor available]` must refer to system variables, for instance, `[exists (doctorAvailable)]`. This expression evaluates to true if there exists a global system variable `doctorAvailable`.

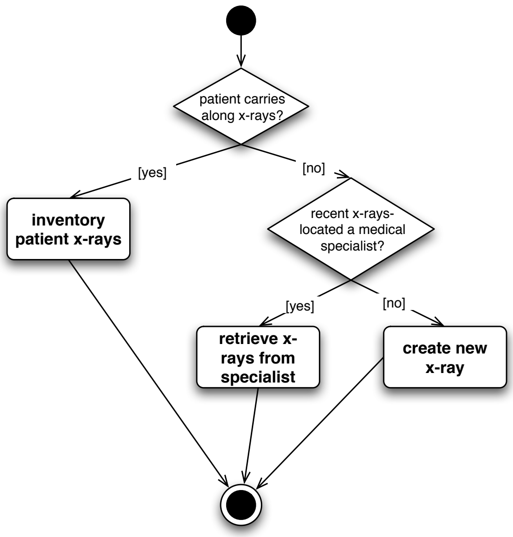
patient carries along x-rays?
[yes]
[no]
inventory patient x-rays
recent x-rays-located a medical specialist?
[yes]
[no]
retrieve x-rays from specialist
create new x-ray

Fig. 2. X-Rays Activity Diagram

Modeled orchestrations can be exposed as composite services, therefore, two special actions for receiving input messages and replying output messages have to be included in the diagrams. These actions are parameterized by name and type of the message parts and variable names which are used to refer to within the orchestration.

3.2 Transformation Tool

The models can be created with any UML standard compliant modeling tool, which supports XMI [2] export. Our execution engine, however, does not process XMI directly but requires a data source which supports sequential data, such as relational databases or XML files.

We have built a transformation tool, which converts XMI files to our sequential representation. States and transitions of state machines are parsed and stored in tabular form. Table 1 shows a subset of the generated state entries. Nesting of states is handled with the state type column for which concurrent or composite states can possess child states. The example shows the examination state from Figure 1 which contains several child states processed concurrently. The child states themselves are composites.

Table 2 depicts how transitions are stored by the transformation tool. For instance, the transition that leads to the `x-rays ready` state in Figure 1 has a guard and invokes the `getXRaysActivity` action.

Activity diagrams are also stored in one action table that stores the name of the action and invocation parameters. To support control flow actions, the transformation tool assigns unique incremental numerical ids to each action in

Table 1. Transformed State Example

state name	state type	parent state	initial child state	entry action	exit action
examinations	concurrent	-	-	-	-
x-ray examination	composite	examinations	initial x-rays	-	-
initial x-rays	initial	x-ray examination	-	-	-
x-rays ready	simple	x-ray examination	-	-	-
x-rays available	final	x-ray examination	-	-	-

Table 2. Transformed Transition Example

transition name	source state	target state	transition action	guards	trigger events
getXRays	x-rays init	x-rays ready	getXRays-Activity	exists (s::xrays)	-
anonymous	x-rays ready	x-rays available	-	-	-

the activity diagram. The execution engine uses these ids to dynamically select the next action.

4 Execution Engine

We have built our lightweight execution engine for the .NET platform to execute the models described in the previous sections. In this section we describe the mechanisms to process models stored in a format provided by the transformation tool. The aforementioned tables act as input to the execution engine.

4.1 Processing State Machines

Processing a state machine is initiated by creating an instance of the `StateMachine` class. This instance fetches the state and transition tables into memory. After the instantiation of the state machine, initial states are immediately entered and followed to the innermost nested state. Figure 3 shows parts of the class diagram for processing state machines.

Transitions between states are triggered by events. We support different kinds of events, primarily GUI events caused by user interactions or custom events caused by programmatic actions. In case one of these events happens, a transition is fired and the target state becomes activated.

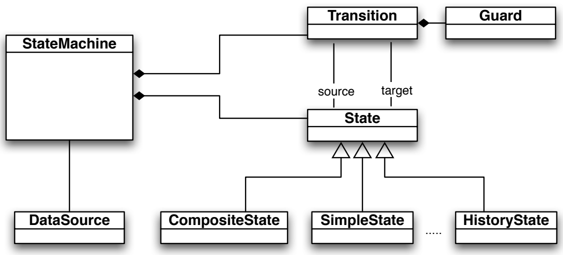

Fig. 3. Execution Engine Class Diagram

On each state entry or state exit, as well as on transitions, an action can be executed. These actions may refer to activity diagrams, state machines, or service invocations. There is no particular difference, if a simple state is entered or if a nested state in a composite state is entered. Furthermore, we emit additional .NET events on state entry, state exit or state transition. This allows the execution of recurring actions for each transition arc without polluting the process models by referring to the same activity diagrams over and over. For instance, if a user is not allowed to enter another state dependent on the context of the data, event handlers may cancel the transition.

In addition to an action that may be executed when a transition event has fired, a guard expression can be provided. This guard expression returns a boolean value and is evaluated before a state exit or an optional transition activity is started. In case the guard evaluates to false the whole transition is canceled and the old state is restored.

Furthermore, our engine also supports history states. A history state stores in which substate a composite state resided before the composite state is left. When the history state is entered again the previous state is restored. Concurrent states split the execution in multiple paths, which can be executed in parallel. The child states of a state of type `concurrent` are in turn composite states that require an explicit initial and an explicit final state. When a concurrent state is entered, the initial states of each child state are entered. A concurrent state is left when each concurrently processed child state has reached a final state. One drawback of our implementation, however, is that we do not support transitions from one concurrent state to another.

4.2 Processing Activity Diagrams

Fine-grained actions may be modeled with UML activity diagrams. Our execution engine processes sequences of actions stored in the data source of the `actiongroup` tables. Figure 4 shows the classes involved in processing of action groups. To execute these action-groups an instance of type `ActionGroup` is created. Similar to states and transitions of state machines, the sequence of actions is loaded on initialization of the action-group. Actions themselves are realized with the flyweight pattern [5]. For each single action within a sequence of actions

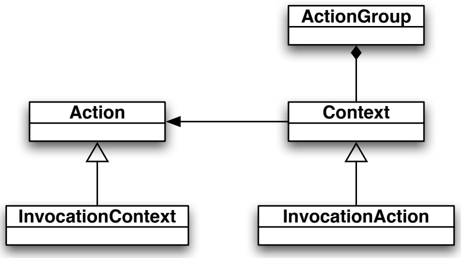

Fig. 4. ActionGroup Class Diagram

a `Context` object is created that parses and stores the parameters originating from the action description in the data source. The classes that implement the actions themselves inherit from the `Action` class.

When an action-group shall be executed a loop processes each context object that in turn delegates the execution to the action object. The context provides not only access to the parameters of the action but supports also access to variables with different scopes, either system global, statemachine-local, or local to an activity diagram. Furthermore, the control flow within an activity diagram may be modified by modifying some predefined fields of the context. We have predefined various action classes that have different tasks:

Control-flow actions: Some actions deal with modifying the flow within the sequence of actions. The execution engine supports an if-then-else construct, too. One parameter provides the conditional expression that is evaluated at runtime. These expressions refer to any variables stored in the context and test its existence, support relational operators for comparison of numbers and strings, and may use logical operators to build complex expressions. Another action *ReturnToOldState*, allows that a transition may be canceled by setting a context flag.

StateMachine actions: A small number of actions allow starting sub-statemachines that will either block the current machine or can also run in parallel.

Container actions: Some actions allow reading and writing to arbitrary variables provided by a blackboard. Actions and the state machines can then access these variables.

Domain-specific actions: We have used the execution engine to also integrate GUI elements. Hence, we have built a couple of actions that load and show GUI forms and other actions that allow modification of GUI widgets.

Invocation actions: Services can be executed by using one of the invocation actions (for CORBA, .NET, COM or Web services). The targets of these calls are configured externally. The execution of further actions may be blocked by another action that supports waiting for notifications.

New action types can easily be added by providing an action class which implements execution semantics and a context class that parses and stores the configuration parameters.

5 Related Work

To the best of our knowledge, there are currently no existing model-driven orchestration approaches, which focus on the integration of heterogeneous services that also include GUI applications.

In the workflow and BPM (Business Process Management) area, numerous approaches exist that focus on coordinating work through software. Most of these approaches are not capable of invoking different components such as Web services, COM, DCOM, Java, EJB. The few existing approaches, such as JOpera [6] or JBPM [7] use proprietary languages and tools for modeling, whereas we rely on using standardized tools (UML) to model our workflows. Furthermore, these tools only support Web services and Java technologies.

Newer approaches, inspired by the service-oriented architecture (SOA), focus on the orchestration and composition of Web services into higher-level processes and composed services. Currently, BPEL [8] is increasingly used for the orchestration of Web services [9]. We believe that one of the major disadvantages of BPEL is that the activity types in the orchestration are limited to Web services only and that there is currently no modeling standard for BPEL processes. BPEL-J [10], a joint effort of BEA and IBM, tries to combine BPEL with Java by adding activities, called *Java Snippets*, which allow to embed Java code into the process and allow to interact with J2EE components. Our decision not to use BPEL as execution language has several origins: Firstly, BPEL is not designed for combining service invocations and human-centric interactions. Secondly, existing BPEL execution engines are rather heavyweight, while our execution engine has only small requirements on the environment and may potentially be executed on PDAs.

Executable UML [11] is one research direction in the MDA area focusing on building directly executable models of software systems which are used as input to execution engines. Usually these approaches focus on rather small domain-specific areas, such as embedded systems [12]. Our approach, however, focus on the integration of heterogeneous services.

6 Conclusion and Future Work

Orchestrating e-services provided by different systems is increasingly important for heterogeneous systems. Current implementations do not focus on a model-driven orchestration of services provided by heterogeneous systems and the integration of GUI based applications. Based on an example from the medical domain, we presented our model-driven approach for specifying service orchestrations, which allow the invocation of various services implemented in .NET, CORBA and COM. Furthermore, a service can even be a GUI application integrated into the orchestration. A previous version of the system is already in use in one hospital in Austria.

We plan to improve modeling the support. Currently, exception handling is only supported by nesting diagrams, which is tedious to do with modeling tools.

By specifying rollback points and compensation actions, nesting can be avoided in the model, and is automatically supported by the transformation tool.

The future work in this area includes porting the execution engine from .NET to Java, which offers the flexibility, to additionally integrate and invoke components and applications written in Java and EJB. Furthermore, a Java solution allows us to invoke functionality implemented with common component models in the invocation classes through the Vienna Component Framework (VCF) [13].

References

1. Papazoglou, M.P.: Service-oriented computing: concepts, characteristics and directions. In: Proceedings of the Fourth International Conference on Web Information Systems Engineering. (2003) 3–12
2. Frankel, D.S.: Model Driven Architecture – Applying MDA to Enterprise Computing. OMG Press (2003)
3. Oberleitner, J., Gschwind, T.: Transparent Integration of CORBA and the .NET Framework. In: Proceedings of On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE (DOA). (2003)
4. Object Management Group (OMG): Unified Modeling Language (UML), Version 1.5. `http://www.omg.org/technology/documents/formal/uml.htm` (2004)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
6. Pautasso, C., Alonso, G.: From web service composition to megaprogramming. In: 5th International Workshop on Technologies for E-Services (TES). (2004) 39–53
7. JBoss: Java business process management. `http://jbpm.org/` (2005)
8. BPEL: Business Process Execution Language for Web Services Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/ (2003)
9. Pasley, J.: How BPEL and SOA are changing web services development. IEEE Internet Computing **9** (2005) 60–67
10. BEA Systems Inc. and IBM Corp.: BPELJ: BPEL for Java. `ftp://www6.software.ibm.com/software/developer/library/ws-bpelj.pdf` (2004)
11. Mellor, S.J., Balcer, M.J.: Executable UML – A Foundation for Model-Driven Architecture. Addison-Wesly (2002)
12. Raistrick, C., Francis, P., Carter, J.W.C., Wilkie, I.: Model Driven Architecture with Executable UML. Cambridge University Press (2004)
13. Oberleitner, J., Gschwind, T., Jazayeri, M.: The Vienna Component Framework: Enabling composition across component models. In: Proceedings of the 25th International Conference on Software Engineering (ICSE). (2003)