Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Complex Event Processing for Monitoring Business Processes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

eingereicht von

Thomas Haitzer

Matrikelnummer 0325781

an der

Fakultät für Informatik der Technischen Universität Wien

Betreuung

Betreuer: Univ.Ass. Dr.rer.nat. Uwe Zdun

Wien, 22.09.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Thomas Haitzer
Haanbaumstraße 9
3001 Mauerbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Mauerbach, 22.09.2010

Thomas Haitzer

Abstract

With Business Processes becoming more and more popular and the applications growing more complex, the need of monitoring and controlling enterprise business process systems grows. The existing solutions do not support model-based compliance control [16] and often have limited event-abstraction capabilities. In this thesis we develop and implement a monitoring system for BPEL processes based on the ideas from Holmes et al. [15, 14], that can be used for monitoring business processes, compliance-constraint monitoring, as well as for debugging. It provides different user-interfaces for different users like high level views for non-technical personnel or debug-views for software engineers. By using model-driven engineering [5] and complex event processing we are able to build a monitoring system that can easily adapted for a variety of business process engines.

From our event meta-model we generate a system, that is split into three independent components which can be distributed a long different servers and are connected via message queues. While the first component (event-listener) collects the events from the event-sources, the second component (event-processor) extract business relevant information like e.g. constraint violations from the event-streams generated by the event-listener. The third component (monitoring application) provides a user-interface for reviewing the data extracted by the event-processor.

Kurzfassung

Mit ständig steigender Relevanz von Geschäftsprozessen in den verschiedensten Bereichen, werden auch die Anwendungen, die damit implementiert werden, stetig komplexer. Damit im Einklang steigt auch die Notwendigkeit der Kontrolle und Überwachung solcher komplexer Applikationen. Bereits bestehende Applikationen sind meist nur mit einem, bestenfalls mit einigen wenigen Applikations-Servern kompatibel. In dieser Diplomarbeit entwickeln wir ein Kontrollsystem zur Überwachung von auf BPEL basierenden Geschäftsprozessen, welches leicht mit bestehenden Applikationsservern integriert werden kann und sowohl technische Informationen zur Fehlersuche als auch abstrahierte Informationen zu den Geschäftsabläufen in Form von sogenannten "Views" darstellt. Zusätzlich ist es möglich, mit diesem System die Einhaltung von Randbedingungen, die während der Laufzeit Gültigkeit haben müssen, zu überwachen und eine Nachricht abzuschicken, falls eine Bedingung verletzt wird. Um die Anpassung an vorhandene Systeme möglichst einfach zu gestalten und rasch und ohne großen Aufwand auf Änderungen im System reagieren zu können, wurde von uns ein Ansatz auf Basis der modellgetriebenen Entwicklung gewählt [5].

Während wir als Laufzeitumgebung Apache Tomcat verwenden, ist unsere Lösung auch auf anderen Java-basierten Applikationsservern wie zum Beispiel JBoss[1] einsetzbar, um diese leicht in vorhandene System-Architektur integrieren zu können.

[1] `http://www.jboss.org/`

Contents

Abstract **iii**

Kurzfassung **v**

Contents **vii**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

2 Context and Problem Description **5**

2.1 Service Oriented Architecture 5

2.2 Challenges of Business Activity Monitoring 6

2.3 Main Contributions 8

3 Solution Overview **9**

3.1 Event-processor 10

3.2 User Interface 11

4 Design and implementation details **17**

4.1 Meta Model 19

4.2 Listener 32

4.3 Event-processor 34
4.4 User Interface 34
4.5 Code generation 37

5 Examples **49**
5.1 Example BPEL-processes 49
5.2 Event rules 54
5.3 Test results 56

6 Related work **59**
6.1 Business process monitoring & Complex event processing 59
6.2 Model-Driven-Developement 61
6.3 Business Activity Monitoring (BAM) Products 63

7 Discussion & Lessions Learned **65**

8 Conclusion & Outlook **67**
8.1 Future work 68

Bibliography **69**

List of Figures

3.1 Architecture Overview 10
3.2 Screen-shot of the live dashboard 14
3.3 Example of a generated view page 15

4.1 Architecture and Message Flow 18

4.2 Event Metamodel: Overview 20
4.3 Event Meta Model: Basic Events 22
4.4 Event Meta Model: Highlevel Events 26
4.5 Visualisation of events for the rule ∀A → B . When visualizing the rule ∀(A → B one has to remove the A3 B3 event (the last arrow in the right column). 28
4.6 Example for a conversion from a business-process engine defined event to a model defined event. 32
4.7 Example for a conversion from a web-service event to a model defined event. 33
4.8 Example Xpand-project layout [19] 38
4.9 Xpand-template for generating the event-beans. 40
4.10 Xpand-templates for generating the esper-rules for basic-events. 42
4.11 Xpand-template for generating the Esper-rules for composite-events. 44
4.12 Example event-rule for a composite event of type every A → (B and not A) 46
4.13 Example event-rule for a composite-event of type every (A or B) → C . . . 47

5.1 Credit-approval business process example 52
5.2 Taken from [3]: Counter process example 53

List of Tables

5.1 Test results 57

CHAPTER 1

Introduction

In the last years service-oriented-architecture and business processes became more and more popular with businesses. The use of this new technologies often led to a great deal of automation, where business-applications often are used as autonomous systems that often run without user-interaction on the company-side. These kind of systems need to be constantly monitored in order to detect performance problems, find improvements and optimizations for the business processes, possible violations of compliance-concerns, or simply debug any faults. The term “Business Activity Monitoring” is defined by Gartner [21] as follows: “"Business activity monitoring" (BAM) is Gartner’s term defining how we can provide real-time access to critical business performance indicators to improve the speed and effectiveness of business operations. Unlike traditional real-time monitoring, BAM draws its information from multiple application systems and other internal and external (interenterprise) sources, enabling a broader and richer view of business activities. As such, BAM will be a natural extension of the investments that enterprises are making in application integration.”

Such a monitoring application needs to be able to gather run-time information for the various business processes from various sources and to store this data for a later review. It also needs to provide options for raising alarms if a compliance-contract is violated and a live view on the business data.

This business data needs to be shown in varying detail for different users and stakeholders and should enable the possibility to react fast to any kind of problem that might arise.

With the growing complexity of business processes not only the monitoring becomes more and more important, also people try to introduce new ways to tackle this problem. Tran, Holmes, Zdun, and Dustdar introduce an approach that breaks “a business process into appropriate architectural views” [31], where each view describes a separate concern of the same business process. In this approach the authors make use of model-driven-development, especially the Eclipse Modeling Framework [29], to generate executable business-processes from the introduced views. When now monitoring a system that is built upon this approach, it is crucial to know not only which processes are monitored but also to which models these processes belong. How this problem can be tackled is described in “Monitoring and Analyzing Service-Based Internet Systems through a Model-Aware Service Environment" [15] and “Model-driven and Domain-specific Architectural Knowledge View for Compliance Meta-data in Process-driven SOAs” [14]. In these articles MORSE [16], a model-aware service environment is used to store models of business processes and compliance-concerns regarding these business processes which need to be monitored. They also say that a monitoring system for MORSE must be able to make use of the modeling-information stored in this system in order to check for compliance violations and other relevant data.

In this thesis we introduce and implement an approach for building a business process monitoring system that is capable of monitoring business processes and compliance concerns by using an event-model from which we generate rules for a complex-event-processing engine. However we do not fully integrate our solution into the described systems above and use and event-model that describes our monitoring rules. We believe that it is possible to create a model-to-model transformation creates instances of our event-model from the compliance-models described in the articles above.

We base the implementation of our approach on the events emitted by business application-servers and Web-Service engines however our approach is not limited to

this event-sources. It is possible to also use events from other event-sources like e.g. data-base triggers or SNMP-alarms, when an according interface, an event-listener, is created. We collect these events from various event-sources and use them to monitor what is going on in a business-application by applying Complex-Event-Processing (CEP) rules.

By using a CEP-engine we are able to obtain high level information from the event-streams that are emitted by the servers. This is necessary since these events normally contain low level data which is very technical and detailed, but most of time not interesting for businesses or compliance checking. These low-level events however also provide the high-level information business are interested in, e.g. the stock of a depot or the business volume for the last week. This data needs to be extracted from this event-streams. For this purpose we introduce an event-model, which we use to model the necessary rules for the CEP-engine. From this model we generate the rules for the CEP-engine that allows us to do live monitoring as well as run-time checking of compliance-concerns. This works the way that, whenever the conditions of an event-rule are met, it fires and produces a new event that contains data from the original events and some possibly new data like e.g. aggregations. We then display the data we gathered in a web-application user-interface that is capable of providing different views for different users.

The rest of the document is structured as follows: In the second chapter we give a detailed description of the problem. Chapter 3 describes the architecture of our approach and shows how CEP for monitoring business processes works in principle. In Chapter 4 we present details of the implementation of our approach. The related work and existing solutions are presented in Chapter 6 and we conclude and give an outlook on possible future work in Chapter 8.

CHAPTER 2

Context and Problem Description

As explained before, business processes and service-oriented architecture (SOA) are nowadays often used concepts that are widely employed in more and more systems. This thesis addresses challenges of business activity monitoring in service-oriented architectures. Before we explain these challenges in detail, let us briefly explain the context of this work.

2.1 Service Oriented Architecture

The systems that make use of SOA have according to Sanders, Hamilton, and MacDonald [27] various characteristics:

- **Integration / Orchestration**: Simple services can easily orchestrated to processes, thus being able to solve complex tasks.
- **Interoperability**: Services have system-independent interface with a specified functionality that is stated in a service-description but users need not take interest in how the service is implemented. They support loose-coupling.
- **Distribution**: Orchestrated services can be distributed across different servers on different continents and still be tied together.

These characteristics lead to series of advantages and disadvantages. While it is a big advantage that services can be distributed in various places, this directly leads to concerns regarding security and availability of the service. For an orchestration to function it is crucial that at least one instance of each service is available, that provides the necessary functionality. Because of the loose-coupling it is also possible to have more than one service that implements a specified functionality thus providing ways of reducing the down-times and providing a high availability. One example for a service-environment where multiple services can exist, that provide the same interfaces, is the OSGi-standard.

On the other hand, distributing services across networks or using services from business partners leads to a number of ranging from a variety of concerns regarding security to other concerns like monitoring, which is of particular interest in a SOA-based environment. Since businesses have the necessity to know how they are performing and where they can improve, as well as detecting any kind of malfunctions. Since this task is particular complex, it lead to the development of business activity monitoring (BAM) where we already gave the definition of [21] in the introduction.

This thesis addresses a number of challenges for BAM in SOAs which are explained in detail in the next section.

2.2 Challenges of Business Activity Monitoring

Business activity monitoring tries to provide enterprises with all the relevant information they need. This ranges from information for different stakeholders to different representations of information and varying detail of the information. The already mentioned definition also names the most important characteristics. BAM solutions do not only provide business data, but provide it in real-time. Additionally the provided information is gathered from multiple internal and external sources, which leads to a “broader and richer view of business activities” [21].

These attributes lead to a number implications for such a monitoring system. For

example it requires flexibility and interoperability, in order to being able to use data from various sources and display the gathered data to the varying stakeholders filtered in a fashion, so that every stakeholder is able to see the things that are important to her.

A number of solutions for this challenge exist. In this thesis we follow the event-driven approach: An event-driven approach to such systems, which has already been described in .e.g. [12, 8], seems to be a good fit for these tasks and a lot of the existing commercial solutions use this approach.

The event-driven or message-based paradigms naturally support the asynchronous, loose connection of systems that is required to integrate the different data-sources. Furthermore it is possible to perform complex event processing (CEP) on the resulting event-streams in order to process the events at run-time. Through this, the abstraction-level of the monitoring data can be increased.

For example is it possible to extract the daily business volume that was generate from a specific business process, which is more interesting to the responsible manager than how many instances of this business process were started and did or did not complete successfully in specific time. This however is very interesting for the process-designer and the responsible software-engineer.

Another key feature to BAM-systems is, that they are able to easily reflect changes in the monitored systems and they provide simple methods for obtaining the needed information.

So far, however, low-level configuration and integration of different technologies for business process, CEP, and BAM, was necessary to reach these goals. The task of implementing and managing a system based on the described approach, is, due to the number of involved sub-systems that have to be kept up-to-date, quite complex and error-prone. Also, when using this approach, it is hard to present the events and event rules for BAM in a way that is suitable for the various stakeholders that are using a monitoring system. Changes imply changing multiple of the involved systems and often programming efforts in low-level languages (like event processing languages and programming languages) are needed to perform a change. Hence, the resulting sys-

tems tend to be inflexible with regard to certain kinds of changes. However there exist systems that provide simple means of configuring the data that is monitored, but these systems are almost all tightly coupled to the application server they are shipped with and can therefor not be integrated into existing systems. In addition, due to the nature of these systems, they sometimes complicate the integration of other event-sources that are not part of the application server and product palette.

2.3 Main Contributions

To address the challenges, summarized before, this thesis proposes a combination of SOA, CEP, BAM, and model-driven software development. By using a model-driven approach, we believe that we can improve the flexibility of such a system and also provide a higher level of abstraction for the configuration of what data is monitored. A defined event-model eases the way how the monitoring rules can be updated and changed. It also provides a layer of abstraction above the language that is used to define these monitoring rules. Independently of the language the system uses for the rules. It does not matter if these rules follow an event- condition-action style like in [36] or an SQL-style approach like it is used by Esper [13] and other CEP-implementations. However it is possible, that this approach is more complex than an implementation without a model-based background. Also our model does not address the matter of visualising the gathered data in any other form then multiple textual views.

CHAPTER 3

Solution Overview

This section provides an overview of the functionality provided by the system we describe and implemented in this thesis. The goal of the system is to provide a possibility to provide a live view on a business-process system. One goal of our solution is that it should also work with systems already existing and not only newly developed systems. Another goal of our solution is to provide the possibility of doing constraint checking for business-processes. Thus being able to detect if a process has violated system constraints is also necessary. An application for this could be a CreditApproval-application (similar to the one described in 5.1) where all loans above a specified threshold have to be approved by more then one person.

Our approach uses Model-Driven-Development(MDD) for various reasons:

- **Event Model**: It is easier to create an event-model or adapt an existing for the use with a new business-application than always writing a new monitoring application from scratch.

- **Modeling of event-rules**: Since our meta-model describes not only basic events that are emitted by a business-application, but also composite events, we can generated the rules for the CEP-engine automatically.

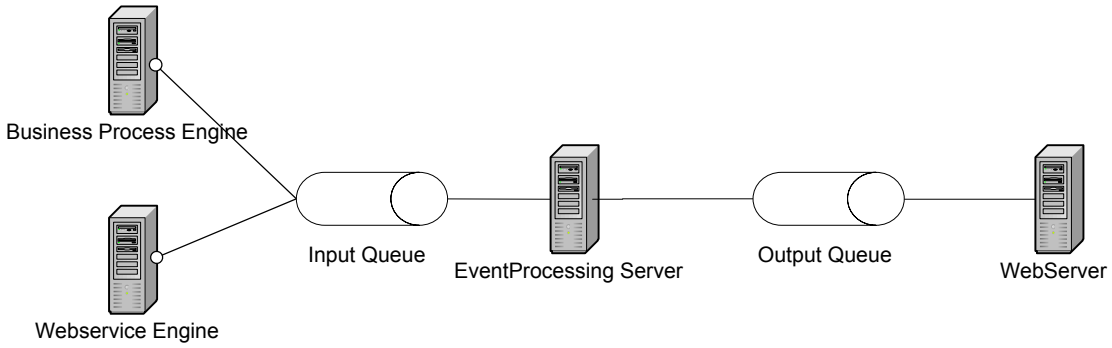

Figure 3.1: Architecture Overview

- **Customization of the user-interface**: Since large parts of our user-interface are described in, and generated from the model, it is easy to customize the UI for the changing needs of different applications and businesses.

- **Easier integration**: Use a model-based approach makes integration of this monitoring system into model-driven business-process solutions like MORSE easier, because the compliance concerns in these models only need to be transformed into an instance of our event-model.

While other approaches use business processes itself for monitoring [2], we designed our system in three basic components that are shown in Figure 3.1. This way the components can be run on different servers if necessary, so that the only extra load for the business-process server are the events it emits, which are collected by the event-listeners and sent to the next component.

3.1 Event-processor

This event-processor component then collects all incoming events and gathers information from the event-stream. For this purpose we use a complex-event-processing (CEP) tool named “Esper” [7] rather then implementing a this functionallity ourselves. Esper works on the basis of an Event Processing Language (EPL) that describes event-rules,

that can be used to describe events and event-patterns which are of particular interest. Because of this we only have to describe generate these event-rules from our model and do not have to generate any other code for this component.

Whenever the conditions of a rule are met, it fires. This result in a new event, which is sent to an output message queue. From there it is picked up by the monitoring application and possible other tools that make use of this information. While the monitoring application we created and describe in the next section displays all the events, other applications might react only to specific events. Possibly event by automatically changing the way the business-process server is working. Since the information exchange between the event-processor and other components is message-basic. It is possible to deploy the monitoring application (and potential other applications) on different machines, as long as the can connect to the message-queue.

3.2 User Interface

The user-interface is the "monitoring application" already mentioned before, which provides the user with information about what is happening in the system. In our implementation, it shows business-data as well as technical-data from business-process and Web-Services and other possible event-sources. The main-page of the application is a "Dashboard" which is shown in Figure 3.2.

Dashboard

The dashboard shows the most important information in one screen. The dashboard shows a live overview of the occurring events by showing a stacked-column-chart on the left hand side. This chart shows the active processes over the last seconds, where every color describes a process and the height of the column shows the number of instances of the according process. Underneath it shows the maximum number of concurrently running process instances and the time-stamp when that system-load occurred.

On the right, a pie-chart shows the state of process-instances that where running within a given interval time from the current time-stamp and before.

Below the two charts the last x events of the views we defined are listed. We describe views in the next section and the details of the debug view and business view in Sections 3.2 and 3.2.

Like the charts, these lists are updated every three seconds to provide a life view on the system. By clicking an event the user can navigate to the event-details in the according view, where all necessary objects are preselected to show the clicked event in detail. Figure 3.2 shows a screen-shot of the dashboard we implemented for our monitoring application.

The dashboard has not been implemented as part of the model (and therefor is not generated from it). Since it is a part of the application, it needs to be completely customized for each specific system for which our solution is used. For example, in a financial environment, it might show transactions and transaction-sums, while an enterprise resource planning system has other key-performance-indicators (KPI).

Views

The data displayed by the monitoring application is organized in views. Each view shows events from specific event-groups. The instances of the event-groups are listed in a column on the left, where they can be filtered and ordered. When selected one or more of the instances, the events of the instances are loaded into the table in the center. Then one can select events that are displayed in detail on the bottom of the screen.

TechnicalView

For example the technical-view that is described in the next section, shows events from the event-groups “BPELProcess” (and its child-group “BPELProcessInstance”) and “WebServices” (and its child-group “Operation”). These groups contain all basic-events that are emitted by the business-process-engine. This view is used mainly for

debugging purposes. It does not display any high-level information on the processes.

Figure 3.3 gives an idea on how the technical-view typically looks like.

BusinessView

This view displays only high-level information. For example this can be if process instances complete successfully or not. This could also be an event if a specific process fails more often then it should. It is also possible to display a message whenever a process like the CreditApproval-process that is described in 5.1, finishes with a positive return-message although the number of approvals necessary was not met.

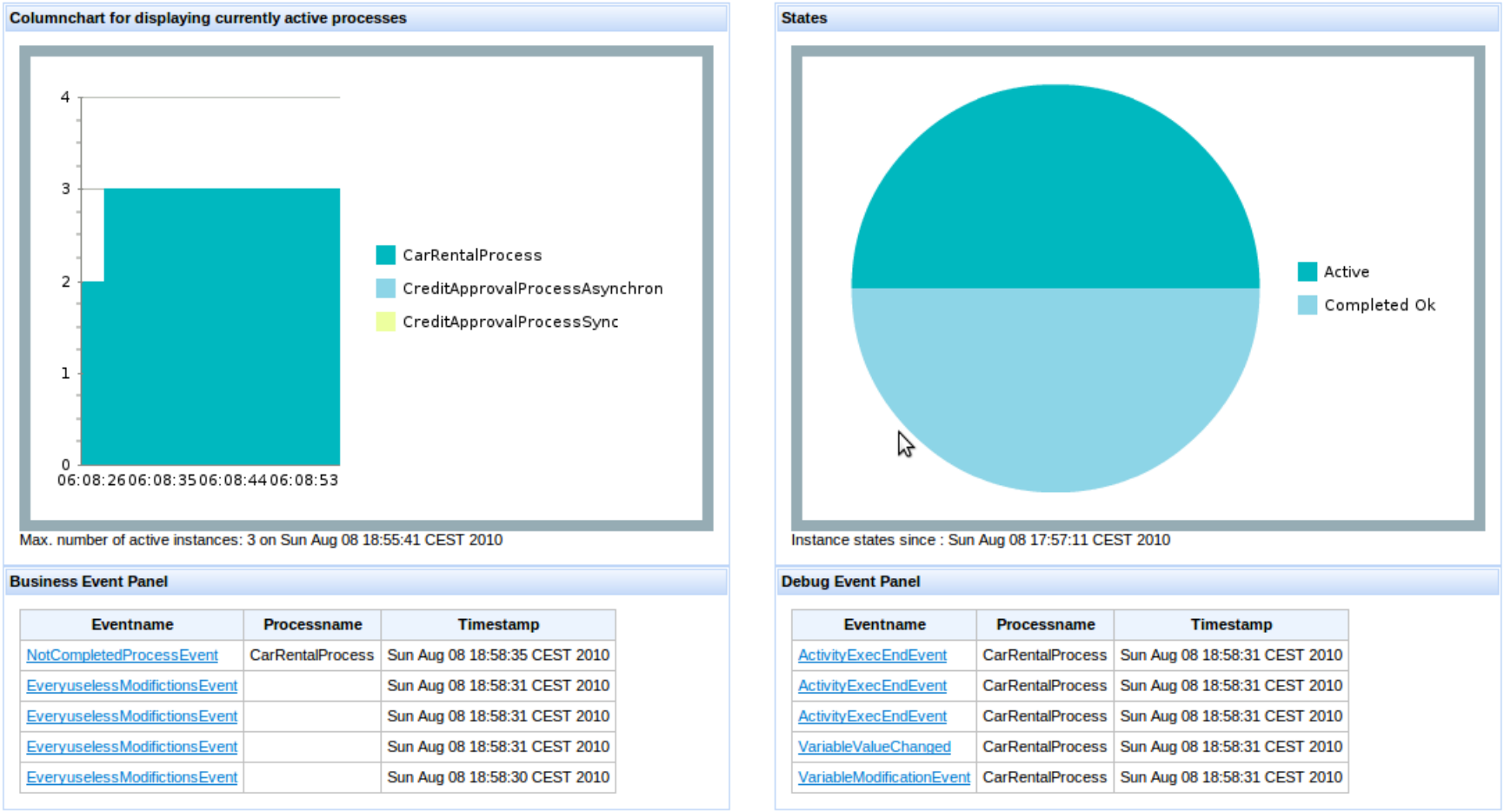

Figure 3.2: Screen-shot of the live dashboard

DashBoard

BPELProcess

InstanceId	NewState	Timestamp
		From To
ProcessName: CarRentalProcess (1)		
251	Completed Ok	Sat Jun 19 08:59:11 GMT 20
ProcessName: TestProcess (2)		
252	Completed Ok	Sat Jun 19 08:55:08 GMT 20
253	Completed Ok	Sat Jun 19 08:55:43 GMT 20

EventType	EventName	ProcessName	InstanceId	ServiceName	OperationName
All					
eventengine.model	ActivityExecStartEvent	CarRentalProcess	251		
eventengine.model	VariableValueRead	CarRentalProcess	251		
eventengine.model	VariableReadEvent	CarRentalProcess	251		
eventengine.model	VariableReadEvent	CarRentalProcess	251		
eventengine.model	ProcessMessageExchangeEvent	CarRentalProcess	251		
eventengine.model	ProcessMessageExchangeEvent	CarRentalProcess	251		
eventengine.model	VariableValueChanged	CarRentalProcess	251		
eventengine.model	VariableModificationEvent	CarRentalProcess	251		
eventengine.model	ActivityExecEndEvent	CarRentalProcess	251		
eventengine.model	ActivityExecEndEvent	CarRentalProcess	251		
eventengine.model	ActivityEnabledEvent	CarRentalProcess	251		
eventengine.model	ActivityExecStartEvent	CarRentalProcess	251		
eventengine.model	VariableModificationEvent	CarRentalProcess	251		
eventengine.model	VariableValueRead	CarRentalProcess	251		
eventengine.model	VariableReadEvent	CarRentalProcess	251		
eventengine.model	VariableReadEvent	CarRentalProcess	251		
eventengine.model	VariableModificationEvent	CarRentalProcess	251		
eventengine.model	ActivityExecEndEvent	CarRentalProcess	251		
eventengine.model	ActivityEnabledEvent	CarRentalProcess	251		
eventengine.model	ActivityExecStartEvent	CarRentalProcess	251		

WebServices

Event Details

VariableModificationEvent_s__newValue:	<?xml version="1.0" encoding="UTF-8"?><parameters><bookResponse xmlns="http://carrental.example.eventengine" xmlns:ns="http://carrental.example.eventengine" xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa="http://www.w3.org/2005/08/addressing"><return>confirm: LuxuryCarRentalProvider:Mercedes S350:364.5</return></bookResponse></parameters>
VariableModificationEvent_s__variableName:	rentalServicePartnerLinkResponse
variableName:	rentalServicePartnerLinkResponse
VariableModificationEvent_s__instanceId:	251
VariableReadEvent_s_a__type:	dataHandling
processName:	CarRentalProcess
instanceId:	251

Figure 3.3: Example of a generated view page

CHAPTER 4

Design and implementation details

In the following sections we first give an overview of the architecture of our system and then describe the different parts of the implemented system in detail.

We use Model Driven Architecture(MDA) [28] to be able to adapt the system fast and easily to different Business Process Engines and interfaces and event-models these engines use. For this purpose we use the Eclipse Modeling Framework(EMF) which "is a modeling framework that exploits the facilities provided by Eclipse" [29]. This framework has a self-describing meta-meta-model named "Ecore" which the authors of the book "EMF: Eclipse Modeling Framework 2.0" describe as "The model used to represent models in EMF [. . .]" [29]. Naturally we use Ecore to define the event-meta-model for our system, which is described in Section 4.1.

In our implementation we created a model for the business-process engine Apache ODE [10] which executes business processes that are defined in the the WS-BPEL standard [24]. Based on the above meta-model we use another project of the EMF called Xpand [19] to generate source-code and event-rules for the complex event processing engine Esper [7]. his tool which was originally developed by the openArchitectureWare project which recently became a part of the EMF. It enables the user to create templates which can generate any kind of source-code from UML and Ecore meta-models.

Figure 4.1 gives an overview of the different subsystems and shows the message

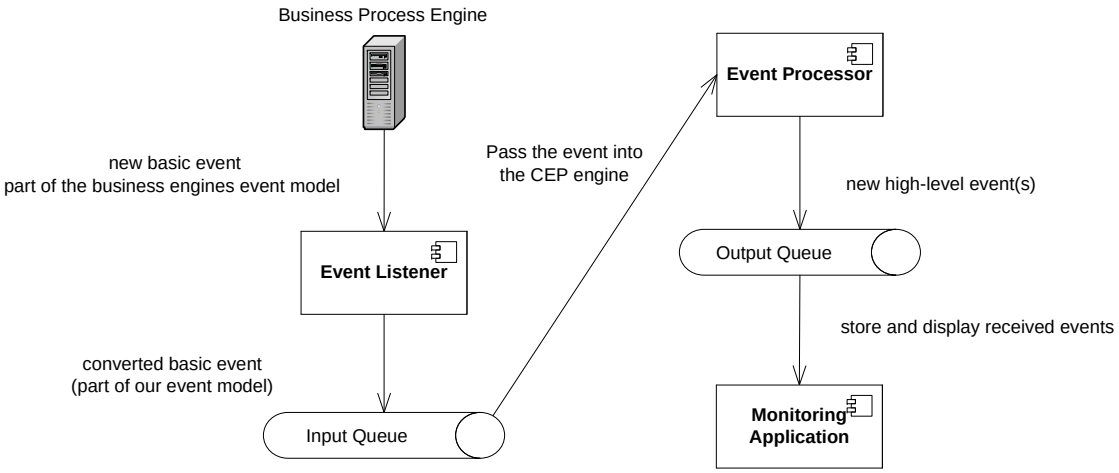

Figure 4.1: Architecture and Message Flow

flow between the different components, which are connected by message queues.

In Section 4.2 we describe how we gather the events from different event sources and then collect them in a message queue [23].

The events are then consumed by the event processor (Section 4.3) which makes use of Esper [7] to extract more interesting complex events from the event stream. For example an event can be generated whenever the run-time of a process instance exceeds a threshold or warnings can be generated when a violation of business constraints has occurred at run-time. Please refer to Section 4.1 where composite events are described in detail.

The resulting business events are then forwarded into an output queue where they can be consumed by various kinds of applications. We also use Xpand to generate a monitoring application (Section 4.4) that provides different views on the monitored data thereby making it possible to address the diverging needs of different users. For example a sales-manager is not interested in specific exceptions and error messages but wants to see the amount of sold goods for a given time-frame or a marketing agent needs figures on how many customers started a process but did not finish it, while the programmer who is responsible for finding and fixing any bugs needs a detailed view of what happened in the system.

The system uses Apache Tomcat [11] as run-time environment, which is also used to deploy the business process engine, services for our example processes, and the monitoring application which is built using the JavaServer Faces (JSF) technology. JSF is described by [13] as "[...] a standard Java framework for building user interfaces for Web applications."

4.1 Meta Model

The most important part of the systems meta-model describes the event-model and the event-meta-data. As one can see in Figure 4.2 events are split into "BasicEvents"(basic events) and "HighLevelEvents"(composite events). The first are events emitted from the business process engine, while the later are more complex events that are composed of "BasicEvents". All events inherit the following properties from event base-class:

- **name**: this events' name
- **eventtypes**: a set of event-types this event has
- **metadata**: a set of meta-data-instances which contain the events properties

For displaying the events in the monitoring application, the system specifies views which directly correspond to single JSF pages. This views display the events by listing one or more event-groups, which in turn, are an aggregation of the events from one or more event-types. For example consider an event ProcessStartedEvent of event-type ProcessEvent. This event is display in a view called "ProcessView", that groups the events first by business-process and then by the process-instance.

Basic Events

Basic events correspond to the events that are emitted by the business process engine, which can be JavaBeans or other simple forms of property lists. In our meta-model we decided to follow the ODE event model and use the event-classes as described in Figure

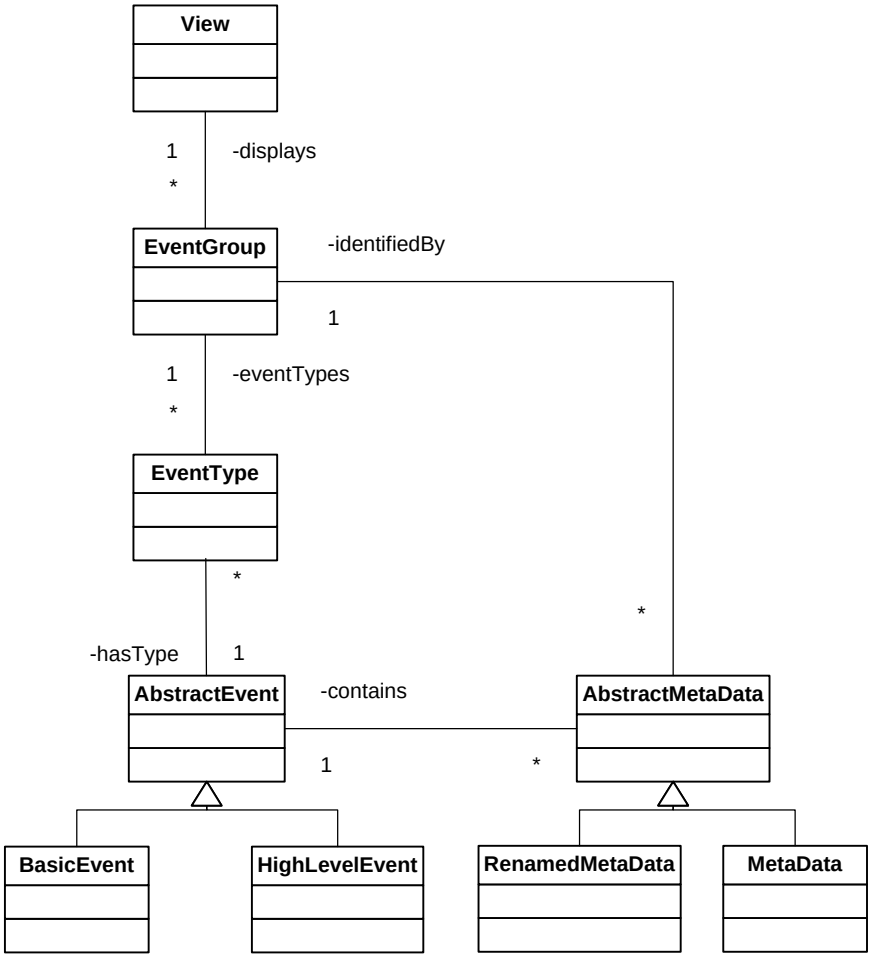

Figure 4.2: Event Metamodel: Overview

4.3. This event-model has a high detail and not all other business-process-engines expose events in the same detail as ODE does. Still we decided to keep these events as they are in ODE because this fine distinctions might be useful for e.g. model-to-model transformations where it can be used to implement transformations that behave differently for different kinds of events. For example one could specifically drop all DataHandling-events or add additional debugging information to instances of the Exception-event-class.

The following list shows what basic-events are supported by our meta-model and describes in short words when they are occur in the system:

- **CreateEvent** is emitted when a new process-instance is created.
- **StartEvent** is emitted when a process-instance is started or a Web-Service is executed.
- **MessageEvent** is emitted whenever a process-instance or Web-Service communicate with one or the other.
- **StateChangeEvent** is emitted whenever a process-instance or a Web-Service have a state transition.
- **DataHandlingEvent** is emitted when a Variable, Correlation, or Expression is used.
- **StopEvent** is emitted whenever a Web-Service-execution or a process instance finish.
- **Exception** is emitted whenever out of the ordinary things happen.

For each basic event a JavaBean™is generated, which has an additional constructor, that takes all event-properties as arguments. Although the different types of basic-events are not used in our system at the moment, they are kept for possible future uses and model-transformations.

Example 4.1.1. *Assume the bpel-process "CarRentalProcess" (described in Section 5.1) is already deployed on an Apache ODE-server. When this process is started the server emits a series of events, which are modeled as instances of the different basic-event meta-model classes:*

1. *ProcessInstanceCreatedEvent*
2. *ProcessStartedEvent*
3. *ScopeStartEvent: This is sent when the default scope for this process is started.*

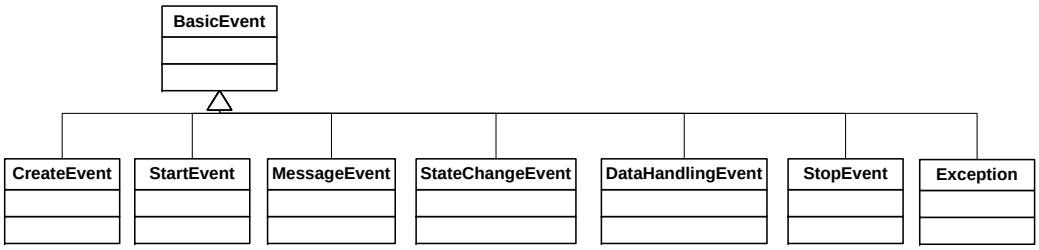

Figure 4.3: Event Meta Model: Basic Events

4. *ActivityEnabledEvent: This event is sent when the outer Sequence-activity of this process is created.*

5. *ActivityExecStartEvent: This event is sent when the execution of the outer Sequence-activity of this process is started.*

6. *ProcessInstanceStateChangeEvent: This is sent whenever the process changes its state. Under normal circumstances the first event describes the transition from "NEW" to "READY" and the second one, which follows shortly after, from "READY" to "ACTIVE".*

7. *. . . (depending on the process-activities)*

8. *ProcessMessageExchangeEvent*

9. *ActivityExecEndEvent: This is the end-event for one of the activities inside the outer Sequence. In this case it is propably the end of the "Return" activity.*

10. *ActivityExecEndEvent: This is the end-event for the outer Sequence-activity.*

11. *ProcessInstanceStateChangeEvent: This is sent when the state transition from "ACTIVE" to "Completed" happens.*

12. *ProcessCompletionEvent: This event is sent, when the process completes without failure.*

MetaData classes

Before discussing HighLevel or composite events in the following Section 4.1 we want to explain the `AbstractMetaData` -class and its subclasses from the meta-model. All event-classes inherit a reference to a list with zero or many instances of meta-data-classes from the abstract Event-class. The meta-data-classes are used to model event-properties in the meta-model.

Example 4.1.2. *We want to take a closer look at the "ProcessStartedEvent" from Example 4.1.1. This event has a set of instances of the class MetaData, which is described in Section 4.1, associated with it. These instances describe the properties of the event:*

- *Name:* ***instanceId***
  Type:`long`
- *Name:* ***processName***
  Type:`String`
- *Name:* ***id***
  Type:`String`
- *Name:* ***type***
  Type:`String`
- *Name:* ***timestamp***
  Type:`java.util.Date`

AbstractMetaData

This class is the abstract super class for the two other meta-data-classes that are described in Figure 4.2. We introduced this class with the RenamedMetaData-class when it became apparent, that it is necessary to be able rename meta-data objects. The RenamedMetaData-class is described in Section 4.1.

This class holds all the common properties of meta-data objects. The first two of the following properties are also the most important ones. Both properties are represented as Strings, in case of the type-property - the fully qualified Java-class-name.

- **name**: the property-name
- **type**: the property-type
- **showAlways**: used for the monitoring application - if true this properties column is by default visible
- **filter**: This is a string that is directly parsed into the where-clause of the Esper-rules. For more information on the where-clauses see Section 4.5.
- **event-name**: This property is used by the Xpand-template to store the event-name of this meta-data during code-generation.
- **repeat**: Like the event-name property before, this is used by the Xpand-template. It stores the number of repetitions, when the meta-data instance is part of a composite event that includes a Repeat-event (Section 4.1).

MetaData

Metadata is the class that describes event-properties. Every BasicEvent has a varying number of properties like e.g. process-name, time-stamp, instance-identifier or variable-name. The only property that is not inherited from `AbstractMetaData` , is the `additionalIdentifierWhereClauses` property, which is used during code generation only. It is used in special cases only where the composite-event has a form like `(A or B) followed by C` which will be explained in Section 4.1.

RenamedMetaData

The instances of RenamedMetaData usually are children of composite events. They change the Esper-rule for an event by adding a new additional name for an already

existing instance of MetaData.

This is feature is useful because composite events always have some properties with the same name, like e.g. a time-stamp property which has every basic event. To be able to select all properties, the system automatically assigns unique names to all properties. This renaming is described in detail in the paragraph "Property renaming" of Section 4.5. This has the result the a property `processName` which is defined in nearly all events, has a slightly different name in every composite-event and therefor would not be listed in the one column in the monitoring application but in a seperate column per composite-event. By renaming this property this can be prevented.

This meta-data-class can also be used to extract more interesting data from the events. E.g. one could define a `start-timestamp` property and a `stop-timestamp` property for every composite event by renaming the timestamp of the event that has occur first (start) and the event that triggers the composite-event to fire(stop). This may be implemented in the future to happen automatically but currently this is not supported by the system.

- **renameProperty** is a reference to the property that is renamed.
- **renameEvent** is a reference to the parent event of the renamed property.
- **DisplayName** is the new name-String that is added to the Esper-rules and the columns in the monitoring-application.

Composite Events

The CEP-engine is the component that enables our system to extract relevant information from the stream of events that we receive from the web-services and business process engines.

To enable the creation of these events, we have modelled the most important operators and constructs provided by Esper into HighLevel-events. By defining these events in the model it should be possible to construct almost any composite-event that is sup-

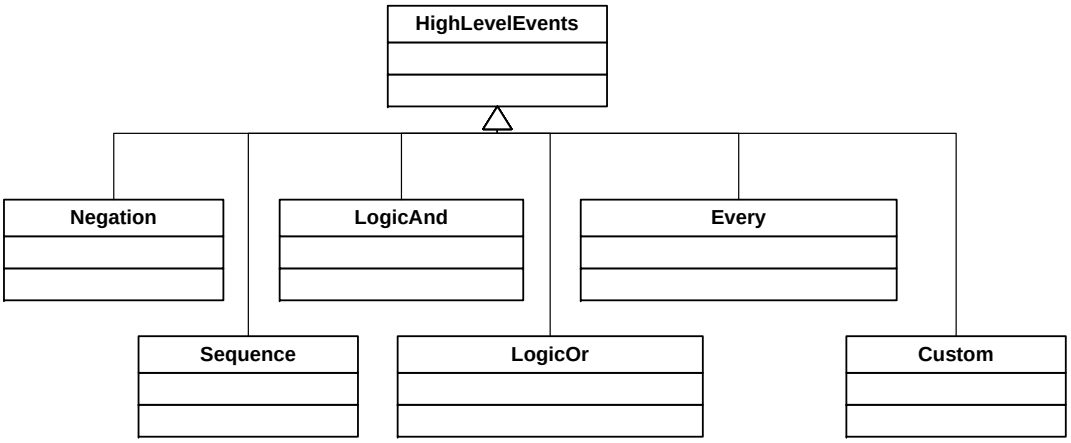

Figure 4.4: Event Meta Model: Highlevel Events

ported by Esper without having to learn the syntax and semantics of Espers Event Processing Language (EPL) or writing any code by hand. Instead one can edit the model in one of the many editors that are available for this task and does not have to use a text-editor for also creating the EPL-statements that would otherwise be necessary.

Sequence

The sequence probably is the most important composite event which will be used most often. It implements event A "followed by" ($\rightarrow$) event B. This means it fires for all events A and B where A occurred before B regardless of how many other events fired in between.

Example 4.1.3. *Given a sequence of events A1,B1,B2,C1,A2,B3,A4,C2. Then* `A → B` *results in only one event: A1B1.*

To create a composite-event that fires on every sequence of events A and B that occur, we need an additional composite event that specifies exactly that:

Every

The second of all composite events we introduce here, is one that is subsequently used in all examples regarding composite-events and event-rules. The "every"-keyword ($\forall$) specifies that the rule matches not only for the next event (say a ProcessStartedEvent) that follows but for all events of that type that follow. This is best visible when compairing Example 4.1.5 and Example 4.1.3.

Please note that the $\forall$-keyword has higher precedence then the Sequence, which is illustrated in Example 4.1.4.

Example 4.1.4. *Given a sequence of events that is similar to Example 4.1.3 but slightly modified: A1,B1,B2,C1,A2,A3,B3,A4,C2. The rule* $\forall$`A` $\rightarrow$ `B` *results in the following events: 1. A1B1, 2. A2B3, and 3. A3B3 . This is because the rule starts to look for other events as soon as it receives another A-event.*

The rule $\forall$`(A` $\rightarrow$ `B)` *only starts to look for events again, after it also received a B event. This results in the following events: 1. A1B1, and 2. A2B3,*

This example is visualized in Figure 4.5 as a UML sequence diagram.

Example 4.1.5. *Given a sequence of events A1,B1,B2,C1,A2,B3,A4,C2. The rule* $\forall$`A` $\rightarrow$ $\forall$`B` *results in the following events: 1. A1B1, 2. A1B2, 3. A1B3, and 4. A2B3*

Obviously a composite-event that contains a $\forall$-operator does not stop looking for new events until the whole system is stopped. As the $\forall$-operator controls when the composite-event starts for new events again, it has to be used very carefully in order not to waste ressources on the server that hosts the event-processing. One of the ways to control when the search for new events is started or aborted is by using additional operators like the following "LogicAnd" or "Negation", which are described in subsequent sections.

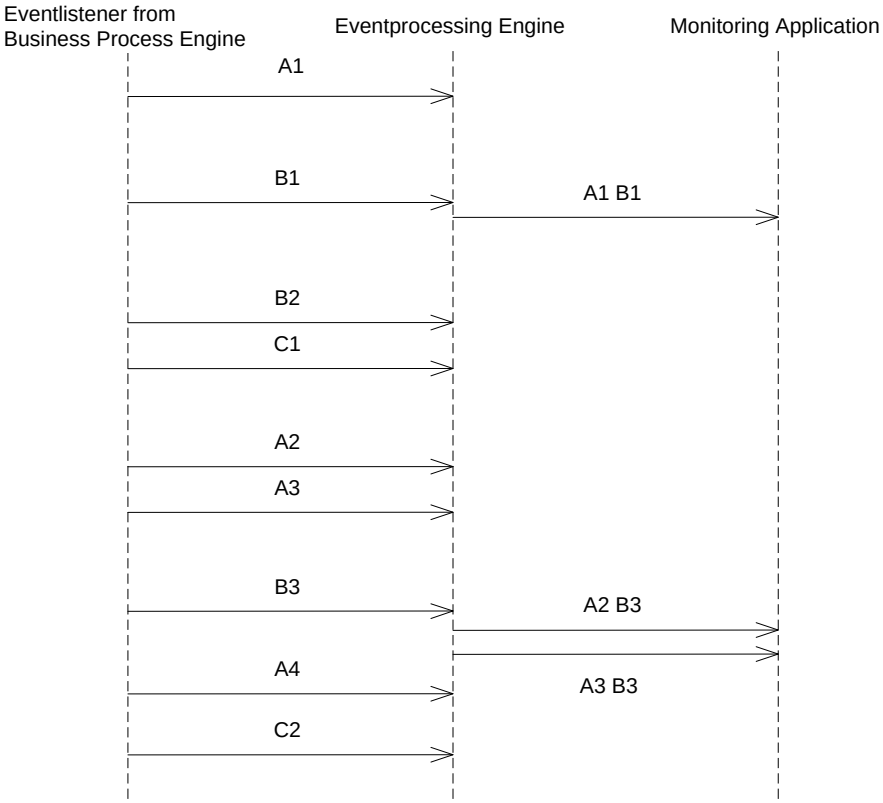

Figure 4.5: Visualisation of events for the rule $\forall A \rightarrow B$. When visualizing the rule $\forall (A \rightarrow B$ one has to remove the A3 B3 event (the last arrow in the right column).

LogicAnd

The composition using the "and" ($\wedge$) is defined in Esper [7] as an event A and an event B that have to occur at the same time. This means that there must not be another event C between A and B. The order of A and B is not important.

Example 4.1.6. *Given the same sequence of events A1,B1,B2,C1,A2,B3,A4,C2 as in Example 4.1.5. Then* $\forall A \wedge B$ *results in the following events: 1. A1B1, 2. A2B3, and 3. B3A4*

Please note that because there is no ordering, the properties of event A cannot be used to filter event B and vice-versa.

LogicOr

Obviously like the “and” before, the “or” ($\vee$) does not impose an ordering on its events and so properties of event A cannot be used as filter for event B and vice-versa either. This rule fires for every event A or B that occurs.

Example 4.1.7. *Given the same sequence of events A1,B1,B2,C1,A2,B3,A4,C2 as in the examples before (Example 4.1.5, Example 4.1.6). Then* $\forall$`(A ∨ B)` *results in the following events: 1. A1, 2. B1, 3. B2, 4. A2, 5. B3, and 6. A4*

Repeat

As the name says, this composition requires the events that follow to occur more than once. It has four properties:

- **event** is the event A that has to occur more than once.
- **lowerBound** is the number of times an event A has at to occur at least, before this pattern can fire.
- **upperBound** is the number of times after which the pattern stops looking for an event A
- **until** specifies an event B until which the pattern looks for an event A, and stops as soon as B occurs.

Depending on its properties values, it has three different ways to operate:

1. If lower- and upper-bound are equal and the until property is empty, then exactly x events A have to occur before this pattern fires. (Where x equals lower- and upper-bound).
2. If no bounds are specified but the until property is set to event B, then the pattern takes all events A and fires when event B arrives.

3. If bounds are specified and the until property is set to event B, the event pattern looks for a number of at least lower-bound events A, but stops looking for events A as soon as the number of A-events reaches the upper-bound. It fires when an event B arrives.

Example 4.1.8. *Given the same sequence of events A1,B1,B2,C1,A2,B3,A4,C2 as in the examples before (Example 4.1.5, Example 4.1.6). Then* `∀[2]B` *results in the following events: 1. B1,B2 2. B1,B3 3. B2,B3*

Example 4.1.9. *Given the sequence of events A1,B1,B2,C1,A2,B3,A4,B4,C2. Then* `[3 .. 5]B until C` *results in the following event: 1. B1,B2,B3,B4,C*

Example 4.1.10. *Given the same sequence of events A1,B1,B2,C1,A2,B3,A4,C2 as in the examples before (Example 4.1.5, Example 4.1.6). Then* `B until C` *results in the following event: 1. B1,B2,C*

Negation

The negation (¬) is not only a useful tool but also needed for being able to express even simple examples. It is possible to add a time-frame, where the pattern fires if the event does not occur within a given time.

Example 4.1.11. *Given the same sequence of events A1,B1,B2,C1,A2,B3,A4,C2 as in the examples before (Example 4.1.5, Example 4.1.6). Then* `∀A → B ∧ ¬A` *results in the following event: 1. A1,B1 2. A1,B2 3. A2,B3 Please note that* `A → B` *would include the additional result* ***A1,B3****.*

Example 4.1.11 is inspired by Apache ODEs default configuration. When unchanged, ODE has a Variable-Read event which fires whenever a variable is read that shows the variable-name but not its value. By using the rule `VariableModificationEvent → VariableReadEvent and not VariableModification` the variable-value can be displayed by reading it from the variables last modification-event.

The following properties are defined for this composite event:

- **event** is the event A that must not occur for the pattern to fire.
- **timewindow** is the number of seconds the pattern waits before it fires, if event A does not occur.

Custom

This event is our way to enable Esper-experts to define events directly, since we implemented a part of what can be done with Esper, but are not near offering a "complete" list of Esper-features in the model. It uses EPL to define what events are selected and from which event-streams they are selected. This composite-event allows for complete direct control over Esper. However it is necessary to add meta-data instances for this event, so that our monitoring application can display custom events properly.

This meta-data instances are used to generate the according columns in the monitoring application, so that the user can make use of them.

Example 4.1.12. *For example Esper supports the aggregation and grouping of values directly in its EPL statements. The following code that, illustrates a possible use for a CustomEvent, is taken from the Esper documentation*[1]*:*

```
select symbol, sum(price), avg(volume)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000 and avg(volume) < 500
```

For using code within our system it would be necessary to rename the columns and then specify meta-data properties accordingly.

[1] http://esper.codehaus.org/esper-3.5.0/doc/reference/en/html_single/index.html

```
if (bpelEvent instanceof ProcessInstanceStartedEvent) {
            ProcessInstanceStartedEvent bpelProcessEvent = (
                ProcessInstanceStartedEvent) bpelEvent;
            ProcessStartedEvent event = new ProcessStartedEvent();
            if (bpelProcessEvent.getProcessId() != null)
                event.setId(bpelProcessEvent.getProcessId().toString());
            if (bpelProcessEvent.getProcessName() != null)
                event.setProcessName(bpelProcessEvent.getProcessName().
                    getLocalPart());
            if (bpelProcessEvent.getProcessInstanceId() != null)
                event.setInstanceId(bpelProcessEvent.getProcessInstanceId());
            if (bpelProcessEvent.getTimestamp() != null)
                event.setTimestamp(bpelProcessEvent.getTimestamp());
            if (bpelProcessEvent.getType() != null)
                this.sendEventToProcessor(event);
        }
```

Figure 4.6: Example for a conversion from a business-process engine defined event to a model defined event.

4.2 Listener

In this section we describe the listener component of our system. At the moment we have implemented two listeners - one for Apache ODE and one for Apache Axis2 [9]. As listeners are part of the meta-model, we defined both in our model. From it we generated a base-class that automatically connects to the input-message-queue and provides a method `sendEventToProcessor(AbstractEvent e)` which sends a new event to the event-processor engine.

Business-process event-listener

The listener for Apache ODE has to be a Java-class that needs to be on the class-path of ODE and is configured in the file `/WEB-INF/conf/ode-axis2.properties` . It has to implement the Java-interface `org.apache.ode.bpel.iapi.BpelEventListener` which defines only one method: `public void onEvent(BpelEvent bpelEvent)` . In this method we convert the `BPELEvent` objects to events generated from our model, which are then sent to the event-processor. In Figure 4.6 a typical conversion is displayed.

```
if (msgContext.getFLOW() == 2) {
            log.info("Service Finished!");
            ServiceStoppedEvent event = new ServiceStoppedEvent();
            if (msgContext.getAxisService() != null)
                event.setServiceName(msgContext.getAxisService().getName());
            if (msgContext.getOperationContext() != null)
                event.setOperationName(msgContext.getOperationContext().
                    getOperationName());
            if (msgContext.getOperationContext() != null) {
                ArrayList<Parameter> parameters = msgContext.
                    getOperationContext().getAxisOperation().getParameters();
                event.setParameterList(parameters);
            }
            event.setTimestamp(new Date());
            this.sendEventToProcessor(event);
        }
        return InvocationResponse.CONTINUE;
```

Figure 4.7: Example for a conversion from a web-service event to a model defined event.

Web-Service event-listener

As we use Apache Axis2 [9] as run-time for our example web-services, we also implemented a listener for retrieving its run-time events. Since web-services only expose methods that are executed and that should be stateless we only need to check if methods are invoked(started), finish(stopped) or ran into an exception. Axis2 does provide for listening on the basis of modules. These modules basically are Java-archive files with a `module.xml` file in the `META-INF/` directory that defines the module[2]. The class that implements the module also has to implement an interface, but Axis2 provides an `AbstractHandler` -class for easier use, which can be extended. One then needs to override `public InvocationResponse invoke(MessageContextmsgContext)` . The `MessageContext` object holds reference to the called service and the parameters of the invocation which can be converted to events from the model. This method should always return `InvocationResponse.CONTINUE` and not thrown an exception - otherwise it would interfere with the execution of the web-service.

[2]For more information on modules see `http://ws.apache.org/axis2/1_5/modules.html`

4.3 Event-processor

The event-processor is the systems component for processing the incoming events and extracting business-relevant information. It is designed to be deployed as a web application although it currently has no user-interface, an interface for managing the esper-rules at run-time could be of interest for future use. The only part of this component, that is generated by the model is the file `/WEB-INF/esper_rules.xml` which is read at start-up. The system reads the event-rules it contains, feeds them directly into Esper and registers a separate instance of `EsperListener` as UpdateListener for each rule.

4.4 User Interface

The user interface we provide for monitoring the business events in our proof-of-concept application consists of a dashboard (described in Section 4.4), different views (Section 4.4), and an event storage which is currently a simple bean that implements the storage interface `IDataProvider` which is generated from the model. This bean listens to the message queue for events from the event-processor and stores them at the moment in memory, grouped by event-groups. The storage interface provides methods for retrieving stored events per event-group, per event-type or all events at once. It also provides a method for retrieving event-group instances where the state variable matches a given state.

This component of the system is implemented as a web-application which can be deployed as web-application archive too. It uses the Apache MyFaces implementation of JSF 1.2 with the RichFaces [26] and the PrimeFaces [30] component libraries.

For each event-group a JavaBean is generated (which is used as a backing-bean) that extends the class `AbstractDataBean` and is tied into the web-application using the xml-configuration component of Spring [18].

Storage

Storing the received events is the task of the `EventConsumer` -class. This bean is configured as a singleton for the whole component and receives the events sent from the event-processor. It extracts the `Map` -objects from the `ObjectMessage` -objects and then stores references to them in an `allEvents` -List as well as in the correct event-groups.

Views

For each view-instance in the model a JSF-page, backing-bean instances of all its event-groups (each view has its own instance of an event-groups bean if a group is displayed in more then one view) and a backing-bean for the view itself (view-bean) are generated. Every view consists of three main areas that are shown in Figure 3.3. On the left is a panelbar with a panel per event-group that is displayed in this view. In the figure the panel for the group "BPELProcess" is active, while the group "WebServices" is minimized. Next to the panel-bar is the area where the events are displayed. On the bottom is the area where detail-information for selected events is presented.

The view-instance that is displayed in Figure 3.3 is named TechnicalView and shows events from the event-groups BPELProcess, BPELProcessInstance, WebService and Operation. Since BPELProcessInstance is the child-group of BPELProcess and operation is the child-group of WebService, it contains only two panels for displaying this groups. As depicted in the screen-shot, the view displays a list of all instances BPEL-ProcesInstances grouped by instances of the BPELProcess-group. The columns of the data-table are generated from the event-groups associated meta-data instances (identifiers). Through using an instance of the `< rich:extendedDataTable >` to display this data, it can be filtered and ordered since the component has built-in support for this features. This component also allows for hiding and reordering columns, as well as minimizing grouped data-rows. (The instances of a bpel process can be hidden by clicking on the "-" icon.) By selecting one more event-group-instances e.g. BPELProcessIn-

stances, the events of the selected instances will be loaded into the table in the center of the screen via AJAX. One can select mulitple rows by holding down the Ctrl-key and left-clicking with the mouse.

This table in the center is another instance `< rich:extendedDataTable >` component. It is used for displaying the events of all selected event-group instances. This component uses the view bean as backing bean, which gathers all selected events from the backing beans used by the event-group-panels. This component provides a column for every event-property of all events the view displays, although the columns of event-group identifiers and the column of meta-data items where the `showAlways` -property is set to true are visible by default. When one more events from the data-table are selected (again by holding the Ctrl-key and left-clicking), they are displayed below the data-table, where an instance of the `< rich:panel >` -component is used for every event. Inside the component the event is displayed using a utility-class that is called `EventFormatter` . It implements the `javax.faces.convert.Converter` -Interface which defines `String getAsString(Object)` and `Object getAsObject(String)` methods. Since this class is only used to convert the event-objects to Strings, only the first method is implemented and the second method always throws a `NotImplementedException` . The class supports all beans that are instances of `AbstractEvent` and `Map` -instances that have a rulename-property. It creates a XHTML-table that has no headings and creates two columns per row. For each property it creates a row where it lists the property-name in the left column followed by a ":" and its value in the right column. If the value is an instance of `org.w3c.dom.Node` it uses another utility-class called `NodeConverter` to display the node properly.

Dashboard

As already mentioned in Section 3.2, the dashboard provides a live view on the monitoring data. It shows a chart with currently active processes and the number of its instances, as well as a pie-chart that shows which number of process-instances that started execu-

tion within a given, configurable time-frame are in what execution-state.

Both charts are components that are provided by the PrimeFaces-library [30] and are implemented as flash-components that can be configured for automatic update. Unfortunately the components do only refresh the already existing chart-series(data-sets) and new chart-series are not added during run-time. Because of this a reloading the components using AJAX is necessary.

These charts receive their data via the `DashBoardDataBean` which provides methods for retrieving the last x events of a given view. This bean-class also provides the data for the charts by implementing a mechanism for saving instance-states, as well as a method for retrieving active instances.

While big parts of our sytem are generated from the model, we saw no use in generating a dashboard, which highly differs from each use-case to the next.

4.5 Code generation

In this section we describe how we obtain source code from our proof-of-concept model. We used the Eclipse Modelling Framework(EMF) where every Xpand project has a layout similar to the one depicted in Figure 4.8 with a folder for the meta-model, a template-folder, a folder that contains the workflow-files, and an instance of the meta-model (in Figure 4.8 named `Model.xmi`). This folder-structure results from the fact that EMF uses a workflow-file with the ending `.mwe` where the programmer configures the options of a workflow-execution (code-generation). For example this file contains which templates to apply to which models, if and which code-beautifiers are used and where the output is written to.

The templates with the ending `.xpt` contain the Xpand-code that describes on the basis of the meta-model how the source-code is generated. The `.ext` -files contain Xtend-functions which can be defined by the programmer to be used in the Xpand-code.

In our case we created three Xpand [19]-templates and the necessary Xtend functions for generating Java source-code, Spring bean-descriptions, and event rules for Es-

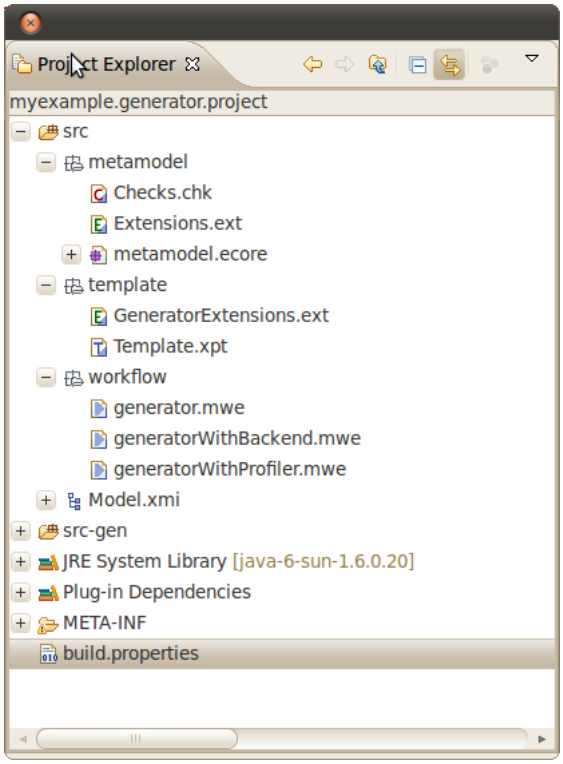

Figure 4.8: Example Xpand-project layout [19]

per. Each one of the three templates is closely connected to a component of the system.

1. The `events.xpt` generates the JavaBeans from the model that are used by the event-listener and event-processor (Section 4.5).

2. The `event-processor.xpt` generates the Esper-rules for the event-processor component (Section 4.5).

3. The `event-consumer.xpt` generates the code for the monitoring application, which includes JSF-files, Spring bean-descriptors, and Java backing-beans (Section 4.5).

Source code

The following subsections describe the source-code that is generated for the instances of the `BasicEvent` -class from event-model (Section 4.5) and how the Esper event-rules are generated for all events defined in the model (Section 4.5).

Model

As mentioned above the JavaBeans for the events from the model are generated by the `events.xpt` . Each bean has properties as described by its meta-data instances with a type defined by the type property of the meta-data instance and a name matching its name property. Furthermore each event-bean has to extend the `AbstractEvent` -class, that defines a logging instance, a `getProperty` -method and an abstract `getEventName-method` . Figure 4.9 shows the listing of the Xpand-code for this purpose.

This file also contains the template for the event-listener base-classes, which have to be extended by the event-listeners.

In this template we also create the utility-class `JMSConnectionFactory` -class, which manages the Java Message Services [23]-connections that are used to connect the single components.

It also generates a Java interface for each event-type in model.

User Interface-code

As already mentioned in Section 4.4 our system generates the biggest part of the user-interface (monitoring application) from the model. This way we can provide an easily adaptable user-interface, that can be configured for displaying multiple views, where each view shows a possibly different type of data.

For this purpose the model has a `View` -class that has only three properties:

- **Groups**: This property is a list of all event-groups this view should list. This means the process lists all events that are associated to these event-groups.

```
1  package «implPackage()»;
2  public class «implClassName()» implements «
      abstractEventClassName()
3          »«IF eventtypes != null && eventtypes.size > 0»
4          «FOREACH eventtypes AS e», «e.fqImplClassName()»«
              ENDFOREACH»
5          «ENDIF»
6            {
7
8      private static String eventName = "«implClassName()»";
9      «FOREACH metadata AS m»
10         private «m.type» «m.name»;
11     «ENDFOREACH»
12
13     public «implClassName()»() {}
14     «IF !metadata.isEmpty»
15         public «implClassName()»(«FOREACH metadata AS m
              SEPARATOR ','»
16             «m.type» «m.name»
17         «ENDFOREACH») {
18         «FOREACH metadata AS m»
19             this.«m.name»=«m.name»;
20         «ENDFOREACH»
21     }
22     «ENDIF»
23     «FOREACH metadata AS m»
24     public «m.type» get«m.name.toFirstUpper()»() {
25         return «m.name»;
26     }
27     public void set«m.name.toFirstUpper()»(«m.type» «m.name») {
28         this.«m.name»=«m.name»;
29     }
30     «ENDFOREACH»
31     public String getEventName() {
32         return eventName;
33     }
34 }
```

Figure 4.9: Xpand-template for generating the event-beans.

- **Name**: This property holds the view's name. This is used for identifying beans, which are automatically instantiated and associated with this view, as well as simple things like the title of the JSF page which is generated for the view.
- Url: This property holds the relative path under which the view is reachable on the server.

The layout of the views which was already described in Section 3.2 consists of a column on the left, where all instances of an event-group (e.g. the event-group "BPEL-ProcessInstance" as a child-group of "BPELProcess") are listed and can be selected. Therefor it is necessary to have a backing-bean for this list of event-group-instances. This backing-bean-class is generated once per event-group. If an event-group is displayed in more then one view, then multiple instances of this bean are created under different names using Spring[18].

For the center-column of the view, where the events of selected event-group instances are displayed, another backing-bean is needed. Here a custom bean is generated for each view.

In order to be able show any data in the user-interface, it is necessary that the events that are collected by the event-listeners are processed by Esper. For processing these events and gathering high-level information from this stream of events we need Esper. As we already described in Section 4.1 we added events to our meta-model for this purpose.

Esper-rules

This section describes the previously mentioned generation of event-rules for the Esper complex-event-processing engine. The Esper-rules are defined in an Event Processing Language (EPL) which is somewhat similar to SQL but has a lot of special features that are designed for working with event-streams.

Just like in SQL every EPL-rule starts with the keyword `SELECT` followed by a list of properties that are selected. After the properties, which can be renamed or where an

```
«REM»This defines the standard xml-clause of events«ENDREM»
«DEFINE event FOR Event»«
        FOREACH getEventTypes() AS currentType»
        <esperrule name="«currentType.fqImplClassName()»">«
        EXPAND subEvent
»       </esperrule>«
        ENDFOREACH
        »«
ENDDEFINE»
«DEFINE subEvent FOR Event»select eventName? as eventName «
        FOREACH this.metaDataTypeNames() AS typeName
        »,«typeName»? as «typeName»«
        ENDFOREACH
        » from «fqImplClassName()
»«ENDDEFINE»
```

Figure 4.10: Xpand-templates for generating the esper-rules for basic-events.

asterisk can be used instead, follows the keyword `FROM` . Then a list of different event-streams can be selected by either defining event-streams yourself (for example using a pattern) or using the name of the classes for which the rule should trigger. In this case it is possible to use the name of super-classes and interfaces as well which can lead to the problem that Esper may not be able to determine if a specific property exists on rule-creation, which causes an error if the property is not followed by a '?'. This marks the property as dynamic and thus unchecked and only resolved at run-time.

In the following section, we explain the generation of EPL-rules for basic-events.

Basic Event rules

As can be seen in Figure 4.10, generating an Esper-rule for a BasicEvent is rather simple. For each meta-data instance the name is put down in the `SELECT` -clause and the event-name is put down in the `FROM` -clause.

While rule-generation for basic-events is rather straight forward, generating syntactically and semantially correct event-rules for composite-events is more complex.

Composite Events

The generation of Esper-rules for composite-events(HighLevel-events) is more complex since they basically have the form of a tree, which consists of complex events along the inner-nodes and basic-events at the leaf-nodes.

Event-rules are only generated for composite-events that are not part of other composite-events. If such an event is needed, we create a second instance of that event and use one of the instances for composition, while the other one remains unused and therefor is generated.

The template for these events starts by overwriting the `subEvent FOR Event` definition from Figure 4.10 with the code from the listing in Figure 4.11. Starting with the `SELECT` -clause this code differs; in addition to selecting the meta-data instances of the event, it selects all the identifiers of all BasicEvents which are part of this composite event. These are obtained by calling the `identifiers()` -method for event. This extension-method then iterates recursively through all child-events selects the meta-data instances and renames them.

Property renaming The renaming is necessary for selecting the properties from the correct event-streams and to ensure that all meta-data instances have unique names. This is done by adding a prefix to the the names that is built from the event-name of the composite event followed by an a unique string and a dot. This String is composed of an underscore and a specific character for each node on path to the meta-data in the leaf-node.

To make this approach work, we overwrite the identifiers-method separately for every composite-event that is defined in the meta-model and use the support for polymorphism built into Xpand. Although this has the drawback, that we have to define new Xpand-templates and Xtend-functions whenever a new composite event is introduced in the meta-model, we prefer this approach over implementing everything within the template directly.

This `identifier` -method also is responsible for setting the `repeat` -property on

```
1  «REM»Overwrite the standard event rule for high level events«
      ENDREM»
2  «DEFINE subEvent FOR HighLevelEvent»
3  select  '«this.name»' as eventName, *, DateConverter.convert(
      current_timestamp()) as timestamp«
4  FOREACH metadata() AS currentdata»
5  ,«getEventNameMeta(currentdata)».«currentdata.originalname()»?
      as «currentdata.name
6  »«ENDFOREACH»
7  «FOREACH this.identifiers(true) AS ident»
8  «IF ident.repeat != null && ident.repeat.size > 0
9  »«FOREACH ident.getRepetitions() AS rep»
10 ,«ident.eventName»«rep».«ident.name»? as
11 «ident.eventName»«rep.replaceBracketPlaceHolder()»«
      separatorString()»«ident.name»«
12 ENDFOREACH »«ELSE»,«ident.eventName».«ident.name»? as «ident.
      eventName»«separatorString()»«ident.name»«
13 ENDIF»«
14 ENDFOREACH»
15  from
16  pattern[
17   «getEventBody()»
18  ]
19 «ENDDEFINE»
```

Figure 4.11: Xpand-template for generating the Esper-rules for composite-events.

meta-data instances that are in the sub-tree of a Repeat-event. This properties are treated as arrays with a number of occurrences depending on the upper and lower-bound of the repeat-event itself. When the value of repeat-property is not null and greater then zero, the property is added to the `SELECT` -clause that often and $[i]$ is appended (where i is the value of the `repeat` -property).

The `FROM` -clause The `SELECT` -clause is followed by the `FROM` -clause that consists of only one pattern, which is defined by another Xtend-function. The method `getEventBody()` is defined for the meta-model class event where it implements the body for all basic events, as well as for all composite-events - again using polymor-

phism.

For basic-events this method returns a string that starts with the event-name of the basic-event and the `name_postfix` . This is the same string that is used within the `SELECT` -clause for prepending the property names. It is followed by a single equals sign which is then followed by the full-qualified class-name of the JavaBean that was generated for this basic-event. After this follows in most of the cases a string, that ties the basic-event to the other basic-events from this composite event by using specific meta-data properties (event-identifiers).

When called, the `getEventBody` -method for composite events first searches for the event-identifiers for this event and then recursively works through the whole tree described by this composite event.

In our implementation the default event-identifier is the first meta-data instance of the first basic-event in the tree of this event which in our model is the `instanceId-property` that holds the unique business-process-instance id.

However normal meta-data instances (not instances of RenamedMetaData) that are listed in the `metadata` -property of the composite events are used as event-idenitifiers instead of the default-identifier when specified. Figure 4.5 shows an example for this use-case. Manually defining the event-identifiers can lead to errors when specified in a composite event where they are not supported by Esper(LogicOr,LogicAnd, . . .). This often is the case when the system cannot obtain a default-identifier automatically.

Although for most of the time it is trivial to obtain the event-identifier, there are cases like Figure 4.5 in which obtaining the event-identifier and adding it to the event-rule can be a more complicated task. In this example using the first event's identifier would lead to an incorrect Esper syntax. The problem is, that the events that are connected by the "or" do not have to occur for the pattern to fire. And thus using their properties to enforce the same instanceId for the following ProcessCompletionEvent results in a run-time error. This problem exists for following the meta-model events: *a*) "Repeat", *b*) "LogicAnd", *c*) "Every", *d*) "Negation", and *e*) the already mentioned "LogicOr" .

When we have obtained the correct event-identifier for the composite event,

```
(EVERY
VariableModificationEvent_s=eventengine.model.
   VariableModificationEvent(TRUE))
-> (
VariableReadEvent_s_a=eventengine.model.VariableReadEvent
(TRUE
AND VariableReadEvent_s_a.instanceId=
   VariableModificationEvent_s.instanceId
AND VariableReadEvent_s_a.variableName=
   VariableModificationEvent_s.variableName)
and ( not VariableModificationEvent_s_a_n=eventengine.model.
   VariableModificationEvent
(TRUE  AND VariableModificationEvent_s_a_n.instanceId=
   VariableModificationEvent_s.instanceId AND
   VariableModificationEvent_s_a_n.variableName=
   VariableModificationEvent_s.variableName))
```

Figure 4.12: Example event-rule for a composite event of type every A → (B and not A)

adding it to the conditions in the brackets that follow the full-qualified event-bean class-name, can still result in run-time errors. This is the case when we modify the rule in Figure 4.5 by adding the clause `VariableReadEvent_s_o.instanceId = ProcessCompletionEvent_s.instanceId` directly after the `VariableReadEvent_s_o` 's definition. The reason for this is that the ProcessCompletionEvent_s is not known at this point.

This can be solved by detecting which basic-events are listed before the basic-event that defines the event-identifier(identifying event) for the composite-event. Subsequently all clauses for basic-events that occur before the identifying event have to be added when the identifying event is defined. For this purpose the property `additionalIdentifierWhereClauses` which is defined for the class MetaData in meta-model is used. Whenever the `whereClause` -method finds a clause that would be before the identifying-events definition, this clause is added to the event-identifier instances. This clauses cannot simply be added by using the `AND` keyword because

```
((EVERY (VariableReadEvent_s_o=eventengine.model.
   VariableReadEvent(TRUE )
or VariableModificationEvent_s_o=eventengine.model.
   VariableModificationEvent(TRUE )))
-> ProcessCompletionEvent_s=eventengine.model.
   ProcessCompletionEvent
(TRUE  AND (FALSE
or (TRUE  AND VariableReadEvent_s_o.instanceId=
   ProcessCompletionEvent_s.instanceId)
or (TRUE  AND VariableModificationEvent_s_o.instanceId=
   ProcessCompletionEvent_s.instanceId))))
```

Figure 4.13: Example event-rule for a composite-event of type every (A or B) → C

of the “or”-events that have to be kept in mind. This can lead to patterns like the one shown in Figure 4.5 where the basic-event that holds the event-identifier is the last event and therefor all conditions have to be appended to the event-identifier. This example also shows how the clauses for the “or”-event are added.

CHAPTER 5

Examples

For the purpose of testing our system, we created a series of Apache ODE [10] BPEL-processes and a set of event-rules that we describe in the following subsections. These processes were used to generate a number of events for testing the generated Esper-rules. We also created a set of composite-rules that do not have a purpose besides testing if they are generated correctly and if they fire as intended to. All our input has been used to test if the monitoring-application stores and displays all received events.

5.1 Example BPEL-processes

These processes contain a variety of different processes, from long-running processes that contain loops and decision nodes to very simple sequential ones. Some of processes are asynchronous, while most of them are synchronous. For most of the asynchronous processes we built synchronous wrappers to ease use and avoid the necessity to implement a client application as well.

Car-rental process

This is one of the few example process for which we implemented a client-application. It was inspired by other car-rental examples found on the Internet but reimplemented for Apache ODE [10] and adapted for our needs. Our version takes a request for a car-rental, which includes the following properties:

- location - pick-up location
- returnLocation - the drop-off location
- category - the category of the car (e.g. van, cabriolet, pick-up truck, . . .)
- start date - the date of the pick-up
- end date - the date when the car is returned
- name - the name of the person that rents the car.

The process then forwards the data from the request to three different web-services that implement a car-rental interface(SmallCarRentalProvider, StandardCarRental-Provider, LuxuryCarRentalProvider). These are dummy web-services that generate pseudo-random offers as responses. The process then lets the requester choose which offer she would like to pick. After this a "RentalService" is invoked which takes the data from the request and simulates a booking. After this the a result string from the rental-service is displayed to the requester.

Train-ticket process

This process implements a train-ticket sale. It receives a request for a train-ticket from location a to location b that also contains the account-number which will be charged for the ticket. It then uses a credit-service to check the balance on the account. After this the price for the ticket is calculated by a train-ticket service. Both services are dummy-implementations and return random numbers. If the balance of the account is smaller

then the price of the ticket, the process aborts. If the balance is high enough, the price of the ticket is sent to the customer to get approval for the ticket-sale. If the user confirms the ticket, her account is charged and a confirmation message is sent.

Credit-approval process 1

This example is a simplified version of the example from [6]. It first checks the customers credit-rating. On the basis of her credit-rating and the requested amount a an appraisal is created. Then this appraisal is approved and the result returned to the customer.

Credit-approval process 2

This example is a modified version of the example from [6] and has also been implemented using dummy Web-Service implementations. It is shown in Figure 5.1. As the name says, it models a simplified credit-approval process where the credit-rating of the customer is checked in the InvokeCreditService-activity. If the rating is high enough an approval-request is sent to an approval-service. Then the process waits until two different persons approved the request. After this the process returns the result to the customer. We also implemented an applicatio which receives the approval request, shows them and gives the possibility to approve them using different names.

Counter process

This bpel example process was introduced by Paul Brown in 2007 [3]. Our reimplementation uses one while-loop and a Pick-activity instead of a flow with four different while-loops. It implements a simple counter that provides three methods and is shown in Figure 5.2:

- init: This method takes a name and initializes the counter (creates the process-instance).

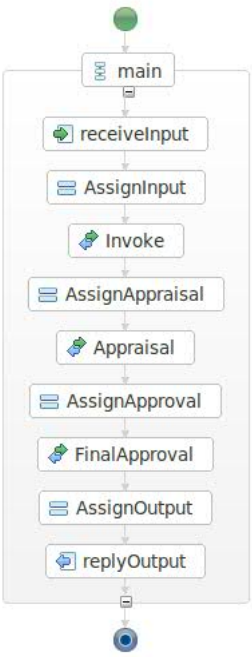

Figure 5.1: Credit-approval business process example

- Get: Takes the name of the counter and returns its value.
- GetAndIncrement: Takes the name of the counter, increments it by 1, and then returns its value.
- Close: Finished the process instance (and so closes the counter).

The correlation is done via the counter-name.

This process is used for testing repeat-composite-events because of the long running nature of its instances with their often called methods.

VarModThenReadProcess

This process was created for testing the VariableValueRead-event we defined. It writes to a variable more then once and then reads the variable. This allows to check if the correct (actual) value of the variable is given in the event. It also tests if the

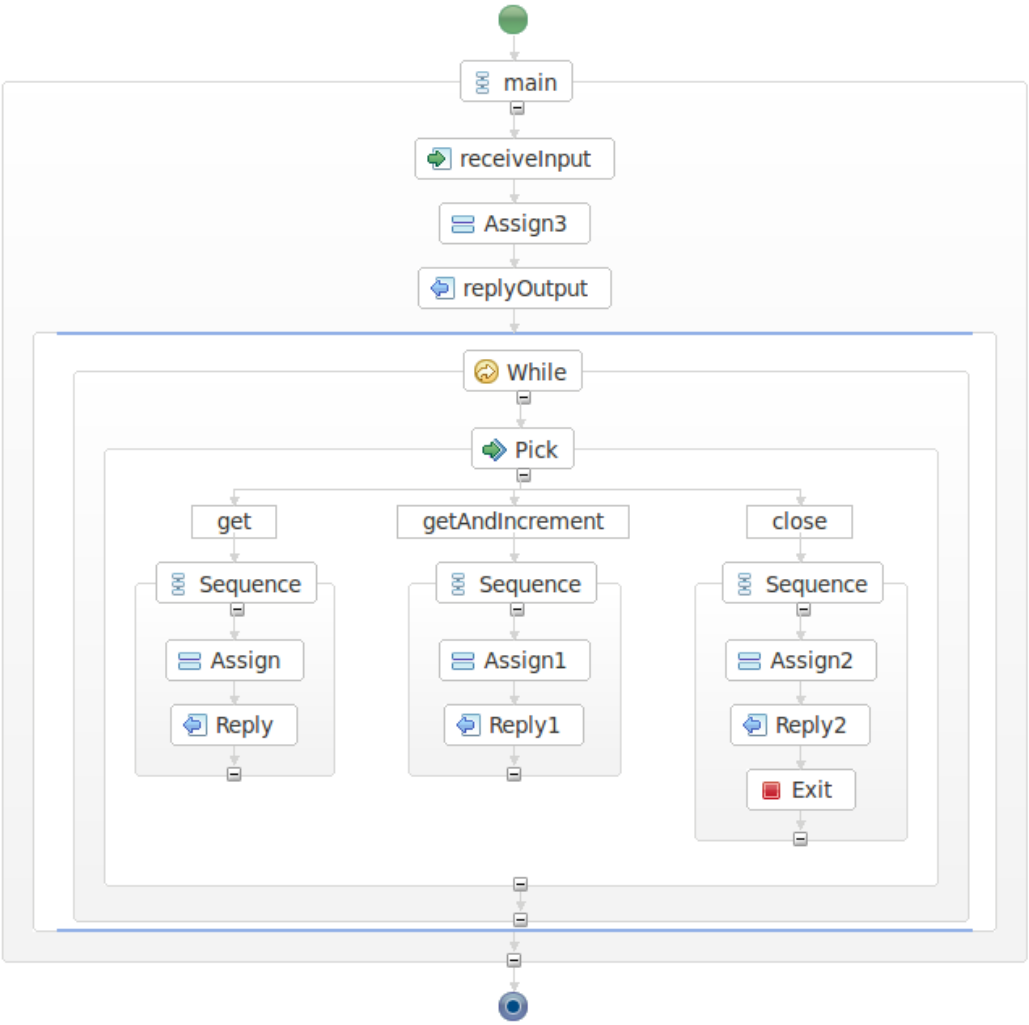

Figure 5.2: Taken from [3]: Counter process example

event only fires for the last VariableModification-event and not for other preceding VariableModification-events too. This process does not call any Web-Service or other BPEL-process.

5.2 Event rules

We also defined a set of event-rules ranging from examples which we regarded to be useful in real-life to examples we added for testing the correct implementation of our Xpand-template that generates the event-rules from the model. This template is named `eventprocessor.xpt` and generates the rules into an xml-file.

1. NotCompletedProcessEvent: Fires whenever a ProcessStartedEvent is not followed by a ProcessCompletionEvent within thirty seconds. The schematic syntax-notation for this rule is: `Every A → (timer:interval(30000 msec) and not B)`

2. CompletedProcessEvent: Fires whenever a ProcessStartedEvent is followed by a ProcessInstanceStateChangeEvent and then a ProcessCompletionEvent. The schematic syntax-notation for this rule is: `Every A → B → C`

3. VariableValueRead: Fires for the last VariableModification event whenever a VariableValueRead event for the same variable is received. The schematic syntax-notation for this rule is: `Every A → (B and not A)`

4. VariableValueChanged: This follows the same pattern as the VariableValueRead event. It fires whenever a variable is changed(VariableModificationEvent) after a call to another BPEL-process or Web-Service(ProcessMessageExchangeEvent).

5. EveryProcessRepeatedlyNotCompletedEvent: This event fires whenever the NotCompletedProcessEvent fires for the third time. However in this form, it is not relevant for which processes the NotCompletedEvents fire. For tieing the repeated events to specific processes a redesign of this rule would be necessary. The

schematic syntax-notation for this rule is: `Every([3](Every A → not B))`. E.g. the rule `([3] A ) → not B` fires when, three times in a row, instances of the same process fail to complete.

6. AfollowedbyBorCSequence: This event is designed for testing the correct implementation of the Xpand-template. The schematic syntax-notation for this rule is: `Every A → (B or C)`

7. SequenceEveryAorBthenCTestOREvent: This event is designed testing the correct implementation of the Xpand-template regarding the special case where the identifying-event is at the end of the rule and therefor the clauses for tieing the composed basic-events together need to be placed correctly. The schematic syntax-notation for this rule is: `Every (A or B) → C`

8. SequenceEveryAandBfollowedbyCTestEvent: This event is like the SequenceEveryAorBthenCTestOREvent above but for testing the correct implementation of the "LogicAnd"-event. The schematic syntax-notation for this rule is: `Every (A and B) → C`

9. EveryRepeatVariableReadAndNotVariableModificationEvent: This event fires whenever four VariableRead-events occur without a VariableModificationEvent. The purpose of this event is testing of the "Repeat" with fixed bounds. The schematic syntax-notation for this rule is: `Every ([4]A → B)`

10. EveryuselessModifictionsEvent: This event fires whenever a VariableRead-event is received and contains all preceding VariableModificationEvents that concerned the same variable-name and process-instance. The schematic syntax-notation for this rule is: `Every ([...]A until B))`

11. PSSSfbFTorC: This is another template-test event whose name describes the events that are part of this composite-event and how they are connected: A ProcessStarted-event that is followed by a ScopeStarted-event, which in turn is

followed by either a ScopeFault-event, that is followed by a ProcessTermination-Event, or a ScopeCompletionEvent, that is followed by a ProcessCompletion event. The schematic syntax-notation for this rule is: `Every A → B → ((C → D) or (E → F))`

12. EveryProcessNotCompletedThreeTimesInARow: This is a possible redesign of the EveryProcessRepeatedlyNotCompletedEvent-example. The schematic syntax-notation for this rule is: `Every ([3] A ) → not B` fires when, three times in a row, instances of the same process fail to complete.

    Please note the EveryProcessRepeatedlyNotCompletedEvent which only differs in the parenthesis.

5.3 Test results

Subsequently we tested the preceding event-rules with all the business-process described in Section 5.1 and some more we do not describe in this document. The results of this tests are shown in Table 5.1.

This table shows how often an event occurred during a test-run. The X means that one or more events of this type occurred. On the left side the according process-name is given and the headline shows the rule-number from the enumeration above.

The first process-run ("Credit 1") was an execution of the process described in Section 5.1 that completed successfully and returned the result "approved". The second row shows the results for a process-execution of the process described in Section 5.1. The row labeled with "Trainticket 1st run" describes an execution of the process from Section 5.1 that terminated because the balance on the account was lower than the price of the ticket. The second execution of this process ended with a successfully a sold-ticket and the confirmation message.

The first execution of the car-rental-process (Section 5.1) started one process-instance that was completed successfully. The second test-scenario of the process

Table 5.1: Test results

Process-name	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
Credit 1		1	8	X		1	X			X		
VarModThenRead		1	5	X		1	X			X		
Credit 2		1	10	X		1	X			X		
Trainticket 1st run	1		4	X		1	2			X		
Trainticket 2nd run		1	9	X		1	X			X		
Carrental 1st run		1	12	X		1	X			X		
Carrental 2nd run	3		X	X	X	1	X			X		1
Counter	X		X	X		1	X		X	X		1

started three different process instances where none of the instances was finished because no offer was selected.

For the counter-process one instance with name "counter" was created. This led to a `NotCompletedEvent` after thirty seconds even though the process-instance was still running. We then called the `get` -method of the instance for four times so that the `EveryRepeatVariableReadAndNotVariableModificationEvent` fired.

The event-rule #11 never fired because no `ScopeCompletedEvent` and no `ScopeFaultEvent` were received. As to our knowledge this is a problem of Apache ODE, as we were not able to receive a single `ScopeCompletionEvent` from the engine. Instead we received `ActivityExecEndEvents` with the property `activityType=OScope` . Still we did not find any bug-trackers or mailing-list entries with support questions regarding this issue.

Event-rule #8 never fired because whenever an `ActivityExecStartEvent` is sent, an `ActivityEnabledEvent` is sent before. And therefor the clause `Scope-StartEvent and ActivityExecStartEvent` never fires.

CHAPTER 6

Related work

We use the approach of Complex Event Processing (CEP) for gathering data to do Business Activity Monitoring (BAM). BAM is an area of research where a lot of work has been done in the last years ([6, 33, 2, 17, **?**, 25]). While this research also expands to the use of CEP for gathering data for BAM, we currently do not know of any other approaches that integrate model-driven-development (MDD). Our approach allows easy integration into different environments as well as fast adaptation of the data that is monitored.

6.1 Business process monitoring & Complex event processing

In "BP-Mon: Query-Based Monitoring of BPEL Business Processes" [2] the authors describe a system for monitoring BPEL-processes with a query language BP-Mon, that allows visual definition of monitoring tasks. These queries are then translated into BPEL-code which is executed on the same execution engine as the monitored processes. While we currently implemented our system only for BPEL, it is possible to use our system with any other execution-engine that emits events. Furthermore our

implementation is designed to run on a different machine than the application-server itself, which should make the performance impact even smaller then the query-translated monitoring-processes that run on the same business server instance.

The authors of [33] developed a process analysis tool (PISA) which is designed for the evaluation of work-flow history data. It provides a plug-able system for analyzing and displaying the data. However this system is based on Java-code classes, which have to implement a specific analysis method, while our system relies on the complex-event-processing capabilities for a live analysis of the system, which does not require any hand-written Java-code. If a new rule for monitoring should be created, we just add it in the model and generate the code. PISA also uses the same "plugin-system" that is used for analysis methods for the display of data. This means that a all classes found in a specific location are loaded at start-up and can in the following be use to view monitoring data. Our system currently only generates textual-views and provides a fixed dashboard.

Mulo, Zdun, and Dustdar [6] describe the extraction of complex-events from web-service event-trails. They also use CEP for monitoring compliance and the ability to react to violations. However they do not use the events emitted by a process-server itself and use a static event model, while we use an event meta-model and create our event-model on the basis of this meta-model. Thus being able to automatically create the according rule-set for the CEP-engine and reacting more flexible to changing requirements.

Donald F. Ferguson and Marcia Stockton provide an overview on business process management in their Article "Enterprise Business Process Management - Architecture, Technology and Standards" [8]. They suggest that "the event- and KPI-models are explicit elements of the business process model" and also mention event-stream and pattern analysis of emitted events and automated reaction to specific patterns, but do not describe any of the topics in detail and keep their focus on the big picture.

In "Events and Streams: Harnessing and Unleashing Their Synergy!*" [4] the authors give a summary of the event-basic computation research and its possible applica-

tions. But they do not describe the use of (complex-)event-processing for the monitoring of business processes nor do they suggest a model-driven approach to using CEP.

6.2 Model-Driven-Developement

In the article "Model-Based Run-Time Monitoring of End-to-End Deadlines" [1] witch was published in the Proceedings of the 5th ACM international conference on Embedded software the authors describe a model-based approached of monitoring QoS-properties in the automotive-domain, where they define interaction specifications either as simple (atomic) messages or compositions of simple messages connected by operators. In this approach also some monitoring functionality is pushed out into the components [1]. This leads to a distributed monitoring, where every component checks and reports if a QoS-property is violated. The authors illustrated their approach on the example of end-to-end deadlines for messages and describe how this principle can be extended to other QoS-properties. However we believe that our approach of using a CEP-engine to extract data from a series of event-streams is more versatile and better suited for our targeted application areas.

Holmes et al. [16] introduce a Model-Aware Service Environment(MORSE), which is a repository for storing and managing models of business processes and services. While MORSE does not describe how, the paper "Modeling process-driven SOAs - a view-based approach" [31] presents a possible approach for the modeling of business processes. Additionally MORSE introduces compliance concern models belonging to process models, that can be used to model compliance concerns such as e.g. business constraints that must not be violated. MORSE provides methods to query, change, and update these models and its relations. Furthermore the authors describe the use of these compliance concern information for online and offline monitoring of systems that are generated from the models in the repository. Therefor they add model-information into the generated entities, which can later be used during monitoring to determine the models where compliance violations occurred. For the monitoring a CEP-engine is

assumed, which uses rules generated from the compliance models.

In “Model-driven and Domain-specific Architectural Knowledge View for Compliance Meta-data in Process-driven SOAs” [14] the authors refine their description of a compliance concern model for MORSE [16] and introduce a view for MORSE that provides the ability to annotate specific SOA elements with compliance meta-data by using compliance controls. These “`Controls` fulfill `Compliance-Requirements` that relate to `ComplianceDocuments` such as a `Regulation` , `Legislation` , or `InternalPolicy` .” [14]

Our work can be extended to provide the monitoring capabilities required by MORSE [16], by making our monitoring application “model aware” and apply model-transformations from the compliance models to our event-model, so that it can be used with Esper. In order for this to work, it is possible that small adjustments to our event meta-model are necessary to make it model aware too.

The authors of “MDD4SOA: Model-Driven Service Orchestration” [20] extend UML2 with model-elements for Services and specifiy an UML-profile. This way they are able to model Services and Business Processes in UML by using existing UML-tools. These models are then converted into an intermediary model that uses structure activities similar to those in structured programming languages. From this intermediary model they then generated BPEL, WSDL, Java and Jolie-code. However the authors focus on modeling the service-orchestration (bpel-processes) and do not mention monitoring of processes that are modeled in this fashion, while we have our focus on the monitoring of possibly existing processes.

In “A Model-driven Approach for Business-Oriented Monitoring of Service Operation” [32] the authors describe a model-based system that uses Event-Condition-Action rules which the authors extend with Past Linear Temporal Logic. This concept is similar to our own concept but differs in which notation is used to describe the events that are monitored. While we use EPL as basis and then subsequently use Esper as processing-engine, the authors have developed their own monitoring engine. Furthermore they do not feed the obtained monitoring data directly into a monitoring application but use an

intermediary database system that stores the monitoring information and from where the information can be queried.

As we already mentioned scientific work concern our topic in the above sections of this chapter, we want to introduce some of the commercial applications regarding this topic in the following section.

6.3 Business Activity Monitoring (BAM) Products

For the purpose of BAM also exists a wide variety of commercial products sold by an event wider variety of companies. Nearly all big players have integrated support for real-time BAM into their business application servers. The following list shows a few examples for these products:

- **Microsoft's BizTalk Server** has support for BAM since version 2006 that allows monitoring the system by exporting data into live Excel templates as well as a graphical web-user-interface called BAM Portal [22].

- Oracle's product which is called **Oracle BAM** is a component that integrates the support for Business Activity Monitoring into Oracle's Application Server. It supports the creation of reports in a WYSIWYG-editor that makes use CEP for gathering its data, which can be displayed in a lot of forms ranging from simple lists to different graphs.

- **IBM WebSphere Business Monitor** also supports dashboards that can be customized using drag and drop of different lists, charts, ... and even supports alerts that can be received on the dashboards, cellphones, pagers, or email.

- Adobe also has a business activity monitoring application in its portfolio called **Adobe LiveCycle Business Activity Monitoring**, that works with IBM, Oracle, or JBoss application servers. It also provides the ability to create dashboards and works with notification-rules.

Not only big companies like the ones mentioned above sell BAM-tools but also Systar, Tibco, wil-co, and others.

CHAPTER 7

Discussion & Lessions Learned

As we described before in Section 2.2, BAM systems face a number of challenges. They need to provide real-time monitoring data from a variety of sources. They need to filter and present the data for different kinds of users ranging from the software-engineer to the sales-manager to name only two examples. The monitoring systems also need to provide means for easy configuration of what monitoring data is provided to adapt to changing businesses.

To tackle this challenges we proposed an approach that provides flexibility and abstraction by using a combination of SOA, CEP, BAM, and model-driven software development, for which we implemented a proof-of-concept system in this thesis.

This implementation shows that in general our approach works successfully, thus providing monitoring data from different sources in different abstraction levels as well as support for compliance monitoring. However to use our implementation in production systems, it would need further extension. The event-model we proposed would need to be extended to support aggregation functions for monitored data and the monitoring application would need a storage-implementation for the monitored data.

Our system also shows that creating or changing existing monitoring-rules in our model provides a good abstraction compared to writing the monitoring-rules in an EPL like the one used by Esper. However changed monitoring rules currently require a restart

of the event-processor component. This problem could be solved by changing rule-loading of the event-processor. Instead of loading the rules at start-up it would need to constantly watch for changes of the rule-file.

We are able to provide different views, that can be created and configured via the model, which show different monitoring data. We currently included a view for showing debugging data and a view for displaying business data. This views can easily be extended without programming further code by adding more views in the model.

Our approach should be easy to integrate into model-based business-process approaches that specify compliance data within their model. These systems only need to create a model-to-model transformation that converts existing compliance data into an instance of our event-model.

Although for the level of abstraction that we introduce in this thesis is a big improvement over writing the rules directly in a low-level language, we have the opinion that there is still room for improvement.

CHAPTER 8

Conclusion & Outlook

In this thesis we presented an approach for monitoring business processes. We accomplish this by creating an event meta-model from which event-models are created. We then proceed by generating the application-infrastructure and Java-classes for these events. Our event-listeners gather the events emitted by business application-servers and Web-Services and convert these events to the model's corresponding events. On this events we perform complex-event-processing by using the "[. . .]open source event stream processing (ESP) and event correlation engine (CEP)[. . .]" Esper [7]. The results are then sent to a monitoring application that provides a web-application user-interface where they are displayed live.

While we used Apache Tomcat as our run-time environment, for easier installation, all components are implemented as web-applications that can be deployed on any Java-based application server or servlet-container.

Our application shows that the model-based approach for the CEP event-rules works in general. It allows us to create an event-model that supports composite events and from which we can generate rules for the CEP-engine that we use for monitoring the business processes. One of the advantages of this approach is that we can adapt the monitoring-rules with only little effort to changing needs.

This approach could also be integrated into other model-based solutions like

MORSE[15]. When the processes and its constraints are not part of the model process-specific monitoring-rules still need to be added to the model for each new process that needs some kind of attention like e.g. A banking process where an alarmed should be raised whenever an amount that exceeds the credit limit, is withdrawn from an account.

8.1 Future work

Our approach fulfills the needs of the constraint-monitoring system described by Holmes et al. [15]. Although we do not fully exploit all features offered by Esper we should be capable of modeling the constraints needed for a business application and if a constraint cannot be expressed by the model we offer the possibility of creating EPL-rules directly. In the future we hope to fully integrate our system into MORSE and extend our monitoring application, so that it is possible to provide model-information for generated processes.

Bibliography

[1] Jaswinder Ahluwalia, Ingolf H. Krüger, Walter Phillips, and Michael Meisinger. Model-based run-time monitoring of end-to-end deadlines. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 100–109, New York, NY, USA, 2005. ACM.

[2] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. Bp-mon: query-based monitoring of bpel business processes. *SIGMOD Rec.*, 37(1):21–24, 2008.

[3] Paul Brown. An Introduction to Apache ODE. `http://www.infoq.com/articles/paul-brown-ode`, September 2007. [Online; accessed 23-August-2010].

[4] Sharma Chakravarthy and Raman Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 1–12, New York, NY, USA, 2008. ACM.

[5] Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors. *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*. Springer, 2008.

[6] Uwe Zdun Emmanuel Mulo and Schahram Dustdar. Monitoring web service event trails for business compliance. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2009.

[7] EsperTech. Esper - Complex Event Processing. http://esper.codehaus.org. [Online; accessed August-2010].

[8] Donald F. Ferguson and Marcia L. Stockton. Enterprise business process management - architecture, technology and standards. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006.

[9] The Apache Software Foundation. http://ws.apache.org/axis2/. [Online; accessed August-2010].

[10] The Apache Software Foundation. Apache ODE. http://ode.apache.org. [Online; accessed August-2010].

[11] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/. [Online; accessed August-2010].

[12] Torsten Greiner, Willy Düster, Francis Pouatcha, Rainer von Ammon, Hans-Martin Brandl, and David Guschakowski. Business activity monitoring of norisbank taking the example of the application easycredit and the future adoption of complex event processing (cep). In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 237–242, New York, NY, USA, 2006. ACM.

[13] James Holmes and Chris Schalk. *JavaServer Faces: The Complete Reference*, page 3. McGraw-Hill, Inc., New York, NY, USA, 2007.

[14] Ta'id Holmes, Huy Tran, Uwe Zdun, and Schahram Dustdar. Model-driven and Domain-specific Architectural Knowledge View for Compliance Meta-data in Process-driven SOAs. In Paris Avgeriou, Patricia Lago, and Philippe Kruchten, editors, *SHARK*, pages 64–71. ACM, May 2010.

[15] Ta'id Holmes, Uwe Zdun, Florian Daniel, and Schahram Dustdar. Monitoring and Analyzing Service-Based Internet Systems through a Model-Aware Service

Environment. In Barbara Pernici, editor, *CAiSE*, volume 6051 of *Lecture Notes in Computer Science*, pages 98–112. Springer, June 2010.

[16] Ta'id Holmes, Uwe Zdun, and Schahram Dustdar. MORSE: A Model-Aware Service Environment. In Markus Kirchberg, Patrick C. K. Hung, Barbara Carminati, Chi-Hung Chi, Rajaraman Kanagasabai, Emanuele Della Valle, Kun-Chan Lan, and Ling-Jyh Chen, editors, *Proceedings of the 4th IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 470–477. IEEE, December 2009.

[17] Daniel Jobst and Gerald Preissler. Mapping clouds of soa- and business-related events for an enterprise cockpit in a java-based environment. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 230–236, New York, NY, USA, 2006. ACM.

[18] Rod Johnson. Spring. `http://www.springsource.org/about`. [Online; accessed August-2010].

[19] Sven Efftinge Karsten Thoms. Eclipse Xpand. `http://wiki.eclipse.org/Xpand`. [Online; accessed 01-August-2010].

[20] Philip Mayer, Andreas Schroeder, and Nora Koch. Mdd4soa: Model-driven service orchestration. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 203–212, Washington, DC, USA, 2008. IEEE Computer Society.

[21] David W. McCoy. Business activity monitoring: Calm before the storm. Technical Report LE-15-9727, April 2002.

[22] Microsoft. Using Business Activity Monitoring. `http://msdn.microsoft.com/en-us/library/aa561326`. [Online; accessed 25-August-2010].

[23] Sun Microsystems. Java Message Service. `http://java.sun.com/products/jms/docs.html`. [Online; accessed August-2010].

[24] OASIS. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf, January 2007. [Online; accessed 25-August-2010].

[25] Carlos Pedrinaci, Dave Lambert, Branimir Wetzstein, Tammo van Lessen, Luchesar Cekov, and Marin Dimitrov. Sentinel: a semantic business process monitoring tool. In *OBI '08: Proceedings of the first international workshop on Ontology-supported business intelligence*, pages 1–12, New York, NY, USA, 2008. ACM.

[26] Inc. Red Hat. Richfaces. http://jboss.org/richfaces. [Online; accessed August-2010].

[27] Derek T. Sanders, James A. Hamilton, Jr., and Richard A. MacDonald. Supporting a service-oriented architecture. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 325–334, San Diego, CA, USA, 2008. Society for Computer Simulation International.

[28] Richard Soley. Model Driven Architecture. Object Management Group, 2000.

[29] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[30] Prime Technology. Primefaces. http://www.primefaces.org/. [Online; accessed August-2010].

[31] Huy Tran, Ta'id Holmes, Uwe Zdun, and Schahram Dustdar. *Chapter 2: Modeling Process-Driven SOAs – a View-Based Approach*, chapter 2, pages 27–48. Information Science Reference, Handbook of Research on Business Process Modeling edition, April 2009.

[32] Zhuohao Wang, Zhuofeng Zhao, and Kaiyuan Qi. A model-driven approach for business-oriented monitoring of service operation. *Service Sciences, International Conference on*, 0:231–236, 2010.

[33] Michael zur Muehlen and Michael Rosemann. Workflow-based process monitoring and controlling-technical and organizational issues. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, volume 2, page 10 pp, 2000.