TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

DIPLOMA THESIS

Design and Implementation of Role-based Architectural Event Modules

submitted by: Frank Rohde

born: 31.12.1985 in Berlin

to obtain the degree of

Diplomingenieur
(Dipl.-Ing.)

First Examiner: Prof. Dr. Uwe Aßmann
Second Examiner: Dr. Sebastian Götz
Supervisor: Dr. Somayeh Malakuti

Submitted: 23.05.2016

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Current technology is developing towards "Systems-of-Systems" (SoS). A System-of-Systems is constituted by a - potentially large - group of individual, collaborating systems. The individual systems are characterized by operational and managerial independence. Moreover, they are developed independently. Those characteristics cause the behaviour of a constituent system to be subject to "emergence". This emergence has to be considered when managing a System-of-Systems (see 2.1).

This thesis is situated in the **context** of "Systems-of-Systems" as a current field of study. In particular, it is concerned with the management of the behaviour of constituent systems on SoS-level. One management goal is to keep the behaviour of the overall system unaffected by the emergent behaviour of its constituent systems. To achieve that goal, the SoS-manager may define and implement certain "coordination rules". Coordination rules are definitions of certain interactions that are forbidden or that are required among constituent systems. The successful application of coordination rules to constituent systems suppresses unintended behaviour that might have been introduced by emergence ([22]).

The concern of coordination is a cross-cutting concern. It involves multiple constituent systems that participate in a SoS. Moreover, this concern crosscuts through multiple components of individual constituent systems. Therefore, the concern of coordination is applicable on a level of abstraction that resides above the level of individual applications. It can be applied on the architectural level. Such concerns can be nicely modeled by architecture description languages (ADLs). An ADL allows for the desired separation of applicational concerns and architectural concerns like "coordination". The ADL "EventArch" is oriented towards coordinated behaviour of systems that are collaborating as "constituent systems" of a SoS. It distinguishes itself from other approaches by providing means to modularize coordination as a cross-cutting concern that is transparent to the code of the constituent system (see [22]).

Systems-of-systems can be divided into static SoS and dynamic SoS. A static SoS is assumed to consist of a static set of constituent systems. A dynamic SoS consists of a dynamic set of constituent systems, i.e., constituent systems may join or leave

a dynamic SoS at any moment in time. Coordination rules are meant to prevent unintended behaviour on the SoS-level that was caused by the simultaneous operation of specific constituent systems. To prevent that behaviour, coordination rules may be designed to be more or less restrictive. For the same coordination rule, several more- or less restrictive variants may exist. The desired level of restrictiveness of a coordination rule may depend on the composition of constituent systems that is prevailing in the SoS at a certain moment in time. The desired level of restrictiveness may rise if more systems join the SoS and may fall if certain systems leave the SoS.

This creates the following **problem**: If the level of restrictiveness of the current variant of the coordination rule would not match the level of restrictiveness that is actually desired, the behaviour of the constituent systems is either over- or under-restricted. Both cases may give rise to unintended behaviour of the constituent systems that may complicate or prevent achieving the goals on SoS-level. The main **goal** of this thesis is to devise a mechanism to dynamically apply a coordination rule to a SoS, depending on the composition of constituent systems. The mechanism that is to be developed in this thesis is understood to be a contribution to solving the problem that has been stated above. To finally solve that problem, another mechanism has to be developed to initialize the coordinators of the more- or less restrictive variant of the coordination rule. A proper initialization is necessary, as the coordinators of the new variant are expected to preserve the state of the former coordinators, to ensure a seamless continuation of the coordination services in the SoS.

The aspired **solution** should allow for a modular implementation of the cross-cutting concern of coordination. The ADL EventArch 2.0 is a promising approach with respect to that condition. Therefore, EventArch 2.0 has been chosen for extension in this thesis. The described problem is solved by extending the architecture description language "EventArch 2.0" to support the dynamic application of a coordination rule to a System-of-Systems. This will increase the applicability of EventArch to dynamic SoS. The application of a coordination rule may be conditioned by the composition of constituent systems in the SoS.

The application of a coordination rule is achieved by composing the coordinators that are associated to that rule, with the constituent systems of the SoS. Each coordinator bears responsibility for a specific constituent system. This architectural setup can be related to concepts that originate in the field of "Role-based Modeling". Therefore, a solution is devised that is based on those concepts, namely "Base", "Role" and "Compartment". An exemplary architectural specification is presented that consists of two constituent systems that dynamically ignore or comply to a specific coordination rule, depending on the presence of the respective other system.

1.2 Overview

In the following, the structure and line of argumentation of this thesis is presented.

In the "**Background**"-chapter, the reader is equipped with knowledge about concepts and fields of study that are related to the concern of this thesis. The description includes "System-of-Systems", "EventArch 2.0", and concepts from the field of "Role-based modeling". The subsequent chapter "**Related Work**" develops requirements in order to solve the problem that has motivated this work and identifies relevant language-features. It introduces current role-based modeling- and programming languages, analyzes their respective approach to "Role-based Design" with respect to the features that have been identified before, and reflects on their capability to solve the problem of coordination in dynamic SoS, that has been described in the preceding section. Then, the thesis' solution to this problem is presented. EventArch 3.0 is described as a loosely-coupled architecture description language that provides the language-elements "Dynamic Composite AEM", "Cross-cutting Roles", "Role-Binder", and "Compartment" to achieve a dynamic application of coordination rules through consistent role-binding. Role-binding may depend on the composition of constituent systems in the SoS. Moreover, the language-support for a modular implementation of the cross-cutting concern of coordination is described. This description is provided on a rather conceptual level in the chapter "**Concepts of EventArch 3.0**". The next chapter, "**Internal Design of EventArch 3.0**", provides a more detailed description that concerns the implementational level as well. This chapter will be useful for readers that intend to extend EventArch 3.0. EventArchs approach to solving the problem of coordination in dynamic SoS, that has motivated that work, is evaluated in the next chapter. This chapter, "**Evaluation of EventArch 3.0**", identifies advantages and disadvantages of EventArchs 3.0 features, reflects on the way in that those features fulfill the requirements that have been stated in the "Related Work"-chapter, and finally applies EventArch 3.0 to a use case that involves two constituent systems.

At the end of this work it is **concluded** that EventArch 3.0 is capable to solve the thesis' problem of coordination in dynamic SoS by providing support for the dynamic application of coordination rules in a dynamic SoS in dependence of the composition of constituent systems in the SoS.

Chapter 2

Background

This chapter is concerned with concepts and notions that are relevant for the rest of that thesis. The reader is introduced to the notion of "System-of-Systems" (SoS). Constituent systems are distinguished from ordinary system-components according to certain characteristics. EventArchs 3.0 tribute to those distinguishing characteristics is described. EventArch 3.0 is an extended version of EventArch 2.0. Therefore, concepts- and implementation of EventArch 2.0 are described in that chapter as well. This part is especially important for understanding later parts of the thesis that are concerned with the language-extension that was applied to EventArch 2.0. In the following, role-related concepts are described. A short introduction into the development of the scientific field of "role-based modeling" is given. This part of that chapter is concluded by the derivation of an interesting mapping from role-related concepts to concepts that relate to peer-to-peer coordination. Finally, the coupling strategies "loose-coupling" and "tight-coupling" are explained. Certain design-decisions of EventArch 3.0 are associated to those strategies.

2.1 System of Systems

Large organizations implement computing- and controlling services that solve increasingly complex tasks. Examples include "traffic and transportation" [12], "power generation and distribution" ([5]) and "space exploration"([13]) The complexity of those tasks has grown out of a level at which they could have been implemented by an individual system. Instead, multiple systems have to achieve the desired complex computing- and controlling services. Such "Systems-of-Systems" are characterized in the following. Those problems of SoS are emphasized, that EventArch 3.0 has been designed to approach. An illustrative example of a System-of-Systems is presented and discussed in conclusion of this section.

To understand the advantage of EventArch 3.0 over EventArch 2.0, the distinction between static SoS and dynamic SoS is crucial. In a static SoS all constituent systems are known at compile-time. Moreover, it can be assumed that all constituent systems will be statically available throughout the lifetime of a static SoS. In a dynamic SoS constituent systems may join or leave the SoS at runtime. Those constituent systems

may be of an unanticipated type.

According to [9], constituent systems can be distinguished from software that is merely a component of an individual system. The relevant criteria would be: "**autonomy**", "**belonging**", "**connectivity**", "**diversity**" and "**emergence**". The pecularities of constituent systems give rise to special tasks for the SoS-manager. Those special tasks should be reflected in an ADL by features that are tailored towards them. EventArch 3.0 can be understood as a step in that direction.

Constituent systems are **autonomous** in that they are independently developed and managed [10]. Moreover, they may be operated independently of each other without failing to achieve their original requirements. [27]. The essential difference between constituent systems and system-components with respect to "autonomy" is, that a system-component has no purpose in its own right, but is incorporated into the system to achieve a system-level purpose. Opposed to that, a constituent system has a purpose in its own right. This purpose can be achieved without contributions of any other constituent system. To illustrate that pecularity, Boardman refers to the example of a brake ([9]), which is merely a component of the system "car", as it was deliberately designed to achieve the system-level purpose "velocity-control" and has no purpose in its own right. Opposed to that, one could regard a car in a sophisticated controlled transport system as a constituent system if it was not deliberately designed to serve a purpose in that system, but can still be used independently of that system, e.g., in areas where this transport system is not prevailing.

To not interfere with the autonomous development of the constituent system, an ADL should not have any impact on the legacy-code of that constituent system. While this is difficult to achieve, EventArch 3.0 respects that characteristic by implementing the concern of coordination transparently to the code of the constituent system.

Constituent systems can also be distinguished from system-components with respect to **"belonging"**. While system-components have been designed for later incorporation into that system, constituent system have been designed to pursue their original purposes. To allow for useful interaction with other constituent systems in order to achieve a SoS-level purpose, additional development effort has to be invested. It can not be assumed for a candidate system that this effort may be justified in order to contribute to the SoS. This characteristic includes that a constituent system might decide to participate in more than a single SoS ([10]).

To allow for reuse in different SoS, that characteristic requires an ADL in the SoS-context to tie a constituent system as tight as necessary, but as loosely as possible to existing SoS-level institutions. Such an institution might be a manager that is concerned with preventing unintended behaviour or a set of coordinators that implement a coordination rule. EventArch 3.0 respects that characteristic by being based on event-based communication to allow for a loosely-coupling between constituent systems and SoS-level institutions.

Constituent systems do also face special conditions with respect to **"connectivity"**.

System-components have been designed to perform specific interactions. Achieving connectivity is a matter of system-design. In a dynamic SoS new types of constituent systems may join or leave throughout the lifetime of the SoS. Therefore, there is recurring pressure for constituent systems towards adding or removing connectivity. This may blur the original system boundary of the constituent system as other systems may achieve "access to some of its inner connectivity that does not normally appear at its surface, or system boundary" ([9]). This interference with the original encapsulation of the systems functionality may be one reason for unintended behaviour.

In EventArch additional connectivity may be added to a constituent system in two levels. The first level is added at compile-time and may provide fundamental services. Based on those fundamental services, more specialized services may be added at runtime. One use case is to provide complex connectivity by deploying special "coordinators" to the second level at runtime. To allow for flexibility between the constituent systems and its coordinators, all communication is based on events. To prevent unintended behaviour that may conflict with SoS-level goals, EventArch 3.0 supports the definition of coordination rules.

Another contrasting issue between a System-of-Systems and an individual system is the type and level of their inherent **"diversity"**. Systems can unfold diversity with respect to their "parts", with respect to the "relationship" between those parts, and with respect to their "system behaviour" as a whole. Individual systems have a comparably short lifetime. Throughout their lifetime they unfold a certain level of diversity with respect to their "system behaviour". But this level of diversity is limited by what was designed in at design time. The diversity of individual systems with respect to their "system behaviour" is limited by design. Opposed to that, Systems-of-Systems have a much longer lifetime. They may be hosted by large organizations that may consider to change their configuration but not to "replace" them. Often, a single instance of a SoS is in operation. Throughout its lifetime a SoS is faced with "rampant uncertainty, persistent surprise, and disruptive innovation" on the side of its constituent systems ([9]). To be able to quickly respond to those challenges, a System-of-Systems should be capable to achieve a much higher level of diversity with respect to "system behaviour" than individual systems.

A System-of-Systems can be also distinguished from an individual system by the way the **emergence** of the respective parts and the overall system can be approached. In the case of an individual system, the desired system-behaviour is relatively obvious to the system-developer. Nevertheless, emergence can take place over the lifetime of the system. This emergence can be anticipated at design-time. This allows the system-developer to design emergence deliberately into the components and into their respective relationships. He may try to anticipate unintended behaviour and provide tests to show its absence. Constituent systems are developed independently. A SoS is committed to a relatively high level of diversity as a whole. Therefore, unintended behaviour is likely to get introduced into the SoS in an unanticipated way. This may

take place throughout the lifetime of the SoS. Moreover, the SoS-manager may not have knowledge about the detailed behaviour of each constituent system. In a changing context, a constituent system may behave in a way that was not expected by the SoS-manager. In this way, the emergence of constituent systems can be described as "unanticipated" from point of view of the SoS-manager. This discourages to show the absence of unintended behaviour by writing tests. Instead, the SoS-manager should "create a climate in which emergence can flourish, and an agility to quickly detect and destroy unintended behaviors, much like the human body deals with unwanted invasions." ([9])

EventArch 3.0 approaches that climate by establishing specific coordination rules among constituent systems and by achieving their acceptance, even in the face of unanticipated emergence.

The SoS-characteristics described above contribute to the capability of a SoS to cope with the increasingly large complexity of todays tasks in computing and controlling. They enable the constituent systems of a SoS to have through collaboration (i.e. as a SoS) a substantially more developed impact on their environment than the same systems could have in isolation. To conclude this section a natural example of a SoS is described.

Researchers may be concerned with multiple fields within their subject. From time to time a research conference is held that brings many representatives of that field together. At such an occasion intense discussions among the assembled researchers can develop. This setup can be understood as a natural example for a System-of-Systems. It is in accordance with all the characteristics that are described above.

Every individual researcher has "**autonomy**". His research was not established in order to make him a participant of such a conference. It was an option to attend at the conference and an investment in time and money. A decision had to be taken to go there or not to go there. Therefore, the researcher also has "**belonging**". At the conference he will meet new colleagues and will have to adhere towards thoughts that are new and challenging to him. He has to put effort in "connecting" to these new colleagues and their thoughts. Therefore, a researcher at such a conference is also subject to "**connectivity**". The conference is an assembly of expert researchers in that field. In the course of several of those conferences, the composition of involved researchers may change. But what is even more striking is the change of ideas throughout the conference or in its aftermath. The conference unfolds a great deal of diversity in that it allows the expert researchers to get in an intelligent discussion that results in the development of new thoughts and insides. In that way, the conference unfolds a great diversity of thought. This can be understood as an extraordinary high level of diversity with respect to "system behaviour". Therefore, the conference has the characteristic of "**diversity**" as well. Finally, the emergence of thought at the conference can not be anticipated prior to the conference. Nobody can test whether the expected inspiration has been gained by each individual researcher at the conference. **Emergence** takes place in an unanticipated way due to the other

factors. Researchers have autonomy, i.e., they are concerned with individual topics within that field and may contribute to the conference the one *or* the other thought that has originated from their work with respect to their individual topics. At every conference the composition of researches may change, i.e., due to "belonging" a new combination of researchers may be achieved. This new combination may cause new inspirations. The well-established researchers have to "connect" to the ideas of the newcomers. This may be a source of inspiration for them. The researchers will finally achieve a diversity of thoughts through intense discussion. All those characteristics give rise to a sort of emergence that is specific for those conferences and could not have been achieved by the individual researchers in isolation.

2.2 EventArch 2.0

EventArch 3.0 has been derived by extension from the architecture description language "EventArch 2.0". In this section EventArch 2.0 is described. An overview of important concepts and intentions of the language is presented first. A detailed explanation of EventArchs 2.0 internal design follows. Due to the close relationship between EventArch 3.0 and EventArch 2.0, this section contains useful background for understanding subsequent chapters that are primarily concerned with EventArch 3.0. Additional material concerning EventArch 2.0 can be found in the appendix. (see 8.2)

2.2.1 Concepts

EventArch 2.0 is an architecture description language (ADL) that was devised in an effort to improve the language-support for coordination in System-of-Systems (see [22] and [23]). In particular it strives for

- Modularizing the cross-cutting concern of coordination at the granularity of coordination rules
- Improving the separation of the concern of coordination from the original concerns of constituent systems.
- Providing flexibility by supporting various coordination patterns
- Achieving a uniform representation of constituent systems and coordination logic
- Supporting constituent systems that are implemented in heterogenous programming languages

Current solutions in the fields of *aspect-oriented programming*, *coordination languages* and *system modeling languages* would fail to provide satisfactory support for coordination in SoS. EventArch would combine all desirable qualities ([23]). To even

improve its unique combination of qualities, EventArch 2.0 was chosen for extension in this thesis.

In EventArch 2.0 all members of a System-of-Systems are uniformly represented as "**Architectural Event Module**" (AEM). One can distinguish two sorts of SoS-members: constituent systems and coordinators. **Coordinators** encapsulate coordination logic that is employed to achieve a coordinated execution of the constituent systems. They may be implemented as a statemachine. EventArch 2.0 provides language-support for that design-decision. (see below 2.2.2). From point of view of "coordination", an AEM can be understood as a system or application that is either in need for coordination or provides coordination-services. From a more technical point of view, an AEM is a wrapper around an existing system or application. The wrapped system is termed "**Reactor**". The AEM is therefore a representation of the reactor on an architectural level that is primarily concerned with coordination. Java and C++ are supported as reactor-implementation language. The AEM extends the reactor by event-based interfaces. AEMs do therefore rely solely on **event-based communication**.

An event-based interface specifies the events that it can receive and process. Those events are specified in the "required"-part of the Primitive Interface (for details see appendix 8.2). Based on that "**required**"-part of the specification of the Primitive Interface, it can select events of interest out of the stream of incoming events. Selected events are further processed by the interface. They may give rise to calls of certain functions of the reactor. The "**provided**"-part of the interfaces specification is meant to describe all events that it can publish. It is a description of all possible outgoing events. Events may be published in response to certain state-changes or function-calls in the reactor. Event-based interfaces are used to publish the state of the reactor that is relevant for coordination to a coordinator. Moreover, they can be used to interfere with the state of the reactor in order to achieve a change in the behaviour of the reactor in response to events that have been sent by a coordinator. The event-based interfaces of an AEM are stateless and are therefore termed **Primitive Interfaces**. Those interfaces are cross-cutting in that they may be employed by multiple components of the constituent system to contribute to the implementation of cross-cutting concerns, e.g., "coordination". They contribute to the separation of the concern of coordination in that they do not rely on available public interfaces that are related to the original concerns of the constituent system, but are solely meant to be used for coordination-purposes. Moreover, they are implemented transparently to the original concerns of the constituent system using aspect-technology (see 8.2).

A coordinator might be used to implement all coordination logic that adheres to a coordination rule. In this case, all constituent systems that are concerned by that coordination rule would have to set up communication relations with that coordinator. Coordination that is organized in that way is said to adhere to the "centralized" coordination pattern. An alternative to that is "Peer-to-Peer" coordination. To achieve coordination according to the "Peer-to-Peer" pattern, each constituent sys-

tem is associated a coordinator that is responsible for providing coordination services exclusively to that constituent system. This coordinator does merely implement that part of coordination-logic that is relevant for that constituent system, i.e., that systems "share" of the coordination logic. To increase modularization of coordination at the granularity of coordination rules, each coordinator can be designed to implement the constituent systems share of coordination logic with respect to a specific coordination rule. As a constituent system may be subjected to several coordination rules, several coordinators might be associated to that system in that way. To improve language-support for the "Peer-to-Peer" pattern, the AEM-concept was extended by the notion of a "**Composite AEM**" (CAEM) in EventArch 2.0. A Composite AEM encapsulates a constituent system and all coordinators that are associated to it according to the "Peer-to-Peer" coordination pattern. In that way, a CAEM constitutes a higher-level facade of the reactor. The reactor is not represented by an AEM that provides Primitive Interfaces, but by a CAEM that provides coordinators. To emphasize their dedication to a specific constituent system, those coordinators that are encapsulated together with that constituent system in a CAEM are called "**Composite Interfaces**" of that CAEM. The AEM that makes use of the coordinators to achieve coordination, is called "**Composite Reactor**" of that CAEM. To distinguish AEMs that do not have that composite structure from the CAEM's, they are termed **Primitive AEM**. Therefore, a CAEM can be described to consist of a Primitive AEM and certain Composite Interfaces.

A concise graphical description of the concepts of EventArch 2.0 is presented in figure 2.1

For further description of the conceptual design of EventArch 2.0, see [22] and [23].

2.2.2 Implementation

The language-implementation has to cope with the following challenges:

- AEMs exchange information using events. Therefore, **events** had to be defined
- The compiler has to achieve an integration of the constituent systems code ("legacy-code") with the **coordination-services** that are provided by its AEM and remote coordinators.
  - Events have to be **published** in response to certain state-changes of the constituent system that may be of interest to other systems or coordinators to achieve coordination
  - Events of interest have to be **selected** out of the stream of incoming events. To achieve coordination, certain state-changes should be commanded to the constituent system in response to receiving an event of interest.
- A **communication platform** for communicating events had to be established
- A **statemachine-language** had to be provided and implemented that could

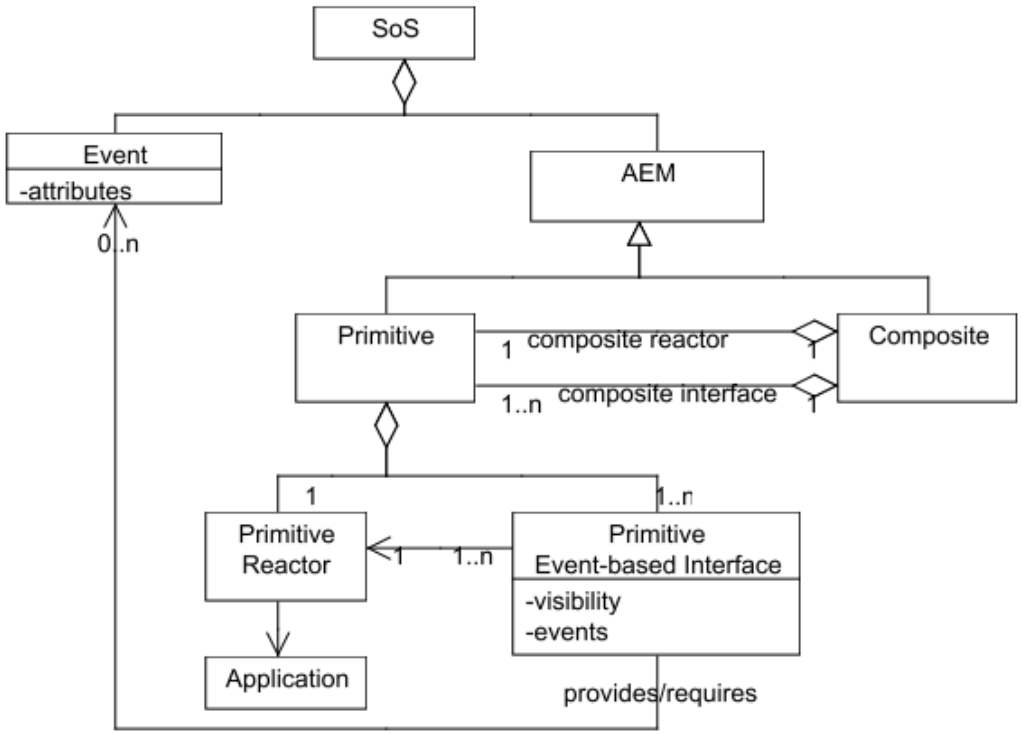

Figure 2.1: EventArch 2.0 concepts (taken from [22])

be used for coordinator-definition.

- Those coordinators that were designed according to the "Peer-to-Peer" pattern, had to be encapsulated in a **Composite AEM**. The constituent system that is coordinated by those coordinators had to be included in that CAEM as well
- The more technical aspects of **compiler-implementation** and code generation had to be tackled

The following description will give an overview about the implementation of EventArch 2.0 with respect to the sketched challenges. Example code of EventArch 2.0 can be found in listing (2.1).

```
1  CoordinatedAdaptiveSoftware[Composite] := {StateCoordinatorASPeer} <->
2                                             {WrappedAdaptiveSoftware}
3  WrappedAdaptiveSoftware[Java] := {ISwitchCoordinatorAS,
4                                    IStateCoordinatorAS} <-> {"AdaptiveSoftware"}
5
6  interface ISwitchCoordinatorAS{
7    requires {
8     CoordinationCommand e_CoordinationCMD =
9                          {E | E instanceof "CoordinationCommand"
10                            && E.publisher == "SwitchCoordinatorLBPeer"
11                            && E.target == "WrappedAdaptiveSoftware"}
12
13    on (e_CoordinationCMD) {
```

```
14        invoke ("org.application.Optimizer", "reconfigure",
15                                                  e_CoordinationCMD);
16     }
17    }
18
19    provides {
20     ConstituentState e_EndedSwitch := after execution (
21         static void org.Optimizer.reconfigure(..)){
22                                     applicationState = "SwitchEnd";}
23    }
24 }
25
26 private interface IStateCoordinatorAS{
27    requires {
28     CoordinationCommand e_CoordinationCMD =
29                          {E | E instanceof "CoordinationCommand"
30                               && E.publisher == "StateCoordinatorASPeer"
31        }
32     }
33
34    provides {
35     ConstituentState e_StartedAS := before execution (
36         void org.ApplicationComponent.execute(..)){
37                          applicationState = "StartExecuting";
38                          target = "StateCoordinatorASPeer";}
39    }
40 }
```

Listing 2.1: EventArch 2.0 code-example: definition of AEM and Primitive Interface

EventArch relies on **events**. The language allows to specify events by type and attribute. An event-type consists of a type-name and a set of attributes. Each attribute has a name and a type. Attributes are of a primitive type. Supported types include *string*, *int*, *float*.

The AEM provides the **coordination-services** "state publication" (provided events) and "command execution" (required). The Primitive Interfaces are responsible for providing both coordination services. State publication is implemented using aspect-technology: AspectJ.

For each event that is specified in the "*provided*"-part of the specification of the Primitive Interface, a pointcut is generated. The pointcut is triggered if certain functions of the legacy-code are executed. In response to a triggered pointcut the specified event is **published**. In that way, Primitive Interfaces can be used to indicate state-changes of the constituent system to other systems or coordinators in the SoS. All pointcuts and advice-code of a Primitive Interface is contained in an aspect. For each Primitive Interface such an aspect is generated. Relying on aspect-technology achieves the transparency of the concern of coordination for the code that implements the original concerns of the constituent system. But still, the coordination-code has to be tightly incorporated into the legacy-code to some extend. Each aspect has to

be woven into the legacy-code at compile-time.

The "*required*"-part of the Primitive Interface specifies those incoming events that are of interest to the module and should therefore be **selected**. The criteria to select an incoming event are encapsulated as a set of selector-specifications. Each selector encapsulates a set of conditions according to which an incoming event might be selected for further processing. The conditions can be concerned with the type of the event and its attributes. They are implemented as a combination of boolean expressions. Expressions can check the received events type and attributes against fixed values. Expressions can be combined using the boolean *and*- and *or*-operator. For each selector that is specified in the "*required*"-part of the Primitive Interface a selector-object is generated at compile-time. This is used to represent the specified selector at runtime. Received events are checked against that object at runtime. In response to a selected event, certain functions of the constituent system can be executed. This feature can be used to command state-changes to the constituent system in an effort to achieve coordination. It is implemented using the reflection-functionality of the respective implementation language.

To implement the coordination-services, events (i.e. "messages") have to be delivered from a source to several targets. To achieve that, a **communication-platform** had to be established. Communication-services are implemented using an existing message-oriented middleware solution: Java Messaging Service (JMS, see [1]). The dissemination of events from a single source to multiple receivers is achieved using JMS' "Topic"-concept. A topic is conceptually a symbolic name that can be mapped to different concrete names. A message-source can publish a message (e.g. an event) to a topic. In response to that, this message will be delivered to all participants that have registered for that topic. The mechanism is implemented by a central entity that is dedicated to providing communication-services: the "communication-provider".

Composite AEMs (CAEM) are implemented as topics. To publish an event into the scope of its CAEM, a constituent system publishes it to the topic that represents its CAEM. The event will be delivered to all coordinators that are contained in that CAEM. No other SoS-member can listen to communication that is published in this way to a CAEM. The communication is private to a CAEM.

In EventArch 2.0, coordination decisions are taken by coordinators. Coordinators are informed about certain state-changes of the constituent systems that may be of interest for the concern of coordination. Coordinators might try to keep an overall state that considers all state-changes of the individual systems. Coordination-decisions can be taken depending on that overall state. To ease the implementation of such stateful coordination, EventArch 2.0 provides a **statemachine-language** that can be used to implement coordinators. Incoming events may be selected to trigger state-transitions. Events might be published in response to entering or leaving a specific state. Depending on the current state, incoming events may cause the publishing of different outgoing events. An extract of a coordinator-definition using the statemachine-language is presented in listing 2.2. The depicted state-machine defines the states *Start*, *Wait-*

ForSwitch and *WaitForRestart*. *Start* is the initial state, i.e., it is the state that the state-machine will assume after startup. On-expressions associate certain actions (state-transition, event-publishing) to selected incoming events. Send-expressions are used to publish a specific event that was specified in the "*provided*"-part of the respective interface. This is done when entering or leaving a specific state (keywords "entry", "exit"). While a specific state is active (keyword "during"), state-transitions can be triggered. They are indicated by the arrow "->". They are performed in response to selecting certain incoming events.

```
1   initial state Start {
2    during:
3      on (ISwitchCoordinator.e_StartedLB) -> WaitForSwitch;
4      on (ISwitchCoordinator.e_TerminatedAS) -> WaitForRestart;
5
6    exit:
7        send ISwitchCoordinator.e_LB_CoordinationCMD =
8                        new CoordinationCommand(){
9                            command = "suspend";
10                           target = "WrappedLoadBalancer";
11                       };
12  }
13  state WaitForSwitch {
14   entry:
15       send ISwitchCoordinator.e_AS_CoordinationCMD =
16                       new CoordinationCommand() {
17                           command = "switch";
18                           target = "WrappedAdaptiveSoftware";
19                       };
20   during:
21     on (ISwitchCoordinator.e_EndedSwitch) ->Start;
22     on (ISwitchCoordinator.e_TerminatedAS) -> WaitForRestart;
23   exit:
24       send ISwitchCoordinator.e_LB_CoordinationCMD =
25                       new CoordinationCommand(){
26                           command = "proceed";
27                           target = "WrappedLoadBalancer";
28                       };
29   }
30  state WaitForRestart {
31   during:
32     on (ISwitchCoordinator.e_InitializedAS) -> Start;
33     on (ISwitchCoordinator.e_StartedLB){
34       send ISwitchCoordinator.e_LB_CoordinationCMD =
35                       new CoordinationCommand(){
36                           command = "proceed";
37                           target = "WrappedLoadBalancer";
38                       };
39     }
40  }
41 }
```

Listing 2.2: EventArch 2.0 statemachine-language code-example

The **compiler-implementation** is based on the Xtext-framework ([4]). It provides automatic parser-generation from a grammar-definition. The classes that model syntax rules are generated likewise. Their names and attributes model the names, references, and attributes of the syntax rules. The generated classes form an abstract syntax tree. The "Model"-class is the root of that tree. Code-generation depends on that tree. While traversing an instance of the tree at runtime ("tree-object"), the code-generator stops at every node and performs code-generation for every attribute and for every reference that was found at the node. Custom validators can be defined to perform semantic-checks on the tree-object that represents the parsed architectural specification.

In the code-example that was presented at the beginning of this subsection (see 2.1), the Composite AEM *CoordinatedAdaptiveSoftware* is defined. It consists of the Primitive AEM *WrappedAdaptiveSoftware* and the Composite Interface *StateCoordinatorASPeer*. This Composite Interface acts as a peer-to-peer coordinator for the Primitive AEM. It is associated to the "State"-coordination rule (see example use case in section 6.4). The Primitive AEM *WrappedAdaptiveSoftware* provides the Primitive Interfaces *ISwitchCoordinatorAS* and *IStateCoordinatorAS*. It is implemented in the programming-language "Java". This interface specifies certain provided events and selectors for required-events. In response to a received event, that could be selected by the selector *e_CoordinationCMD*, the function *reconfigure* of the class *org.application.Optimizer* is executed. The received event is passed as an argument. The event *e_EndedSwitch* is provided in response to finishing the execution of the *reconfigure*-function of the class *org.application.Optimizer*.

2.2.3 Diagrams

This section gives a broad overview about the organization of EventArchs 2.0 code-implementation. It presents two package-diagrams. One is concerned with packages that contribute to providing runtime-services like event-reception, command execution, and state-dependent behaviour. The other is concerned with providing compilation-services like parsing, code-generation, and validation.

The implementation-code of EventArch 2.0 is organized according to the abstract design depicted in figures 8.1 and 8.2. They just contain packages and their relationships to one another. The short hints at the "uses"-dependencies clarify what the supplier-package is used by the client-package for. The abstract design allows a birds-eye view on the implementation.

For single packages, compile-, and runtime-behaviour refer to the appendix 8.2.

The packages that are concerned with code-generation and the compilation-process are depicted in figure 8.1. Those concerned with runtime-issues are depicted in figure 8.2. Both parts of the implementation make use of one another. Compilation-related packages are denoted by numbers, runtime-related packages by letters.

The parser of the EventArch 2.0-language (package 4) is automatically generated by the Xtext-framework. It accepts an architectural specification that is written in the EventArch 2.0-language. Its output is an object-tree representing the specification. Package 2 contains classes whose instances serve as nodes of this tree.

A more detailed description of the internal design of EventArch 2.0 can be found in the appendix (8.2).

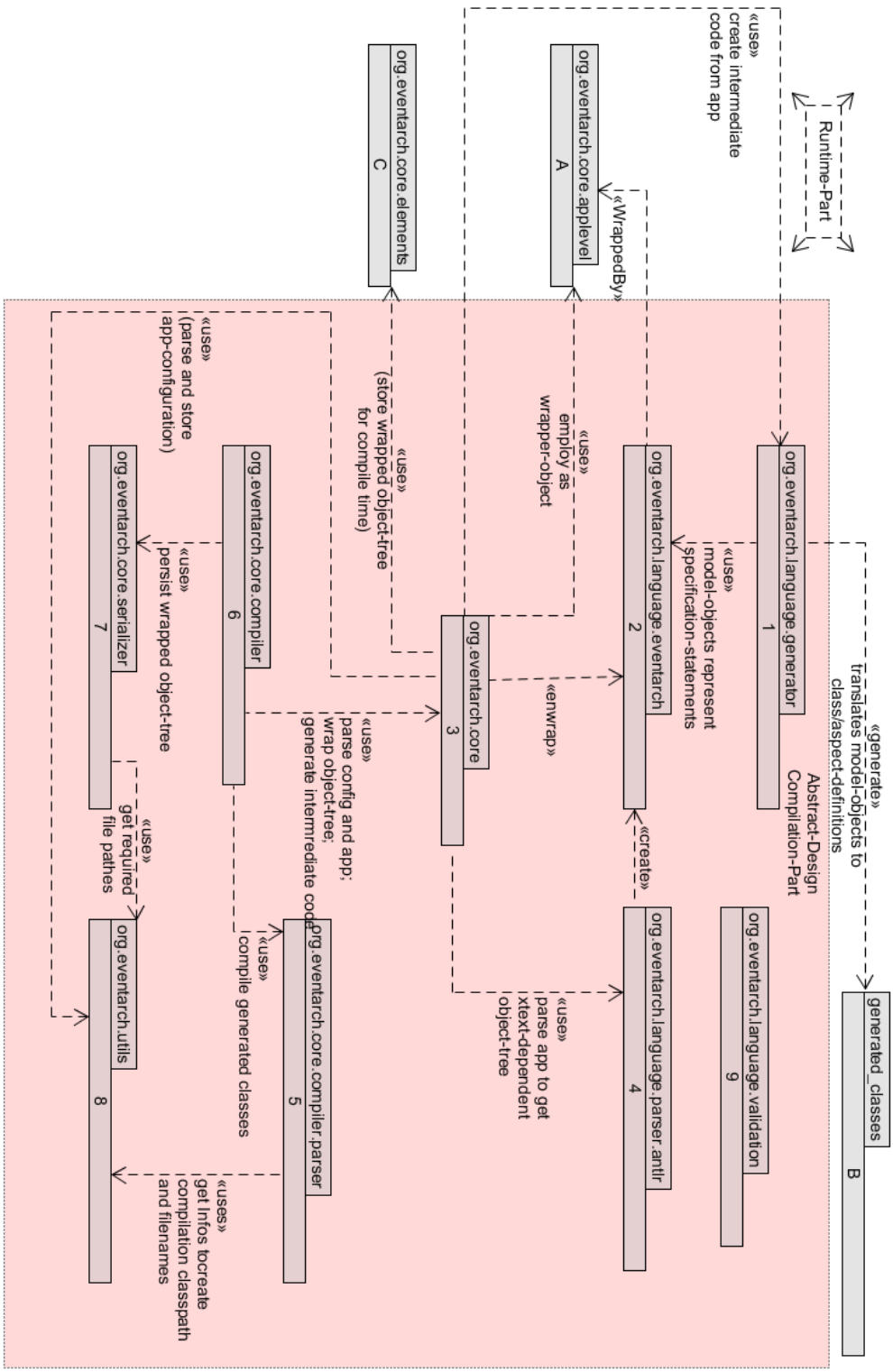

Figure 2.2: Package diagram of the compilation-part of the EventArch 2.0-implementation

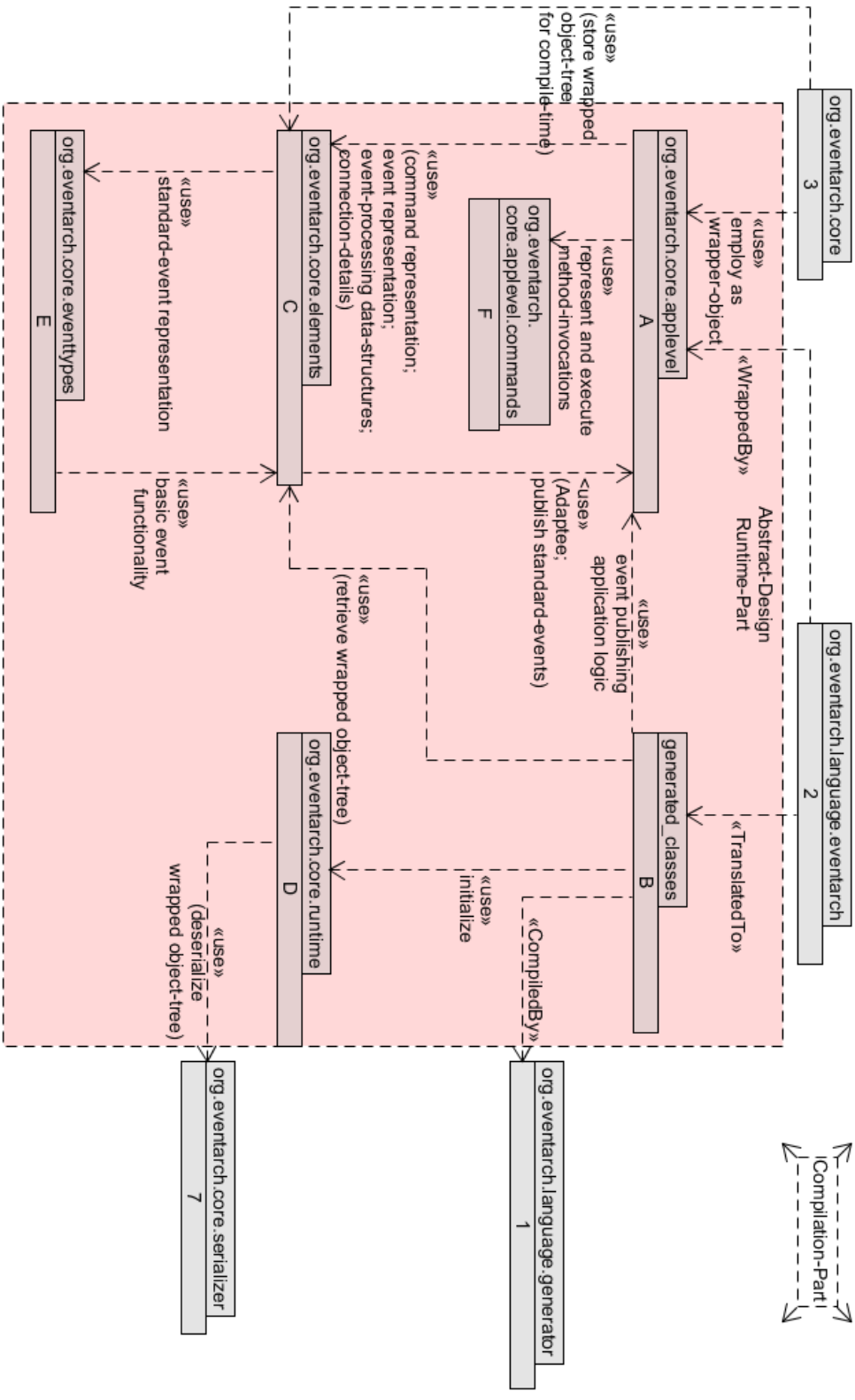

Figure 2.3: Package diagram of the runtime-part of the EventArch 2.0-implementation

2.3 Role-based Modeling

The behaviour of objects or systems can be modeled. "Role-based Modeling" is one modeling approach. In this thesis a modeling solution with respect to the concern of coordination in dynamic SoS is developed (1.1). Role-based modeling is a promising modeling approach to model such implementations. The reason is that core-concepts of peer-to-peer coordination ([18]) can be mapped very closely to concepts that are native to role-based modeling. Therefore, the concepts of role-based modeling are of interest to this thesis. This section gives a short introduction to the modeling approach "Role-based Modeling". The early scientific work on that topic is sketched and some concepts are introduced that are relevant for understanding the application of this approach to the modeling-problem of this thesis. Finally, those concepts are mapped to concepts of peer-to-peer coordination.

Early works
"Role-based Modeling" is a modeling approach that has received a certain attention inside the research community. A basic observation to motivate that approach has been done by Beck and Cunningham in 1989: "No object is an island". Objects would "stand in relationship" with each other and "reply on services and control" ([8]) To cultivate that notion for practical modeling tasks, they devised the "CRC-cards" concept ([11]). A CRC-card associates a class with responsibilities and collaborations (CRC: Class, Responsibility, Collaboration). To capture the contributions of the same class for different collaboration contexts, different CRC-cards can be written for the same class. A collaboration context that involves different classes, can be captured by a group of CRC-cards, each being concerned with one of those classes. To reflect this connection between CRC-cards and role-based modeling, Kendall has suggested to rename them into RRC (Role, Responsibility, Collaboration) ([15])

Based on the outlined development, Zhao ([28]) describes a role to characterize for an object a "position in a context meaningful from a particular viewpoint". Such a viewpoint would constitute an "abstraction" that would select for an object the detail relevant to its position and "suppress the irrelevant information". Roles would be "stereotypical, describing an object from different viewpoints".

Current support
Motivated by those early works, role-related concepts have been supported by various languages. Existing role-based languages can be categorized as role-based modeling languages and role-based programming languages. A survey on the existing language support has been done by Kühn et al. ([20]). In this paper, 26 characteristic features of roles have been identified. Available languages have been rated with respect to those features. They are centered around the concepts of Role, Relationship and Compartment and correspond to the three natures of roles: behavioural-, relational-, and contextual nature. Those concepts are explained in the following. Moreover, the important notion of shared identity is explained and some significant differences between role-binding and inheritance are explained.

Natures of roles

In this thesis a "nature" is understood to be a property that can not be dropped. As stated above, a role can be regarded to hold three natures: behavioural-, relational-, and contextual nature. Each nature is reflected by certain features of the role. The **behavioural nature** is reflected by the roles feature to have properties and behaviour. Moreover, it is related to the notion that a role-object is meant to act influencing the behaviour of a base-object. Whatever a role does is done on behalf of a base-object.

A role-object acts on behalf of a base-object, but there is one property that holds true for all role-behaviour in the same way: a role-object is concerned with behaviour that involves specific other role-objects. This is the essence of the **relational nature** of a role. It may be cultivated in a role-based language by the notion of a "relationship". A relationship may be understood to represent (in part or as a whole) a collaborative behaviour between two specific roles. At runtime, relationships may be instantiated. Those instances can be understood to represent an ongoing collaborative behaviour between two specific roles. This concept shall be illustrated by an example.

Example: In an effort to model educational processes in an university, one could define the roles "Supervisor", "Student", and "Professor". Two relationships "Weekly Consultation" and "Monthly Consultation" could be defined. The "Weekly Consultation" could be constrained to include one instance of a "Supervisor" role and one instance of a "Student" role and the "Monthly Consultation" could be constrained to include additionaly one instance of the "Professor" role. At runtime, the existing relationship-instances could represent the currently ongoing consultations in the university.

Roles are not just involved in certain "relations" between bases, but in certain relations with respect to a certain context. Each role is involved in this context and adheres to the conditions that are imposed by this context. These conditions include the presence of specific other roles and can include additional properties and behaviour that may be exclusively available to roles that participate in this context. The inevitable involvement of a role in such a context can be understood as the **contextual nature** of a role. It can be cultivated by a role-based programming language by representing this context by a dedicated language-construct. Kühn suggests to denote such a language-construct by the general term "compartment". The term "context" is specifically avoided as it would be suffering of inconsistent usage in the research community ([20]). The contextual nature of a role shall be illustrated by an example as well.

Example: The roles "Supervisor", "Student", and "Professor" of the previous example could be encapsulated inside a compartment "University". Besides those three roles this compartment would comprise other roles, e.g., "Researcher", "Research Assistant", "Teaching Assistant". Moreover, additional properties and behaviour might be defined in the university: the name of the university and administration services. Note, that the compartment-concept allows the programmer to define

context-dependent behaviour. In our example one could define different versions of the three roles. They may be included in another compartment named "Engineering School". Depending on whether a *Person*-object plays the "Supervisor"-role in the "University"-context or in the "Engineering School"-context, it may urge the respective "Student"-role to write a research-oriented thesis or to find a practice partner for the final assignment to gain some desirable practical experience.

The mapping of role-related concepts to concepts related to peer-to-peer coordination

Some of the described concepts of role-based modeling can be nicely mapped to concepts that are relevant for peer-to-peer coordination in a dynamic SoS.

According to the role-based modeling approach, base-objects are objects whose behaviour should be modeled context-dependently. In a coordinated dynamic SoS-scenario the behaviour of constituent systems should be modeled somehow in depence of coordination rules that are defined in a dynamic SoS. A coordination rule can be understood as a special context that imposes certain conditions on constituent systems. Therefore, the concept "**base-object**" of the role-based modeling approach is promising to be mapped to the concept "**constituent system**" in a coordinated dynamic SoS-scenario.

According to the role-based modeling approach, role-objects are meant to act on behalf of a specific base-object with respect to a specific context. They are bound to- and unbound from a specific base-object at runtime. In a peer-to-peer coordination scenario a coordinator is associated to every constituent system. This coordinator is meant to act on behalf of the constituent system with respect to a specific coordination rule. In a dynamic SoS coordination rules are applied at runtime. This requires the coordinators to be attached to the constituent systems at runtime as well. Therefore, the concept "**role-object**" of the role-based modeling approach is promising to be mapped to the concept "**coordinator**" in a peer-to-peer coordination scenario within a dynamic SoS.

According to the role-based modeling approach compartments represent a context that constitutes a specific viewpoint on several related objects. This viewpoint selects certain detail of those objects that is relevant with respect to this context and suppresses unnecessary details. The concerned objects are collaborating with respect to this viewpoint. In a dynamic peer-to-peer coordination scenario, a coordination rule is implemented by a set of coordinators. Each coordinator is designed to provide functionality that is relevant for the constituent system that it is associated to. It is relevant to achieve compliance to a specific coordination rule. The coordinators are interacting with each other to achieve compliance with respect to the coordination rule. Due to the apparent compatibility of the concepts "**Compartment**" and "**Coordination Rule**" it is promising to map both concepts to each other in a role-based architecture description language that aims at supporting peer-to-peer coordination in a dynamic SoS. In a peer-to-peer coordination scenario a compartment can be understood to encapsulate the behaviour of the constituent systems that is

relevant from point of view of a specific coordination rule.

The compatibility of certain concepts of both domains motivated the application of the role-based modeling approach to implement a solution to the modeling problem of this thesis that adheres to the peer-to-peer coordination pattern.

Shared identity
The usage of role-based programming languages allows to apply the notion of roles to runtime objects. An important question in the design of a role-based programming language concerns the way to bind and unbind a role from a base-object at runtime. A role-based programming language may implement role-binding by making access to a specific role available to a base-object at runtime. This is done in EventArch 3.0 as well. As a consequence, the identity of the obtained "complex-object" (i.e. the object that is composed of the base-object and several role-objects) has to be considered. In EventArch 3.0, the base-object and the role-objects share a common identity.

"Sharing identity" means that the base and its roles are not regarded as "base" and "roles" from their collaborators point of view, but as a self-contained entity that can be uniquely identified. The practical meaning of this condition is that a collaborator can not refer to the base or to a specific role, but just to the "group" of the base and its associated roles.

Role-binding vs inheritance
Objects may achieve functionality through role-binding or by means of inheritance. Both mechanisms can be distinguished from each other. Relevant points include the following:

- Role-binding is dynamic, i.e., the functionality can be achieved and lost at runtime, while inherited functionality is added to/removed from the objects class at compile-time
- A base-object may be allowed to play several different instances of the same role, while inherited data can be instantiated at most once
- Base and role may be implemented in different programming languages and be combined at runtime using classical component technology (CORBA, EJB) or architecture description languages (EventArch 3.0). Opposed to that, a sub-class is conceptually an extended variant of the super-class and is therefore implemented in the same programming language.

2.4 Coupling Strategies

In this section the coupling strategies "loose-coupling" and "tight-coupling" are explained. They are described with respect to the facets "binding", "interaction", "model", "state", "conversation", and "identification". The coupling level of EventArch 3.0 is described for the facets "binding" and "interaction".

The phrase "coupling" is intuitively associated with the notion of certain entities being subject to mutual dependency. In software engineering, people will typically think of interdependent components as being concerned. The level of impact that the established coupling may have on the behaviour of those components depends on the algorithmic decisions that have been taken in the design of those components. Nevertheless, the systems infrastructure to achieve interdependency may encourage a rather loosely or rather tightly coupling. Both strategies are shortly described in the following. EventArchs support with respect to the one or the other strategy is indicated.

Wilde ([24]) identifies 12 facets to describe the level of coupling intensity for service-oriented systems that rely on web-services. The facets "binding" and "interaction" are especially meaningful for EventArch. **Binding** refers to the "process of resolving symbolic names into identifiers" ([24]). Components are regarded as tightly-coupled with respect to the binding-facet if symbolic names are resolved into identifiers long before being looked up during operation. Therefore, compile-time- or deploy-time binding is regarded as tightly-coupled, while runtime-binding is regarded as loosely-coupled. An example for tightly-coupled binding is the internets naming system: DNS. An example for loosely-coupled binding is the mapping of the identifier of a mailing-list to the email-adresses of its receivers.

In EventArch 3.0, components may communicate by publishing events to a symbolic name. The mapping of that name to specific component-names is specified at runtime. EventArch 3.0 is not restricted to that mode of communication (see "Composite AEM" in 2.2.1). Nevertheless, EventArch 3.0 may be used in that way to achieve loosely-coupling with respect to the facet "binding".

Components can be loosely- or tightly-coupled with respect to **interaction**. The coupling is considered to be tightly if all concerned components have to be simultaneously available in the system to interact successfully. If messages can be delivered to an unavailable component when it becomes available again, the coupling is considered to be "loosely" with respect to the facet "interaction". Therefore, the asynchronous mode of communication gives rise to loose-coupling and the synchronous mode of communication to tight-coupling. EventArch employs synchronous communication.

The facet of interaction can be described in further detail. A characteristic property of that facet is the significance of feedback that is typically expected during interactions. Feedback may be a response-value or an acknowledgement/error-message to indicate successful/erroneous processing. In a tightly-coupled interaction a component would block until an awaited feedback has arrived. In a loosely-coupled interaction a component would not expect any feedback. Loosely-coupled interactions may be implemented using event-based interfaces. Nevertheless, acknowledgements and response-values may be implemented in an event-based system using callbacks. Tightly-coupled interactions can be implemented using function-based interfaces. EventArch 3.0 employs event-based interfaces and can be described to be rather loosely-coupled than tightly-coupled with respect to the facet "interaction". Nevertheless, it is possible to

achieve blocking-behaviour if the need arises ("wait-when-block", see 8.2).

According to ([24]) the facets "model", "state", "conversation", and "identification" may be considered as well. Components are tightly-coupled with respect to the facet **model** if they share a common application-level data-model. In this case, all components have to base their algorithms on this model. Components with a shared data-model are often restricted to a certain toolset to achieve marshaling/unmarshaling. In contrast to that, components that are loosely-coupled with respect to "model" do not base their algorithms on a common model. Messages are processed as documents. No special toolset is required to achieve marshaling/unmarshaling. Components may employ an individual data-model. In this case, marshaling/unmarshaling would produce instances of that model.

Another facet of a coupling strategy is **state**. In a stateful interaction it is possible to relieve a service-provider from having to manage the state of interaction of each individual client by encapsulating the relevant state-information in the messages that are exchanged between service-client and service-provider. This approach is regarded as loosely-coupled with respect to the facet "state". Managing the state of each interaction in the service-provider is regarded as a tightly-coupled solution.

The desired behaviour of a system may require components to apply themselves to several related interactions. In this case, the desired behaviour is said to be achieved by a "conversation". The interactions of two components are tightly-coupled with respect to the facet **conversation** if the permitted sequence of interactions has been strictly defined at compile-time. They are regarded as loosely-coupled if the permitted sequence can be discovered at runtime. To achieve that, a component would have to provide its interaction-partner a set of possible interaction-options to choose from at runtime.

Finally, coupling strategies can be distinguished with respect to the facet **identification**. Tightly-coupled solutions rely on a central identification service. This renders components to be unable to interpret a known identity of another component if this identification service should become unavailable. Opposed to that, in a loosely-coupled solution, all components bear the competence to interpret a known identity correctly. To achieve that, comprehensive identities have to be used. An example are the Uniform Resource Identifiers (URIs) ([2]).

Chapter 3

Related Work

EventArch 3.0 was designed in an effort to improve language support for the concern of coordination in dynamic SoS. Certain requirements and features of a programming language are desirable to achieve that support. Those requirements and features are described in that chapter. EventArch 3.0 is a solution that provides all those features. The devised solution relies on concepts that originate from the field of "role-based modeling". This field has inspired other languages as well. In this chapter current role-based languages are described. They are analyzed with respect to the desirable features. Areas of improvement of their role-based language support for the concern of coordination in dynamic SoS are pointed out. The presentation will use OT/J as a reference-example. Therefore, this language is considered in greater detail.

3.1 Requirements

In this section requirements are defined and clarified, that would be desirable to be met by an ADL in order to provide a proper support for the dynamic application of coordination rules in a dynamic SoS.

In a static SoS no constituent systems will join or leave the SoS at runtime. This relieves an ADL from providing a mechanism for applying/withdrawing a coordination rule to/from a group of constituent systems at runtime. Instead, coordination rules can be applied at compile-time. Compile-time mechanisms are not sufficient for scenarios that involve dynamic SoS. In such scenarios constituent systems may join or leave the SoS at runtime. The following requirements have been identified to achieve applicability for an ADL to scenarios that involve dynamic SoS.

1. **To cope with coordination scenarios that involve a dynamic SoS, ADLs should support to condition the application of coordination rules with respect to the presence and absence of constituent systems in the SoS.**

   Compile-time application of coordination rules is insufficient for the following reason: applying a specific coordination rule may just be appropriate if specific constituent systems have joined the SoS. While the presence of those systems

can be assumed for a static SoS, it can not be assumed for a dynamic SoS.

In those cases, the coordination should not be applied prematurely, i.e., before all required systems have joined the SoS. Premature application may impose unnecessary restrictions to the systems that are present. Instead, the coordination rule should be applied if all required systems have finally joined the SoS. To support those use cases, an ADL should support defining conditions for the application of coordination rules. Those conditions should be based on the presence and absence of certain constituent systems.

2. **In order to dynamically integrate constituent systems with coordinators of the coordination rules, connectors should be employed that can be activated/deactivated at runtime and that provide a modular implementation of that cross-cutting concern, i.e., the concern of connecting all components of the constituent system to the respective coordinators.**

   The goal of this thesis is to devise a mechanism to allow for the application of coordination rules at runtime. To impose the behavioural restrictions of a coordination rule on the behaviour of constituent systems, certain coordinators can be installed. Those coordinators have the task to take influence on the behaviour of the constituent systems as a whole. To connect the constituent systems to those coordinators, connectors are required that are able to process the commands that have been issued by those coordinators to take influence on the behaviour of the constituent system as a whole. To implement those commands, the connectors have to take influence on the constituent system as a whole. That means that they may have to take influence on each of the components of the constituent system. Therefore, those coordinators can be thought of having to implement a cross-cutting concern. An ADL would therefore benefit from providing means to achieve a cross-cutting implementation of those connectors.

   Moreover, activating and deactivating those coordinators at runtime would allow the constituent system to dynamically uphold and give up the cooperation with the coordinators. This can be used to implement the dynamic application of coordination rules and should therefore be provided by an ADL that aims at supporting the dynamic application of coordination rules.

3. **In a dynamic SoS-environment the dynamic application of coordination rules should be not performed by the individual constituent systems, but by a specialized SoS-member.**

   In a static SoS coordination rules are applied at compile-time. That means that the rules state of application will not change at runtime. There is no transition from “unapplied” to “applied”. This is different for a dynamic SoS. In a dynamic SoS there will be such a transition at a certain moment in time. At this moment in time all concerned constituent systems have to establish compliance with that

coordination rule consistently. In case of inconsistent application, malfunctions might occur (e.g. a system will not receive the expected response-messages as it is the only system that is currently complying to that coordination rule). To relieve the constituent systems from the necessary synchronization effort to achieve consistent application, a specialized SoS-member should be concerned with that task.

4. **A constituent system should be open to new coordination rules and allow for integrating them into its behaviour at runtime.**

   In a dynamic SoS, constituent systems may join or leave at runtime. The presence of those systems may require coordination with respect to the other systems. To achieve that coordination, new coordination rules have to be defined and integrated into the behaviour of the existing systems. Stopping and recompiling the constituent systems in order to integrate the coordination rules into its behaviour, may be not feasible. In a static SoS this pressure towards runtime-integration of new coordination rules is not present. Opposed to that, the constituent systems of a dynamic SoS are more likely to face the problem of runtime-integration of new coordination rules into their behaviour. Therefore, a core concern of an ADL that aims at coordination in dynamic SoS-scenarios should be to allow for the integration of coordination rules into its behaviour at runtime.

   If recompiling the constituent system in order to integrate a new coordination rule is unavoidable, the extent of necessary code-changes should be kept to a minimum. This is true for both, dynamic SoS and static SoS. In other words, constituent systems should be open for new coordination rules in that their integration should require as few code-changes as possible. This requirement may be supported by an ADL by a proper separation of the concern of coordination from the original concerns of the constituent system.

   Systems-of-Systems are characterized by the emergent behaviour of constituent systems. Those systems may be managed independently from each other. New constituent systems may join the SoS unanticipatedly at runtime and the behaviour of existing constituent systems may change at runtime in an unanticipated way. Therefore, in a SoS the need may arise to integrate new coordination rules into the behaviour of constituent systems at runtime that have not been anticipated at design-time. Therefore, to be "open" for new coordination rules, an ADL should also provide means to easily integrate new coordination rules into the behaviour of constituent systems at runtime that have not been anticipated at design-time.

5. **In a dynamic SoS-environment all communication that is done on behalf of a specific coordination rule should be performed in a separated scope.**

   A coordination rule concerns a group of constituent systems. Those systems are

somehow related with respect to their SoS-level functionality and are in need of coordination to achieve their SoS-level goals. Such a scenario encourages to not rely on individual communication relations, but to rely on a common message scope for all related systems. In a dynamic SoS the global scope can not be used for that as it is unsafe, as unknown systems could join this scope and thereby get access to confidential information that is contained in the stream of coordination-related messages. Therefore, all communication that is done on behalf of a specific coordination rule should be performed in a separated scope.

3.2 Features

In this section language-features are described that may be provided in order to meet the requirements that were described above. The description is given on an abstract level. A more detailed description that is specifically concerned with EventArch 3.0 can be found in the section 6.1. A description in what way the identified features do actually contribute to the requirements is given in section 6.3

1. **Event-based System**: To facilitate communication between components, an architecture description language (ADL) might solely rely on events. The purposeful exclusion of function-based interaction is understood to be a feature in this thesis. This feature facilitates base-role integration by means of a common message scope (see 6.1). To achieve event-based communication, an ADL might allow for the definition of event-types and event-based interfaces.
2. **Dynamic Composite AEM**: An architecture description language might provide for the integration of base and roles by means of a common message scope. This would simplify role-binding from a technical point of view. Moreover, it would simplify integrating roles with bases at runtime that have not been anticipated at design-time. Such a common message scope will be termed "Dynamic Composite AEM" in those chapters of this thesis that are primarily concerned with EventArch 3.0.
3. **Cross-cutting Roles**: An architecture description language might provide a solution to encapsulate systems in a way that allows for both, modularizing concerns that cross-cut through multiple components of a system (like "coordination") and for the dynamic activation and deactivation of the code that implements those modularized concerns. Such a solution is termed "Cross-cutting Role" in this thesis.
4. **Programmable Role-Binding - Role-Binder**: The task of role-binding may involve complex information that originated in different constituent systems of the SoS. Role-binding may be required to facilitate the consistent application of a specific coordination rule to all constituent systems of the SoS that are concerned by it. To implement that complex task - that may be subject to change throughout the lifetime of the SoS - an architecture description language

may prefer a modular solution. A "Role-Binder" is presented as such a modular solution in this thesis.

5. **Support Notion of Compartment**: Entities that are related by collaboration might be encapsulated in a common message scope. This achieves privacy and increases the comprehensibility of the purpose of collaboration. On the architectural level, entities may collaborate in order to achieve compliance with a certain collaboration rule. In this thesis an architecture description language is presented that facilitates the encapsulation of those entities by means of a common message scope. That message scope is termed "Compartment".
6. **Two-layered Role-Binding**: A role may be concerned with interfering with the behaviour of its base at certain critical points of execution. To achieve that, the behaviour of the base has to be monitored and analyzed, the interference has to be planned, and the interference has to be performed, i.e., executed. The tasks of monitoring, analyzing, and executing can be separated from the task of planning. Both, the entity that is responsible for monitoring, analyzing, and executing and the entity that is responsible for "planning for interference" may be exchanged at runtime, depending on certain criteria. Both entities may be designed as interacting roles. In that way, it might be useful for an ADL to introduce two levels of role-binding. Such a feature is termed "Two-layered Role-Binding" in this thesis.
7. **Base-Role Integration on Architectural Level**: An architecture description language may introduce base and roles as separated components that provide relatively fixed interfaces. To achieve that, the base-code and the role-code is encapsulated in a component that adheres to a certain component model. This would allow for the implementation of a role in a programming language that is different from the programming language that the base has been implemented in. An ADL that features the described separation of base and roles can be described to support the feature "Base-Role Integration on Architectural Level".

3.3 OT/J

This section gives a short description of the role-based language "OT/J". Support for the role-related concepts and the mechanics of role-binding are emphasized. The presentation is enhanced by a code-example. Familiarity with the role-concept is assumed (see 2.3).

ObjectTeams/Java (OT/J) is a role-based language extension for the "Java" programming-language ([17]). It was devised in an effort to improve support for the role-based concepts in mainstream programming languages. Its support was analyzed and rated in ([20]). OT/J is interesting for this thesis as it is the "most sophisticated approach to context-dependent roles so far" ([20]) Moreover, it is actively developed. OT/J's lan-

guage support for role-based programming primarily relies on the language elements "Team", "Role", "played-by clause", "Call-In-/Call-Out Binding" and "Guards".

OT/J's language-element "**Team**" is an implementation of the role-related concept "Compartment" (see 2.3). A teams definition may contain class-definitions and other properties and functionality. Teams can be related by inheritance. They also can be nested. Each class-definition that is contained in a team is understood by the OT/J-compiler as definition of a role-type.

A "**Role**" is therefore an instance of a class that is defined within a team. A role is restricted to the scope of its Team-definition. That means that it may only reference other classes that are defined within this definition and only employ properties and functionality that are defined within it likewise. In other words, a role may only collaborate with other roles of the same team and resort to functionality that is collectively owned by all its members.

Roles are associated to a specific base-class. Role-instances can only be played by instances of their base-class. This association is defined by a "**played-by clause**" that is added to the head of the role-types class definition.

Role-functionality is not deliberately called by the respective base-class. The availability of role-functionality is actually transparent to the base-class. Instead, a call to a method of the base-class may be forwarded to a method of the role-class. The call to the roles method may either replace the call to the base-method or be performed before or after it. The described behaviour can be specified in the definition of the role by so-called "**Call-In Bindings**". A call-in binding contains the names of the methods to be associated and one of the qualifiers *replace*, *before*, *after*.

The language element of "Call-In Binding" is complemented by the language element "**Call-Out Binding**". Call-out bindings are specified in the role-class as well. They associate a method that is defined in a role-class to a method that is defined in a base-class. Calls to the roles method will be replaced by calls to its associated base-method at runtime.

To further condition the execution of role-methods due to call-in binding, guards can be defined. A **guard** is a boolean expression that is evaluated before a role-method is invoked due to a call-in binding. Guards can apply to individual call-in bindings, all call-in bindings that refer to a specific role-method, all call-in bindings of a role, or all call-in bindings of all roles that are defined within a specific team.

Listing 3.1 contains a part of the definition of a role-class named *Observer*. It is contained in a team named *ObserveLibrary*. A played-by clause is defined to associate the role-class to the base-class *BookManager*. Calls to the roles method *update* are forwarded to the bases method *updateView* due to a call-out binding. Three call-in bindings are defined. For example, calls to the bases method *buy* are followed by calls to the roles method *afterBuy*.

```
1 public team class ObserveLibrary extends ObserverPattern {
2     public class Observer playedBy Bookmanager  {
```

```
3     // Callout method binding: bind an action to the update event.
4     update -> updateView;
5
6     // Callin method bindings: bind events to trigger the start/stop
7                                                          operations.
8     start      <- after buy;
9     stop       <- before drop;
10    afterBuy   <- before buy;
```

Listing 3.1: OT/J example code: role definition

The following information are relevant with respect to the mechanics of role-binding:

- Roles are not bound individually, but collectively at Team-activation
- Teams can be activated imperatively by calling the *activate*-method of a Team-instance.
- On activation of a Team-instance, all objects that are instances of a base-class that is a role-player of one of the teams role, get a role-instance bound.
- A role-instance is created and associated to a base-instance when the Team-instance is activated for the first time. The same role-instance will be bound to the same base-instance at subsequent activations of the Team-instance.

3.4 Other Role-based Languages

This section shortly describes the role-based languages "SMAGs", "SCROLL", "Helena Approach", "powerJava", "Rumer", "E-CARGO". It presents background- information to understand the next section, which is concerned with areas of improvement that those languages have with respect to supporting coordination in dynamic SoS. The description focuses on the central language-intentions and on the support for role-related concepts.

SMAGs

Smart Application Grids (SMAGs) is a role-based composition system that aims at providing support for unanticipated, dynamic adaptation of the grids components (application-components or systems). Unanticipated, dynamic adaption can be distinguished from anticipated, dynamic adaption. It is achieved at runtime by adaptation-measures that have not been foreseen or planned for at design-time. Dynamic adaption can be modeled by means of a feedback-loop ([14]). SMAGs is primarily concerned with applying modifications to applications at runtime, i.e., with the "act"-phase of that feedback-loop. In the following, SMAGs component model, -composition technique, and -composition language are outlined.

An application grid consists of interacting applications. A running application can be understood as a runtime-component of that grid. According to SMAGs **component model** a SMAGs-component implements certain "PortTypes". Those PortTypes are

specified by the components "ComponentType". Therefore, a SMAGs-component implements a ComponentType. PortTypes are implemented by "Ports". Ports are instantiated as well.

Ports can be attached and detached to applications at runtime. Their instances can be therefore understood as "Roles". Consequently, Ports can be understood to implement a (concrete) "Role-Type" whose interfaces have been (abstractly) specified by the PortType. The described conceptual setup of SMAGs component model is depicted in figure 3.1. The figure has been taken from [25].

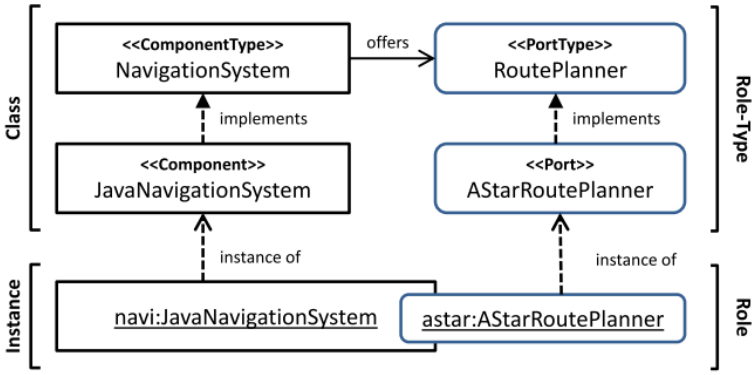

Figure 3.1: Illustration of SMAGs component model

A PortType may specify functions and properties ("BehavioralPortType") or events ("EventPortType") that are provided by this SMAGs component to other components. Other components can register for events of an EventPortType. A SMAGs component may require certain other PortTypes to provide the specified functionality. Those required PortTypes are specified in a PortType as well.

The functions that are specified by a PortType may be implemented by both, a Port and the application that is encapsulated by the component. Calls to the functions are delegated to the Port-implementation first. The Port-implementation can process the call partially and pass it to the application-implementation. Ports can be stacked, i.e., a Port-instance can decide to pass it to a subsequent Port-instance as well.

To allow for automatic composition at runtime, metadata can be added to a component.

SMAGs offers means to compose applications and to achieve interface-compatibility. This **composition technique** includes the concepts of "passive connectors", "Adapter-Ports", and "FilterPorts". For details see [25]. Components can be composed with PortTypes and Ports using the composition operators "extend" and "bind/unbind". The extend-operator can be used to add new PortTypes to a component at runtime. This is a prerequisite to achieve unanticipated, dynamic adaption of applications. The bind- and unbind-operator are used to attach a specific Port-instance to a component-instance at runtime.

The composition of components can be described at two levels. SMAGs can therefore be thought of to provide a two-tiered **composition language**. At the metaarchitectural level, ComponentTypes and PortTypes can be composed with each other. At the architectural level, ComponentTypes can be mapped to components and PortTypes can be mapped to Ports. The architectural specification is represented at runtime by a runtime-model. The extend-operator (see above "composition technique") is implemented as an extension of that runtime-model of the architectural representation. Changes to that model are reflected by corresponding changes to the interface-structure of the running component-instance.

SCROLL

SCROLL (SCala ROles Language) is "a simple implementation pattern for role-based objects" ([21]) to solve programming tasks that are characterized by increased complexity and context-dependence. SCROLL emerged out of the desire to popularize the application of the role-concept in the software-development practice. Therefore, the pattern is defined independent of a specific runtime system and can be implemented as a library for an existing system. The "SCROLL library" is a lightweight-implementation of the pattern for the programming language SCALA. Certain requirements for a system to implement the pattern are identified, but workarounds are presented as well in [21].

Besides popularizing the role-concept, SCROLL wants to demonstrate its applicability for enabling view-based programming. Views are understood as partial representations of a system whose composition might achieve the system as a whole. In this case views are described to be "constructive". Views represent a related set of concerns and can therefore be used to further the separation of concerns in a system. In SCROLL, views are implemented as a set of related roles which are encapsulated by compartments. The compartment represents a certain "viewpoint" on the system. The activation of a compartment causes all encapsulated roles to be activated that are currently bound to a role-player.

Roles can be bound dynamically by the play- and unplay-operation. They can contain additional functionality and be related by inheritance. SCROLL features an elaborate dispatch-mechanism that allows the programmer to dynamically define dispatching rules prior to each function call. Role-binding functionality can be encapsulated by means of a statemachine (see [3]). SCROLL is subject to an ongoing development which strives to increase its feature-base with respect to the criteria identified by [20]. The evaluation in this thesis is primarily based on the description published in ([21])

Helena Approach

Helena Approach provides a formal foundation for the modeling of massively distributed systems that involve "complex interaction structures of concurrently running individuals" ([16]). Roles encapsulate attributes and operations that enable the respective role-player to collaborate as participant of a specific ensemble. They condition participation to the ensemble by requiring their role-players to be of one of several component types. Role-relations can be restricted by "role-connectors". A

role-connector permits the use of certain functions of two role-types for the purpose of interaction. The allowed sequence of operation-calls can be further restricted for a specific role by a so-called "Role Behavior".

Helena Approach clearly distinguishes the description of type-level restrictions and the description of the behaviour of ensemble-instances at runtime. A specific runtime-state is represented by a "Σ-ensemble state". The modeled state-properties include the current instances and instance-data of base and roles and the current played-by relation. A "Σ-ensemble automaton" specifies allowed state-transitions. State-transitions can be restricted by transition-guards. A state-transition can be associated with a role-binding/-unbinding action or with a function call.

powerJava
powerJava is an extension of the programming language "Java" ([6]). It adds support for role-based concepts. powerJava intends to improve available solutions like Object-Teams/Java by providing a better support for the inherent dependency of role, context, and role-player. Role-instances are created as part of a context ("institution"), empowering a role-player to affect the state of this context and access its behaviour through his role. A role-instance can be bound to any instance of a base-type that implements the "required interface" specified within the role-definition. Role-binding is done manually, constrained by interface-compatibility. The programmer can employ Java application-logic to restrict role-binding. Grouped role-binding is not possible. A context is not activated, each role of an institution is bound manually one by one. The institution imposes a soft relational constraint: collaborations are just allowed among roles within the institution. However, any collaboration is allowed within this institution. A role-player can directly call his roles methods to affect institution-state and access institution-behaviour from outside the institution. The player can not call any role-method, but just those specified by the "offered interface" of the role. Roles are bound at instance level.

Rumer
Rumer is a relational model that allows for the specification of systems that are composed of classes and relationships ([7]). It was designed to introduce a representation of object-collaborations by a specific language construct: relationships. A relationship-definition describes the structure of a collaboration by defining the participating roles and the necessary class-types of the corresponding role-players. At runtime, a relationship is a container for sets of tuples of instances of the respective classes. Roles are just placeholders for logical "places" within the tuples of a relationship. They have no behaviour, but can have attributes. Those attributes ("interposed member") are understood to describe the base-object with respect to the relationship. Roles are assigned automatically to a certain base-object in the moment in that it is added to the relationship. The language supports the definition of different types of constraints to restrict the participationship of base-objects to the relation. Constraints may have an intra-relational or inter-relational orientation. Moreover, they can be distinguished to be structural- or value-based. Relationships

can be understood as mechanism to group collaborating roles. A base-object can be arranged to play several roles of a relationship, having multiple roles bound to it at once.

E-CARGO
E-CARGO has been proposed as a "model [...] for role-based collaboration"([29]). It supports modeling human collaboration scenarios like the collaboration of several employees of a firm in different working-groups. Its application domain is the development of computer-supported cooperative work (CSCW) - systems. Roles are used to represent rights and responsibilities of an employee. Roles provide certain capabilities (support certain incoming-/outgoing-messages) and resources (other roles/classes with other i/o-messages) to their role-players. They do not encapsulate additional state or behaviour. Therefore, the same role can be assigned to multiple role-players. The model distinguishes between human-participants: "users" ; and their logical representations: "agents". Roles are played by agents, not by users. Users use agents to make use of the capabilities granted by the role. An employee is assumed to play at a certain moment in time at most one role. An agent can therefore not play multiple roles simultaneously. The decision whether a role is to be bound or not, is not taken by logic or other predefined conditions. A participant willing to play a role or change his role, starts a negotiation with another participant. This other participant plays a role that allows him to take the decision. Depending on his decision the role is bound or not.

There are special roles that allow a participant to define new roles at runtime that can be played afterwards by other participants. Roles enable the role-players to access certain resources. Roles and their respective capabilities and resources are grouped in "Environments" and "Groups". To obtain all i/o-messages of all roles of a Group, the Group is simply added to the roles set of resources.

3.5 Areas of Improvement

This section describes in what way OT/J and the other languages still lack support for the concern of coordination in dynamic SoS. Areas of improvement are identified. The languages are rated with respect to their support for the features that were described in section 3.2.

3.5.1 OT/J

OT/J's approach to role-based programming shows certain disadvantages if applied to the coordination-problems that are prevailing in a dynamic SoS (see 1.1). The following disadvantages have been identified:

- Call-in-/Call-out bindings constitute a tight-coupling between base-class and role-classes. No newly created role-classes can be integrated with the base-class

at runtime. Therefore, role-behaviour that was unanticipated at design-time can not be integrated at runtime.

- Role-functionality that cross-cuts through several base-classes can not be modularized.
- The language does not support stateful role-binding. Instead, the user has to implement a Role-Binder on his own.

Call-in-/Call-out bindings constitute a tight-coupling between base-class and role-classes.
In OT/J, role-base integration is not achieved through the exchange and interpretation of messages. Instead, an involvement of base and role is created using call-in- and call-out bindings. A call-in binding defines a runtime-interference of a roles method with a method of the base. It can be implemented by equipping the source-code of the base with suitable calls to role-functions at compile-time. Call-out bindings can be implemented in a similiar way. This introduces a tight-coupling between the implementation of the base-class and the implementation of the role-classes. While this would allow for the dynamic application of coordination rules, this tightly-coupled mechanism would be not sufficient to integrate a coordination rule into a running system that has been unanticipated at design-time. Roles could be bound to- and unbound from a base that act as a coordinator for the base with respect to a specific coordination rule. But it would not be possible to bind a role to a base that has been defined after the OT/J-compiler has finished the compilation-process. The reason is, that certain function-calls have to be incorporated into the base-code at compile-time to implement a call-in binding. An alternative would have been to connect a base with its roles by means of a common message-scope. Then, a message-based integration of base and roles would have been possible.

That solution would have allowed to exchange the implementation of a role at runtime. The common message-scope would have decoupled the base from the roles. The call-in bindings could have been designed to facilitate an involvement between the base-object and the common message-scope. The base-code would have had to be equipped with new code at compile-time, but this new code would not have called a specific role-method. Instead, that code would have published a certain message to the common message scope of base and roles. With such a mechanism at hand, it would have been possible to load the new role-implementation from a jar-file using a custom classloader and to instantiate a new instance of that class using Javas reflection-capabilities.

A workaround in todays OT/J to decouple base and roles with respect to call-in binding would be to define a role that is responsible for creating all instances of the other roles. This role would process a call-in binding by publishing a suitable message to all other role-instances. The implementation of that workaround is rather unintuitive and can not be recommended.

Role-functionality that cross-cuts through several base-classes can not be

modularized. In certain use cases, concerns exist whose relevance is distributed across the structural units of an entity. The concerns of logging and security are examples for that. Those cross-cutting concerns can be implemented in a modular way. Moreover, those concerns may gain and loose applicability, depending on the current context of the entity. In another case, a different implementation of those concerns may be desirable depending on that context. In both cases, roles may be used to modularize those cross-cutting concerns. In OT/J, those "Cross-cutting Roles" are not optimally supported as it is not possible to associate a role-class with several base-classes. Instead, multiple roles have to be defined and individually bound to multiple base-classes. The following code-example illustrates that problem with respect to the use case that is described later in this thesis in section 6.4.

```
1  public team class StateCoordinationRule {
2    public class LoadBalancerMigrator playedBy Migrator {
3      notifyAdaptiveSoftware <- after migrate; //when the Migrator-
4           //component has finished the migration-process, it has to
5           //be interrupted
6
7      public void notifyAdaptiveSoftware(){
8        updateAppCompCoordinator();
9        appCompCoordinator.releaseCurrentLock(); //migration has been
10        //finished -> the lock of the AdaptiveSoftware can be released.
11     }
12   }
13
14   public class LoadBalancerMigrationPlanner playedBy MigrationPlanner {
15     waitForProceed <- before plan; //when the MigrationPlanner-
16       //component is about to start planning, it has to be interrupted.
17
18     public void waitForProceed(String load){
19       updateAppCompCoordinator(); //update the teams reference to the
20                                   //coordinator of the other system
21       if (load.equals("underutilized")){
22         LockRequest lbRequest = new LockRequest("loadBalancer");
23          //LoadBalancer and AdaptiveSoftware should not run
24          //simultaneously. Therefore, the respective other has
25          //to be locked.
26
27         enqueueRequest(lbRequest); //dont forget that you, the
28                          //LoadBalancer, have requested a lock.
29         appCompCoordinator.enqueueRequest(lbRequest); //the
30          //coordinator-role of the AdaptiveSoftware has to know,
31          //that you requested to lock the AdaptiveSoftware
32
33         LockRequest headRequest = getQueueHead(); //what is the current
34                                          //lock that we have to respect?
35         while(true){
36           if (headRequest.responsibleSoftware.equals("loadBalancer")){
37             return; //proceed if not the AdaptiveSoftware has requested
38                     //a lock.
39           }else{
```

```
40                  waitForLockRelease(); //if the AdaptiveSoftware has
41                      //requested a lock, wait until the AdaptiveSoftware
42                      //releases that lock.
43              }
44          }
45      }
46  }
```

Listing 3.2: OT/J code-example implementation of "State" coordination rule

According to this use case, the "State"-coordination rule has to be applied if the LoadBalancers *MigrationPlanner*-component is about to start migrating the VM from an "underutilized" server to another server. The "State"-coordination rule requires the LoadBalancer to wait until the current request for software-adaption has been serviced by the AdaptiveSoftware. After finishing the migration-process, the LoadBalancer has to indicate this to the AdaptiveSoftware as well. In order to implement that coordination rule the respective coordinator of the LoadBalancer would have to

- Interrupt the LoadBalancers *MigrationPlanner*-component when it is about to start the migration process
- Continue the LoadBalancers *MigrationPlanner*-component when the AdaptiveSoftware has indicated to have finished servicing the current request
- Interrupt the LoadBalancers *Migrator*-component when it has finished the migration process

MigrationPlanner and *Migrator* are different components of the constituent system "LoadBalancer". Hence, the concern of "State"-coordination is a cross-cutting concern in this example. Therefore, the coordinator is favourably implemented as a Cross-cutting Role. That coordinator can not be implemented in OT/J as Cross-cutting Role. Instead, the coordination logic has to be distributed across two roles. They are named *LoadBalancerMigrator* and *LoadBalancerMigrationPlanner* in the code-example. This example has shown, that in OT/J the coordination logic to implement a cross-cutting coordination rule has to be scattered around the code of multiple role-classes. A possible improvement would be to support the feature of Cross-cutting Roles.

The complete code of this example can be found in the Appendix (see 8.1.1).

The language does not support stateful role-binding. Instead, the user has to implement a Role-Binder on his own. Language-support for stateful role-binding is especially interesting for this thesis. OT/J provides language features that allow for the implementation of a stateful Role-Binder. Nevertheless, it does not support the feature "stateful role-binding" by a dedicated language construct.

Implementing a stateful Role-Binder is not particularly difficult in OT/J. The programmer can take advantage from the following language-features:

- OT/J's team-activation mechanism automatically creates role-instances and associates them to base-objects.

- OT/J is an extension of the programming language "Java". Therefore, all Java-features can be used.
- In OT/J, teams may provide properties and behaviour. A Role-Binder that shall be responsible for a specific team can be implemented as an additional team-function.

While OT/J provides useful language features for implementing a stateful Role-Binder, the user still has to discover those features and their applicability for that purpose. Moreover, the user has to discover the stateful role-binding strategy on his own and is not encouraged by the language to consider a stateful design of the role-binding mechanism. Those disadvantages could be relieved by supporting the feature of "stateful role-binding" by introducing a language-construct that is dedicated to implementing a stateful Role-Binder.

Figure 3.3 depicts an extract from an example-implementation of a stateful Role-Binder in current OT/J.

```
1  public team class StateCoordinationRule {
2
3  private State noApplicationThere = new NoApplicationThere();
4  private State oneApplicationThere = new OneApplicationThere();
5  private State twoApplicationsThere = new TwoApplicationsThere();
6
7  private State currentState = noApplicationThere;
8  private State nextState = noApplicationThere;
9
10 public void informRoleBinder(String information){
11   nextState = currentState.getNextState(information);
12
13   if (!currentState.equals(nextState)){
14     currentState.doExitAction();
15     nextState.doEntryAction();
16     currentState = nextState;
17   }
18 }
19
20 public class TwoApplicationsThere implements State{
21
22   @Override
23   public State getNextState(String information) {
24     switch(information){
25      case "LoadBalancerLeft": return oneApplicationThere;
26      case "AdaptiveSoftwareLeft": return oneApplicationThere;
27      default: return twoApplicationsThere;
28     }
29   }
30
31   @Override
32   public void doEntryAction() {
33     activate(Team.ALL_THREADS);
34   }
```

```
35
36    @Override
37    public void doExitAction() {
38      deactivate(Team.ALL_THREADS);
39    }
40  }
```

Listing 3.3: OT/J code-example implementation of stateful Role-Binder

In this example, the stateful Role-Binder is implemented according to the "State" design pattern. Incoming informations are passed to the *currentstate*-object to determine whether a state change is necessary. If this is the case, the *currentstates* exit-action- and the *nextstates* entry-action are executed. The respective constituent system may inform the Role-Binder about its presence by calling the *informRoleBinder*-function. If all awaited systems are present, the Role-Binder will activate the team. This is done on entering- and leaving the state *TwoApplicationsThere*. Role-instances are automatically bound to all instances of all relevant system-components. On leaving the *TwoApplicationsThere*-state, the Role-Binder will deactivate the team. This is done if one system has informed the Role-Binder that it is going to leave the SoS.

The complete code of this example can be found in the Appendix (see 8.1.1).

3.5.2 Other Role-based Languages

The features that have been identified in section 3.2 further the ability of an architecture description language to dynamically apply coordination rules, based on the composition of constituent systems in the SoS. Most of the role-oriented programming languages, that have been described in section 3.4, do not intend to describe behaviour on the architectural level. Nevertheless, their language-design provides a certain applicability for the task that is of interest to this thesis, namely the dynamic application of coordination rules in dependence of the composition of constituent systems in the SoS. To determine the individual applicability of each language with respect to this purpose, its fittness to support the features that have been identified in section 3.2 has been analyzed. The result of that analysis is presented in the feature-table 3.1.

In the following, for each language the most striking areas of improvement are highlighted, that exist with respect to its respective support for the concern of coordination in dynamic SoS.

SMAGs
SMAGs does currently not support the notion of a common message scope of base and roles. ("**Dynamic Composite AEM**). To achieve that notion, PortTypes may publish in response to each function call a message that represents that function call to a message scope that includes all Ports that are currently associated to the SMAGs-component. Nevertheless, it is possible to exchange the implementation of a Port at runtime and to extend the SMAGS-component by new PortTypes at runtime,

Feature	EventArch 3.0	SMAGs	SCROLL	OT/J	EventArch 2.0	Helena Approach	powerJava	Rumer	E-CARGO
Event-based System	+	+/-	-	-	+	-	-	-	+
Dynamic Composite AEM	+	-	-	-	+/-	-	-	-	-
Cross-cutting Roles	+	+/-	+/-	-	+/-	+/-	-	-	+/-
Role-Binder	+	+/-	+	-	-	+/-	-	-	+
Compartment	+	+/-	+	+	-	+	+	+/-	+/-
Two-layered Role-Binding	+	-	+/-	-	+/-	-	-	-	-
Architectural-level	+	+	-	+/-	+	-	+/-	+/-	-

Table 3.1: Feature table for EventArch 3.0 and related languages

even without a dcaem. But still, supporting the notion of a DCAEM may simplify the implementation of those features. To attach a certain Port to a specific PortType, it would just have to be included in a certain message scope. All PortTypes would publish all messages by default to this message scope. As a disadvantage, the described solution would require an additional mechanism to select a Port out of multiple Ports of the scope that implement the function that is represented by that message.

In SMAGs, Ports can be attached to SMAGs-components. SMAGs-components represent applications or systems. Therefore, Port-functions can be called by different classes or system-components. Nevertheless, SMAGs ports do not modularize a cross-cutting concern, as the function-calls still scatter around those classes and system-components and are tangled with the implementation of other concerns. Therefore, SMAGs partly supports the notion of "**Cross-cutting Roles**".

SCROLL

In SCROLL, a base can start to play a role-object at runtime. The playedBy-relation is defined dynamically. When a compartment is activated, all role-objects that are instances of role-classes defined in it are bound to their respective role-players. This is implemented by SCALAs feature "dynamic mixin". To implement the SCROLL-architectural pattern in another programming language, this language has to support "dynamic mixins" as well. Currently, this feature is not widely supported among programming languages. A common message scope of a base and *all* its roles ("Dynamic Composite AEM") would be easier to implement for most languages and might be therefore a factor in easing the task of implementing the SCROLL-architectural pattern in other programming languages.

To implement that message scope, function-calls would have to be represented as messages. A mechanism would have to be implemented to deliver messages to all

members of that scope. If multiple members implement the function that is represented by the message, a specific member would have to be selected that is responsible for processing that message. Possibly, the dynamic-dispatch mechanism that is available in the current version of the language can be reused in a certain way for that purpose. It may be used to determine an order in that the messages are published to the members of the scope. The first member that implements the function may be responsible to compute the return value.

Currently, SCROLL does not support a notion of Cross-cutting Roles. Nevertheless, there exist cross-cutting concerns that may be subject to context-dependency on the applicational level. Examples include logging and security. For example, a Cross-cutting Role could be used to attach logging-functionality to a set of objects. This logging functionality may depend on the context that the object is experiencing at logging-time. For example, a more exhaustive logging could be required if the object is experiencing at logging time a context that is more critical to achieving the applications goal than another context.

EventArch 2.0
EventArch 2.0 is concerned with integrating constituent systems with coordinators that implement coordination rules. It aims at achieving comprehensibility and preserving the reusability of the constituent systems in other SoS or outside any SoS-context. While EventArch 2.0 is able to achieve the integration of constituent systems and coordinators for static SoS, it is not capable to achieve that integration for dynamic SoS. Composite AEMs are static entities. They can not gain and loose Composite Interfaces at runtime. Therefore, coordinators can not be bound to- and unbound from a DCAEM at runtime. Primitive interfaces modularize a cross-cutting concern. Nevertheless, they lack support for dynamic behaviour as well. They can not be activated and deactivated. Moreover, the coordinators of a coordination rule that is implemented according to the "Peer-to-Peer"-coordination pattern can not be encapsulated in a common message-scope. Instead, messages are published into global scope.

E-CARGO
E-CARGO has been proposed as a model for CSCW (computer supported collaborative work) scenarios. This model integrates several concepts that are relevant in that application domain (e.g. human-collaborators, resources, groups). E-CARGO is described on a rather conceptual level. No syntax for specifying a CSCW-scenario has been proposed ([29]). The description does not approach questions concerning the implementation of that model in current programming languages. Nevertheless, it can be argued in what way the features that have been identified in section (3.2) could help to improve the design of that model and could allow for a seamless implementation in current programming languages.

Agents may act independently or on behalf of a human user of the CSCW-system. An agent can send messages to other roles that are provided by its own roles. An agent could be integrated with its roles by means of a common message scope (DCAEM).

This scope would contain at most a single role, as a human can not play multiple roles simultaneously, according to the E-CARGO model. With a common message scope, the agent could keep publishing messages to that scope and would not have to switch the destination of its messages whenever a new role is played. Moreover, the common message scope would allow for a feasible implementation in current programming languages.

Helena Approach
Helena Approach has been proposed as a "formal foundation for ensemble modeling" ([16]). Certain concepts that are relevant for ensemble modeling are developed and formally defined. Examples include the "Ensemble Automata", which defines the desired behaviour of an ensemble, and the "Ensemble Specification", which encapsulates a specification of allowed role-to-role relationships and defines certain behavioural restrictions for individual roles. Helena Approach is open for different implementations of those concepts. Nevertheless, it might be advisable to associate a component to all its roles of a certain ensemble by means of a common message scope. In that way, an individual message scope could be established for each ensemble that the component is participating in. Calls to specific role-functions could be represented by specific messages. Role-binding and -unbinding could be implemented by adding and removing roles to that message-scope. But still, a mechanism would have to be implemented to determine which role should be responsible for processing a specific message if multiple roles implement that function.

In Helena Approach, roles act on behalf of components. Components can be thought of as consisting of multiple structural units. Every role of an ensemble can be understood as a representation of "capabilities that a component needs when participating in a specific ensemble" ([16]). A "capability" may be functionality to implement a concern that cross-cuts through multiple structural units of the component. Therefore, it would be advantageous to allow for a role-implementation that modularizes this cross-cutting concern and prevents its scattering around the structural units and its tangling with other concerns of the component. Such a role-implementation could be achieved by supporting the notion of "Cross-cutting Roles".

Supporting the notion of "Cross-cutting Roles" would make it advisable to separate the task of extracting the state-information that is relevant for the cross-cutting concern from the component and the task of implementing the concern itself. This would encourage to support the notion of "inner roles" that are responsible for observing the current state of the multiple structural units of the component and translating that state into a representation that the outer-roles can use to implement the concern on their own.

Rumer
In Rumer, roles are used to associate an object with a planned position in a relationship. They may have properties and their state can be processed by the methods of their relationship. Nevertheless, a role can not define own methods and is therefore not capable to change the state of other roles in the relationship. Moreover, roles are

automatically bound to an object when it is added as part of a tuple to a relationship. This design can hardly be improved by most of the features that have been identified in section 3.2.

But still, different relationships may be associated to a common context. In ([7]) the example relationships “teaches”, “assists”, “worksFor”, and “attends” are defined. They capture the relationship between professors and courses (teaches), student assistant and courses (assists), students and faculties (worksFor), and students and lectures (attends). All those relationships can be associated to the context “university”. Therefore, it might be useful to group those relationships in a certain way. To achieve that, support for the notion of “Compartment” could be introduced to the language. A compartment may represent a common context for specific relationships. It may contain additional properties that are specific to that context, as well. Therefore, grouping of relationships may be regarded as an area of improvement in Rumer.

powerJava
powerJava is a role-oriented language that is situated at the applicational level. In powerJava, compartments (institutions) can be instantiated and different implementations of the same role can be defined within the same compartment. Therefore, powerJava requires the programmer to specify at every function-call the compartment-instance and the type of the role-implementation explicitly. To allow for a greater flexibility and to separate function-calls from function-dispatch, Role-Binders could be employed. A Role-Binder could be concerned with selecting the compartment-instance that should be used. Moreover, another Role-Binder could be concerned with selecting the appropriate role-implementation that is available within a specific compartment. Moreover, base and roles could be integrated by a common message-scope. The Role-Binders decision to select the one or the other compartment-instance for activation might be implemented by letting certain roles join or leave the common message scope. In the same way, its decision to select the one or the other role-implementation could be implemented.

Chapter 4

Concepts of EventArch 3.0

In this chapter core-concepts of EventArch 3.0 are explained. This includes role-related concepts like "Base", "Role", and "Compartment" as well as concepts like "Dynamic Composite AEM" and "Atomic Block" that are inspired by other fields of knowledge. The description is concerned with the implementation of this concepts in EventArch 3.0, but it is given on a rather conceptual level. It is enhanced by certain diagrams that illustrate the concepts on an abstract level: package-diagrams, a class-diagram, and a sequence diagram. A more detailed description can be found in the subsequent chapter.

4.1 Base, Role, and Compartment

This section introduces EventArchs 3.0 concepts of "Base", "Role", and "Compartment". Their relationship to the concepts "Constituent System", "Coordinator", and "Coordination Rule" is sketched.

As stated in the motivation (see 1.1), EventArch 3.0 was designed to support dynamic application of coordination rules. This task is related to the following concepts that originate in the problem area "coordination of constituent systems": *constituent system*, *coordinator*, and *coordination rule*. EventArch 3.0 is dedicated to coordination according to the "Peer-to-Peer" pattern (see [18]). According to this pattern, a coordinator is responsible for a single constituent system to achieve compliance with a specific coordination rule. Related constituent systems may be subjected to the same coordination rule. This rule guarantees a certain operational compatibility of the constituent systems that are concerned by it, with respect to achieving the SoS-level goals. It imposes certain restrictions on the behaviour of the constituent systems, e.g., to stop operation if another system has reached a critical state.

Those concepts can be mapped to the concepts *base*, *role*, and *compartment* that originate in the modeling approach "Role-based Modeling" (see 2.3). Roles can be dynamically bound to- and unbound from a base. A compartment comprises a set of related roles. The roles may be related by a common purpose. In this case a compartment may be understood to be dedicated to achieving that purpose. From

point of view of a role, a compartment establishes a context of potential collaboration partners. A base is placed into this context by becoming a role-player, i.e., by getting a role bound.

In EventArch 3.0, a **role** is understood to represent a coordinator that is associated to a single constituent system ("Peer-to-Peer" pattern). The constituent system is understood as the "**Base**" of that role. The role can be dynamically bound to- and unbound from that base. This is a prerequisite to achieve the dynamic application of coordination rules. The preceding description applies to EventArchs 3.0 "Outer Roles". A description of the related concept "Inner Role" is given below (see 4.3).

In EventArch 3.0, a **compartment** is understood to represent- and implement a certain coordination rule. The contained roles represent coordinators that are associated to this coordination rule. A base represents a constituent system that is subjected to this coordination rule by getting a role bound. In EventArch 3.0, a compartment establishes a common scope for all contained roles. It may contain a Role-Binder (see 4.2), but no additional functionality. It contains a fixed set of roles that can not be reduced or extended at runtime (see 6.2).

4.2 Dynamic Composite AEM and Role-Binder

In this section EventArchs 3.0 concepts "Dynamic Composite AEM" and "Role-Binder" are explained. Their relation to the role-concept and their relevance for the concern of coordination in dynamic SoS is pointed out.

The dynamic application of coordination rules is backed by the concepts "Dynamic Composite AEM" and "Role-Binder". Coordination rules are applied by binding coordinators to constituent systems. A coordinator is responsible for implementing a specific coordination rule into the behaviour of a constituent system. The binding of a coordinator is done by establishing a communication relation between a constituent system and that coordinator. Therefore, the dynamic application of a coordination rule can be achieved by dynamically establishing a communication relation between coordinators and constituent systems.

In EventArch 3.0, this is achieved by the language element "**Dynamic Composite AEM**" (DCAEM). A DCAEM is implemented as a common scope for a constituent system and all coordinators that act on behalf of this constituent system. This may include coordinators of different coordination rules. Coordinators can join and leave this scope at runtime. Therefore, a DCAEM can be understood as a dynamic mapping from a symbolic name (i.e. the name of the DCAEM) to a set of coordinator-identifiers. In this way, the concept is contributing to establishing a loose-coupling between a constituent system and its coordinators (the mapping is not fixed, but can be changed at runtime).

From point of view of role-based modeling, the DCAEM can be understood as a

"complex object". It comprises the base-object (constituent system) and all its role-objects (coordinators). The DCAEM-concept of EventArch 3.0 is a variant of the concept "Composite AEM" of EventArch 2.0 (see 2.2.1). In EventArch 2.0, constituent systems and coordinators are encapsulated as Architectural Event Module (AEM). A DCAEM is therefore a common scope for certain AEMs that represent a constituent system and its coordinators. Certain Primitive Interfaces of the AEMs can be associated to the DCAEM. Communication within the scope of a DCAEM is just possible for an AEM using Primitive Interfaces that are currently associated to this DCAEM.

Coordination rules may be employed to prevent unintended behaviour in a SoS. In a dynamic SoS, coordination rules have to be applied at runtime (see 1.1). Therefore, in a dynamic SoS, a point of execution has to be selected at which the coordination rule should be applied to the concerned constituent systems. In EventArch 3.0, this decision is taken by a **Role-Binder**. The Role-Binder decides upon the point of execution and achieves the application of a coordination rule by composing relevant coordinators with constituent systems that are concerned by that rule. A Role-Binder can be responsible for applying a single coordination rule or for applying several coordination rules. In the latter case, it may be used to select one coordination rule out of several alternatives for application. The Role-Binders decision is based on events that have been published by constituent systems. The entry- and exit of a constituent system can be indicated to the Role-Binder by publishing those events. In this way, the Role-Binder may be brought in a position to consider the current composition of constituent systems in the SoS with respect to its binding decisions, and therefore with respect to its decision to dynamically apply a specific coordination rule to the SoS. A Role-Binder provides special statements ("bind", "unbind") to bind a coordinator to a constituent system, i.e., establish a communication relation between a coordinator and a constituent system.

Moreover, a Role-Binder can create and destroy a DCAEM (statements "create", "destroy"). To create a DCAEM, one AEM that represents a constituent system is determined to be the "Base" of that DCAEM (a DCAEM can be understood as a complex object of base and roles, see above). Additionally, a Primitive Interface of this AEM has to be selected to get associated to the DCAEM. In this way, the Role-Binder may be used to select one Primitive Interface out of several alternatives (this is one of the advantages of EventArch 3.0, see 6.1).

Role-Binders are defined as statemachines. State-transitions can be performed in response to certain events that have occured within the SoS. Binding actions can be performed on entering a specific state. This can be used to apply a coordination rule consistently to all constituent systems that are concerned by it. (see 6.1) In this way, the Role-Binder can be used to activate a specific coordination rule, or to select several coordination rules for application.

4.3 Inner Roles and Atomic Block

This section introduces the concepts of "Inner Roles" and "Atomic Block". The significance of these concepts is motivated by their contribution to EventArchs 3.0 language-support for the concern of coordination in dynamic SoS.

In a SoS, several constituent systems may be in need for coordination. Coordination can be used to prevent unintended behaviour on SoS-level. The task of coordination can be decomposed into the following subtasks:

- *Monitor the current state of the constituent systems*
- *Analyze the current state and plan your interference with the constituent systems behaviour*
- *Interfere with the constituent systems behaviour*

The first task may be accomplished by the constituent systems themselves. They might be concerned with informing relevant members of the SoS about their current state. The constituent systems do also have their share in accomplishing the third task. They have to execute the commands that have been sent to achieve the interference. In dynamic SoS a problem arises with respect to accomplishing that task. The need for coordination may appear and vanish at runtime. Therefore, in a dynamic SoS, constituent systems should refrain from monitoring their current state and should refrain from executing received commands if no need for coordination exists. In EventArch 3.0, the concept of "**Inner Roles**" was introduced to cope with that problem.

The concept of "Inner Roles" is derived from the concept of "Primitive Interface" of EventArch 2.0 (see 2.2.1). An inner role is a Primitive Interface that can be deactivated. It is associated to an AEM that represents a constituent system. Deactivation suspends the ability to publish state-changes of the constituent system to coordinators. It also suspends the ability to translate command-events that were received by the coordinators into actions of the constituent system. Deactivating an inner role equals establishing ignorance of all coordination rules whose coordinators depend on this inner role to communicate with the constituent system.

Inner roles are meant for communication of a constituent system within a DCAEM, i.e., between a constituent system and its coordinators. They can not be used for communicating events outside of a DCAEM. Therefore, all inner roles are deactivated if the constituent system is currently not the "Base" of a DCAEM. When a DCAEM is created around a constituent system (i.e. when the constituent system is declared to be the "Base" of a DCAEM), certain inner roles of the constituent system are activated. This is done by a Role-Binder. The Role-Binder can decide which inner roles to activate. In this way, the Role-Binder can decide which state-changes of the constituent system should be communicated to the coordinators and which command-events should be processed in which way by the constituent system. This increases the adaptivity of constituent systems with respect to the concern of coordination

(6.1).

Opposed to the outer roles (or just termed "roles" in this description), the inner roles are tightly-incorporated into the constituent system (see 5.1.3). They encapsulate the tasks of translating the current state of the constituent system into a stream of events and of translating received command-events into actions of the constituent system. Moreover, inner roles can be activated and deactivated. In that way, the inner roles of a constituent system establish an "inner layer of role-binding". This layer can be distinguished from the "outer layer of role-binding" that is constituted by the (outer-)roles of the DCAEM.

A coordination rule is concerned with multiple constituent systems. Activating a coordination rule does therefore require to bind several constituent systems to their coordinators. To prevent malfunctions, a consistent application of the coordination rule to all constituent systems is necessary. (6.1) In EventArch 3.0, this consistent application can be achieved by a Role-Binder. To do that, the Role-Binder can execute several binding commands on entering a specific state (see preceding section). Each command would bind a specific coordinator to a specific constituent system. To guarantee consistency, the successful execution of all binding commands has to be guaranteed. This successful execution can be guaranteed by means of an "**Atomic Block**".

An "Atomic Block" is a sequence of binding commands that is executed atomically. Either all binding commands of this set are executed or none of them. An atomic block is in fact a transaction that is concerned with changing the state of a SoS with respect to binding relations. The beginning and end of this block do mark the BoT-point (begin of transaction) and the commit-point of the transaction respectively. If one of the contained commands can not be executed, the state of the binding relations inside the SoS is restored that was active at the beginning of the atomic block. This operation can be understood as a "rollback" of the state of the binding relations. The following commands can be executed in an atomic block: create/destroy DCAEM bind/unbind role.

4.4 Diagrams

This section presents certain diagrams that describe the implementation of EventArch 3.0 on an abstract level. Two package-diagrams are contained that describe the dependencies among packages that have been changed or extended with respect to the implementation of EventArch 2.0. The role-binding process is illustrated by a sequence-diagram. Two class diagrams illustrate on a conceptual level the relationship between role-related concepts and concepts that are related to coordination in dynamic SoS.

Language-concepts
Figure 4.1 contains two UML class-diagrams to support the above description by a concise graphical presentation. One diagram captures EventArchs 3.0 concepts from

point of view of "Role-based Modeling". The other captures EventArchs 3.0 concepts from point of view of "coordination in dynamic SoS".

Role-base management
Figure 4.2 describes different Role-Binder activities to manage role-base relations (create, bind) on a conceptual level. The issued binding-commands were defined in an atomic block. This diagram clearly shows the dynamic nature of the Dynamic Composite AEM (DCAEM).

Figures 4.3 and 4.4 describe the code-changes on package-level (only changed/extended dependencies are contained). There were two new package-dependencies introduced in the compilation-part (4.3): code-generation employs a validator. It gets interrupted if the validator detects an error. Also, the core-package provides standard interfaces to the *applevel*-package (see next section for details). Moreover, two package-dependencies were extended. The *generator* makes use of the model-objects that represent the newly created binding- and creation-statements. Also, the *generator* added the functionality to the *generated classes* to respect the new DCAEM scope-restrictions.

Some new dependencies have been added to the packages of the runtime-part. Also, some code-changes have been made that contribute to existing dependencies. Figure 4.4 depicts all new and changed dependencies. The details of necessary code-changes are discussed in the next section.

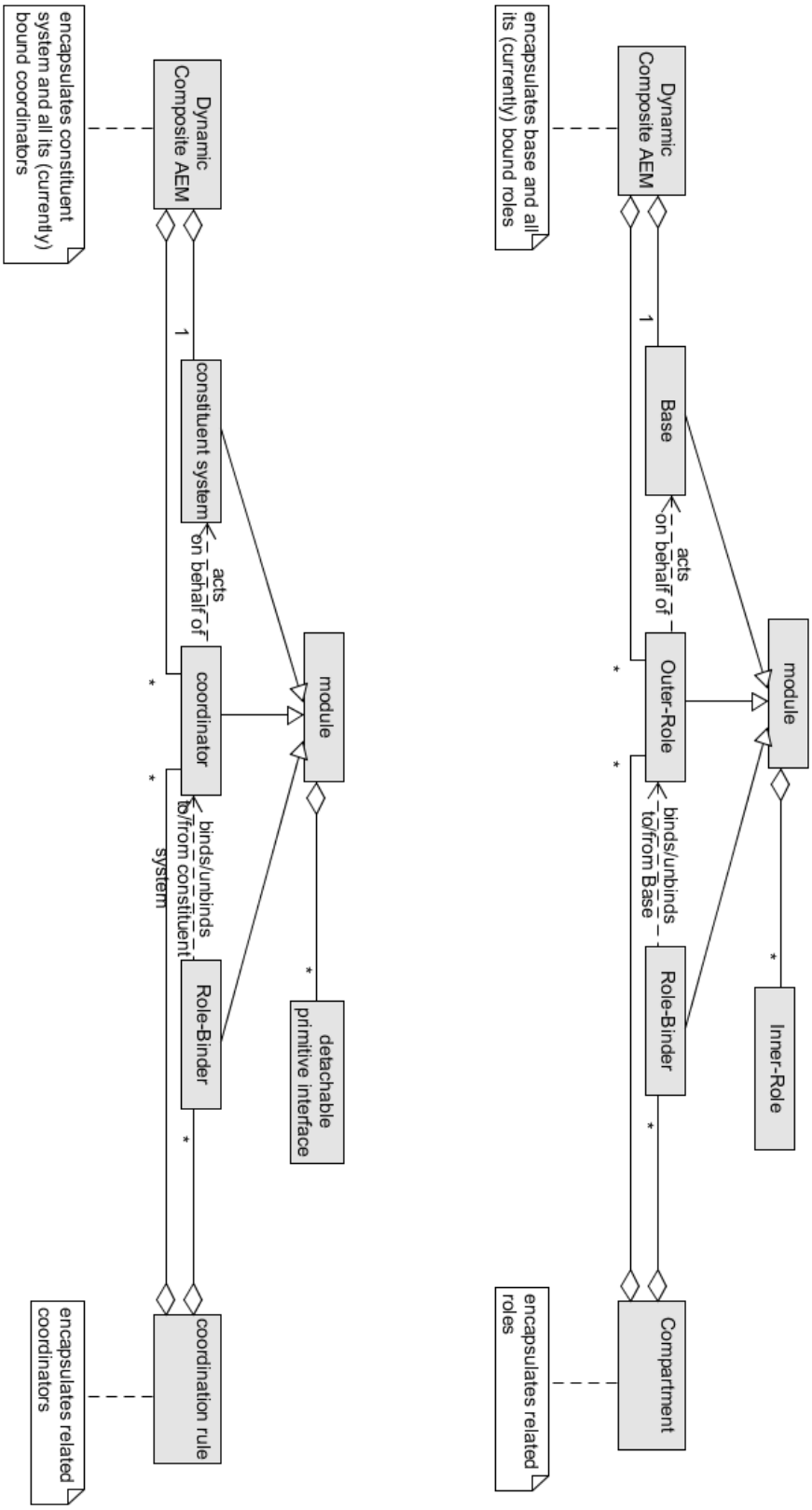

Figure 4.1: EventArchs 3.0 concepts from point of view of "Role-based Modeling" (right) and "coordination in dynamic SoS" (left)

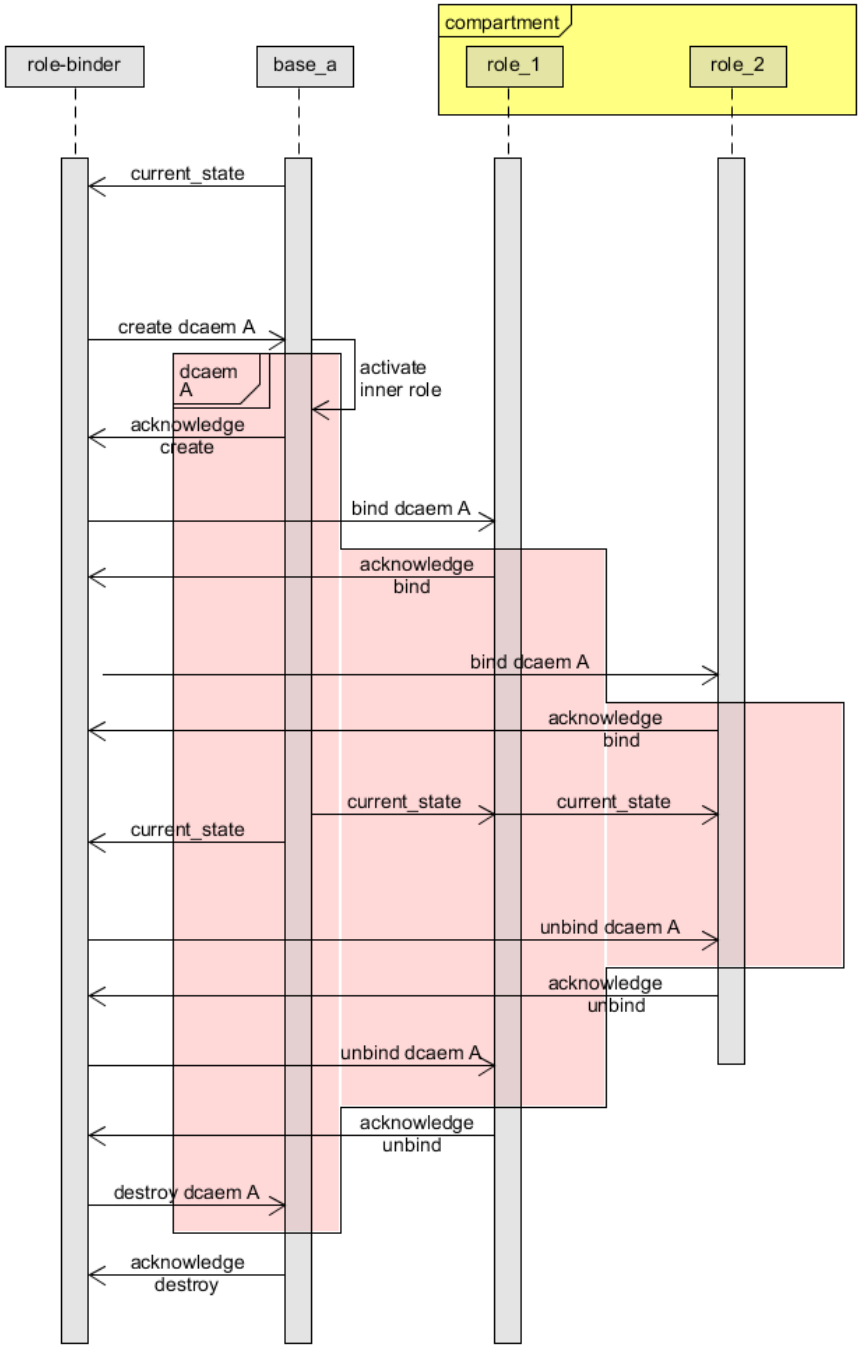

Figure 4.2: Conceptual role-base management

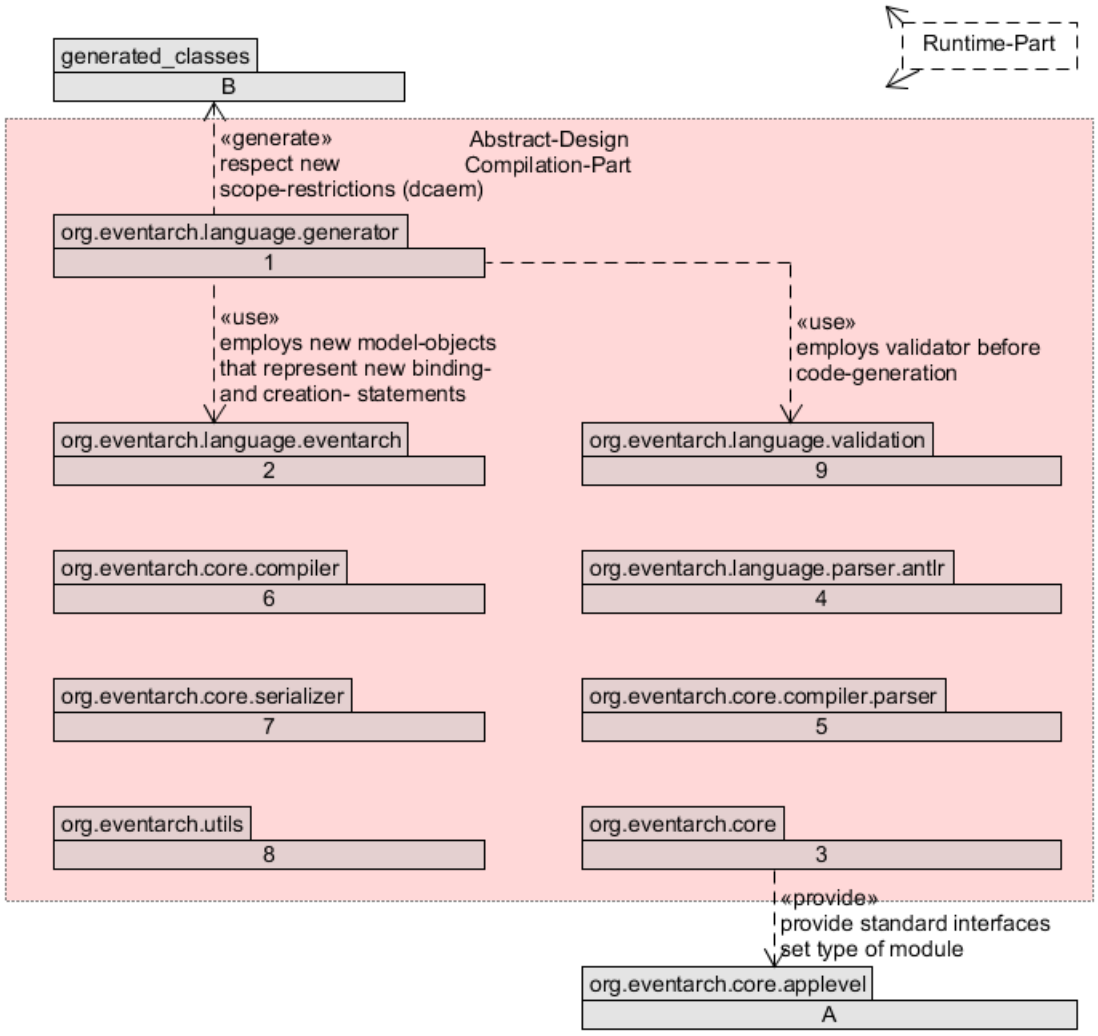

Figure 4.3: Changed dependencies within compilation-part

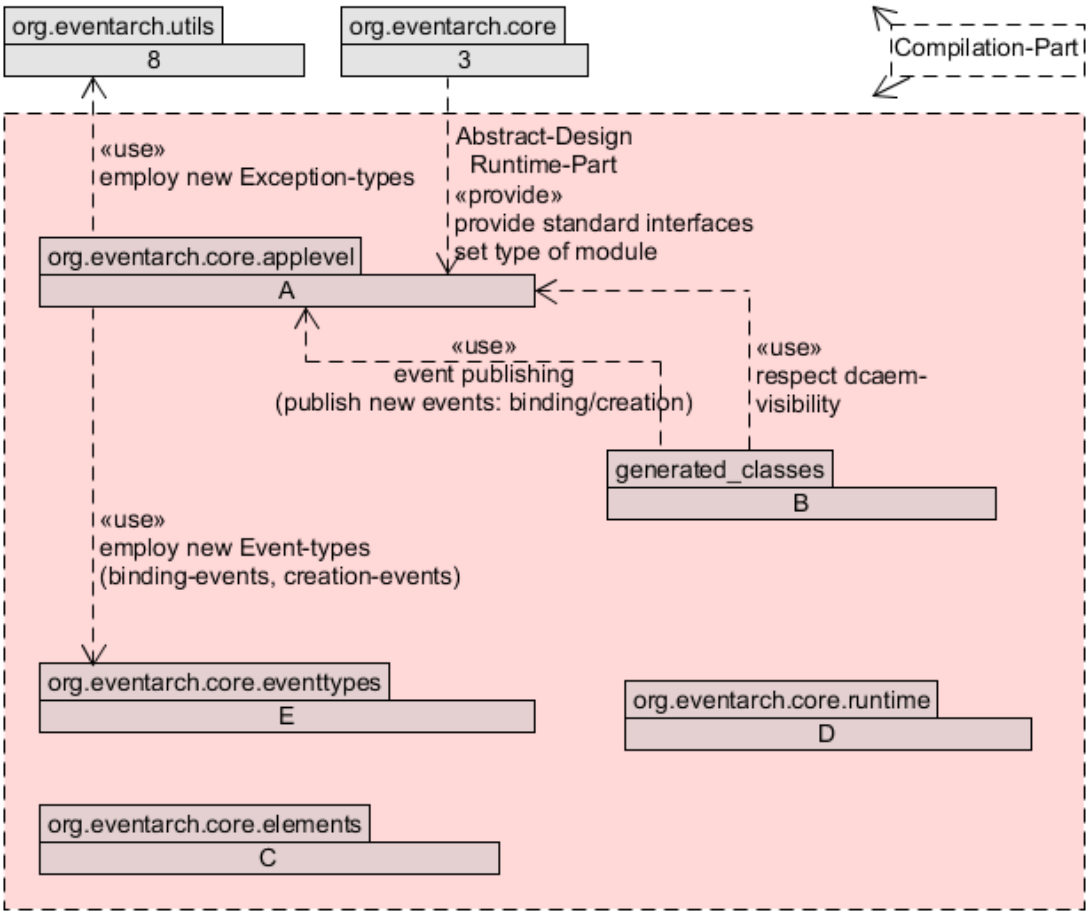

Figure 4.4: Changed dependencies within runtime-part

Chapter 5

Internal Design of EventArch 3.0

This chapter presents an in-depth discussion of the implementation of concepts of EventArch 3.0 that are relevant for the concern of coordination. Moreover, design alternatives are considered. An even more detailed discussion of the internal changes that had to be applied to EventArch 2.0, can be found in the appendix (see 8.4). The presentation goes down to the level of names of individual classes and functions. It will provide a useful introduction for readers who intend to extend EventArch 3.0. Some additional lower-level concepts are explained as well.

5.1 Implementation of the Concepts

The new language concepts (see chapter 4) had to be incorporated into the EventArch-language. To achieve that, the grammar of EventArch 2.0 had to be changed. New language elements had to be introduced. Listing 5.1 shows a piece of code that employs these new elements. A Role-Binder is depicted that employs the binding- and creation commands inside of an atomic block. This section will explain the detailed design solutions that have been taken to introduce the described concepts of "Role-based Modeling" into EventArch 2.0 to increase its language-support for the concern of coordination in dynamic SoS. Certain other concepts that are relevant for EventArchs 3.0 implementation are explained as well. EventArchs 3.0 implementation is described in detail. The description includes class-diagrams that illustrate the structural changes that have been made to the implementation of EventArch 2.0. The detailed description is given on a per-package level. This chapter will especially be useful for readers that intend to extend EventArch 3.0.

```
1  roleBinder SwitchCoRoleBinder[StateMachine] := {IExternal} <-> {
2    initial state noSystemThere{
3      during:
4        on(IExternal.lb_started || IExternal.as_started) {} ->
5                                                    oneSystemThere;
6    }
7
8    state oneSystemThere{
9      during:
10       on(IExternal.lb_terminated || IExternal.as_terminated) {} ->
```

```
11                                                          noSystemThere;
12        on(IExternal.lb_started || IExternal.as_started){} ->
13                                                          twoSystemsThere;
14    }
15
16    state twoSystemsThere{
17      entry:
18        atomic {
19          DcaemLoadBalancer[composite] := {LoadBalancerCoord.
20                                                  IBaseDirectedLB} <->
21                                  {WrappedLoadBalancer.ISwitchCoordinatorLB}
22          DcaemAdaptiveSoftware[composite] := {} <->
23                              {WrappedAdaptiveSoftware.ISwitchCoordinatorAS}
24          DcaemAdaptiveSoftware[composite] += {AdaptiveSoftwareCoord.
25                                                  IBaseDirectedAS}
26        }
27
28      during:
29        on(IExternal.lb_terminated || IExternal.as_terminated) {} ->
30                                                          oneSystemThere;
31
32      exit:
33        DcaemLoadBalancer[composite] -= {LoadBalancerCoord.
34                                                  IBaseDirectedLB}
35        destroy DcaemLoadBalancer[composite]
36        DcaemAdaptiveSoftware[composite] -= {AdaptiveSoftwareCoord.
37                                                  IBaseDirectedAS}
38        destroy DcaemAdaptiveSoftware[composite]
39    }
40  }
```

Listing 5.1: EventArch 3.0 example-code: Role-Binder with binding- and creation commands in atomic block

5.1.1 Base, Role, and Compartment

This subsection describes the implementation of EventArchs 3.0 "Base"-, "Role"-, and "Compartment"-concept. Important properties of "Roles" are described from point of view of language-implementation. Based on that description, the implementation of the concepts "Base" and "Compartment" is described.

From point of view of implementation, EventArchs 3.0 **roles** are characterized by the following properties:

- A module that is able to receive and process bind- and unbind-events
- A module that can be distinguished from modules of the types "Role-Binder" and "Base"
- A module that belongs to a compartment and publishes its events by default into the scope of that compartment

- A module that may have interfaces that are associated to a DCAEM and are therefore restricted to publish to the scope of that DCAEM

Those properties are implemented in the following way:

A module that is able to receive and process bind- and unbind-events: Every module that has been identified as "Role" by the compiler is added a so called "StandardBindingInterface". This standard interface can select *Bind-* and *Unbind* events that have been sent by the Role-Binder. The standard interface is created and associated to the role-module by the *Loader*-class. Those binding-events are processed by establishing or suspending connectivity to the topic (see section 2.2.2) that represents the DCAEM that the role is to be bound to/unbound from. This functionality is implemented in the class *ERGeneratedAppEventmodule*.

A module that can be distinguished from modules of the types "Role-Binder" and "base": roles, bases and Role-Binders are specified by the SoS-manager in the EventArch 3.0 specification. Roles are modules that are marked by the keyword "role" in the specification. Role-Binders are marked by "roleBinder". The definitions of modules of type "Base" are not marked in the specification. The pieces of information that have been provided by the SoS-manager in the specification are represented in the compiler by objects of type "PrimitiveAEM". This is done for modules of type "Role", as well as for modules of type "Base" and "Role-Binder". Alternatives to this solution exist. But in case of EventArch 3.0, the indicated solution should be prefered. (see 5.2) To allow for a distinction of the three module types at runtime and in the process of compilation, the compiler associates each module with the PrimitiveAEM-object AND with a *TypeOfModuleMarker*. The *TypeOfModuleMarker* allows for qualifying a module as "Base", "Role", or "Role-Binder". The *TypeOfModuleMarkers* are created and associated to the respective modules by the *Loader*-class.

A module that belongs to a compartment and publishes its events by default into the scope of that compartment: EventArch 2.0 provides a concept called "instance-group". An AEM-instance can be specified to belong to an instance-group. Instance-groups are implemented as topics. All events of an AEM-instance are published by default to its instance-group. The concept of "instance-group" could be reused in EventArch 3.0 to restrict a role-module to publish its events by default to its compartment. The compiler creates an instance-group for each compartment. All roles that are defined within that compartment are joined into this instance-group at compile-time. No further code-changes had been necessary. Primitive interfaces of a role-module that are currently not associated to a DCAEM do publish their events by default to the scope of the compartment that they belong to. This is actually not enough to guarantee that the role would be unable to publish its events deliberately to another scope than the compartments scope. This guarantee is not achieved by means of code-generation, but by means of semantic-checks (see below).

A module that may have interfaces that are associated to a DCAEM and are therefore restricted to publish to the scope of that DCAEM: Interfaces

that are associated to a DCAEM can not publish their events to another scope than that of their DCAEM. To achieve that behaviour, the state of DCAEM-association was represented in the class *ERAppInterface*. The class was extended by a new property that provided information about the current DCAEM-association. The compiler was adjusted to make the generated code (Primitive Interface-implementation, statemachine-implementation) aware of the current DCAEM-association of the *ERAppInterface*-objects. If the interface is currently associated to a DCAEM, the event is published to the topic of this DCAEM by force. If not, it is published to the default-topic. In case of role-modules, this is the topic that represents the roles compartment.

The concept of "**Base**" is implemented in a similiar way like the concept of "Role". A base-module receives a standard interface of the type "StandardCreationInterface". This enables the module to receive *Create*- and *Destroy* events that have been sent by the Role-Binder. Those events are processed by creating a topic at the communication-provider (JMS, see 2.2.2). Base-modules are associated a *TypeOfModuleMarker*, as well. The marker allows for qualifying them as base-modules. In contrast to roles, bases are not specifically marked in the EventArch 3.0 specification. Modules that are not marked in the specification are qualified by the compiler as base-modules. Primitive Interfaces of base-modules may also get associated to a DCAEM. Opposed to roles, not arbitrary Primitive Interfaces can get associated to a DCAEM but just those Primitive Interfaces that have been marked "private" in the EventArch 3.0 specification. Private interfaces are restricted to publish events to the scope of that DCAEM that they are associated to. Moreover, they are deactivated as long as they are not associated to a DCAEM. Private interfaces implement the "Inner Role"-concept of EventArch 3.0 (see 4.3).

In EventArch 3.0, **compartments** are implemented as a message scope. Therefore, a compartment is represented by a topic (see section 2.2.2). More specifically, they are implemented reusing EventArchs 2.0 concept "instance-group" (see above, implementation of the concept "Role"). Compartments do not provide behaviour and can not be nested.

5.1.2 Dynamic Composite AEM and Role-Binder

In this subsection EventArchs 3.0 implementation of the concepts "Dynamic Composite AEM" and "Role-Binder" is explained. The DCAEMs implementation as a message-scope is revealed. The implementation of the Role-Binder is explained according to the characteristics "Statefulness", "Accessibility", "Exclusiveness", "Qualification", "Dedication".

DCAEMs: Implementation and membership
A DCAEM is conceptually a complex object of a base and all its roles (role-based point of view). From point of view of coordination, it encapsulates a constituent system and all its coordinators. As a DCAEM is not meant to implement any additional

functionality, it was possible to implement it as a common scope for a base and all its roles, or a constituent system and all its coordinators respectively.

A scope can be understood as a symbolic name that is dynamically mapped (i.e., at runtime) to a set of names of event-receivers (roles/coordinators). JMS provides a solution for scope-implementation: topics (see 2.2.2). Therefore, DCAEMs are implemented as a topic that a base and all its roles can get assigned to. The creation of a DCAEM is commanded by a Role-Binder (see below), but the topic is physically created by the base-module itself. The Role-Binder specifies, which Primitive Interfaces of the base-module should publish/subscribe to this topic. If one of those Primitive Interfaces is not marked "private", the base rejects to create the DCAEM (see also above, implementation of concept "Base"). Multiple Primitive Interfaces of the base-module can be assigned to the same DCAEM, but a base-module can be the "Base" of at most one DCAEM. A Primitive Interface is either assigned to a DCAEM or to a Composite AEM (CAEM), not to both. To initiate DCAEM-creation, the Role-Binder sends a Create-event to the base. It contains the following information: *name of DCAEM* and *names of Primitive Interfaces* that should get connected to the DCAEM.

Roles are added to a DCAEM at runtime (binding). On binding, certain Primitive Interfaces of the role are granted to publish/receive events to/from the topic that represents the DCAEM, i.e., connectivity to that topic is created/removed. A specific Primitive Interface can get assigned at most to a single DCAEM. A role can not be a member of a Composite AEM. Binding is implemented by the Role-Binder by sending a Bind-event. It contains the following information: *name of DCAEM* and *names of Primitive Interfaces* that should get access to the DCAEM. The role is responsible for checking whether the DCAEM that it should be bound to, is actually existing. If not, it rejects to process the event.

Role-Binder
From the "Role-Binder" concept the following characteristics can be identified, which are relevant for implementation:

- Statefulness: Role-Binders provide stateful role-binding.
- Accessibility: Role-Binders are meant to handle events of many different constituent systems (e.g. "notification of entry" to the SoS) and should therefore maintain a high level of accessibility.

Moreover, the Role-Binder implementation has to ensure:

- Exclusiveness, i.e., no role-modules or base-modules should be able to provide binding-services that are reserved for Role-Binders
- Qualification, i.e., Role-Binder modules should be qualified as such, and be distinguishable from base-modules and role-modules
- Dedication, i.e., Role-Binders can be dedicated responsibility for a specific coordination rule.

The mentioned characteristics of Role-Binders are implemented in the following way: **statefulness** is achieved by reuse of EventArchs 2.0 statemachine-concept. Role-Binders are a special type of statemachine. Role-binding is achieved in the following way: Role-Binders are not concerned with technically binding a role to a base, but by commanding this to a role-module. To achieve that, Role-Binders have been provided access to special binding-commands: "bind" and "unbind". Those commands cause the transmission of Bind- and Unbind-Events to a role-module. Role-Binders are moreover concerned with commanding the creation or destruction of a DCAEM to a base-module. This is accomplished by "create" and "destroy" commands and CreateDcaem-Event and DestroyDcaem-Event respectively.

Role-Binders are sent "notifications of entry" by potentially every system that joins the SoS. To achieve **accessibility**, all Role-Binders are associated to a common scope (see below "standard-scopes"). Instead of publishing events to individual Role-Binders, constituent systems publish their "notifications of entry" to the standard-scope that includes all Role-Binders.

Role-Binders should be exclusively allowed to issue binding- and creation-commands. **Exclusiveness** is achieved by semantic checks (see below). The compiler checks whether other modules than "Role-Binders" make use of binding- and creation-commands and rejects compilation if this is the case. **Qualification** is achieved in the same way like it is achieved for role-modules and base-modules (see above). Role-Binder modules are associated a specific *TypeOfModuleMarker* that allows for their qualification as "Role-Binder". **Dedication** is currently not enforced, but encouraged by allowing to define a Role-Binder within a compartment. This can be used by the SoS-manager to indicate the responsibility of this Role-Binder for applying the coordination rule that is represented by that compartment.

5.1.3 Inner Roles and Atomic Block

This subsection explains the implementation of the concepts "Inner Roles" and "Atomic Block" in EventArch 3.0. The process of activating and deactivating an inner-role is explained, as well as the implementation of the inner roles scope-restrictions. Two features of the language-element "Atomic Block" are identified: "error-sensitivity" and "recoverability". The implementation of both features is described.

Atomic Block

The implementation of the atomic block (see 4.3) has to ensure recoverability and error-sensitivity. Recoverability is the ability to restore the state that the SoS had adopted with respect to binding relations prior to entering the atomic block. This should be done in case of errors. Therefore, error-sensitivity is required as well. To achieve **error-sensitivity**, acknowledgements are employed. Every role-module and every base-module that has received a binding- or creation-event has to acknowledge the successful processing of this event. If the acknowledgement fails to appear, roll-back is triggered. To achieve **recoverability**, the strategy "sequential processing of

statements" is employed. The commands of the atomic block are executed one-by-one, i.e., the Role-Binder publishes a binding- or creation event and does not proceed until the awaited acknowledgement has arrived or a certain period of time has elapsed. By this approach the Role-Binder has at any point of execution definite knowledge about the successful command-execution. When executing a specific binding- or creation-command, the Role-Binder can assume that all preceding commands in the atomic block have been executed successfully. This allows a rollback-implementation in the following way: the Role-Binder internally stores information about all binding- and creation-commands that have been executed successfully up to the current point of execution. In case of a failed acknowledgement, this information can be employed. To achieve rollback, all commands that have been successfully executed are undone. To undo a Bind-Event an Unbind-Event is sent and vice versa. To undo a CreateDcaem-Event a DestroyDcaem-Event is sent and vice versa.

Inner Roles
The concept of "Inner Roles" requires dedication to a DCAEM. The inner roles should be **disabled** as long as no DCAEM has been created around their base-module. Inner Roles should not be able to publish events outside the **scope** of the DCAEM that they are associated to.

The concept of "Inner Roles" is closely related to EventArchs 2.0 concept of "Primitive Interface". An inner role can be defined by defining a Primitive Interface and marking it by the keyword "private". Due to the close relationship of the two concepts, the concept of "Inner Roles" could be implemented by extending the existing implementation of the "Primitive Interface"-concept. In particular, the *ERAppInterface*-class has been extended by the property *currentActivationState*. This property is checked whenever an event is published by the module or an event is received by the module. Received events can not be selected by an *ERAppInterface*-object that is currently deactivated. State-changes of a constituent system that would give rise to an event-transmission using that *ERAppInterface*-object, are ignored by that object if it is currently deactivated. *ERAppInterface*-objects that represent an inner role are deactivated by default. This renders inner roles to be **disabled** as long as no DCAEM has been created around their base-module.

As described above, roles are "modules that may have interfaces that are associated to a DCAEM and are therefore restricted to publish to the scope of that DCAEM". The description that was given above applies to the "Inner Roles"-concept as well. The DCAEM is represented by another property of the *ERAppInterface*-object. This property is employed to enforce **scope-restrictions**.

Diagrams
A detailed listing of the changes that had to be applied to the implementation of EventArch 2.0 in order to implement the described concepts is presented in the appendix (8.4). The presentation contains numerous class-diagrams and a sequence-diagram.

5.1.4 Other Concepts

This subsection explains certain other concepts that are relevant for the implementation of EventArch 3.0. Standard-scopes are a means to facilitate communication between modules and Role-Binders. The SystemMonitor may be used to visualize the ongoing activity among constituent systems. Semantic checks are employed to relieve the runtime-environment of EventArch 3.0 from having to check on all semantic rules that an EventArch 3.0-specification has to comply to.

Standard-scopes
To facilitate communication between certain participants of the role-based system, certain standard scopes had to be established:

- All modules are connected to a global scope
- Each Role-Binder is listening to a standard scope that is meant to make all Role-Binders globally available ("Role-Binders standard scope").

The global scope is used for the events that are directed from Role-Binders to base-modules (for a discussion see next section). All base-modules use the Role-Binders standard scope to notify all Role-Binders about relevant state-changes. The modules are connected to their respective standard scopes at initialization.

SystemMonitor
EventArch 3.0 provides a possibility to visualize the ongoing activities (event-reception, successful event-processing) of the SoS. A new built-in module has been defined: the "SystemMonitor". The SystemMonitor depicts running base-modules, role-modules, and compartments by named boxes. The boxes are also coloured. Colours are mapped to DCAEMs and indicate the current membership of base and roles to a DCAEM.

The described behaviour has been implemented in the following way: modules (role, base, Role-Binder) can decide to notify the SystemMonitor about successfully processed events. To notify the SystemMonitor, a special event named "Acknowledgement" is sent. The Acknowledgement-event holds a copy of the event whose reception/processing is to be acknowledged. The relevant information of the acknowledged event is extracted at the SystemMonitor and changes are performed to the graphical representation accordingly. The changes are encapsulated as commands. The "Command" design-pattern is used. Warnings are issued if received events indicate an inconsistent state in the SoS (e.g., a role is bound to a DCAEM that has not been created yet).

Semantic checks
The semantical correctness of an EventArch-specification can be checked at compile-time or at runtime. EventArch 3.0 relies primarily on compile-time checks. For a discussion see chapter 5.2.

The Xtext-framework (see 2.2.2) provides several methods to define compile-time checks. The following methods are employed in the EventArch 3.0 language- implementation:

- Language-specific validation
- Scope-provider

The language-specific validation (in contrast to automatic validation, which is a built-in feature of Xtext) is performed by a language-specific validator. The EventArch 3.0 validator checks the following semantic rules:

- Only Role-Binders are allowed to make use of bind-, unbind-, create-, destroy-statements.
- A role is only allowed to send events to a topic that either represents its compartment or the DCAEM that it is currently bound to.
- If a destroy-statement, that is meant to destroy a specific DCAEM, is present in the specification, a corresponding create-statement to create this DCAEM has to be present as well.

The following scopes are provided by the EventArch 3.0 scope-provider:

- Bind- and unbind-statements can only bind role-modules to a DCAEM. (i.e., can only "see" role-modules)
- Create- and destroy-statements can only create a DCAEM at a base-module (i.e., can only "see" base-modules).

5.2 Further Discussion and Design Alternatives

In this section certain alternatives with respect to the design of EventArchs 3.0 implementation are considered. This section is mostly concerned with low-level concepts like the internal representation of the architectural specification or the validation of semantic rules. Moreover, alternatives to the current standard-scopes and standard-interfaces are considered. Additionally, alternatives to EventArchs 3.0 current syntax are concerned and it is discussed in what way advantageous use can be made of a Role-Binder.

Syntax

Figure 5.1 also shows the new binding- and creation-expressions of EventArch 3.0 (see listing 5.2). Roles can be added to existing DCAEMs and DCAEMs can be created having multiple roles initially bound.

```
1 atomic
2  {
3     DcaemLoadBalancer[composite] := {LoadBalancerCoord.IBaseDirectedLB}
      <->{WrappedLoadBalancer.{ISwitchCoordinatorLB, IStateCoordinatorLB}}
4     DcaemAdaptiveSoftware[composite] := {} <-> {WrappedAdaptiveSoftware.{
      IStateCoordinatorAS, ISwitchCoordinatorAS}}
5     DcaemAdaptiveSoftware[composite] += {AdaptiveSoftwareCoord.
      IBaseDirectedAS}
```

```
6 }
```

Listing 5.2: Extract of definition of EventArch 3.0 Role-Binder

An extract of EventArchs 3.0 grammar is presented in the appendix (see 8.3). The syntax to create DCAEMs and bind/unbind roles to/from it shows an apparent similiarity to the EventArch 2.0 syntax to define Composite AEMs (see 2.1). An alternative syntax is used in listing 5.3.

```
1 atomic
2  {
3      createDcaem DcaemLoadBalancer at WrappedLoadBalancer.{
           ISwitchCoordinatorLB, IStateCoordinatorLB} with LoadBalancerCoord
           .IBaseDirectedLB
4      createDcaem DcaemAdaptiveSoftware at WrappedAdaptiveSoftware.{
           IStateCoordinatorAS, ISwitchCoordinatorAS}
5      bind AdaptiveSoftwareCoord.IBaseDirectedAS to DcaemAdaptiveSoftware
6 }
```

Listing 5.3: Alternative syntax for definition of EventArch 3.0 Role-Binder

The advantage of the first solution is, that similiar concepts are syntactically represented in a similiar way (all that the DCAEM-concept adds to the CAEM-concept are member-changes at runtime). It saves the programmer from having to get used to different notations of similiar concepts.

Standard-interfaces
An advantage of having standard-interfaces is that they encapsulate the conditions to select/discard events in the established way of the EventArch 2.0 language. Primitive interfaces contain a certain set of selectors (see 2.2.2) that represent those conditions. Making binding- and creation-selectors part of "just another interface" of the module integrates them well into the existing infrastructure of the language-implementation to access and employ selectors. All future language-features that rely on selector-checking will have seamless access to the modules binding- and creation-selectors. An alternative would have been to not encapsulate the binding- and creation-selectors in an interface, but to make them available to the module as another variable that is globally available. This solution would have taken away the described advantage from the language.

Representation of information that is provided in the specification
The information that have been provided in the specification are represented as an object of type "PrimitiveAEM". This is done for all module-types: role, base, Role-Binder. An alternative would have been to introduce specific object-types for each type of module. Those object-types could be subclasses of "PrimitiveAEM". This would have avoided to introduce the *TypeOfModuleMarkers*. On the other hand, it would have been an interference with the type "PrimitiveAEM" which is used all over the code of EventArch 2.0. The classes *ERGeneratedAppEventModule* and *ERAppEventModule* are responsible for publishing and processing events for all types of modules: role, base, Role-Binder. A clean solution to allow the mentioned classes

to take advantage of those subclasses of "PrimitiveAEM" would have been to subclass them as well. Moreover, the code that implements serializing/deserializing to achieve persistence would have had to be enabled to process the subclasses as well. This would have been an acceptable solution if the concepts base, role, and Role-Binder would require the SoS-manager to provide different information in the specification. But in EventArch 3.0 those concepts share a common information-base. The main difference are the type of events that the respective module-types are allowed to process or to publish. But this difference is respected during operation due to the standard interfaces. Hence, there is no necessity to interfere with the well-established object-type "PrimitiveAEM".

Role-Binders
Role-Binders are not intended to be used as wrapper for another Java or C++-application. They should be used as an EventArch-statemachine. The events that are sent to the Role-Binder inform it about the present state of execution of the SoS. They can be used to trigger state-changes. A state of the Role-Binder would then correspond to a state of the SoS *as observed* by the Role-Binder. Binding and creation-statements can be put in every defined state, but to take advantage of the Role-Binders advantage "consistent application of a coordination rule to all constituent systems that are concerned by it", (see 6.1) all commands that are relevant for activating a specific coordination rule should be executed as entry-actions of a specific state. Example code is presented at the beginning of this chapter. (see 5.1)

Scopes
The communication between Role-Binder and base-modules is done in a global scope. An alternative would have been to establish an individual destination point for each base-module. The advantage of having a global scope is a lower occupation of resources at the jms-provider. Disadvantages are a decreased scalability through a much higher count of events sent through the system and a greater vulnerability to sniffing-attacks, as all modules do basically receive events that were addressed to specific unrelated base-modules.

Validation
The validator was used to enforce several semantic rules for the language (see section 5.1.4). It is a general advantage to rely on compile-time checks instead of runtime-checks, as present errors are communicated to the programmer earlier and definitely. An alternative would have been to implement these semantic rules by additional functionality of the language core ("runtime-check"). In this case, the language-core would have had to distinguish between bases, roles, and Role-Binders, which would have made it severely more difficult and practically impossible to implement the distinction between base-modules, role-modules, and Role-Binder-modules by the simple *TypeOfModuleMarker*s. This would have taken away all the described advantages that are connected with that (see above "Representation of information provided in the specification"). Another disadvantage is, that the error would not have been detected if the erroneous statement was not executed at runtime.

Chapter 6

Evaluation of EventArch 3.0

This thesis is concerned with the problem area of "coordination in dynamic SoS". An architecture description language is devised that allows for a dynamic application of coordination rules. In this chapter the obtained solution, EventArch 3.0, is evaluated argumentatively by analyzing its advantages and disadvantages and reflecting on its capability to fulfill the requirements that have been stated in the "Related Work"-chapter. Moreover, a coordination problem that originated in the field of energy-efficient computing is solved using EventArch 3.0. The application to this use case shall demonstrate the applicability of the obtained solution to practical coordination problems.

6.1 Advantages

This section provides a further discussion about the features that have been identified in section 3.2. It is concerned with the implementation of those features in EventArch 3.0. The advantages that have been gained by that implementation are pointed out.

Event-based System
Event-based interfaces are superior to function-based interfaces with respect to **communicating the entry and exit of a system** to the scope of a SoS and with respect to **establishing a dynamic, common scope of base and roles**. The claimed advantages are shown in the following.

In an inter-system communication scenario both, event-based interfaces and function-based interfaces give rise to the transmission of messages between specific systems. Those systems may define provided- and required interfaces. Communication between two specific systems may be restricted by interface-compatibility. Despite that commonality there is a fundamental difference between event-based interfaces and function-based interfaces. Functions are typically used to obtain a required piece of information. Moreover, functions are typically used to introduce a dependency of the caller on the success of information-processing at the callee's side. If information processing can not be accomplished successfully, the caller may resort to exception-handling.

In contrast to that, messages that were sent through an event-based interface are typically used to indicate a specific state that may be of interest to the receiving party. The sender is not in need of a desired piece of information. There are typically no return-messages expected in response to an event. Moreover, acknowledgements concerning successful event-processing are not typically expected.

These characteristics render event-based interfaces to be **advantagous with respect to communicating the entry and exit of a system** to the members of a SoS. The reason for that is that the entry- and exit-information are such informations that a sender does typically not expect a response on. Nor it typically expects an acknowledgement.

Functions are typically used to obtain a specific piece of information. This also implies, that it would be impractical to allow a message that originated from a function-call to be delegated to multiple receivers. The caller would have to take the decision which of the multiple received pieces of information to make use of. Moreover, determining the identity of the sender of a response to a function call is external to the function-concept and would require additional language support. Opposed to that, events can be dispatched to multiple receivers, as typically no response is expected. Therefore, relying on event-based interfaces is a prerequisite to connect a constituent system to all of its coordinators (i.e., "multiple" receivers) by means of a common message scope. Messages can be published to this scope instead of to be addressed to individual receivers. From point of view of role-based modeling, event-based interfaces allow for connecting a base (constituent system) to all of its roles (coordinators) by means of a common message scope. Therefore, event-based interfaces are **superior to function-based interfaces with respect to establishing a dynamic, common scope of base and roles**.

Dynamic Composite AEM

The feature "Dynamic Composite AEM" (DCAEM) puts a SoS-participating system in the position to cope with the need for dynamics in system-to-system relations which is prevailing in a dynamic SoS, especially with respect to compliance with relevant coordination rules. This is shown in the following.

A DCAEM is conceptually a mapping between a symbolic name and a set of addresses of specific members in the scope of a SoS, which can be changed at runtime ("dynamic mapping"). These members can be coordinators that are associated to a specific coordination rule. Compliance to a coordination rule can be achieved by establishing a communication relation between the system-to-be coordinated and a suitable coordinator. The roles of a compartment act as coordinators for the coordination rule that is represented by the compartment. Changing the mapping at runtime to include a suitable role ("binding a role") is therefore a proper means to dynamically achieve compliance to a coordination rule for a SoS-participating system.

Supporting the concept of Dynamic Composite AEM does also allow for the integration of coordinators that have been unanticipated at design-time. Messages are not published to individual members of the SoS, but to the DCAEM, which is a common

message scope of base and roles (i.e., coordinators). Integrating a coordinator that has been unanticipated at design-time can be performed solely by including this coordinator into the DCAEM at runtime. The code of the constituent system does not have to be changed. There is no need to address the new coordinator directly. The constituent system may keep publishing to the common message scope of base and roles. This can be used to exchange the implementation of a coordinator at runtime as well. While other solutions allow for the same dynamics in base-role relations (e.g., SCROLL achieves this feature by means of SCALAS feature "dynamic traits", see 3.4), the integration of base and roles by means of a common message scope is a solution that can be implemented relatively easy. No special language features are required. Available message-oriented-middleware (MOM)-solutions can be employed. EventArch 3.0 employs the messaging-platform "JMS" to implement that feature.

Three further advantages of this feature can be identified.

- A DCAEM preserves the privacy of the SoS-participating system by excluding all members of the SoS-scope from the coordination, but those that are relevant for coordination.
- DCAEMs can be used to allow a base-application to provide an identical, steady stream of events to all currently bound roles. This synchronizes the roles with respect to their knowledge of the base-applications by providing a consistent pool of information.
- The feature can be used to exchange the implementation of an existing coordination rule by remapping a symbolic name to roles of a different compartment.

The described advantages of the feature "Dynamic Composite AEM" open up the static "Composite AEMs" to contribute to EventArchs 3.0 support for coordination scenarios in dynamic SoS, where the application of a coordination rule depends on the presence or absence of certain participating systems.

Cross-cutting Roles

The possibility to dedicate roles to a cross-cutting concern increases the modularity and adaptability of a SoS-participating system with respect to the concern of coordination. Moreover, the process of role-binding has become less complex due to Cross-cutting Roles. Both claims are shown in the following.

Cross-cutting concerns are characterized by a widespread dissemination across different parts of the application in question. Aspect technology is a means to modularize a cross-cutting concern and to separate it from other concerns of the application. It even may provide an implementation of a cross-cutting concern that is transparent to the other concerns of the application. A code-unit that implements a cross-cutting concern may be dynamically bound and unbound from the base-code that it is attached to. Such an implementation of a cross-cutting concern that can be bound and unbound at runtime is termed a "Cross-cutting Role" in this thesis.

The "Inner Roles" of EventArch 3.0 are understood as Cross-cutting Roles, as they encapsulate services that are concerned with a cross-cutting concern ("coordination"),

and can be activated and deactivated at runtime. A specific inner role can be associated to a specific coordination rule. It implements connection-services with respect to that coordination rule that may cross-cut through all components of the constituent system. An alternative would have been to associate an inner role with a specific component of the constituent system. This would be a less valuable solution as it implies the scattering of the concern of coordination across several components of the SoS-participating system. The inner-roles of EventArch 3.0 encapsulate the concern of coordination from point of view of the constituent system that they are bound to. They can be transparently employed by each of its components. In particular, the coordination-related code is neither scattered around the base nor tangled with the original concerns of the original codebase. This **increases the modularity** of the concern of coordination. Moreover, this provides an increased reusability of the constituent system in other SoS with different coordination rules.

The transparency that is gained by the cross-cutting implementation of roles has another advantage with respect to the adaptability of the legacy code of the constituent system. If a new inner role is to be incorporated into the legacy-code of the constituent system, this legacy-code does not have to be changed. The incorporation is not achieved by manually changing the legacy-code, but by weaving additional functionality into this code at compile-time. Moreover, the reflection-functionality of the respective programming language is used to perform function calls. This also furthers the ability of EventArch to incorporate inner roles at runtime that have not been anticipated at design-time. The legacy-code would not have to be changed at runtime. This **increases the adaptability** of the SoS-participating system with respect to the concern of coordination. But still, the cross-cutting implementation of the inner roles would have to be woven into the constituent system at runtime. This is currently not possible, but may be achieved by future work.

The feature of "Cross-cutting Roles" also renders the process of role-binding to be less complex. The Role-Binder may be concerned with binding one inner role per coordination rule to the constituent system. It would be even possible to encapsulate all functionality that is relevant to connect the constituent system to all relevant coordinators within a single inner role. Without the possibility to implement the cross-cutting concern of coordination in a modular way, the Role-Binder would be concerned with binding a specific inner role to each individual component of that constituent system that is concerned with the respective coordination rule. This would render the process of role-binding more complex. Therefore, the process of role-binding has become **less complex** due to the feature of "Cross-cutting Roles".

Programmable Role-Binding: Role-Binder

Supporting programmable role-binding by means of a Role-Binder increases the modularity, evolvability, and comprehensibility of a SoS with respect to the concern of coordination. Moreover, it increases the feasibility of the consistent application of a specific coordination rule to a set of constituent systems. The claimed advantages are shown in the following.

Having support for dynamic application of coordination rules requires a SoS-oriented composition system to take the decision at which point of execution which coordination rule should be applied to which SoS-participating system. This decision may be put in the field of responsibility of the individual systems. Alternatively, a Role-Binder can be employed that is responsible for taking this decision for several systems. This Role-Binder may be responsible for applying a specific coordination rule to certain applications. If it is responsible for multiple coordination rules, it may select one out of many coordination rules for application to the participating systems. In both cases, not the individual system is concerned with the binding decision, but the Role-Binder. Depending on this decision, compliance with a specific coordination rule is achieved at a specific point of execution. The decision is encapsulated by the Role-Binder. It is a specific member of the SoS-scope that provides a well-known interface. This **increases the modularity** of the SoS with respect to the concern of coordination.

This member can be replaced without interfering with the share of the architectural specification that concerns the participating system in question. This **increases the evolvability** of the SoS with respect to the concern of coordination.

Coordination rules are applied to a SoS to achieve a particular coordinated behaviour. A coordination rule in general concerns multiple constituent systems. The application of a coordination rule may be just appropriate if all constituent systems that the rule is concerned with, have joined the SoS (e.g., coordination to achieve mutual exclusion). In those cases, a consistend application of the coordination rule to all concerned constituent systems is required. The reason for that is the following:

Acting in compliance with a specific coordination rule induces into a specific constituent system dependencies on the behaviour of certain other systems that are participating in the SoS. In a coordination scenario, these dependencies may be a determining factor for the behaviour of the constituent system. In those cases, applying the coordination rule constitutes a tight-coupling between the constituent systems that are concerned with that coordination rule. Tight coupling implies that a participant may depend on return values that another participant would send in response to a request. The participant may be even unable to proceed, unless this return value or an awaited acknowledgement has been received. A coordinative behaviour under those conditions requires all concerned systems to have applied that coordination rule at the same time and therefore being ready to act according to it in a tightly-coupled manner. If one concerned system is not ready to act according to the coordination rule, malfunctions may occur due to unnecessary blocking of the other applications. To prevent such malfunctions, compliance to the coordination rule should be established consistently.

As a Role-Binder maintains communication relations to numerous SoS-participating systems and decides upon applying specific coordination rules on them or not, it may decide to apply a specific coordination rule consistently to all relevant SoS-participating systems. If the individual systems would be responsible for taking

binding decisions, they would have to maintain an elaborate mechanism to achieve synchronization with respect to conditions for a consistent application of a specific coordination rule. The Role-Binder relieves the systems from having to establish that synchronization-mechanism. It achieves the *simultaneous* composition of *all* coordinators of the *same* coordination rule with those constituent systems that are concerned by that rule. This **increases the feasibility of the consistent application** of a specific coordination rule to a set of constituent systems.

A SoS-oriented composition system that supports the dynamic application of a coordination rule has to provide means to identify the point of execution of the SoS at which the application of the coordination rule should be performed. The conditions that identify this point of execution can be nicely modeled as a state-machine. This is true for the following reason: role-binding is a cross-cutting concern that involves several systems. Each system maintains local state information. The aggregated knowledge about the local state of several systems can be represented as an overall system-state. The role-binding conditions can be described in terms of this overall system state. A state-machine is therefore well-suited to model role-binding condition and may provide nice specifications that are easy to understand. A Role-Binder is dedicated to the purpose of taking role-binding decisions and may therefore be defined as state-machine. EventArch 3.0 features a Role-Binder that is implemented as a state-machine. This **increases the comprehensibility** of the SoS with respect to the concern of coordination.

Support Notion of Compartment
Supporting the notion of Compartment increases the evolvability, modularity, and comprehensibility of a SoS with respect to the the concern of coordination. This is shown in the following.

The notion of Compartment originated in the field of role-based modeling. It can be understood to encapsulate a context of related roles. With respect to the problem that this thesis is concerned with, the "relation" between roles can be interpreted as a relation with respect to the aspect of coordinated collaboration. It is therefore promising to use the notion of Compartment as an encapsulation of coordinators that are related by a common coordination rule. All roles inside a compartment can be understood to be coordinators that try to achieve for certain related constituent systems compliance with respect to a certain coordination rule. A compartment may be used to encapsulate all coordinators of a specific coordination rule. This associates each coordinator with a certain coordination goal and relates it to a context of other coordinators. Therefore, it is easier for the SoS-maintainer to understand the purpose of each coordinator, as well as its behaviour. This **increases the comprehensibility** of the SoS with respect to the concern of coordination.

The definition of a compartment contains the definition of the roles that act as coordinators and may contain the code of relevant Role-Binders. All code that is relevant for a specific coordination rule may therefore be packaged as compartment. A compartment sets up a common scope for all roles that act as coordinators with respect

to the associated coordination rule. This separates the coordination-related communication from communication that is done on behalf of other concerns. Both points **increase the modularity** of the SoS with respect to the concern of coordination.

The coordinators that are responsible for a specific coordination rule are designed as roles of a compartment. This prevents the SoS-maintainer from tangling their behaviour with behaviour that corresponds to other concerns (e.g., coordination according to a second coordination rule). Having all code that is relevant for a specific coordination rule packaged as compartment, allows for replacing this code by a different implementation. Both points **increase the evolvability** of a SoS with respect to the concern of coordination.

Two-layered Role-Binding
Partitioning role-responsibilities into an inner- and outer layer increases the separation of the concern of coordination from the original concerns of the SoS-participating systems and increases the adaptivity of the constituent system with respect to the concern of coordination. Both claimed advantages are shown in the following. Moreover, it is shown, in what way EventArchs 3.0 "Inner Roles" contribute to the solution of the problem of this thesis.

The coordinated execution of several SoS-participating system requires interactions between the individual systems and their coordinators. The interactions between a constituent system and its coordinators can be understood as interactions between a base-application and its role-applications. To achieve the desired transparency of the concern of coordination from the original concerns of the SoS-participating system, the concern of coordination was implemented in EventArch transparently to the original concerns of the constituent system. This was achieved using aspect-technology. This causes the original concerns to be unaware of the concern of coordination. Therefore, it is more likely that changes to the code that implements the original concerns of the system will be performed in ignorance of the coordination-related code and break it.

To ease this problem, the code that the base-application is unaware of, should be not concerned with any coordination logic. Instead, it should be solely concerned with translating the observed base-applications activity into a representation that can be processed by downstream entities that are concerned with the desired coordination logic. Realizing this functional model will reduce and encapsulate the portion of the exogenous code which could be broken by changes that are committed to the code that implements the original concerns of the constituent system.

This model is realized in EventArch 3.0 by the partitioning of the role-responsibilities into a layer that is only concerned with observing and representing the base-applications state (inner roles) and another layer that is only concerned with the coordination logic, i.e., achieving compliance to a specific coordination rule (outer roles). This model reduces the maintenance effort in case of conflicting base-code changes to the inner roles and renders the outer roles to be independent from code-changes to the base-application. This **increases the separation of the concern of coordination**

from the original concerns of the SoS-participating systems.

In EventArch 3.0, the inner roles are concerned with observing the state of the base-application and providing a representation of it to the outer roles. Moreover, they provide a command-interface to the outer roles that allows them to get certain functions executed. The SoS-maintainer may define several different implementations in the specification of the Architectural Event Module. When creating a DCAEM, the Role-Binder may choose one out of several available inner-role implementations. This allows to adapt the published state-representation and the provided command-interface at creation-time of the DCAEM. This **increases the adaptivity of the constituent system with respect to the concern of coordination.**

The inner roles connect the constituent systems to coordinators that are associated to certain coordination rules. Every coordinator employs coordination logic to issue certain event-based commands to the constituent systems in order to achieve compliance with the behavioural restrictions that are set up by that coordination rule. The inner roles do also have their share in achieving that rule-compliant behaviour of the constituent systems. They execute the commands and can cause the constituent system to suspend execution and wait for further commands of a coordinator. This is implemented by EventArchs 2.0 "wait-when blocks" (see 8.2.1). In EventArch 2.0, it was not possible to deactivate the Primitive Interfaces. Therefore, the execution of a constituent system has been suspended by a wait-when block regardless of whether the necessity to comply to a coordination rule actually existed for that constituent system or not. This was possible for EventArch 2.0 as this ADL assumes a static SoS. In a static SoS all constituent systems that have been anticipated at design-time are assumed to have joined the SoS at runtime. Under that condition it was feasible to not allow a Primitive Interface to get unbound from the constituent system at runtime. This is not sufficient for EventArch 3.0 as this ADL assumes a dynamic SoS. The need to comply to a coordination rule may vanish at runtime, depending on the composition of constituent systems in the SoS. Therefore, the necessity for a constituent system to have its execution suspended at certain points of operation may vanish as well. The "Inner Roles" of EventArch 3.0 are Primitive Interfaces that can be deactivated at runtime. The deactivation of a Primitive Interface deactivates all its "wait-when blocks" as well. Therefore, this feature allows EventArch 3.0 to relieve a constituent system from having to stop its operation at runtime if no need for coordination exists. This feature is central for EventArchs 3.0 ability to **solve the problem** that is relevant for this thesis. It enables this ADL to allow constituent systems to dynamically comply to a coordination rule or ignore it.

Base-Role Integration on Architectural Level

EventArch 3.0 is an architecture description language (ADL). An Architectural Event Module (AEM) encapsulates a running application or a running system. EventArchs 3.0 outer roles are AEMs that are bound to other AEMs - their bases - at runtime. Both, bases and roles are independent processes that have been provided an AEM-representation on the architectural level. Therefore, EventArchs 3.0 role-base

integration is situated on the architectural level. Base and roles can be understood as components that communicate across relatively fixed interfaces. The implementation of base and roles is decoupled from each other by the AEM (a wrapper around the legacy code) with its Primitive Interfaces. This basically allows EventArchs 3.0 roles to be implemented in a different programming language than the base. Nevertheless, the advantage of "language independent base-role integration" is currently not exploited. "Dcaem" and "Inner Roles" can currently just be employed in conjunction with constituent systems that are implemented in the programming language "Java". Extending EventArch 3.0 to support other programming languages (e.g., C++) would be a subject for future work.

6.2 Disadvantages

In this section disadvantages that have been introduced through the respective features to the language are pointed out.

Event-based System
With event-based interfaces a base can be dynamically connected to its roles by a common message scope. But the roles of EventArch 3.0 act as coordinators of their bases with respect to a certain coordination rule.

And the constituent systems do not merely provide state information to their coordinators, but also require a specific piece of information. It is the information, whether they can safely continue with their execution or whether they should suspend. For that use case event-based interfaces have a poor applicability. Therefore, a mechanism had to be introduced that allowed for selectively introducing a tight-coupling between sender and receiver for specific events (wait-when block). The solution is somewhat clumsy and less intuitive to the programmer than a function-based solution would have been.

Dynamic Composite AEM
In a Composite AEM, events of the base-application are published to multiple Composite Interfaces (see 2.2.1). The scope that is established by the Composite AEM will not change at runtime. To prevent unintended behaviour, the SoS-maintainer has to choose for the event such a type and such attributes that it is not selected and processed by a Composite Interface that was actually not intended to select it.

The SoS-maintainer can relatively easy check for that problem in case of Composite AEMs, as all members of its scope are known at compile-time. With Dynamic Composite AEMs, the SoS-maintainer has to run through all Role-Binders and check whether they might bind a coordinator to that DCAEM or not. Moreover, the binding decision may depend on the state of the SoS at runtime. Depending on that state, it may be safe or may not be safe to use a specific event type and attributes. The problem of unintended event-processing is therefore harder to solve for the SoS-maintainer using a Dynamic Composite AEM.

Cross-cutting Roles

In EventArch 3.0, the inner roles are understood as Cross-cutting Roles. Cross-cutting Roles are not associated to an individual component but to a group of related components. This expansion of responsibilities makes it more difficult to depend event-processing on the component that the event originated from.

It follows an example for the cross-cutting concern of logging: a system designer might decide to associate a priority level to a logging message. In this case, the cross-cutting logging implementation would have to assign a priority level to a logging message. An easy criteria for setting the priority level would be the name of the component that the message originated from. This introduces a dependency between the cross-cutting logging information and individual components.

It is more difficult to implement such behaviour with Cross-cutting Roles as they maintain responsibility for multiple components. The relevant component has to be identified out of several components that come into question. The ease of implementation depends on the support for such problems in the employed aspect-technology.

In EventArch 3.0, the inner roles would be responsible for determining the relevant component and making it available to the outer-role as an additional event-attribute. The inner roles are implemented in AspectJ. In this language one would have to introduce additional pointcuts or introduce a single complex pointcut that depends on the type of the calling object. Both solutions increase the maintenance effort of the aspect-code, as it is more likely to be affected by changes to the base-code.

Moreover, in EventArch 3.0 the inner roles connect a constituent system to its coordinators in a transparent way. The legacy-code of the constituent system does not have to be changed in order to provide a representation of the coordination-relevant state of the constituent system to its coordinators. The inner roles can process commands that have been sent by the coordinators in a transparent way, as well. This renders the concern of coordination to be transparent to the original concerns of the SoS-participating system. While this increases the evolvability of the concern of coordination, it also increases the maintenance effort of the coordinated system. This is shown in the following.

Transparency allows the maintenance of the core-system to be separated from the maintenance of the coordination-code. If this is the case, all changes to the base-code are performed in ignorance of the coordination-code which makes them more likely to break that code. If they are not separated, the maintenance effort of the core-system is increased. The reason is, that due to transparency the system maintainer is basically unaware of the potential consequences of his code-changes for the coordination code. To be on the safe side, he would have to assume for every code-change that it is incompatible with the coordination code and check for possible consequences. Both cases **increase the maintenance effort** of the coordinated system.

Even if all changes to the base-code are compatible with the coordination-code, there remains a basic uncertainty about the behaviour of the base-application at runtime.

Due to the transparency of the concern of coordination, one can not safely understand the runtime-behaviour of the system from the base-code alone. For each function, one would have to expect interference of the coordination code and have to check for that. This **decreases the comprehensibility of the coordinated system**.

Support Notion of Compartment
EventArch 3.0 supports the notion of Compartments, but it is a relatively poor support with respect to the classifying features identified in [20]. Compartments can neither be nested nor play roles. Compartments do not provide additional properties and behaviour that might be used by the contained roles. Roles are defined inside a specific Compartment and can not be part of several Compartments. Compartments can not inherit structure and behaviour from each other.

Programmable Role-Binding: Role-Binder
Having a Role-Binder may complicate coordinative behaviour for the constituent systems. This is shown in the following.

The Role-Binder is a member of the SoS-scope, but no participant of the SoS. This renders all state-information that was sent to it to be lost for the purpose of coordination. That means, that the SoS-participating systems have to deal with the problem that the specific state-information that caused the Role-Binder to decide in favour of a specific coordination rule is lost for the purpose of coordination. But this state-information is typically relevant for this purpose.

The reason is, that this state-information allowed the Role-Binder to take a decision on a specific coordination rule. That means, that it allowed the Role-Binder to tell that a certain coordinative behaviour will be necessary in the near future for a specific SoS-participating system. The fact that he typically used for his decision is the fact, that this information is relevant for particular that coordinative behaviour. But in those cases it would be relevant for the entity that implements the rule for the system as well. This is the coordinator that the Role-Binder is meant to bind to the system. So, the constituent system may have to await binding and republish this information after successful binding to the coordinator. This complicates coordinative behaviour for the SoS-participating System.

The Role-Binder is no single point of failure, as it may be defined for a particular coordination rule. Nevertheless, it can not be called a decentralized solution, as it is still responsible for a group of constituent systems. While this group is less likely to grow that much to achieve a size that would overburden the single Role-Binder, the probability for that is not zero. So, if there is a high volatility with respect to accepting compliance (binding) and giving up compliance (unbinding) with the coordination rule of that single Role-Binder, the Role-Binder is more likely to fail due to overloading, than a truly decentralized solution.

Two-layered Role-Binding
In EventArch 3.0, the outer roles are decoupled from the code of the base-application by the inner roles. Therefore, changes to the base-code are transparent to the outer

roles. The outer roles process the state-representation of the constituent system that is provided by the inner roles. This prevents the SoS-maintainer from having to change the code of the outer roles in response to changes to the base-code, but the code of the inner roles still has to be changed. The necessary effort is lowered due to the fact that the inner roles encapsulate the translation of base-behaviour into state-representation and are not concerned with taking coordination-decisions. Nevertheless, a **seamless integration of base-code changes into the coordinated system is not possible**. The reason is, that EventArch 3.0 employs aspect technology that is based on compile-time weaving and not on runtime-weaving. The consequence is that the coordinated system has to be recompiled after every change to the base-code that required an inner role to be changed.

6.3 Reflections on the Fulfillment of the Requirements

In this section it is analyzed in what way EventArch 3.0 fulfills the requirements that have been stated in section 3.1

To cope with coordination scenarios that involve a dynamic SoS, ADLs should support to condition the application of coordination rules with respect to the presence and absence of constituent systems in the SoS. EventArch 3.0 fulfills this requirement due to the features "Event-based System", "Dynamic Composite AEM" and "Programmable Role-Binding".

Event-based System: Constituent systems may indicate their entry and exit to the other constituent systems of the SoS. This is done by messages. Those messages are meant as a general information and may be published to all participants of the SoS. They do not necessarily require a response or acknowledgement from a specific constituent system. Nor do they require specific exception-processing in the case of failed processing. For such use cases, event-based interfaces are superior to function-based interfaces (see 6.1). Therefore, the information concerning the entry or exit of a constituent system is preferably disseminated based on events. Due to EventArchs feature "Event-based System" all communication among constituent systems is based on events. In this way, the feature "Event-based System" contributes to fulfilling this requirement.

Dynamic Composite AEM: This requirement demands from ADLs to provide means to condition the application of coordination rules. A prerequisite for that is the ability to apply a coordination rule at runtime. The dynamic application of a coordination rule may be implemented by dynamically establishing a communication relation between a constituent system and a coordinator. EventArchs feature "Dynamic Composite AEM" allows such a dynamic establishment of a communication relation. This is conceptually achieved by changing the mapping of a symbolic name to a set of addresses at runtime (see 6.1). The feature "Dynamic Composite AEM" is the core contribution of EventArch 3.0 to fulfill this requirement.

Programmable Role-Binding: A conditioned application of a coordination rule is a dynamic application of a coordination rule. It includes a transition of the activation-state of the coordination rule from "unapplied" to "applied". To prevent malfunctions from the concerned constituent systems due to this transition at runtime, the application has to be performed consistently to all constituent systems that are subjected to that coordination rule (see 6.1). The Role-Binder improves the ability of EventArch to achieve a consistent application of a coordination rule by relieving the constituent systems from having to synchronize in order to achieve a consistent application of that coordination rule. Instead, the Role-Binder allows for an effortless application of a coordination rule to several constituent systems in a consistent way. By this means, the Role-Binder contributes to fulfilling this requirement.

In order to dynamically integrate constituent systems with coordinators of the coordination rules, connectors should be employed that can be activated/deactivated at runtime and that provide a modular implementation of that cross-cutting concern, i.e., the concern of connecting all components of the constituent system to the respective coordinators.

Cross-cutting Roles: To achieve this requirement, entities are required that can be activated- and deactivated at runtime and that provide a modular implementation of a cross-cutting concern. EventArchs 3.0 "Cross-cutting Roles" can be regarded as an implementation of such entities. They are incorporated into the constituent systems using aspect-technology and are designed to be a modular implementation of the cross-cutting concern of coordination. They can be used as a connector to connect the constituent system to the coordinators that are responsible for coordinating it with the other systems according to a specific coordination rule. Moreover, they can be activated and deactivated by the Role-Binder.

In a dynamic SoS-environment the dynamic application of coordination rules should be not performed by the individual constituent systems, but by a specialized SoS-member.

Dynamic Composite AEM: Like the first requirement, this requirement expects an ADL to provide means to perform an application of a coordination rule at runtime. The feature "Dynamic Composite AEM" does therefore contribute to the fulfillment of this requirement in the same way as it does for the first one.

Programmable Role-Binding: This requirement expects from an ADL to concentrate the responsibility for applying a coordination rule to related constituent systems at a specific member of the SoS. This member would be specialized on that task. The "Role-Binder", which was introduced in EventArch 3.0 to provide the feature "Programmable Role-Binding", can be understood as such a specialzed member. The Role-Binder allows for the consistent application of a coordination rule to all constituent systems that are concerned by it. It can be made responsible for a specific coordination rule or for multiple coordination rules.

A constituent system should be open to new coordination rules and allow

for integrating them into its behaviour at runtime.

Two-layered Role-Binding: In EventArch 3.0, the implementation of a new coordination rule requires the implementation and deployment of new outer roles. The outer roles are independent processes that act as coordinators for the constituent systems. In EventArch 3.0, the outer roles are not concerned with extracting relevant state-information. Alle state-information that is relevant for the concern of coordination is extracted by the inner roles. This allows new outer roles to reuse the existing inner roles in order to get access to relevant state-information. This simplifies the implementation of the outer roles and contributes to the desire to render ADLs to be open for new coordination rules, which is expressed in this requirement.

As the outer roles are independent processes, they can be deployed independently of the constituent systems. A new outer role may start servicing the constituent system at runtime. This relieves the SoS-manager from having to recompile several constituent systems to integrate a new coordination rule. It is a prerequisite to allow for the integration of new coordination rules at runtime. In this way, the feature "Two-layered Role-Binding" contributes to fulfilling this requirement.

Inner roles provide a representation of the state of the constituent system that is relevant for the concern of coordination. This state-representation is provided to the outer roles. The inner roles can be used as a modular implementation of the task of providing that state-representation. This allows for exchanging inner roles more easily. If a particular state-representation is not sufficient for a new outer role, a new inner role may have to be defined and incorporated into the constituent system. This would still require a recompilation of the constituent system. Nevertheless, the adaption can be done more easily due to the high level of modularity of the inner roles. In this way, the feature "Two-layered Role-Binding" eases the integration of a new coordination rule as well and thereby contributes to fulfilling the requirement to be open for new coordination rules.

Dynamic Composite AEM: A coordination rule can be integrated into the behaviour of a constituent system by activating a coordinator that is responsible for implementing that coordination rule. The activation of this coordinator can be done by establishing a communication relation between the constituent system and this coordinator. To integrate a new coordination rule at runtime, the communication relation has to be established dynamically. The feature "Dynamic Composite AEM" allows for establishing communication relations between a constituent system and its coordinators at runtime. Coordinators can be bound and unbound from that constituent system at runtime. In this way, DCAEMs allow for integrating coordination rules into the behaviour of constituent systems at runtime.

A DCAEM is implemented as a common message scope of base and roles. To bind a role to a base, it is included in that common message scope. This is a relatively simple solution. It can be used to integrate coordination rules into the behaviour of constituent systems at runtime that have not been anticipated at design-time. This can be achieved by simply including the relevant coordinator of the coordination

rule into that message scope at runtime. By providing the possibility to achieve that dynamics in a relatively simple way, DCAEMs contribute to fulfilling the requirement of being "open" for new coordination rules.

Base-Role Integration on Architectural Level: In EventArch 3.0, the outer roles do conceptually provide coordination services to a specific constituent system. Depending on the nature of the desired services, an implementation in the one or the other programming language may be preferable. SoS are characterized by emergence and longevity (see [9]). This reinforces the need for language-independent base-role integration, as new coordination services may be diverse in nature and require implementation in different programming languages. Moreover, new software technology may emerge over the lifetime of a SoS. This may put pressure towards reimplementing some coordination services in a new programming language on the SoS-manager.

The feature "Base-Role Integration on Architectural Level" is a prerequisite to allow the SoS-manager to implement an outer role independently from the constituent system. Therefore, the SoS-manager is basically not restricted to implement the outer role in the same programming language like the constituent system. This renders EventArch 3.0 to be more open towards new coordination rules. In this way, the feature "Base-Role Integration on Architectural Level" contributes to fulfilling the requirement to be open towards new coordination rules.

Cross-cutting Roles: Due to this feature, the concern of coordination is not tangled with the original concerns of the constituent system and does not scatter around the base-code. Inner roles are implemented transparently to the base-code, e.g., by means of aspect technology. The increased modularity allows to adjust or even exchange the implementation of an inner role more easily. This renders EventArch to be more open towards new coordination rules.

Other Features: Certain other features contribute to fulfilling this requirement as well. The comprehensibility of the concern of coordination is increased through the features "Support Notion of Compartment", "Programmable Role-Binding", and "Cross-cutting Roles" (for details see section 6.1). The implementation of a new coordination rule is simplified by "Programmable Role-Binding", especially with respect to the necessary synchronization effort. The mentioned features allow the SoS-manager to understand and maintain the SoS more easily. Moreover, new coordination rules can be implemented more clearly. Both advantages render EventArch to be more open for new coordination rules.

In a dynamic SoS-environment all communication that is done on behalf of a specific coordination rule should be performed in a separated scope.

EventArch 3.0 is able to meet this requirement due to the features **Support Notion of Compartment** and **Dynamic Composite AEM**. This is shown in the following.

In a SoS, two types of members are relevant with respect to coordination: constituent systems and coordinators. According to this requirement, all communication that is related to coordination should be encapsulated in a separated scope. A coordination-

rule can be implemented according to the "Centralized"-coordination-pattern or the "Peer-to-Peer"-coordination pattern (see [18]). In case of a "Centralized" implementation, communication between constituent systems and the central coordinator is relevant. In case of an implementation according to the "Peer-to-Peer"-coordination pattern, in addition, the communication between coordinators that implement a common coordination rule is relevant.

EventArch 3.0 encapsulates both sorts of communication in a separated scope. Due to the feature "Dynamic Composite AEM", DCAEMs can be created in EventArch 3.0. DCAEMs provide a common scope for a constituent system and all its coordinators. This scope encapsulates all communication of a constituent system that is done on behalf of the concern of coordination. Due to the feature "Support Notion of Compartment", Compartments can be created in EventArch 3.0. Compartments can be used to provide a common scope for all coordinators that implement a specific coordination rule in a peer-to-peer coordination-scenario. Therefore, in EventArch 3.0 all communication that is related to the concern of coordination is encapsulated in a separated scope.

6.4 Use case

To increase the value of this evaluation of EventArch 3.0, the language is applied to a use case that originated in the field of energy-efficient computing. This use case is introduced in the following.

The worlds energy ressources are constantly diminishing, while the worlds energy demand is constantly growing. To cope with the problem of limited energy ressources, the available ressources have to be used efficiently. Due to that challenge, the question of how to increase **energy-efficiency has become an interesting topic** in science and technology. Several approaches have been developed to increase energy-efficiency. Energy-aware software-adaption and load-migration are two of them.

In [26], possible interference between the load-balancing- and software-adaption approach have been shown. In [22], a use case is concluded from that to evaluate EventArch 2.0. This **use case shall be reused** in this thesis to evaluate EventArch 3.0.

An increased energy-efficiency can be achieved by migrating a virtual machine that is executing some application, from one computing-service ("server") to another. This approach is termed "load-balancing". It is situated on "platform-level". As an alternative approach, the algorithm of the running application can be adapted at runtime ("software-adaption"). Both approaches can be combined. To evaluate EventArch 3.0, two coordination rules are defined for a SoS that consists of a constituent system "**LoadBalancer**" and a constituent system "**AdaptiveSoftware**". The LoadBalancer consists of the components "LoadMonitor", "LoadAnalyzer", "MigrationPlanner", and "Migrator". The AdaptiveSoftware consists of several "Application Com-

ponents" and several "Optimizer Components". To prevent unintended behaviour that may be caused by the simultaneous operation of both constituent systems, two coordination rules are defined.

The "State" coordination rule shall apply if the AdaptiveSoftware is running and the LoadBalancer detects the underutilization of the server. The "Switch" coordination rule shall apply if the LoadBalancer detects the overutilization of the server. According to the "**State**" **coordination rule**, the LoadBalancer does not stop the current adaption-process immediately, but waits until the current request for software-adaption has been processed. Subsequent requests shall be postponed until the LoadBalancer has finished migration. According to the "**Switch**" **coordination rule**, the AdaptiveSoftware is stopped and the migration is performed by the LoadBalancer immediately. Prior to that, the AdaptiveSoftware shall be commanded to switch to the least energy-consuming algorithm. The use case is illustrated by the figure 6.1

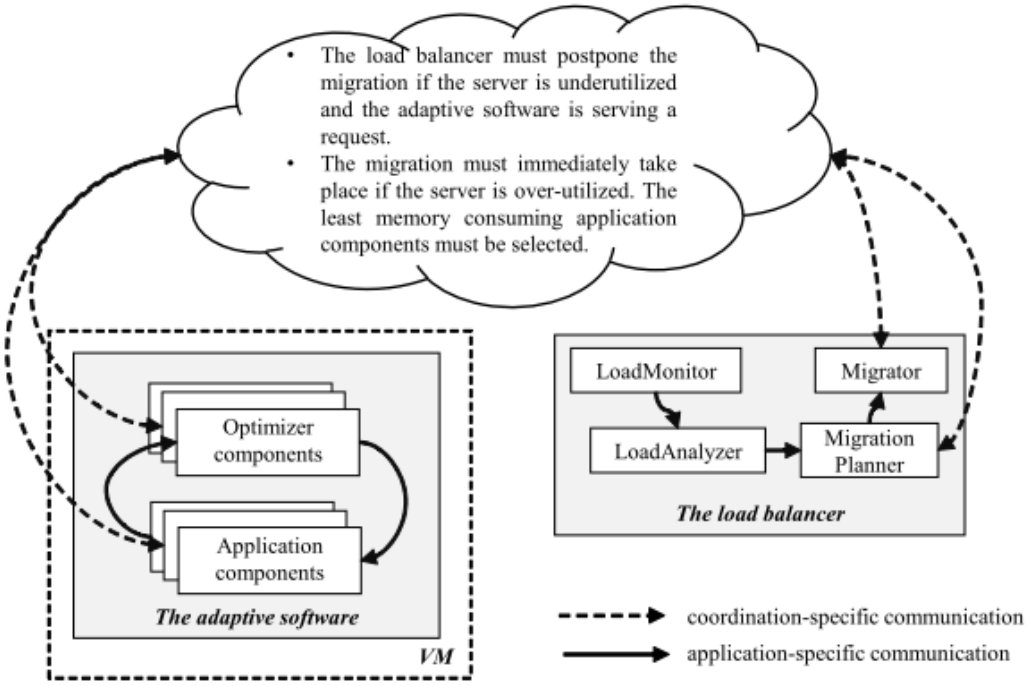

Figure 6.1: Illustration LoadBalancer/AdaptiveSoftware use case (taken from [22])

It is assumed that the load-balancing facilities and the software-adaption facilities of the respective constituent systems can be enabled and disabled at runtime. It is further assumed that if one of the two constituent systems is disabled, the respective other can not continue operating if the coordination rules remain active. Such a scenario would require the dynamic application of the two coordination rules in dependence of the composition of constituent systems in the SoS.

6.5 Application to the Example Use case

This section presents an application of EventArch 3.0 to the use case that has been described in the preceding section. An architectural specification is presented that specifies the dynamic architectural setup of the use case. It is pointed out, in what way the devised specification could take advantage from the distinguishing features of EventArch 3.0.

6.5.1 Presentation of the implementation

The described use case involves two approaches for optimizing the energy-efficiency of the running software: the LoadBalancer achieves an optimized energy-efficiency by migrating the running software to a different server, i.e., to different computing hardware. The AdaptiveSoftware achieves an optimized energy-efficiency by exchanging the employed algorithms, i.e., by adapting the software-algorithms at runtime. To prevent mutual interference, both LoadBalancer and AdaptiveSoftware should not impact the running software simultaneously, i.e., the software should not be migrated in a moment in that its algorithms are exchanged.

The described scenario can be solved by the dynamic application of coordination rules. Depending on the internal state of the LoadBalancer, the "Switch"- coordination rule or the "State"-coordination rule is dynamically applied. The dynamic application is conditioned by the composition of constituent systems in the SoS. If both systems are available, the rules are applied, i.e., the coordinators are composed with the constituent systems. If one constituent system leaves the SoS, the coordination rules are unapplied from the respective other. Due to those dynamic facets, implementing this scenario in EventArch 3.0 would show the fittness of this ADL for specifying the dynamic application of different coordination rules in dependence of the composition of constituent systems in the SoS.

In the following, the architectural setup according to the "Switch"-coordination rule is presented. The presentation includes the specification of relevant inner roles and outer roles of the constituent systems. Moreover, the specification of the Role-Binder is explained. The specification of the "State"-coordination rule can be found in the appendix (see 8.1.2). The implementation of the "State"-coordination rule is not described, as the Role-Binder is currently not concerned with selecting one of the two coordination rules for application(see future work: 7.1). The Role-Binder is just concerned with determining the current composition of constituent systems in the SoS. Moreover, the presentation is dedicated to the inherent dynamics and not to the implementation and functionality of the individual coordinators.

In the architectural specification, constituent systems are represented as "modules". The specification of the constituent systems is shown in figure 6.1.

```
1 modules {
2    WrappedLoadBalancer[Java]:= {ISwitchCoordinatorLB,
```

```
3                                    IStateCoordinatorLB} <-> {"LoadBalancer"}
4   WrappedAdaptiveSoftware[Java]:= {ISwitchCoordinatorAS,
5                                 IStateCoordinatorAS} <->{"AdaptiveSoftware"}
6 }
```

Listing 6.1: Specification of the module-section of the LoadBalancer/AdaptiveSoftware use case

The LoadBalancer and the AdaptiveSoftware are wrapped inside an Architectural Event Module (AEM). Therefore, the modules are called *WrappedLoadBalancer* and *WrappedAdaptiveSoftware*. For each constituent system, Primitive Interfaces are defined in the specification. The module-definition associates each constituent system with its Primitive Interfaces.

Figure 6.2 contains the specification of the Primitive Interface "ISwitchCoordinatorLB" of the LoadBalancer. This Primitive Interface is marked "private". That means, that it is regarded as an "Inner Role" of the module *WrappedLoadBalancer*. Inner roles are deactivated by default and can be activated at DCAEM-creation. Inner roles are deactivated at DCAEM-destruction.

```
1 private interface ISwitchCoordinatorLB for WrappedLoadBalancer{
2   requires {
3     CoordinationCommand e_CoordinationCMD = {E | E instanceof
4                                                 "CoordinationCommand"
5                                 && E.coordRule == "SwitchCoordinationRule"
6                                                 }
7   }
8   provides {
9     ConstituentState e_StartLB := before execution (
10       void org.dummy.loadbalancer.MigrationPlanner.plan(String load))
11         if (load =="OverUtilized") {
12           serverLoad = load;
13           applicationState = "StartExecuting";
14         }
15
16     wait when (e_StartLB) until (e_CoordinationCMD){
17         switch (e_CoordinationCMD.command){
18           case "proceed": proceed;
19           case "suspend":      suspend;
20         }
21       }
22
23     ConstituentState e_EndLB := after execution (
24                   void org.dummy.loadbalancer.Migrator.migrate(..)){
25                     applicationState = "EndExecuting";
26                   }
27   }
28 }
```

Listing 6.2: Specification of LoadBalancers inner role that is responsible for connecting it to the coordinator of the "Switch"-coordination rule

The inner roles connect the constituent system to its coordinators. The constituent system is associated to one coordinator per coordination-rule. The coordinators are understood to be the "Outer Roles" of the constituent system. They are encapsulated in a Compartment. A Compartment may represent a coordination rule. It would then contain all coordinators that implement that coordination rule. In figure 6.3

```
1 compartment SwitchCoordination {
2         role LoadBalancerCoord[StateMachine] := {IBaseDirectedLB,
3                                                  ICompDirectedLB} <-> {...}
4         interface IBaseDirectedLB for LoadBalancerCoord{...}
5         interface ICompDirectedLB for LoadBalancerCoord{...}
6
7         role AdaptiveSoftwareCoord[StateMachine] := {IBaseDirectedAS,
8                                                      ICompDirectedAS} <-> {...}
9         interface IBaseDirectedAS for AdaptiveSoftwareCoord{...}
10        interface ICompDirectedAS for AdaptiveSoftwareCoord{...}
11
12        roleBinder SwitchCoRolebinder[StateMachine] := {IExternal,
13                                                        ICompIntern} <->{...}
14        interface IExternal for StateCoRoleBinder{...}
15        interface ICompIntern for StateCoRoleBinder{...}
16
17 }
```

Listing 6.3: Sketch of the definition of a Compartment that represents the "Switch"-coordination rule in the architectural specification.

a Compartment is sketched that contains all coordinators of the "Switch"-coordination rule. It can therefore be understood to represent the "Switch"-coordination rule in the architectural specification. This Compartment contains a Role-Binder as well. The Role-Binder is dedicated to this Compartment.

In listing 6.4,

```
1 role LoadBalancerCoord[StateMachine] := {IBaseDirectedLB,
2                                          ICompDirectedLB} <-> {
3   initial state Idle{
4     during:
5       on(IBaseDirectedLB.e_StartedLB){//If the LoadBalancer is started
6                                       //...suspend it
7         send IBaseDirectedLB.e_LB_CoordinationCMD =
8                                             new CoordinationCommand(){
9                                   command = "suspend";
10                                  coordRule="SwitchCoordinationRule";
11                                            };
12
13        //...and command the AdaptiveSoftware to switch algorithms
14        send ICompDirectedLB.e_AS_CoordinationCMD =
15                                            new CoordinationCommand(){
16                                command = "switch";
17                                coordRule = "SwitchCoordinationRule";
18                                            };
19      } -> LB_underutilized_waiting_for_switch; //change to
```

```
20                                                    //another state
21    }
22
23    state LB_underutilized_waiting_for_switch {
24      during:
25        on(ICompDirectedLB.e_EndedSwitch){ //if the AdaptiveSoftware has
26                                           //finished switching algorithms...
27                                           //...let the LoadBalancer proceed
28          send IBaseDirectedLB.e_LB_CoordinationCMD =
29                                                  new CoordinationCommand(){
30                                        command = "proceed";
31                                        coordRule="SwitchCoordinationRule";
32                                                  };
33        }-> LB_migrating;
34    }
35
36    state LB_migrating {
37      during:
38        //if the LoadBalancer has finished the migration-process,
39        //switch to the Idle-state
40        on(IBaseDirectedLB.e_EndedMigrate) -> Idle;
41    }
42  }
```

Listing 6.4: Definition of the coordinator that will act as outer-role of the LoadBalancer to achieve compliance with the "Switch"-coordination rule

the specification of the coordinator is presented that will be responsible for coordinating the LoadBalancer with the other modules according to the "Switch"-coordination rule. It is therefore contained in the Compartment that represents that rule. This coordinator can be dynamically bound to the LoadBalancer by a Role-Binder. The coordinator is therefore understood to be an outer-role of the LoadBalancer. It is labeled with the keyword "role" and named "LoadBalancerCoord".

The specification of the Primitive Interfaces of this coordinator is presented in figure 6.5

```
1  interface IBaseDirectedLB for LoadBalancerCoord{
2
3    requires {
4      ConstituentState e_StartedLB = {E | E instanceof 'ConstituentState'
5                                && E.applicationState == 'StartExecuting'
6                                && E.serverLoad == 'OverUtilized'}
7
8      ConstituentState e_EndedMigrate = {E | E instanceof 'ConstituentState'
9                                 && E.applicationState == 'EndExecuting'}
10   }
11
12   provides {
13     CoordinationCommand e_LB_CoordinationCMD;
14   }
15
```

```
16 }
17
18 interface ICompDirectedLB for LoadBalancerCoord{
19
20   requires{
21     ConstituentState e_EndedSwitch = {E | E instanceof 'ConstituentState'
22                                     && E.applicationState == 'SwitchEnd'}
23   }
24
25   provides {
26     CoordinationCommand e_AS_CoordinationCMD;
27   }
28 }
```

Listing 6.5: Definition of the Primitive Interfaces of the coordinator that will act as outer-role of the LoadBalancer to achieve compliance with the "Switch"-coordination rule

The Primitive Interface *IBaseDirectedLB* is intended to be used for DCAEM- communication. It is therefore marked by the keyword "private". The Primitive Interface *ICompDirectedLB* is not marked "private". Therefore, its events are not published to the roles DCAEM, but to the scope of the Compartment that it is defined in.

The coordinator of the AdaptiveSoftware and its interfaces are presented in figure 6.6.

```
1  role AdaptiveSoftwareCoord[StateMachine] := {IBaseDirectedAS,
2                                                ICompDirectedAS} <-> {
3    initial state forward {
4      during:
5        on (ICompDirectedAS.e_SwitchCommand_required){
6          send IBaseDirectedAS.e_SwitchCommand_provided =
7            new CoordinationCommand(){
8                command = 'switch';
9                coordRule = 'SwitchCoordinationRule';};
10       }
11
12       on (IBaseDirectedAS.e_EndSwitchState_required){
13         send ICompDirectedAS.e_EndSwitchState_provided =
14           new ConstituentState(){applicationState = 'SwitchEnd';};
15       }
16   }
17 }
18
19 interface ICompDirectedAS for AdaptiveSoftwareCoord{
20   requires {
21     CoordinationCommand e_SwitchCommand_required = { E | E instanceof
22                                                      'CoordinationCommand'
23                                                   && E.command == 'switch'
24                                                      }
25 }
26   provides{
27     ConstituentState e_EndSwitchState_provided;
```

```
28   }
29 }
30
31 interface IBaseDirectedAS for AdaptiveSoftwareCoord{
32   requires {
33     ConstituentState e_EndSwitchState_required = { E | E instanceof
34                                                    'ConstituentState'
35                                 && E.applicationState == 'SwitchEnd'
36                                                    }
37 }
38
39   provides {
40     CoordinationCommand e_SwitchCommand_provided;
41   }
42 }
```

Listing 6.6: Definition of the behaviour and the Primitive Interfaces of the coordinator that will act as outer-role of the AdaptiveSoftware to achieve compliance with the "Switch"-coordination rule.

It is just concerned with forwarding certain messages from the LoadBalancers coordinator to the AdaptiveSoftware and vice versa. That coordinator could have been omitted, but according to EventArchs 3.0 compartment-concept each constituent system that is concerned with the coordination rule is represented by a role in the Compartment that represents that coordination rule.

Listing 6.7 contains the definition of an EventArch 3.0 Role-Binder.

```
1  roleBinder SwitchCoRoleBinder[StateMachine] := {IExternal} <-> {
2    initial state noSystemThere{
3      during:
4        on(IExternal.lb_started || IExternal.as_started) {} ->
5                                                    oneSystemThere;
6    }
7
8    state oneSystemThere{
9      during:
10       on(IExternal.lb_terminated || IExternal.as_terminated) {} ->
11                                                     noSystemThere;
12       on(IExternal.lb_started || IExternal.as_started){} ->
13                                                   twoSystemsThere;
14   }
15
16   state twoSystemsThere{
17     entry:
18       atomic {
19         DcaemLoadBalancer[composite] := {LoadBalancerCoord.
20                                                   IBaseDirectedLB} <->
21                                 {WrappedLoadBalancer.ISwitchCoordinatorLB}
22         DcaemAdaptiveSoftware[composite] := {} <->
23                               {WrappedAdaptiveSoftware.ISwitchCoordinatorAS}
24         DcaemAdaptiveSoftware[composite] += {AdaptiveSoftwareCoord.
25                                                   IBaseDirectedAS}
```

```
26        }
27
28      during:
29        on(IExternal.lb_terminated || IExternal.as_terminated) {} ->
30                                                        oneSystemThere;
31
32      exit:
33        DcaemLoadBalancer[composite] -= {LoadBalancerCoord.
34                                                        IBaseDirectedLB}
35        destroy DcaemLoadBalancer[composite]
36        DcaemAdaptiveSoftware[composite] -= {AdaptiveSoftwareCoord.
37                                                        IBaseDirectedAS}
38        destroy DcaemAdaptiveSoftware[composite]
39    }
40 }
```

Listing 6.7: Definition of the rolebinder that is responsible for binding the coordinators of the "Switch"-coordination rule

Listing 6.8 contains its Primitive Interface *IExternal*.

```
1  interface IExternal for SwitchCoRoleBinder{
2    requires {
3      ApplicationStarted lb_started = {E | E instanceof
4                                                  'ApplicationStarted'
5                                  && E.publisher == 'WrappedLoadBalancer'
6                                  }
7      ApplicationStarted as_started = {E | E instanceof
8                                                  'ApplicationStarted'
9                                && E.publisher == 'WrappedAdaptiveSoftware'
10                                 }
11
12     ApplicationTerminated lb_terminated = {E | E instanceof
13                                                  'ApplicationTerminated'
14                                 && E.publisher == 'WrappedLoadBalancer'
15                                 }
16
17     ApplicationTerminated as_terminated = {E | E instanceof
18                                                  'ApplicationTerminated'
19                               && E.publisher == 'WrappedAdaptiveSoftware'
20                                 }
21   }
22 }
```

Listing 6.8: Definition of the Primitive Interface of the rolebinder of the "Switch"-coordination rule

The Role-Binder is contained in the Compartment that represents the "Switch"-coordination rule. It is therefore just responsible for binding the coordinators of that rule to the constituent systems. The Role-Binder has a critical task in the architectural setup. It has to process information concerning the current composition of constituent systems in the SoS to take a binding decision. Moreover, it has to execute this binding decision by performing role-binding. The Role-Binder *integrates* the

other features, that have been added to EventArch, to achieve a dynamic application of coordination rules in dependence of the composition of constituent systems in the SoS.

The Role-Binder is informed by *events* about the presence and absence of constituent systems in the SoS. Whenever a new constituent system enters the SoS, an *ApplicationStarted* event is sent to the Role-Binder. Whenever a consituent system leaves the SoS, the Role-Binder is informed by an *ApplicationTerminated* event, likewise. The Role-Binder employs a special representation of the current composition of coordination-rules. It introduces *states* to represent a specific composition. For that purpose, the states *noSystemThere*, *oneSystemThere*, and *twoSystemsThere* have been introduced into the Role-Binder. Changes to the composition of the constituent systems in the SoS are reflected by state-changes of the Role-Binders statemachine. When both constituent systems, WrappedLoadBalancer and WrappedAdaptiveSoftware, are available in the SoS, the respective **Dynamic Composite AEMs** (DCAEM) are created. Relevant **inner roles** are activated and **outer roles** are included into the DCAEM. The described binding- and creation-actions are performed on entering the state *twoSystemsThere*.

The statement

```
1 DcaemLoadBalancer[composite] := {LoadBalancerCoord.IBaseDirectedLB} <->
      {WrappedLoadBalancer.ISwitchCoordinatorLB}
```

creates the DCAEM *DcaemLoadBalancer*. The module *WrappedLoadBalancer* is included as the base of this DCAEM. The Primitive Interface *ISwitchCoordinatorLB* serves as inner role of the DCAEM. The coordinator *LoadBalancerCoord* (see listing 6.4) serves as an outer role of this DCAEM.

Additional outer roles may be bound after the creation of the DCAEM as well. The DCAEM *DcaemAdaptiveSoftware* is created by the line

```
1 DcaemAdaptiveSoftware[composite] := {} <-> {WrappedAdaptiveSoftware.
      ISwitchCoordinatorAS}
```

An outer role is bound to this DCAEM after its creation. This is done by the line

```
1 DcaemAdaptiveSoftware[composite] += {AdaptiveSoftwareCoord.
      IBaseDirectedAS}
```

A formal description of the syntax of EventArch 3.0 can be found in the appendix (8.3). It is an extract of the grammar that defines the language of EventArch 3.0. A similiar Role-Binder has been defined inside the Compartment that represents the "State"-coordination rule.

6.5.2 Advantages shown by the implementation

The application of EventArch 3.0 to the use case has shown the following advantages:

- The integration of the base and roles could be achieved by means of a **common message scope**. The coordinators *LoadBalancerCoord* and *AdaptiveSoftwareCoord* could be included as "**outer roles**" into this message scope. From point of view of the architectural developer, defining role-binding is not so much different in EventArch 3.0 from defining it in other languages. Like in SCROLL and SMAGs, roles are explicitly added to a base by specific statements. Two syntactical variants have been presented in the presentation of the Role-Binder. The real advantage is hidden from the architectural developer: a role is not bound to a base by dynamically incorporating it into the base-object (like it is done in SCROLL using SCALAs dynamic traits feature), but by a simple message-based mechanism. Messages are sent using a message-oriented-middleware platform (EventArch 3.0 uses JMS). This would allow to easily exchange the implementation of a role at runtime. It would not be necessary to interfere with a base-object (or "base-system" in the case of EventArch 3.0), but to merely include the new implementation of the role into a message scope and to exclude the old implementation from that scope.
- The implementation of the use case has concerned the behaviour of multiple components of the constituent systems in order to implement the concern of coordination. In the example-implementation, those cross-cutting concerns could be implemented without scattering the relevant code across those components and without having to tangle it with the original concerns of the constituent system. This could be achieved by EventArchs 3.0 **Cross-cutting Roles**. EventArchs 3.0 "private" Primitive Interfaces can be understood as Cross-cutting Roles in that they could be activated and deactivated by the Role-Binder. Moreover, they could be concerned with multiple components of the constituent systems. For example, the Primitive Interface *IStateCoordinatorLB* of the constituent system *WrappedLoadBalancer* (see appendix: 8.5) is concerned with the LoadBalancers components *MigrationPlanner* and *Migrator* and can therefore be understood as an implementation of a cross-cutting concern.
- The information concerning the entering and leaving of the LoadBalancer and AdaptiveSoftware could be disseminated by **events**. The constituent systems could take advantage from that, in that they could publish the *ApplicationStarted*- and *ApplicationTerminated*-events to both available Role-Binders. This could be done due to the loose-coupling that was induced by this scheme of communication. Event-publishers do not have to process a return value and can therefore publish the information to multiple receivers. The Role-Binder could also take advantage from the event-based communication. Instead of mapping a received message to specific function-code, they were encouraged to install a dispatch- and handling mechanism. The defined statemachine can be understood, together with the defined "on(...)" expressions, as a state-dependent dispatch- and handling mechanism. For example, depending on the current state, the *ApplicationStarted*-event of the *WrappedLoadBalancer*

triggers a state-transition to the state *oneSystemThere* or *twoSystemsThere*. That means, depending on the current state, the "on(...)"-dispatch statements, dispatch the event to a "handler" whose action consists in switching to the *oneSystemThere*-state, or the "on(...)"-dispatch statements dispatch the event to a "handler" whose action consists in switching to the *twoSystemsThere*-state.

- The current composition of constituent systems of a SoS could be represented by a stateful **Role-Binder**. Changes in the composition of constituent systems in the SoS could be reflected by statechanges in the Role-Binder. The states could be given intuitive names: *noSystemThere*, *oneSystemThere*, *twoSystemsThere*. The *simultaneous* composition of *all* coordinators of *the same* coordination rule with concerned constituent systems could be accomplished. Currently, it is not possible to select a specific coordination rule for application based on the internal state of the constituent systems. This prevented the devised solution to select the "State"-coordination rule for application if the LoadBalancer has found the server-load to be "UnderUtilized" and the "Switch"-coordination rule if the LoadBalancer has found the server-load to be "OverUtilized". The inability of the solution to select a specific coordination rule for application, not only based on the composition of constituent systems in the SoS, but also on the current state of the constituent systems, is a serious disadvantage that has to be solved by future-work.
- The architectural setup could be specified **transparently** to the code of the constituent system. The base-code did not have to be changed in order to achieve compliance to coordination rules. Instead, Primitive Interfaces werde defined outside the base-code. To interfere with the constituent systems, pointcuts were defined and internal functions were triggered. For example, the Primitive Interface *IStateCoordinatorLB* (see appendix: 8.5) of the constituent system *WrappedLoadBalancer* defined the pointcuts

```
1 before execution (void org.dummy.loadbalancer.MigrationPlanner.plan
    (String load))
2 after execution (void org.dummy.loadbalancer.Migrator.migrate(..))
```

Moreover, the Primitive Interface *ISwitchCoordinatorAS* of the constituent system *WrappedAdaptiveSoftware* called the internal function "*org.dummy.application.Optimizer.reconfigure*" of the AdaptiveSoftware by the line.

```
1 on (e_CoordinationCMD) {invoke ('org.dummy.application.Optimizer',
    'reconfigure',e_CoordinationCMD); }
```

In the OT/J implementation, that was presented in section 3.3, the code of the base-classes had to be changed. The start of the system had to be explicitly communicated to the team that implemented the coordination rule, and relevant components had to be registered at the *ExampleRunner*. The relevant figure can be found in the appendix: 8.1.1

Chapter 7

Conclusion

The conclusion of this thesis presents a restatement of the problem that it has been motivated by. Then, the prevailing line of argumentation of this thesis is reproduced. Finally, the main contributions of EventArch 3.0 for achieving a solution for the stated problem are emphasized.

Currently, the value of independent systems is increased by arranging them as constituents of a "System-of-Systems" (SoS). Constituent systems may be subject to emergence. To prevent the failure to fulfill certain SoS-level goals due to unintended behaviour that is caused by emergence, coordination rules can be defined. In a dynamic SoS, constituent systems may join or leave at runtime. This creates a **problem** with respect to the application of coordination rules. The level of restrictiveness of a coordination rule that is applied to the SoS at a certain moment in time should match the level of restrictiveness that is actually required by the current composition of constituent systems in that SoS. This creates the desire to dynamically apply variants of a coordination rule to the SoS that differ with respect to their level of restrictiveness. More-restrictive coordination rules should be generally applied if constituent systems join the SoS and less-restrictive coordination rules should be generally applied if constituent systems leave the SoS. Depending on the current composition of constituent systems, a coordination rule should be applied that matches the level of restrictiveness that is actually required with respect to that specific composition of constituent systems. As a prerequisite, those application scenarios require means to dynamically apply coordination rules in dependence of the current composition of constituent systems in the SoS.

In this work, several **requirements** have been identified that would be desirable for an architecture description language in order to provide support for the dynamic application of coordination rules in a dynamic SoS in dependence of the current composition of constituent systems in the SoS.

Moreover, certain **features** have been identified, that would render an architecture description language capable to fulfill all requirements. Current object-oriented programming- and modeling-languages have been considered as **related work** and have been analyzed with respect to those features. Their support for those features has been rated. Focus has been put on the role-based programming-language "OT/J".

The architecture description language "EventArch 2.0" has been selected for extension in order to achieve support for the features that have been identified. As a solution, the architecture description language "EventArch 3.0" has been devised. The **concepts of EventArch 3.0** that contribute to the described application scenario have been thoroughly described in that work. Moreover, an in-depth description of **EventArchs 3.0 implementation** has been given.

Finally, it has been shown by **evaluating EventArch 3.0** that this ADL fulfills all requirements that would be desirable to achieve support for the dynamic application of coordination rules in a dynamic SoS in dependence of the current composition of constituent systems in the SoS. An **example** has been presented that involved two constituent systems and consequently features the alternating ignorance- or compliance with respect to a specific coordination rule, depending on the presence of the respective other system.

The major contributions of EventArch 3.0 with respect to providing support for the dynamic application of coordination rules in a dynamic SoS in dependence of the current composition of constituent systems in the SoS, can be described as follows:

1. The language-element "**Dynamic Composite AEM**" (DCAEM) allows for the integration of coordinators with constituent systems at runtime. A DCAEM is a common message-scope for a constituent system and its coordinators. It implements a loose coupling between them. A DCAEM can be used to apply a coordination rule at runtime, depending on the current composition of constituent systems in the SoS. That feature is a prerequisite to allow for an integration of new coordination rules into the behaviour of constituent systems, that have been unanticipated at design-time. This runtime-integration may be necessary in response to emergence that has been experienced with respect to the behaviour of individual constituent systems.
2. The capability of an architecture description language to modularize cross-cutting concerns can be used to modularize the cross-cutting concern of coordination. In the design of EventArch 3.0, the capability to modularize cross-cutting concerns has been combined with the notion of adding and removing functionality at runtime. That functionality has been represented as "Role". Consequently, the feature "**Cross-cutting Role**" was devised and implemented into the language EventArch 3.0. It has been shown, that it is feasible to represent coordinators as Cross-cutting Roles and to dynamically apply coordination rules by combining those coordinators with constituent systems at runtime.
3. A **Role-Binder** has been devised as a modular solution that allows for the consistent application of a coordination rule to constituent systems. The consistent application of a coordination rule requires the *simultaneous* application of *all* coordinators that are associated to the *same* coordination rule to those constituent systems that are concerned by that rule. A Role-Binder may be defined as a statemachine and state-transitions may be conditioned by the entry and exit of constituent systems to the SoS. In that way, the Role-Binder can

be used to achieve the desired dependence of its decisions with respect to the current composition of constituent systems in the SoS.

4. Coordination rules can be implemented according to the "Peer-to-Peer" coordination pattern. In EventArch 3.0, the role-related concept "Compartment" has been used to represent a coordination-rule that is implemented according to that pattern. A **compartment** encapsulates all coordinators that implement a coordination rule and creates a common message scope for them. This message scope separates their communication from the other constituent systems of the SoS. In that way, communication that is related to the coordination rule is separated from constituent systems that are unrelated to that coordination rule. This achieves privacy with respect to the concern of coordination and is therefore desirable in an effort to achieve the dynamic application of coordination rules in a dynamic SoS.

7.1 Future Work

In this thesis an extension of the architecture description language EventArch 2.0 has been devised that has increased EventArchs support for adaptive coordination in dynamic SoS, considerably. Nevertheless, further improvements could be achieved by applying some further extension to the language. In this section potential starting points for future extensions are pointed out.

Provide support for the integration of unanticipated coordination rules into SoS.
In a dynamic SoS, constituent systems may join or leave at runtime. This requires the dynamic application of coordination rules to the SoS by composing coordinators with those constituent systems that are concerned by the rule. Currently, it is only possible to integrate coordination rules that have been anticipated at design-time into the behaviour of constituent systems. Coordination-rules that have been unanticipated at design-time can not be integrated without stopping and restarting all constituent systems and all coordinators of the SoS. For future support for the integration of unanticipated coordination rules at runtime, a mechanism is needed to integrate changes to the architectural specification into a running SoS. To achieve that, it would be required to update the metadata that represents the architectural representation at runtime and to selectively compile those Architectural Event Modules whose specifications have been changed.

Provide support for the replacement of a coordination rule by a variant of itself at runtime.
As indicated in the motivation (see 1.1), EventArch 3.0 does not intend to support the replacement of a coordination rule by a variant of itself at runtime. Nevertheless, this would be a useful feature, as it would allow to integrate a variant of the coordination rule into a running SoS. This variant could be designed to match the

level of restrictiveness that is currently desired in the SoS. This would help to prevent unintended behaviour on SoS-level.

To achieve that support, a mechanism would have to be implemented that allows for a proper initialization of the coordinators of the new variant. Those coordinators would have to be initialized in a way that allows for the seamless continuation of the coordination services in the SoS. To achieve that, the state of the coordinators of the deselected variant has to be translated into a suitable initial state of the coordinators of the selected variant. This state-translation has to be performed in the moment of rule-change.

Improve support for the selection of a coordination rule for application based on the internal state of the constituent systems.
EventArch 3.0 supports the dynamic application of coordination rules based on the current composition of constituent systems in the SoS. Currently, poor support is provided for basing that decision on the internal state of constituent systems. To allow the Role-Binder to take an informed role-binding decision, the constituent systems have to provide a representation of their internal state to the Role-Binder. Currently, EventArch 3.0 assumes that a constituent system is in need of coordination when it publishes certain state-informations. Therefore, the inner roles have to be activated to publish state-information and the wait-when blocks are activated as well. In an improved version of EventArch it might be possible to activate the inner roles, publish state-information to the Role-Binder, but to depend the activation of the wait-when blocks of the inner roles on the current need for coordination, i.e., on the current composition of constituent systems in the SoS.

A possible solution would be to allow the Role-Binder to command the activation of specific wait-when blocks in the inner roles based on the current composition of constituent systems in the SoS. For example, the LoadBalancer might have to suspend execution when it publishes the current server-load to the Role-Binder if the AdaptiveSoftware is currently present in the SoS. But it would not have to suspend execution when it publishes the current server-load to the Role-Binder while the AdaptiveSoftware is currently not present in the SoS. This would allow the Role-Binder to select the application of the “Switch”-coordination rule or the “State”-coordination rule based on the internal state of the LoadBalancer (“server-load”), without interfering with the LoadBalancers behaviour by unnecessary suspension.

Provide extended support for the dispatching and handling of events.
Functions are often used as a tight mapping between a symbolic name and a piece of implementation-code. Opposed to that, events are not used to identify a specific piece of implementation-code. Instead, they are typically dispatched at the receiver. Event-based Systems typically provide a dispatch- and handling-mechanism. They can be dispatched according to certain criteria and be handled by specific implementation code.

In EventArch, coordinators can be implemented as statemachines. A statemachine can be understood as a state-oriented dispatch- and handling-mechanism. Depending

on the current state, incoming events are selected by different "On"-expressions. Depending on the "On"-expression that has selected the event, different actions are executed. The defined statemachine-actions that process the event, can be understood as the "handler" of the event. Then, the "On"-expressions and the state-dependence of event-selection can be understood as the dispatch-mechanism. Currently, both mechanisms (dispatch- and handling-) are defined using EventArchs statemachine-language. Other programming languages can not be employed.

To allow the SoS-manager to implement event-handling in a programming language of his choice (e.g. Java or C++), but to still take advantage of EventArchs event-dispatch mechanism, it would be desirable to separate event-dispatch from event-handling in EventArchs statemachine. To achieve that, EventArchs statemachine would have to be implemented as a wrapper around a legacy-application. Then, it would be possible to integrate calls to specific functions of that legacy-code as a specific type of "statemachine-action" into the definition of a statemachine. As the Role-Binder is implemented as a statemachine, this extension could be used to enhance the process of role-binding as well.

Other extensions

In the following, further possible extensions are briefly sketched.

- EventArch 3.0 supports the notion of Compartment. Nevertheless, it is a relatively poor support with respect to the criteria defined in [20]. In the future, EventArch could be extended to support more of those criteria. For example, compartments could be allowed to be nested.
- To increase the support for anticipated adaptation of coordination rules at runtime, statemachines could be allowed to dynamically change the selectors of event-based interfaces and to create new selectors at runtime.
- In a dynamic SoS, the number of constituent systems that are concerned by a specific coordination rule may be unclear. New constituent systems that are concerned by the coordination rule could join the SoS unanticipatedly at runtime. To allow for a dynamic application of such coordination rules, it would be necessary to instantiate new roles at runtime. Currently, EventArch 3.0 is not up to that. Therefore, EventArch 3.0 could be extended in the future to support the dynamic instantiation of roles.
- Currently, the features of "Inner Roles" and "Dynamic Composite AEM" are just supported for constituent systems that are implemented in the programming language "Java". That support could be extended by including other programming languages, e.g., C++. This would exploit the advantage "Language-Independent Base-Role Integration".
- EventArchs 3.0 support for the dynamic application of coordination rules that are implemented according to the "Centralized"-coordination pattern could still be improved. Primitive interfaces could be allowed to be bound to multiple DCAEMs, or a common message scope for all constituent systems that are

interacting with a specific central coordinator could be introduced.

- To allow for the runtime-integration of new coordination rules on the "Inner Roles"-level, EventArchs 3.0 implementation of the inner roles could be changed to employ aspect technology that provides the possibility of runtime-weaving. This would increase the ability of the ADL to integrate a coordination rule into the behaviour of constituent systems at runtime that has not been anticipated at design-time.

Chapter 8

Appendix

The Appendix contains an in-depth discussion of the implementation of the EventArch 2.0-compiler and additional source-code. The grammar of EventArch 3.0 is described. Moreover, the extension that has been applied to EventArch 2.0 is described in detail by numerous class-diagrams and a sequence-diagram.

8.1 Additional Source-Code

8.1.1 OT/J source-code

In the following, the complete source code of the OT/J-solution of the described use case 6.4 is printed. The first listing contains the definition of the team that implements the "State"-coordination rule. The following listing contains a dummy-implementation of the constituent system "LoadBalancer".

```
1  public team class StateCoordinationRule {
2    private State noApplicationThere = new NoApplicationThere();
3    private State oneApplicationThere = new OneApplicationThere();
4    private State twoApplicationsThere = new TwoApplicationsThere();
5
6    private State currentState = noApplicationThere;
7    private State nextState = noApplicationThere;
8
9    public void informRoleBinder(String information){
10     nextState = currentState.getNextState(information);
11
12     if (!currentState.equals(nextState)){
13       currentState.doExitAction();
14       nextState.doEntryAction();
15       currentState = nextState;
16    }
17   }
18
19   public class TwoApplicationsThere implements State{
20
21     @Override
```

```
22      public State getNextState(String information) {
23        switch(information){
24          case "LoadBalancerLeft": return oneApplicationThere;
25          case "AdaptiveSoftwareLeft": return oneApplicationThere;
26          default: return twoApplicationsThere;
27        }
28     }
29      @Override
30      public void doEntryAction() {
31        activate(Team.ALL_THREADS);
32      }
33
34      @Override
35      public void doExitAction() {
36        deactivate(Team.ALL_THREADS);
37      }
38    }
39
40    public interface State{
41
42      public State getNextState(String information);
43      public void doEntryAction();
44      public void doExitAction();
45    }
46
47    public class NoApplicationThere implements State{
48
49      @Override
50      public State getNextState(String information) {
51        switch(information){
52          case "LoadBalancerJoined": return oneApplicationThere;
53          case "AdaptiveSoftwareJoined": return oneApplicationThere;
54          default: return noApplicationThere;
55        }
56      }
57
58      @Override
59      public void doEntryAction() {}
60
61      @Override
62      public void doExitAction() {}
63
64    }
65
66    public class OneApplicationThere implements State{
67
68      @Override
69      public State getNextState(String information) {
70        switch(information){
71          case "LoadBalancerJoined": return twoApplicationsThere;
72          case "AdaptiveSoftwareJoined": return twoApplicationsThere;
73          case "LoadBalancerLeft": return oneApplicationThere;
```

```
74          case "AdaptiveSoftwareLeft": return oneApplicationThere;
75          default: return oneApplicationThere;
76        }
77      }
78
79      @Override
80      public void doEntryAction() {}
81
82      @Override
83      public void doExitAction() {}
84    }
85
86    public class LoadBalancerMigrator playedBy Migrator {
87      notifyAdaptiveSoftware <- after migrate; //when the Migrator-
            component has finished the migration-process, it has to be
            interrupted
88
89      public void notifyAdaptiveSoftware(){
90        updateAppCompCoordinator();
91        appCompCoordinator.releaseCurrentLock(); //migration has been
              finished -> the lock of the adaptive software can be released.
92      }
93    }
94
95    public class LoadBalancerMigrationPlanner playedBy MigrationPlanner {
96      waitForProceed <- before plan; //when the MigrationPlanner-component
            is about to start planning, it has to be interrupted
97
98      public void waitForProceed(String load){
99        updateAppCompCoordinator(); //update the teams reference to the
              coordinator of the other system
100       if (load.equals("underutilized")){
101         LockRequest lbRequest = new LockRequest("loadBalancer"); //
              LoadBalancer and AdaptiveSoftware should not run
              simultaneously. Therefore, the respective other has to be
              locked.
102
103         enqueueRequest(lbRequest); //dont forget that you, the
              LoadBalancer, has requested a lock.
104         appCompCoordinator.enqueueRequest(lbRequest); //the coordinator-
              role of the adaptive software has to know, that you
              requested to lock the adaptive software
105
106         LockRequest headRequest = getQueueHead(); //what is the current
              lock, that we have to respect?
107         while(true){
108           if (headRequest.responsibleSoftware.equals("loadBalancer")){
109             return; //proceed if not the adaptivesoftware has requested
                  a lock.
110           }else{
111             waitForLockRelease(); //if the adaptiveSoftware has
                  requested a lock, wait until the adaptiveSoftware
```

```
                releases that lock.
112           }
113         }
114       }
115     }
116
117
118     public void enqueueRequest(LockRequest lockRequest){
119       //attach the lockRequest to the Request-Queue
120     }
121
122     public LockRequest getQueueHead(){
123       //return the head of the Request-Queue
124     }
125
126     public void waitForLockRelease(){
127       //wait until the adaptive software-coordinator releases the
              current lock
128     }
129
130     public void releaseCurrentLock(){
131       //release the current lock
132     }
133
134   }
135
136   protected AdaptiveSoftwareAppComponent appCompCoordinator;
137
138   public void updateAppCompCoordinator(){
139     ExampleRunner.updateTeamsApplicationComponent(this);
140   }
141
142   public void updateApplicationComponent(ApplicationComponent as
        AdaptiveSoftwareAppComponent appCompCoordinator){
143     this.appCompCoordinator = appCompCoordinator;
144   }
145
146   protected LoadBalancerMigrationPlanner migPlanCoordinator;
147
148   public void updateMigPlanCoordinator(){
149     ExampleRunner.updateTeamsMigrationPlannerComponent(this);
150   }
151
152   public void updateMigrationPlannerComponent(MigrationPlanner as
        LoadBalancerMigrationPlanner migPlanCoordinator){
153     this.migPlanCoordinator = migPlanCoordinator;
154   }
155
156   public class AdaptiveSoftwareAppComponent playedBy
        ApplicationComponent {
157
158     waitForProceed <- before execute;
```

```
159     notifyLoadBalancer <- after execute;
160
161     public void waitForProceed(){
162       updateMigPlanCoordinator();
163       LockRequest asRequest = new LockRequest("adaptiveSoftware"); //
              this represents your requested lock
164       enqueueRequest(asRequest); //dont forget, that you have requested
              a lock
165       migPlanCoordinator.enqueueRequest(asRequest); //announce your
              request for a lock to the loadbalancer
166       LockRequest headRequest = getQueueHead(); //what is the current
              lock, that we have to respect?
167       while(true){
168         if (headRequest.responsibleSoftware.equals("adaptiveSoftware")){
169           return; //proceed if not the loadbalancer has requested a lock
                  .
170         }else{
171           waitForLockRelease(); //if the loadBalancer has requested a
                  lock, wait until the loadbalancer releases that lock.
172         }
173       }
174     }
175
176     public void notifyLoadBalancer(){
177       updateMigPlanCoordinator();
178       migPlanCoordinator.releaseCurrentLock();
179     }
180
181     public void waitForLockRelease(){
182       //wait until the loadbalancer-coordinator releases the current
              lock
183     }
184
185     public void releaseCurrentLock(){
186       //release the current lock
187     }
188
189     public LockRequest getQueueHead(){
190        //return the head of the LockRequest-queue
191     }
192
193     public void enqueueRequest(LockRequest lockRequest){
194       //engueue lockRequest into the queue of LockRequests
195     }
196
197   }
198
199   public class AdaptiveSoftwareOptimizer playedBy Optimizer {
200     //call ``reconfigure''-function if needed
```

```
201   }
```

Listing 8.1: Definition of the team that implements the “State”-coordination rule of the example-implementation of the use case in the role-based language OT/J

The second listing contains a dummy-implementation of the constituent system “LoadBalancer”. As can be seen from this listing, the code of the LoadBalancer had to be changed in order to connect this system to its “State”-coordinator that is defined as a role in the respective team-definition.

```
1  package LoadBalancer;
2
3  import AdaptiveSoftware.ApplicationComponent;
4  import lb_asw_usecase.ExampleRunner;
5  import lb_asw_usecase.StateCoordinationRule;
6
7  public class LB_Main extends Thread {
8
9    private StateCoordinationRule stateCoordinationRule;
10   private MigrationPlanner migrationPlanner;
11   private Migrator migrator;
12   private ApplicationComponent appcomp;
13
14   public LB_Main(StateCoordinationRule stateCoordinationRule){
15     this.stateCoordinationRule = stateCoordinationRule;
16     this.migrationPlanner = new MigrationPlanner();
17     this.migrator = new Migrator();
18     ExampleRunner.registerMigrationPlannerComponent(migrationPlanner);
19   }
20
21   public void run(){
22     stateCoordinationRule.informRoleBinder("LoadBalancerJoined");
23     migrationPlanner.plan("underutilized");
24     migrator.migrate();
25   }
26 }
```

Listing 8.2: Dummy-implementation of the constituent system “LoadBalancer” in the role-based language OT/J

The following listing contains the “ExampleRunner”. It maintains a reference to the components of the constituent systems, that are of interest to the coordinators of the “State”-coordination rule. It provides a mechanism for component-registration to the constituent systems “LoadBalancer” and “AdaptiveSoftware”.

```
1  package lb_asw_usecase;
2
3  import AdaptiveSoftware.ASW_Main;
4  import AdaptiveSoftware.ApplicationComponent;
5  import LoadBalancer.LB_Main;
6  import LoadBalancer.MigrationPlanner;
```

```
 7
 8 public class ExampleRunner {
 9   private static ApplicationComponent appComponent;
10
11   public static void updateTeamsApplicationComponent(
         StateCoordinationRule stateCoordinationRule){
12     stateCoordinationRule.updateApplicationComponent(appComponent);
13   }
14
15   public static void registerApplicationComponent(ApplicationComponent
         appComponent){
16     ExampleRunner.appComponent = appComponent;
17   }
18
19   private static MigrationPlanner migPlanComponent;
20
21   public static void updateTeamsMigrationPlannerComponent(
         StateCoordinationRule stateCoordinationRule){
22     stateCoordinationRule.updateMigrationPlannerComponent(
           migPlanComponent);
23   }
24
25   public static void registerMigrationPlannerComponent(MigrationPlanner
         migPlanComponent){
26     ExampleRunner.migPlanComponent = migPlanComponent;
27   }
28
29   public static void main(String[] args) {
30     StateCoordinationRule stateCoordinationRule = new
           StateCoordinationRule();
31
32     LB_Main lb_Main = new LB_Main(stateCoordinationRule);
33     lb_Main.start();
34
35     ASW_Main asw_Main = new ASW_Main(stateCoordinationRule);
36     asw_Main.start();
37   }
38 }
```

Listing 8.3: Module “ExampleRunner” of the example-implementation of the use case in the role-based language OT/J

8.1.2 “State”-coordination rule

In the following, additional parts of the architectural specification of the “State”-coordination rule are printed. For brevity, the respective Compartment is not printed in full. The specification of the “State”-coordinator of the AdaptiveSoftware and the “State”-Role-Binder are omitted. The specification of the “State”-coordinator of the AdaptiveSoftware is very similiar to the printed specification of the “State”-coordinator of the LoadBalancer. The specification of the “State”-Role-Binder is

very similiar to the specification of the "Switch"-Role-Binder. The specification of this Role-Binder is printed in section 6.5

```
1  Compartment StateCoordination{
2
3  role LoadBalancerCoord[StateMachine] := {IBaseDirectedLB,
4                                           ICompDirectedLB} <->  {
5    PriorityEventQueue pending_requests;
6      initial state Idle {
7        during:
8          on (IBaseDirectedLB.e_StartedLB_UnderUtilized){
9            print ">>>>>> During Idle>>>>> Load Balancer  >>> insert in
10                                                              queue";
11           ICompDirectedLB.e_LockRequestByLB = new LockRequest();
12           pending_requests.add(ICompDirectedLB.e_LockRequestByLB );
13           send ICompDirectedLB.e_LockRequestByLB;
14           send IBaseDirectedLB.e_LB_CoordinationCMD =
15                                          new CoordinationCommand(){
16                                  command = 'suspend';
17                                  coordRule = 'StateCoordinationRule';
18                                          };
19         } -> Waiting;
20
21         on(ICompDirectedLB.e_LockRequestByAS){
22           print ">>>>>> During Idle>>>>> LB  >>> insert AS in queue";
23           pending_requests.add(ICompDirectedLB.e_LockRequestByAS);
24           send ICompDirectedLB.e_LockReplyByLB = new LockReply();
25         }
26
27         on (ICompDirectedLB.e_LockReleaseByAS){
28           print ">>>>>> During Idle>>>>> LB  >>> remove queue";
29           pending_requests.remove ();
30         }
31
32         on (IBaseDirectedLB.e_TerminatedLB)-> AlwaysGrant;
33     }
34
35   state AlwaysGrant {
36     during:
37       on(ICompDirectedLB.e_LockRequestByAS){
38         print ">>>>>> During AlwaysGrant >>>>> LB  >>> insert AS in
39                                                              queue";
40         pending_requests.add(ICompDirectedLB.e_LockRequestByAS);
41         send ICompDirectedLB.e_LockReplyByLB = new LockReply();
42       }
43         on (ICompDirectedLB.e_LockReleaseByAS){
44           print ">>>>>> During AlwaysGrant>>>>> LB  >>> remove queue";
45           pending_requests.remove ();
46         }
47
48         on(IBaseDirectedLB.e_InitiatedLB) -> Idle;
49     }
50
```

```
51    state Waiting {
52      during:
53        on(ICompDirectedLB.e_LockReplyByAS)[pending_requests.peek().get(
54                        'publisher') == 'LoadBalancerCoord'] -> Running;
55
56        on(ICompDirectedLB.e_LockReleaseByAS){
57          print ">>>>>> During Waiting>>>>> LB >>>remove AS from queue";
58          pending_requests.remove ();
59          pending_requests.peek().get('publisher')=='LoadBalancerCoord'
60                                                              -> Running;
61        }
62
63        on(ICompDirectedLB.e_LockRequestByAS){
64          print ">>>>>> During Waiting>>>>> AS >>> insert AS queue";
65          pending_requests.add(ICompDirectedLB.e_LockRequestByAS);
66          send ICompDirectedLB.e_LockReplyByLB = new LockReply();
67        }
68    }
69
70    state Running {
71      entry:
72        print ">>>>>> Entry Running >>>>> Load Balancer >>> Proceed LB";
73        send IBaseDirectedLB.e_LB_CoordinationCMD = new
74                                                    CoordinationCommand(){
75                                        command = 'proceed';
76                                        coordRule = 'StateCoordinationRule';
77                                                      };
78
79      during:
80        on (IBaseDirectedLB.e_EndedLB) -> Idle;
81        on(ICompDirectedLB.e_LockRequestByAS){
82           print ">>>>>> During Run>>>>> LB >>> insert AS in queue";
83           pending_requests.add(ICompDirectedLB.e_LockRequestByAS);
84           send ICompDirectedLB.e_LockReplyByLB = new LockReply();
85        }
86
87      exit:
88        print ">>>>>> During Run>>>>> LB >>> remove LB queue";
89        pending_requests.remove ();
90        send ICompDirectedLB.e_LockReleaseLB = new LockRelease();
91    }
92  }
93
94  private interface IBaseDirectedLB for LoadBalancerCoord{
95    requires {
96      ConstituentState e_StartedLB_UnderUtilized = {E | E instanceof
97                                                    'ConstituentState'
98                                  && E.applicationState == 'StartExecuting'
99                                  && E.serverLoad == 'UnderUtilized'}
100
101     ConstituentState e_EndedLB = {E | E instanceof 'ConstituentState'
102                                  && E.applicationState == 'EndExecuting'}
```

```
103
104     ApplicationTerminated e_TerminatedLB = { E | E instanceof
105                                                 'ApplicationTerminated'}
106     ApplicationStarted e_InitiatedLB = { E | E instanceof
107                                                   'ApplicationStarted'}
108 }
109
110   provides {
111     CoordinationCommand e_LB_CoordinationCMD;
112   }
113 }
114
115 interface ICompDirectedLB for LoadBalancerCoord{
116   requires {
117     LockRequest e_LockRequestByAS = { E | E instanceof 'LockRequest'}
118     LockRelease e_LockReleaseByAS = { E | E instanceof 'LockRelease'}
119     LockReply e_LockReplyByAS =     { E | E instanceof 'LockReply'}
120   }
121
122   provides {
123     LockRequest e_LockRequestByLB;
124     LockReply e_LockReplyByLB;
125     LockRelease e_LockReleaseByLB;
126   }
127 }
```

Listing 8.4: Specification of the LoadBalancers "State"-coordinator

```
1 private interface IStateCoordinatorLB for WrappedLoadBalancer{
2   requires {
3     CoordinationCommand e_CoordinationCMD = {E | E instanceof
4                                                 'CoordinationCommand'
5                             && E.coordRule == 'StateCoordinationRule'}
6   }
7   provides {
8     ConstituentState e_StartedLB :=
9                                     before execution ( void
10            org.dummy.loadbalancer.MigrationPlanner.plan(String load))
11                                    if (load =='UnderUtilized') {
12                                       serverLoad = load;
13                                       applicationState =
14                                                   'StartExecuting';
15                                    }
16
17    wait when (e_StartedLB) until (e_CoordinationCMD){
18      switch (e_CoordinationCMD.command){
19        case 'proceed': proceed;
20        case 'suspend': suspend;
21      }
22    }
23
24    ConstituentState e_EndedLB := after execution (void
25                          org.dummy.loadbalancer.Migrator.migrate(..)){
```

```
26                                  applicationState = 'EndExecuting';
27                                }
28    }
29 }
```

Listing 8.5: Specication of LoadBalancers inner role that is responsible for connecting it to the coordinator of the "State"-coordination rule

8.2 Internal Design of EventArch 2.0

This section gives a textual description of the internal design of EventArch 2.0 and a graphical overview of the defined packages and their relations to each other. This description is given on a rather abstract level. An overview of the internal design of EventArch 2.0 is presented. It is an extended version of the section of the thesis that is concerned with the implementation of EventArch 2.0. In the following, a more detailed description is given. The design of each package is presented in the form of a UML class-diagram. The description is intended to provide background knowledge for future extension of the language.

8.2.1 Abstract

This subsection gives a textual description and a graphical overview of the internal design of EventArch 2.0. The purpose of each major package is explained shortly.

Design-Description
The internal design of the language can be divided in a compilation-part and a runtime-part. The overall task of the compilation part is to read an architectural specification that was written by the architectural designer in the EventArch 2.0 language (short: "EventArch 2.0 specification") and to translate it to certain target-languages (Java, C++, AspectJ, AspectC++). Several files are created from that specification, each in a specific target language.

The overall task of the runtime-part is to implement the representation of the to-be-coordinated applications on the architecural-level and to provide an environment that allows for integrating them with each other. Integration is achieved according to the architectural setup that is defined in an EventArch 2.0-specification. The runtime-part can be further devided into classes that have been generated from the EventArch 2.0 specification and into classes that constitute the runtime-environment of the language. Important to note here is, that not every Architectural Event Module (AEM, see section 2.2.1) is wrapping legacy code. The architectural designer can define EventArch-statemachines in the specification. Those are translated into Java-code and can be used at runtime. In case of statemachines, the AEMs are wrapping generated applications. For further details on this topic, consult the paragraphs about the packages of the runtime-part in this section.

As described in section 2.2.2 of the thesis, incoming events are selected by "Selectors". Every interface-action (e.g. method invocation and wait-when-block) is associated to a specific selector. Selector-design is a crucial part of defining an EventArch 2.0 specification.

Another language-feature is the "wait-when-block". A defined wait-when-block causes the wrapped application to suspend operation after having published a certain event. Execution can be proceeded if a specific event has been selected. The wait-when-block commands the wrapped application to wait for a specific event after having sent a specific event. It is a means to induce tight-coupling between AEMs.

In EventArch 2.0, AEM-instances can be encapsulated into "instance-groups". An instance-group is a common scope for the contained AEM-instances. Instance-groups can be the target of an event-transmission. The implementation of instance-groups relies on JMS-topics as well. Instance-groups can be nested.

Composite AEMs are implemented as topics (see 2.2.2). To associate each event to a certain scope, EventArch 2.0 introduced a standard-attribute "targetGroup". Every event is published to the scope that is indicated by its targetGroup-attribute.

Graphical Overview
The package-diagrams that have been presented in section 2.2.2 are reprinted here for the sake of completeness.

Compilation-part and runtime-part are briefly described in the following. The output of parsing a specification is an "object-tree". That object-tree represents the specification at runtime and contains all information that is relevant for code-generation. As the object-tree was generated by an Xtext-generated parser, it has to be considered Xtext-dependent and is therefore likely to be subject to future change. To become independent from future Xtext-changes, this object-tree is wrapped by Wrapper-classes that are contained in the applevel-package (package A). The wrapping is performed by the core-package (package 3). The wrapped object-tree is used at runtime from numerous other packages. It is encapsulated in a container ("DataCatalogue") that is defined in the elements-package (package C). It is persisted after compilation and loaded at runtime by the Serializer-package (package 7). The Xtext-dependent object-tree is used in the generator-package (package 1) to generate the target code (Java, C++, AspectJ, AspectC++). The code-generation maps the architectural setup that was defined in the EventArch-specification to application-code (wrappers, event-definitions, statemachine-definitions, aspect-code). This code is written in one of the target-languages (Java, C++, AspectJ, AspectC++). The generated code employs the EventArch-runtime environment to implement the desired behaviour. The target code is compiled to executable code by the parser-package (package 5). To do this, the language-processors of the respective target languages are employed. The whole compilation-process is controlled by the compiler-package (package 6).

The runtime-behaviour is implemented mainly by the applevel-package (package A) and the generated classes (package B). Generated aspects are concerned with trig-

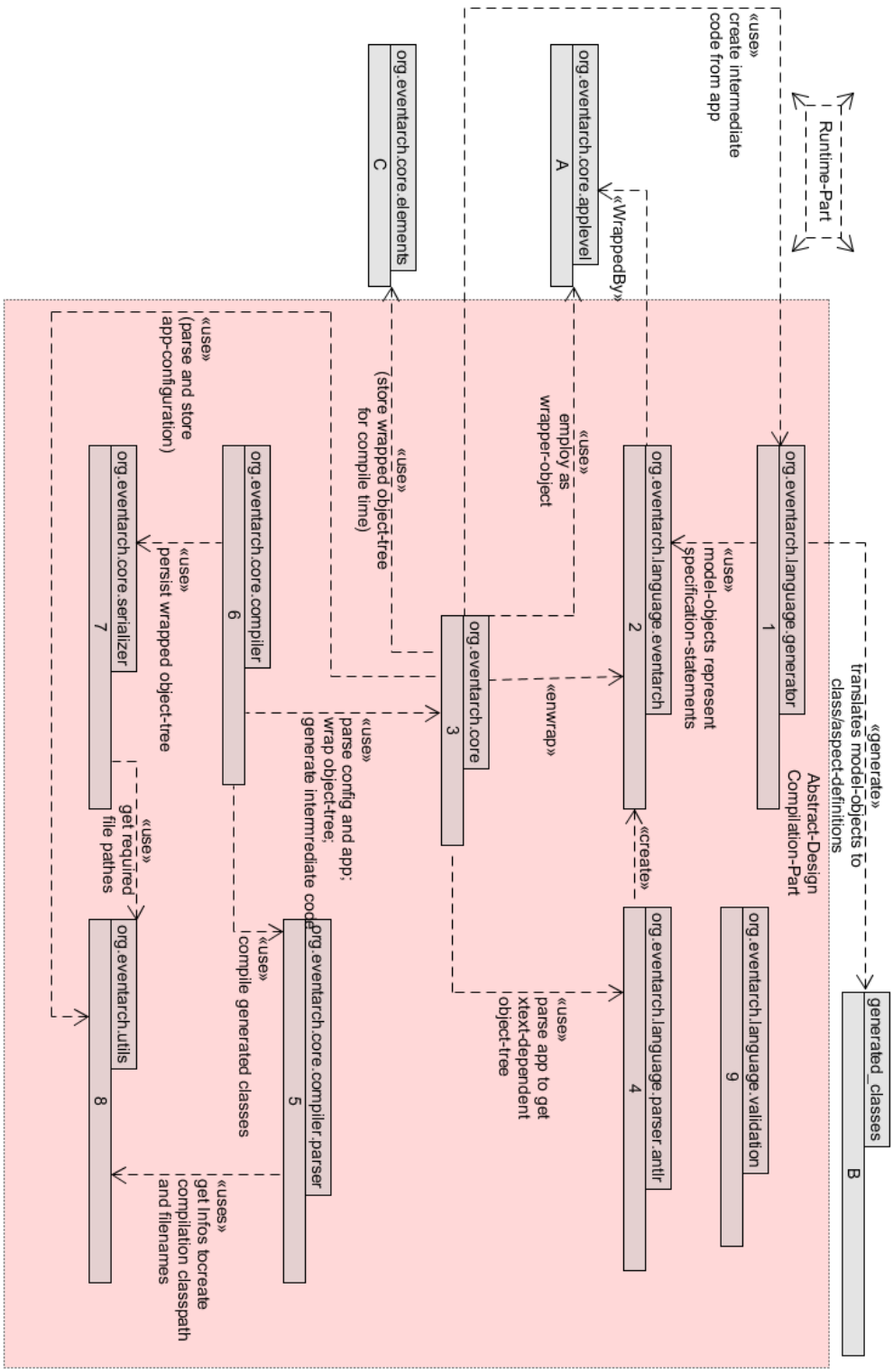

Figure 8.1: Package diagram of the compilation-part of the EventArch 2.0-implementation

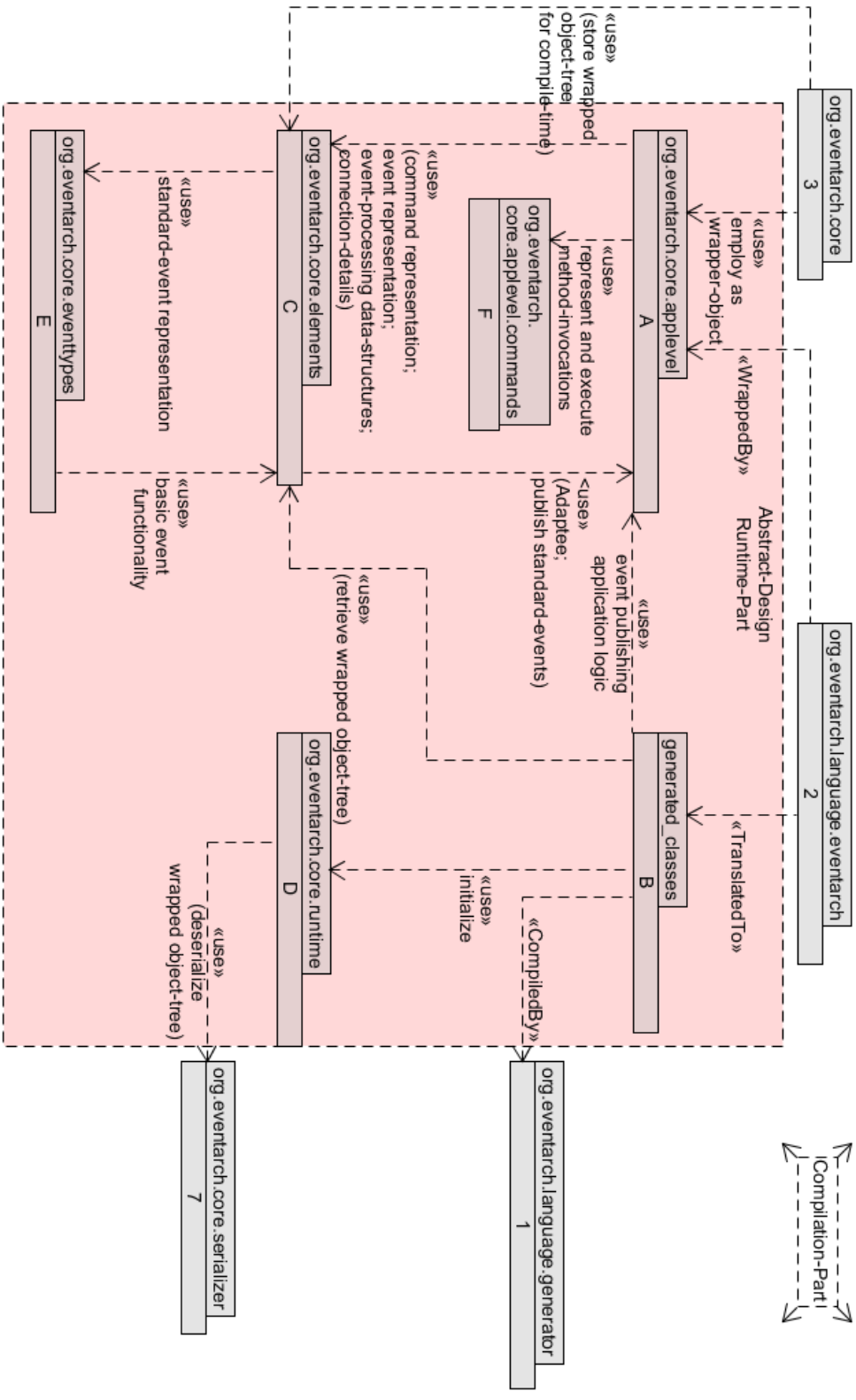

Figure 8.2: Package diagram of the runtime-part of the EventArch 2.0-implementation

gering event-publishing. The employed pointcuts are triggered (as defined by the EventArch-programmer) at certain "state-changes of interest" (function calls) of the wrapped application. The generated statemachines implement the state-based behaviour that was defined by the architectural designer in the EventArch-specification. Moreover, the applevel package implements general communication services like event-reception and JMS-interfacing. For details see the respective sections in the "detailed design"-subsection of this section and the following sequence-diagrams.

Sequence-Diagram The sequence diagram in figure 8.3 describes the sequence of actions for the compilation process on package-level. As part of abstract design, there are no concrete method-names used in the figure. Instead, the purpose of each action is described. The sequence diagram of the compilation part presents the compilation-process. The actions have been described in the "Graphical Overview"-part of this subsection.

Figure 8.4 indicates the sequence of actions that are typically performed at runtime. At AEM-startup, the wrapped object-tree has to be loaded from hard disk. It has been saved to hard disk at the end of the compilation process. The Serializer-package is also responsible for this deserialization of the wrapped object-tree. Thereafter, the unwrapped object-tree has to be recreated again, as it could not be persisted at the end of the compilation process. The references between the wrapped object-tree and the unwrapped object-tree also have to be recreated. In the following, the JMS-connection is set up by classes of the applevel-package. Having JMS-connectivity established, the applevel-package and the generated classes are responsible for event-processing. The approach depends on whether the AEM wraps an applicatioin that was written in EventArchs 2.0 statemachine-language, or whether the AEM wraps legacy-code. For details see the generator/generated-classes description in subsection 8.2.2.

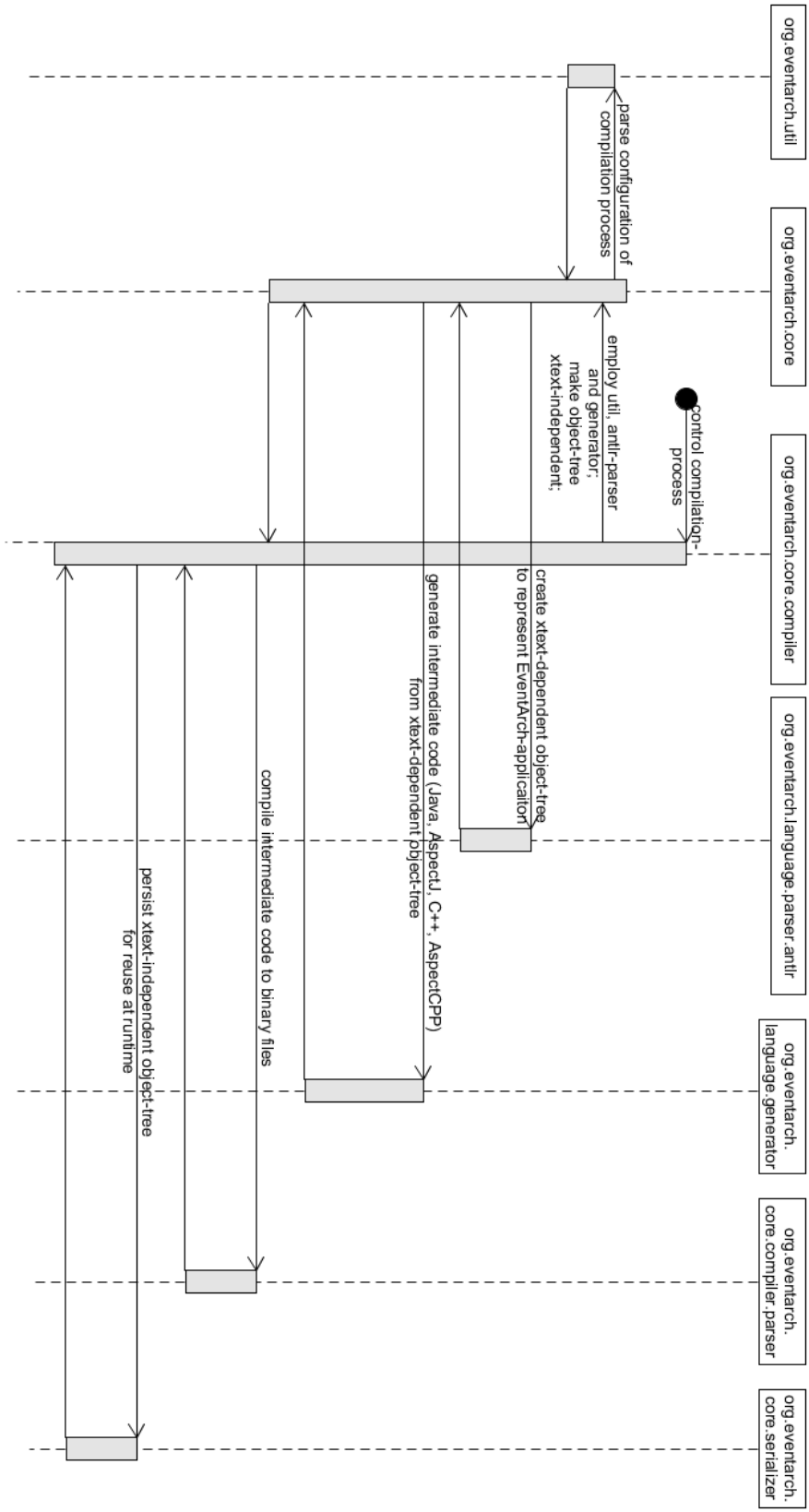

Figure 8.3: Sequence diagram of the compilation-part of the EventArch 2.0-implementation

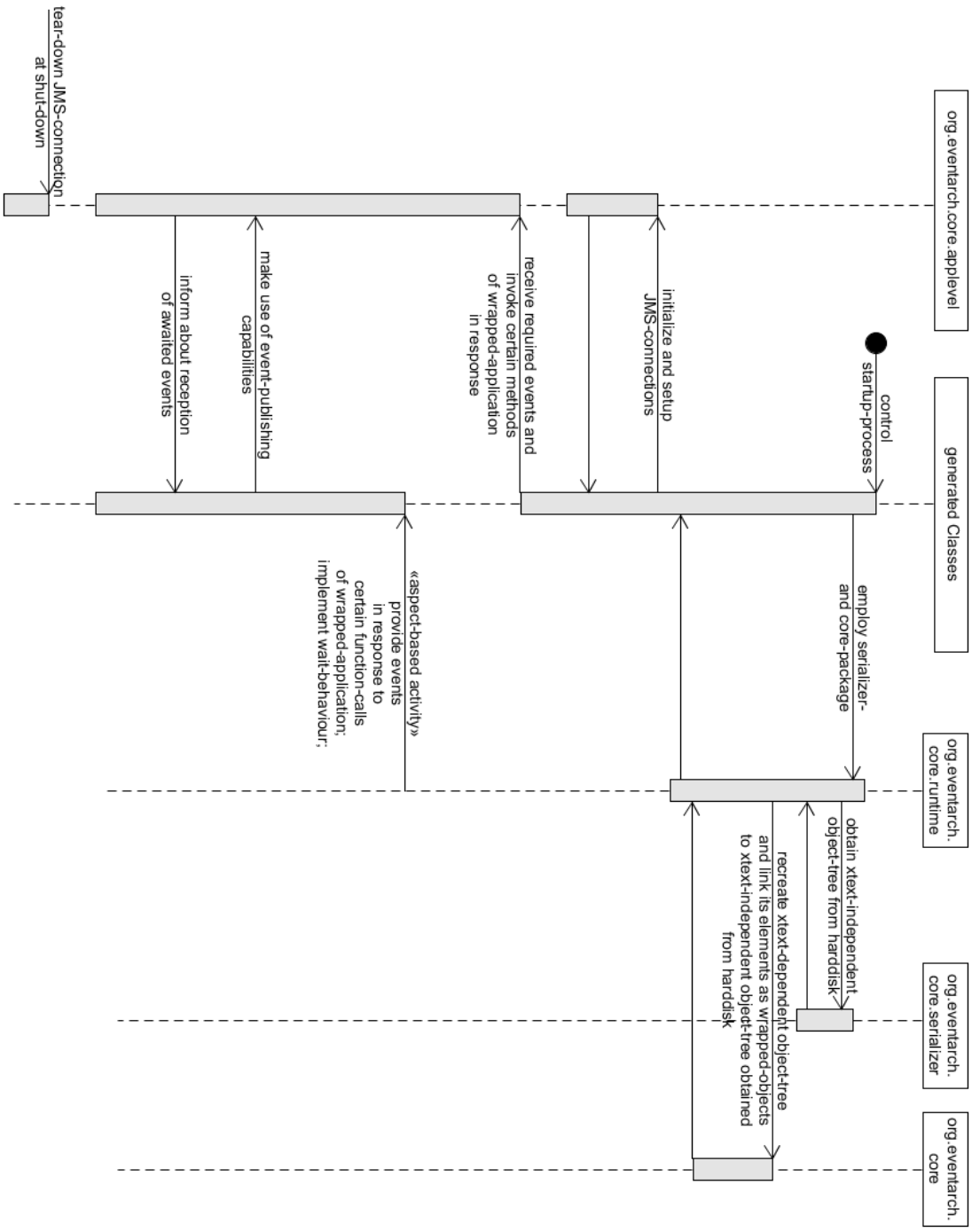

Figure 8.4: Sequence diagram of the runtime-part of the EventArch 2.0-implementation

8.2.2 Detailed

This subsection describes major packages of the internal design of EventArch 2.0 in further detail. The design of each package is presented in the form of a UML class-diagram. The diagrams contain all classes of the respective package with important functions, plus those classes of other packages that serve as major collaboration partner. This description will prove to be useful for possible future extensions of the language.

org.eventarch.core

Diagram: 8.5.
This package consists of a single class, the Loader. It is responsible for wrapping the Xtext-generated object-tree (therefore: Xtext-*dependent*). This object-tree contains all information of the EventArch 2.0-specification that is to be compiled. The result of the wrapping-operation is an Xtext-independent object-tree. The loader employs classes of the *applevel-package* as Wrapper-classes. Several other packages are employed to achieve the following tasks:

- *utils-package:* parse configuration file
  - contains pathes to the to-be-coordinated applications
  - contains output-pathes of the code that was generated from the EventArch 2.0 specification
- *antlr-package:* parse EventArch 2.0 specification (antlr-parser generated by Xtext-framework)
- *generator-package:* translate parsed specification (object-tree) to target languages (Java, AspectJ, C++, AspectC++)

The loader is called by the *compiler-package* and finally returns control to it.

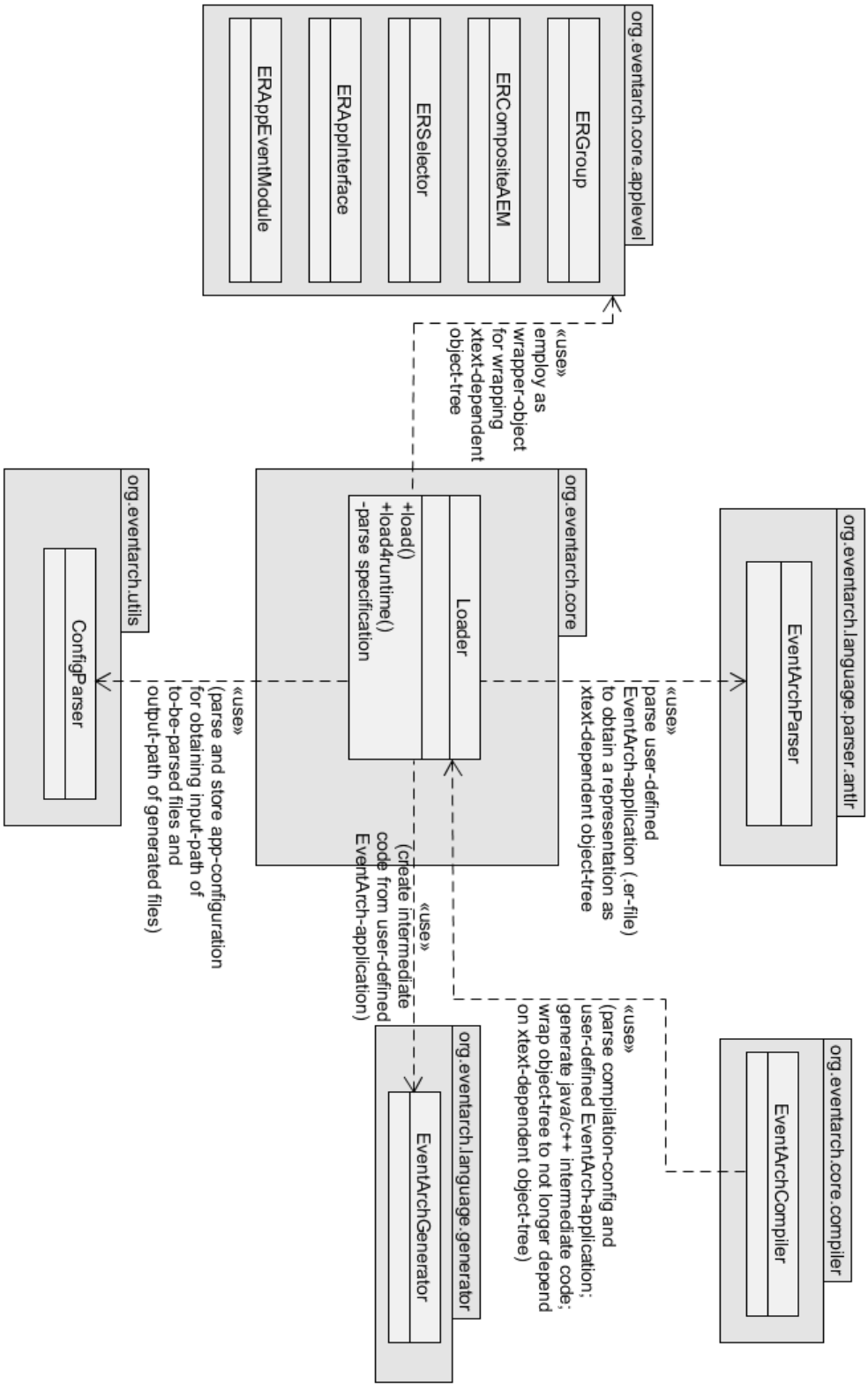

Figure 8.5: Class-diagram of EventArchs 2.0 core-package

org.eventarch.core.compiler

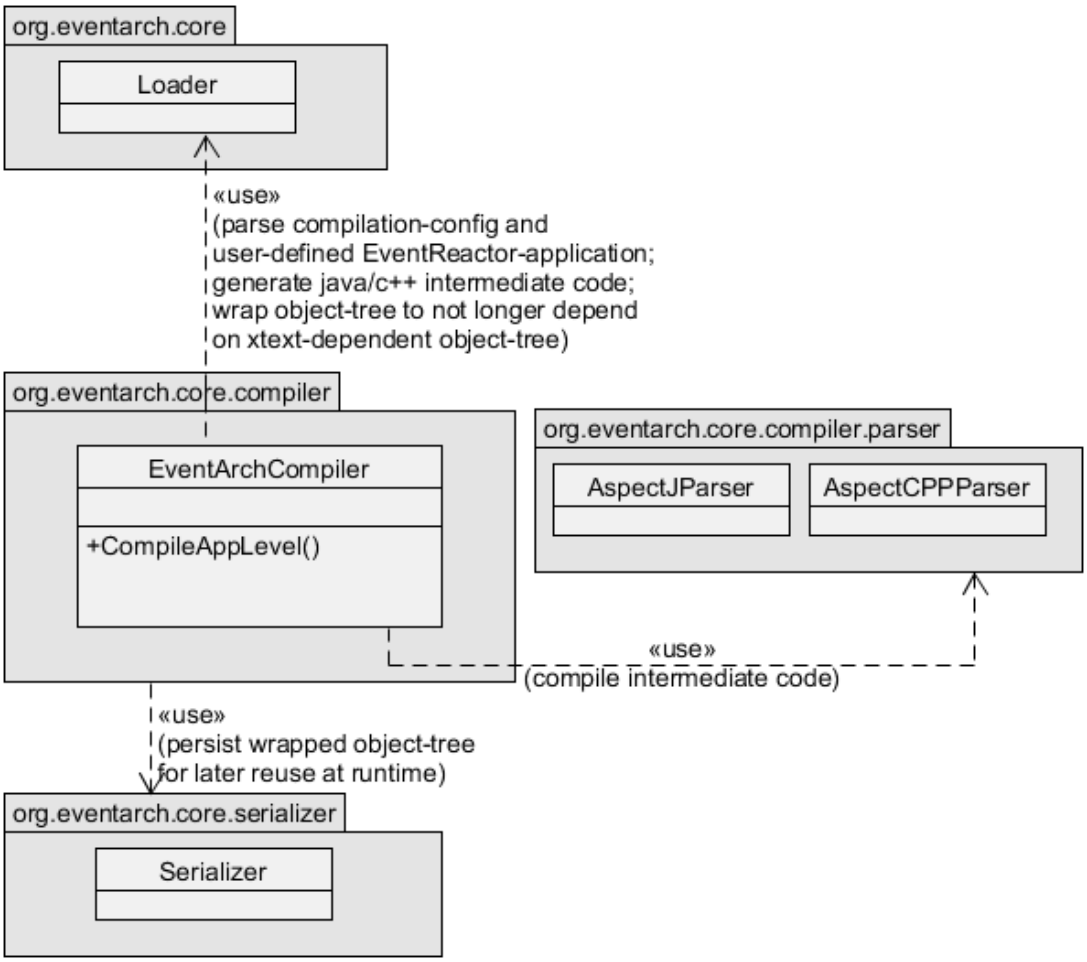

Figure 8.6: Class-diagram of EventArchs 2.0 compiler-package

Diagram: 8.6
The compiler-package consists of a single class, EventArchCompiler. It controls the compilation process by delegating certain tasks to other packages:

- *Loader-package:* translate EventArch 2.0 specification to target languages (Java, AspectJ, C++, AspectC++) and wrap object-tree (see above)
- *parser-package:* translate the target-code into executable-code by employing the respective compilers
- *serializer-package:* persist wrapped object-tree for later use

org.eventarch.core.applevel

Diagram: 8.7
The applevel-package implements all standard-functionality of an Architectural Event Module (AEM). This functionality is mainly encapsulated in the classes *ERGeneratedAppEventModule*, *ERGeneratedAppEventModule*, and *ERAppInterface*. The standard-functionality is used by the generated classes, i.e., the code that has been generated from the EventArch 2.0 specification. The following functionality is provided:

- receive incoming events and determine the matching selectors of the respective Primitive Interfaces (*ERGeneratedAppEventmodule.getMatchingSelectors()*)
- employ *commands-package* to invoke methods of the wrapped applications (*ERGeneratedAppEventModule.addInvocation()*)
- employ generated statemachine to perform event-processing according to the EventArch 2.0 specification
- publish events according to their targetGroup-attribute (*ERAppEventModule.publish()*)
- inform wrapped application if an event has arrived, that it is currently waiting for (wait-when-block) (*ERGeneratedAppEventModule.getCommand()*)
- provide interfacing to communication-platform (JMS, see chapter 2.2.2; *ERGeneratedAppEventModule.init()*)

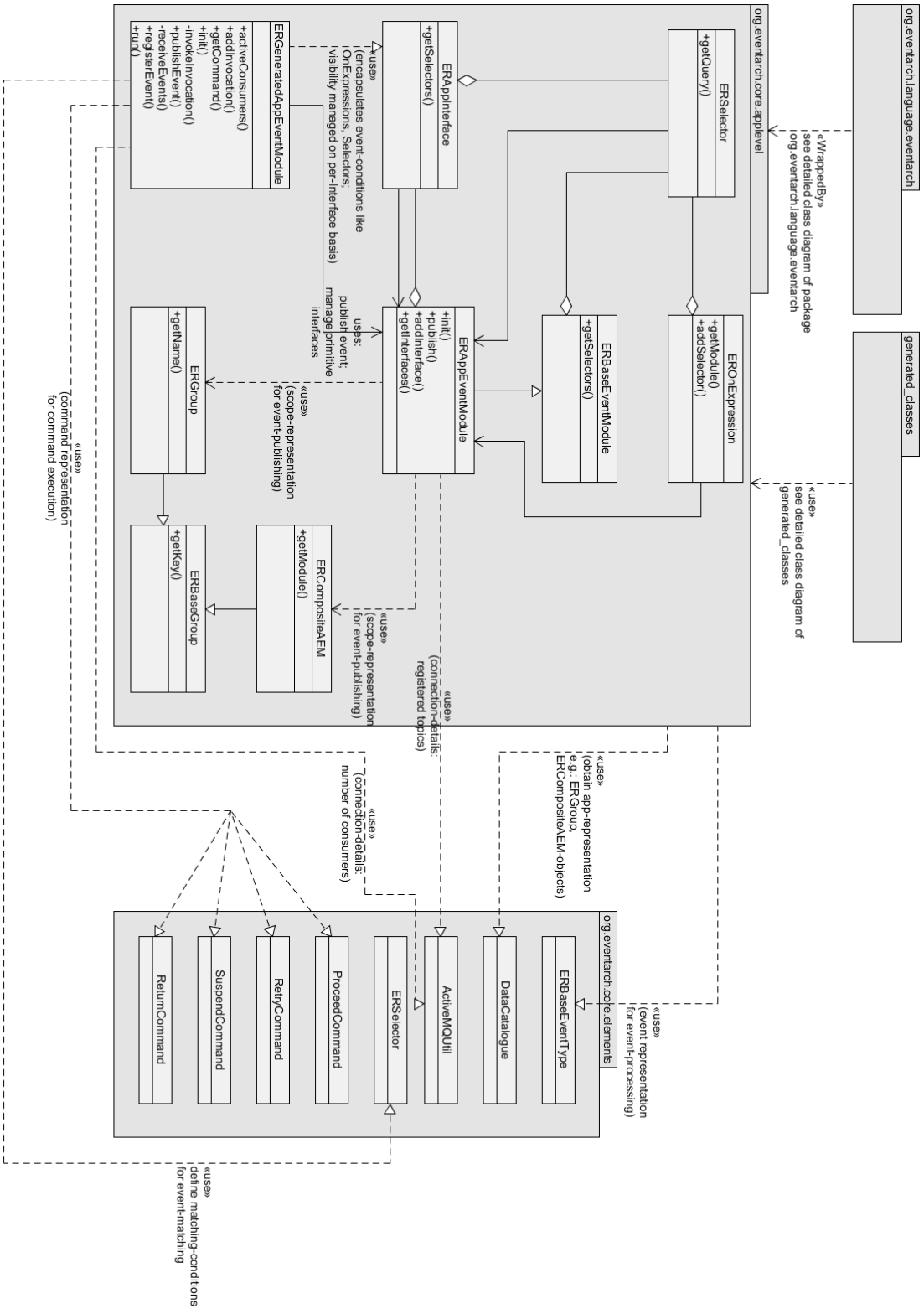

Figure 8.7: Class-diagram of EventArchs 2.0 applevel-package

generator

Diagram: 8.8

The generator is responsible for generating classes that represent types and instances of Primitive AEMs that were defined in the EventArch-specification. The following classes/aspects are generated (curly-braces represent names that the generator extracts from the specification):

- **{EventModuleName}**: class represents the type of a Primitive AEM. This class inherits most of its functionality from the class ERGeneratedAppEventModule (applevel-package)
- **{EventModuleName}** _ **{InstanceName}**: class represents a Primitive AEM-instance of a specific type. It is responsible for attaching the instance to an instance-group (see 8.2.1) and triggering initialization and start of the Primitive AEM
- **{EventModuleName}StateMachine**: class implements an EventArch-statemachine. It is responsible for determining and executing state-dependent actions
- **{EventModuleName}{InterfaceName}Interface**: aspect implements the wrapped legacy-applications event-publishing which is done in response to calls on certain methods of the legacy-code
- **{EventName}**: class implements a user-defined event

The generator performs code-generation as a mapping of the object-tree (represents the EventArch-specification) to source-code. More specifically, specific model-objects of the object-tree are mapped to specific classes, functions, aspects of the target language(Java or C++). To generate code, the object-tree is traversed and the code-blocks that were obtained from translating specific model-objects into code are combined. Specific functions of the generator are responsible for translating specific types of model-objects. The call-graph of the functions resembles the hierarchy of the model-objects in the object-tree. There are specific functions to translate the following model-objects:

- user-defined events (*compileEventTypeJava()*, *compileEventTypeCPP()*)
- Primitive Interfaces (*compilePrimitiveInterfaceJava, compilePrimitiveInterfaceCPP*)
- Primitive AEMs (*compilePrimitiveAEMJava, compilePrimitiveAEMCPP*)
  - with EventArch 2.0-statemachine (*compileJavaStateMachine()*)
- Composite AEMs (*generateElementsCompositeModuleAssociation()*)

The following functions are concerned with generating code for

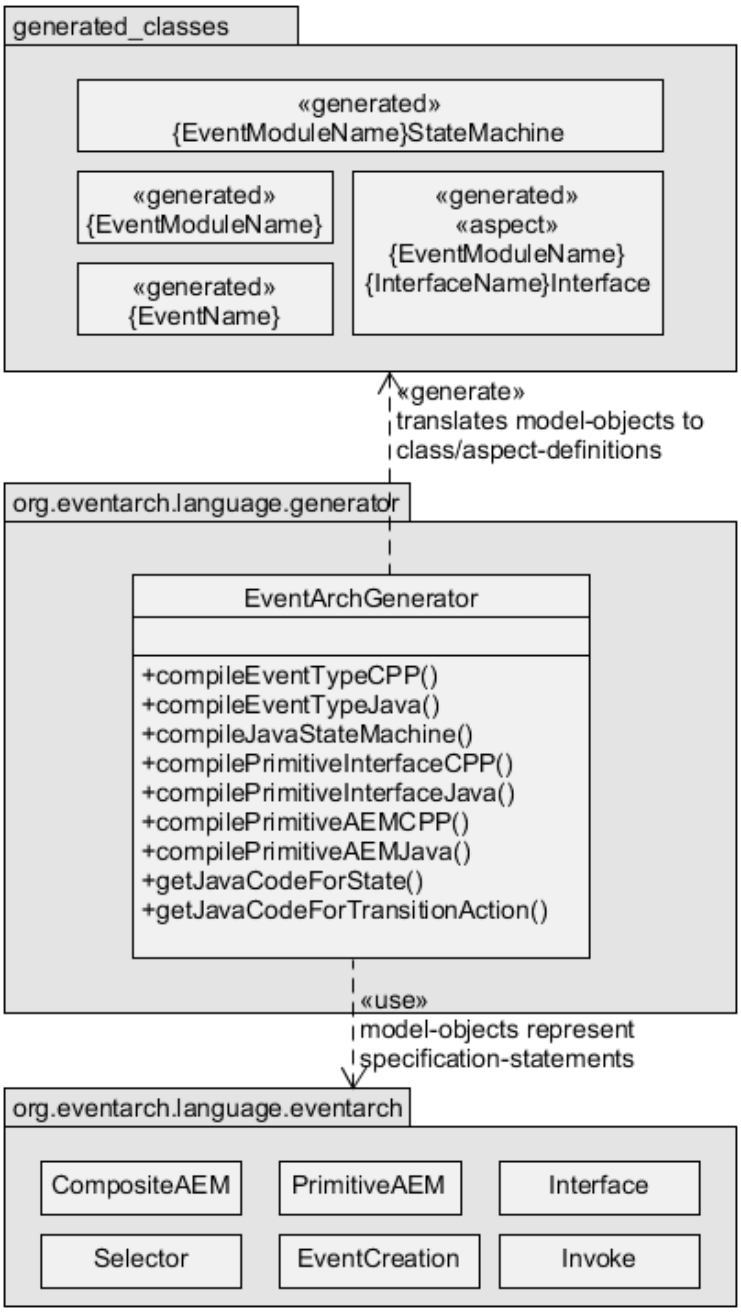

Figure 8.8: Class-diagram of EventArchs 2.0 generator-package

EventArch 2.0-statemachines.

- *compileJavaStateMachine()*: structure the code, call the other functions
- *getJavaCodeForState()*: generate code to map a matched selector to a transition action for a specific state
- *getJavaCodeForTransitionAction()*: generate code to perform a transition action, e.g., "send an event", "call a function in the wrapped system"

generated classes

Diagram: 8.9

The generated classes are concerned with processing received events and publishing new events. To perform both activities, the runtime-environment is used. Events to be sent are created in the interface-aspects or the statemachine-class. They are sent using the runtime-environment, especially the applevel-package. Sending events is necessary in response to a triggered pointcut or in response to a received event (statemachine-action). Events are received by the runtime-environment (applevel-package) and either passed to the generated code for further processing (statemachines) or translated into method-invocations (legacy-applications). Method-invocations are implemented using the respective languages reflection-features. The WaitOn-behaviour is implemented by the generated aspect: the execution is not returned to the wrapped legacy-application until the expected event has been received.

8.3 Grammar of EventArch 3.0

This subsection contains an extract of the grammar that defines the architecture description language "EventArch 3.0". According to this grammar

- roles are PrimitiveAEMs that have been marked by the keyword "role"
- Role-Binders are PrimitiveAEMs as well. They may be defined inside Compartments or in the module-section
- the binding- and creation commands are special "TransitionActions" (used inside statemachine)
- the atomic-block may just include binding- or creation commands
- roles can be bound at DCAEM-creation or by a subsequent "Bind"-command

```
1
2 ModuleSection:
3         {ModuleSection} 'modules' '{' (primitivemodules+=PrimitiveAEM |
              compositemodules+=CompositeAEM | roleBinders+=RoleBinder)
              *'}'
4 ;
```

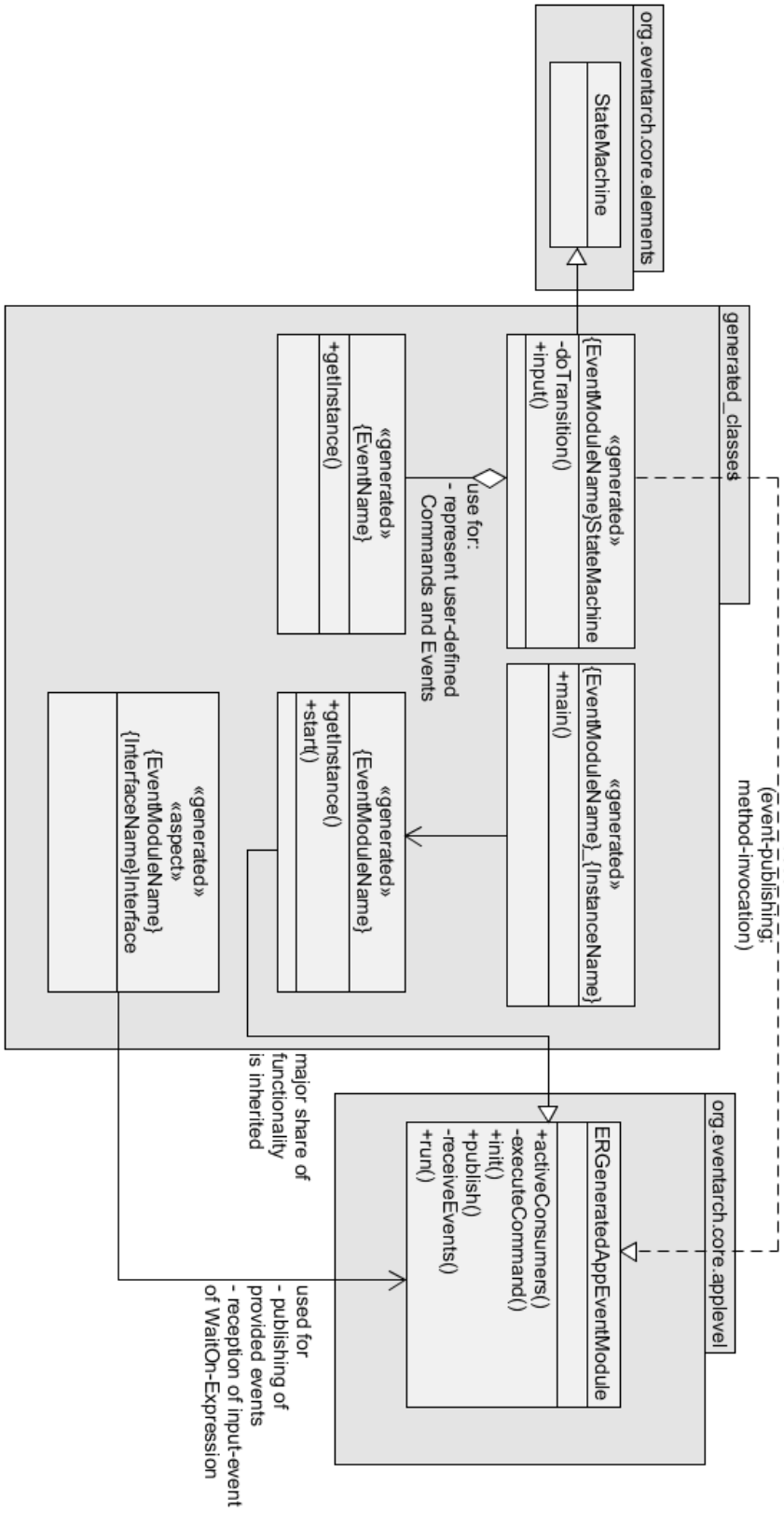

Figure 8.9: Class-diagram of EventArchs 2.0 generated classes

```
5
6 Compartment:
7         'compartment' name=ID '{' (roleAEMs += RoleAEM | interfaces+=
              Interface | roleBinders+=RoleBinder)*  '}'
8 ;
9
10 RoleAEM:
11         'role' (primitiveAEM=PrimitiveAEM)
12
13 ;
14
15 RoleBinder:
16         'roleBinder' primitiveAEM=PrimitiveAEM
17 ;
18
19 Transition:
20         'on' '(' (events+=[Selector| QualifiedName] ('||' events+=[
              Selector| QualifiedName])*)? ')' ('[' condition=Condition
              ']')? ('{' transitionActions+=TransitionAction* '}')? name
              ='->' nextState=[State] ';'
21 ;
22
23 TransitionAction:
24 (action=EventCreation /*| action=Send */| action=Invoke | action=Print |
       action=VarAssignment | action=EventSend | action=
      ConditionalTransition | action=DcaemAction  | action=AtomicBlock)
25 ;
26
27 AtomicBlock:
28         'atomic' '{' actions+=DcaemAction+ '}'
29 ;
30
31 DcaemAction:
32         action=Bind | action=Unbind |  action=CreateDcaem | action=
              DestroyDcaem
33 ;
34
35 Bind:
36         dcaem=ID '[' 'composite' ']' '+=' '{' (roleReference =
              RoleReference) '}'
37 ;
38
39 Unbind:
40         dcaem=ID '[' 'composite' ']' '-=' '{' (roleReference =
              RoleReference) '}'
41 ;
42
43 CreateDcaem:
44         dcaem=ID '[' 'composite' ']' ':=' '{' (roleReferences+=
              RoleReference (',' roleReferences+=RoleReference)*)? '}'
              '<->' '{' (baseReference=BaseReference) '}'
45 ;
```

```
46
47 BaseReference:
48         primitive=[PrimitiveAEM] '.' (('{' primitivesInterfaces+=[
               Interface] (',' primitivesInterfaces+=[Interface])* '}') |
               primitivesInterfaces+=[Interface])
49 ;
50
51 RoleReference:
52         (compartment=[Compartment] '.')? (role=[PrimitiveAEM]) '.' (
               rolesInterface=[Interface])
53 ;
54
55 DestroyDcaem:
56         'destroy' dcaem=ID '[' 'composite' ']'
57 ;
```

8.4 EventArch 3.0 Diagrams

This section presents several class-diagrams to describe the necessary changes to EventArch 2.0 in standard UML-notation. The following class diagrams just contain functions that have been added or severely modified in EventArch 3.0. One sequence-diagram is contained in that subsection. It is concerned with the process of respecting scope-restrictions when publishing an event.

org.eventarch.core.applevel
class-diagram: figure 8.10
As indicated by the package-diagram 4.4, the applevel-package got several new or changed dependencies:

- It is used by the generated classes (see figure 8.12) to determine the current DCAEM-association of a Primitive Interface
  - ERAppEventModule.*getInterfaceByName()*
  - ERAppInterface.*isLocalToDcaem()*
- It is used by the generated classes to publish the new built-in events
  - ERGeneratedAppEventModule.*publishBindingEvent()*
  - ERGeneratedAppEventModule.*publishCreationEvent()*
- To receive special-events (bind/unbind, create/destroy), the applevel-package relies on the standard-interfaces provided by the core-package
  - ERGeneratedAppEventModule.*getMatchingSelectors()*

Several functions have been added to the applevel-packages, especially to the class ERGEneratedAppEventModule:

- these functions are concerned with processing the received bind-/creation-events:

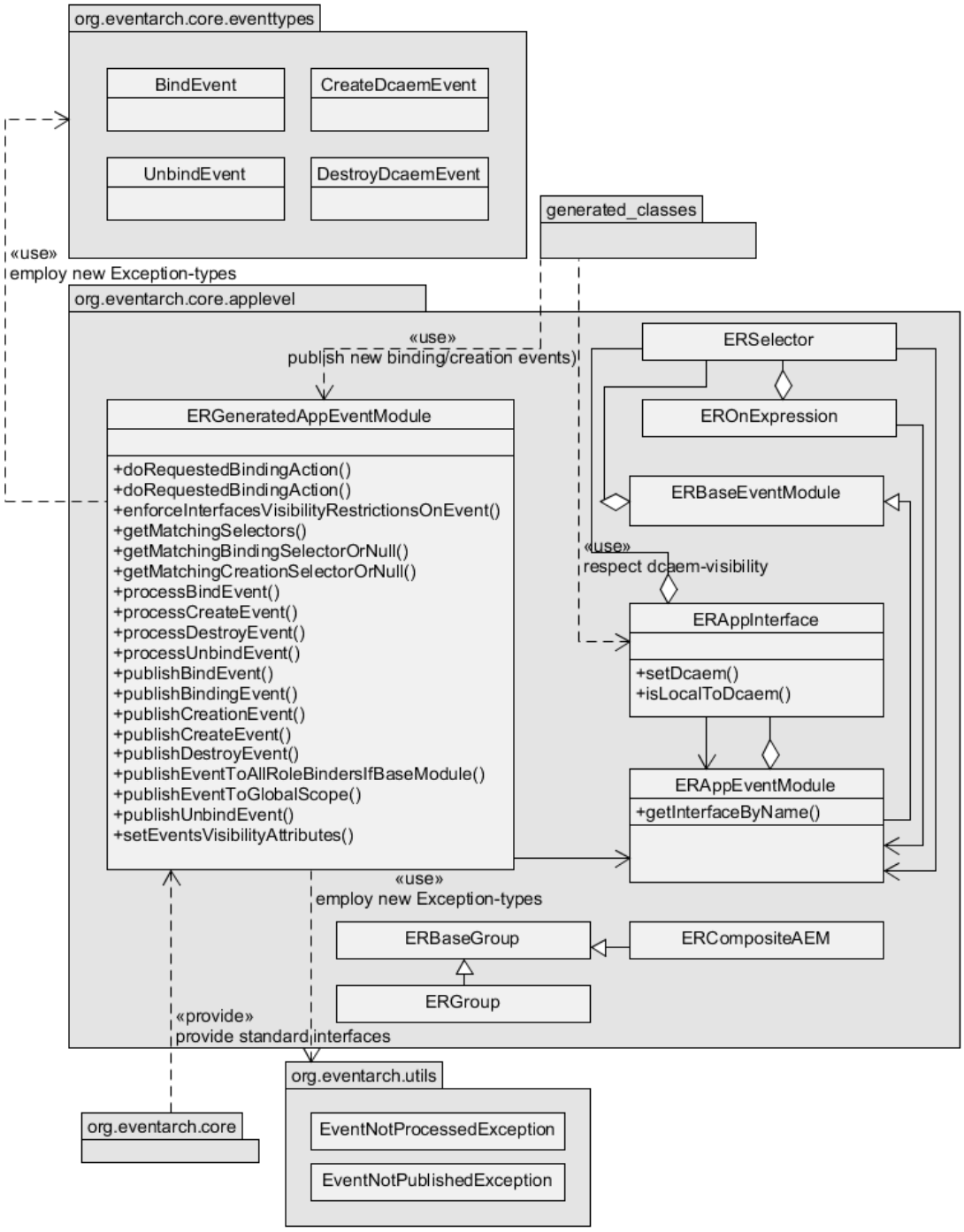

Figure 8.10: Internal changes to applevel-package

  - *getMatchingBindingSelectorOrNull()* ,
  - *doRequestedBindingAction()*
  - *processBindEvent()*
  - *processUnbindEvent()*
  - *getMatchingCreationSelectorOrNull()*
  - *doRequestedCreationAction()*
  - *processCreateEvent()*
  - *processDestroyEvent()*
- many functions are concerned with publishing the new special events or with publishing events to the new standard scopes:
  - *publishBindEvent()*
  - *publishUnbindEvent()*
  - *publishBindingEvent()*
  - *publishCreateEvent()*
  - *publishDestroyEvent()*
  - *publishCreationEvent()*
  - *publishEventToAllRoleBindersIfBaseModule()*
  - *publishEventToGlobalScope()*
- many of the added functions of the applevel-package are concerned with managing- or determining the visibility of events and the visibility-restrictions imposed by their Primitive Interfaces:
  - ERGeneratedAppEventModule.*enforceInterfacesVisibilityRestrictionsOnEvent()*
  - ERAppInterface.*setDcaem()*
  - ERAppInterface.*isLocalToDcaem()*
  - ERGeneratedAppEventModule.*setEventsVisibilityAttributes()*

org.eventarch.core
class-diagram: figure 8.11
The core-package experienced the following changes:

- an enumeration was introduced to mark a module as role-module, base-module, or Role-Binder module:
  - TypeOfModuleMarker
- the Loader associates the module with the respective marker:
  - *extractPrimitiveAEMsFromCompartment()*
  - *extractPrimitiveAEMsFromModuleSection()*

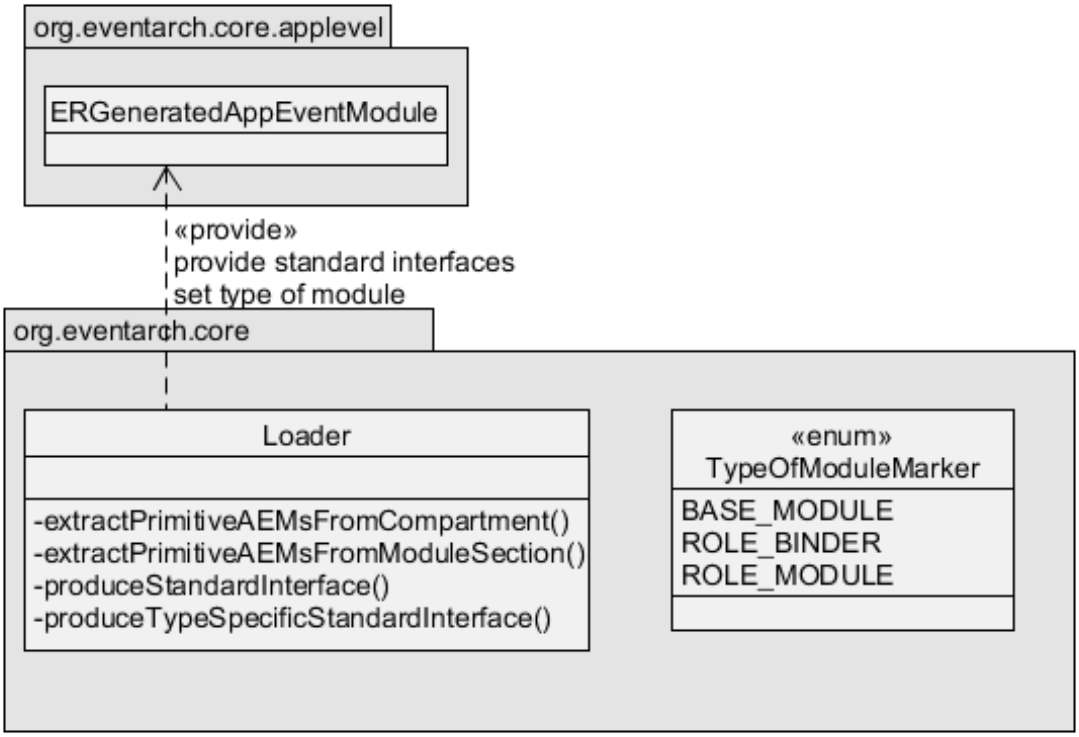

Figure 8.11: Internal changes to core-package

- the Loader creates a standard interface for the module. The type of which depends on the marker:
  - *produceTypeSpecificStandardInterface()*
  - *produceStandardInterface()*

Generator and generated classes
class-diagrams: 8.13 *and* 8.12

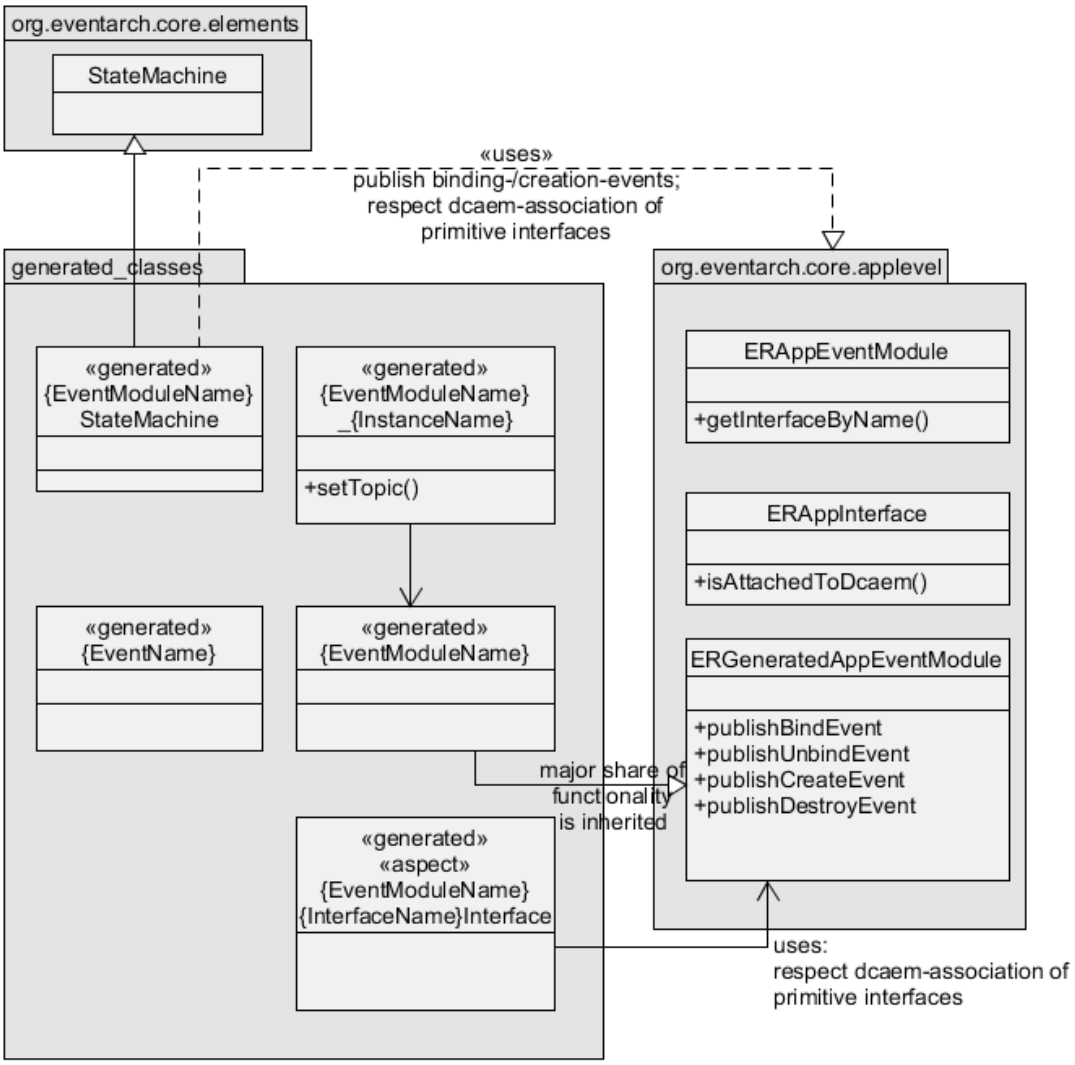

Figure 8.12: Internal changes generated-classes

To implement EventArch 3.0, the generator also had to be changed:

- the generator can handle the new binding- and creation-statements. Relevant

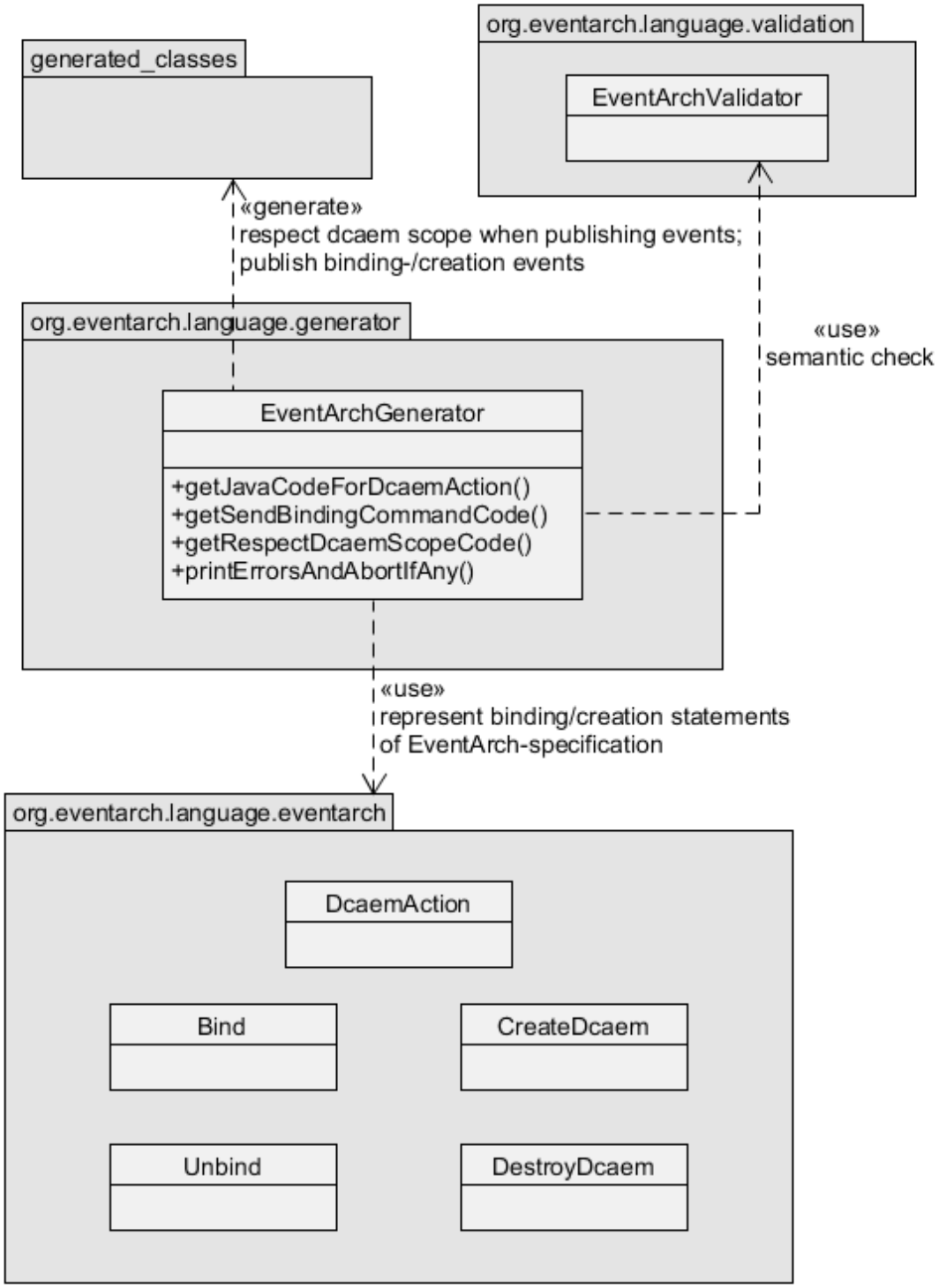

Figure 8.13: Internal changes generator-package

functions:

  - *getJavaCodeForDcaemAction()*
  - *getSendBindingCommandCode()*

- the generator is aware of the scope-restrictions that are imposed on a Primitive Interface by its current DCAEM-association. *Changed behaviour:* event-publishing in statemachine; event-publishing on triggered pointcut. Relevant function:
  - *getRespectDcaemScopeCode()*
- the generator employs the validator for a semantic check and stops if errors are found. Relevant function:
  - *printErrorsAndAbortIfAny()*

Consequently, the generated classes have experienced some property- and behavioural changes:

- those generated classes that represent module-instances (class-diagram: {EventModuleName} _ {InstanceName}) that are contained in a Compartment, have built-in access to the scope of this Compartment. Responsible function:
  - *setTopic()*
- generated statemachines that are used as Role-Binder employ the applevel-package to send binding- and creation-events. Relevant class:
  - {EventModuleName}StateMachine
- generated statemachines employ the applevel-package to check the DCAEM-association of a Primitive Interface. Relevant class:
  - {EventModuleName}StateMachine

Sequence: respect DCAEM scope
The sequence diagram 8.14 describes the sequence of actions to check the DCAEM-association of a Primitive Interface before event-publishing. These actions are performed whenever a user-defined statemachine creates and publishes an event. The depicted sequence of actions constitutes a dependency between the generated classes-package (class {EventModuleName}StateMachine) and the language-core (applevel-package). This in part implements the loose coupling of base and roles: neither roles nor bases have to determine the receiver of the event specifically. Instead, the events "scope" -attribute (determines destination-scope of the event) is set automatically to the name of the common scope (DCAEM) of base and roles.

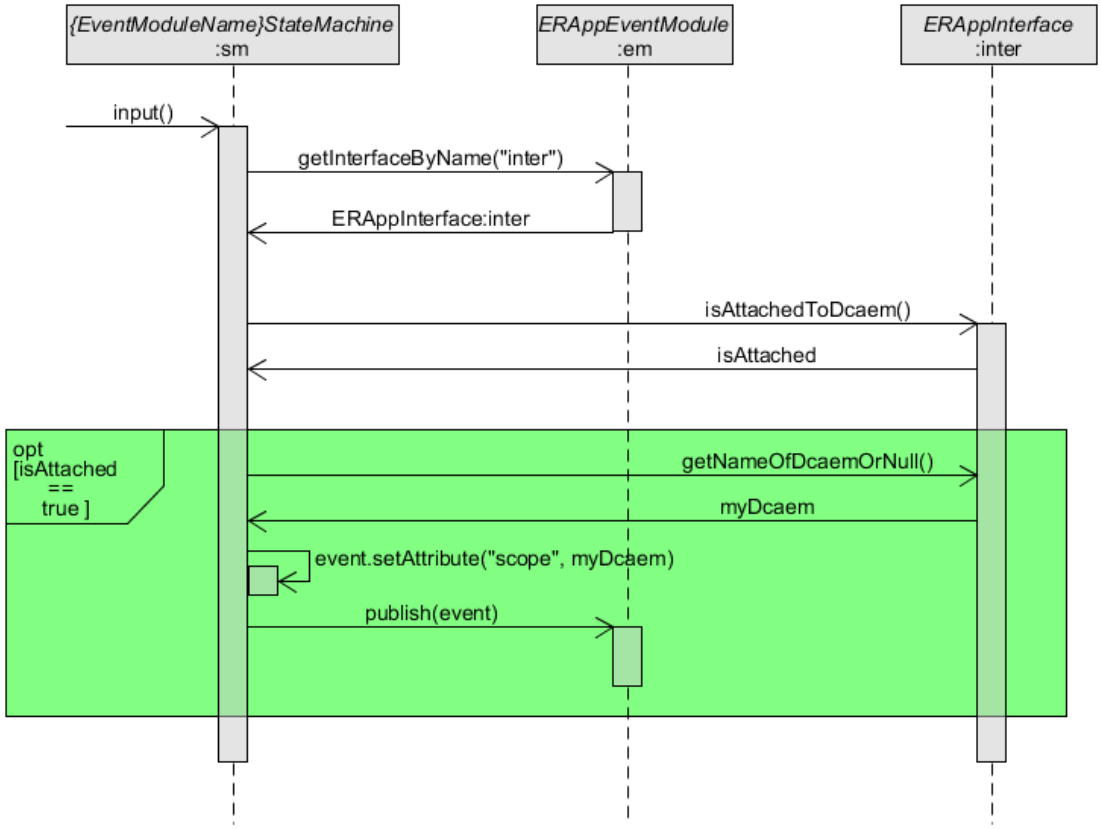

Figure 8.14: Sequence-Diagram: Respect DCAEM scope

Bibliography

[1] *JSR 343: JavaTM Message Service 2.0.*

`https://jcp.org/en/jsr/detail?id=343`

[2] *RFC3986. Uniform Resource Identifier (URI): Generic Syntax.*

`https://tools.ietf.org/html/rfc3986`

[3] *SCROLL role-binder feature.* https://github.com/max-leuthaeuser/SCROLL/wiki/Implementation-Role-Feature-6.

[4] *XText web-presentation.*

`https://eclipse.org/Xtext/`

[5] Datu Buyung Agusdinata and Daniel DeLaurentis. Specication of system-of-systems for policymaking in the energy sector. *IAJ - The Integrated Assessment Journal*, 8(2):1–24, 2008.

[6] Matteo Baldoni, Guido Boella, and Leendert van der Torre. powerjava: Ontologically founded roles in object oriented programming languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1414–1418, New York, NY, USA, 2006. ACM.

[7] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. *ECOOP 2007 – Object-Oriented Programming: 21st European Conference, Berlin, Germany, July 30 - August 3, 2007. Proceedings*, chapter A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships, pages 323–346. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[8] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 1–6, New York, NY, USA, 1989. ACM.

[9] John Boardman and Brian Sauser. System of systems the meaning of of. In *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*, 03 2006.

[10] Radu Calinescu and Marta Kwiatkowska. *Foundations of Computer Software. Future Trends and Techniques for Development: 15th Monterey Workshop 2008, Budapest, Hungary, September 24-26, 2008, Revised Selected Papers*, chapter Software Engineering Techniques for the Development of Systems of Systems, pages 59–82. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[11] Ward Cunningham and Kent Beck. A diagram for object-oriented programs. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 361–367, New York, NY, USA, 1986. ACM.

[12] Daniel DeLaurentis. *43rd AIAA Aerospace Sciences Meeting and Exhibit*, chapter Understanding Transportation as a System-of-Systems Design Problem, pages 323–346. American Institute of Aeronautics and Astronautics, Berlin, Heidelberg, 2005.

[13] Daniel DeLaurentis, Oleg Sindiy, and William Stein. *SPACE Conferences and Exposition*, chapter Developing Sustainable Space Exploration via System-of-Systems Approach. American Institute of Aeronautics and Astronautics, 2006.

[14] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, December 2006.

[15] E.A.Kendall. Agent roles and role models: New abstractions for multiagent system analysis and design. International Workshop on Intelligent Agents in Information and Process Management, 1998.

[16] Rolf Hennicker and Annabelle Klarl. *Foundations for Ensemble Modeling – The Helena Approach*, pages 359–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[17] Stephan Herrmann. Programming with roles in objectteams/java. *Papers from the AAAI Fall Symposium*, November 2005.

[18] Valentin Kennke. *Adopting Event-Based Modularization for modular implementation of coordination patterns.* Diplomarbeit, TU Dresden, 2015.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[20] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, chapter A Metamodel Family for Role-Based Modeling and Programming Languages, pages 141–160. Springer International Publishing, Cham, 2014.

[21] Max Leuthäuser and Uwe Aßmann. Enabling view-based programming with scroll: Using roles and dynamic dispatch for etablishing view-based programming. In *MORSE/VAO '15 Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, 2015.

[22] Somayeh Malakuti. An overview of event-based facades for modular composition

and coordination of multiple applications. *Technical Report*, 08 2015. http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-203204.

[23] Somayeh Malakuti and Mariam Zia. Adopting architectural event modules for modular coordination of multiple applications. *Technical Report*, 07 2015. http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-175973.

[24] Cesare Pautasso and Erik Wilde. Why is the web loosely coupled? a multi-faceted metric for service design. In *IN: PROC. OF THE 18TH WORLD WIDE WEB CONFERENCE*, 2009.

[25] Christian Piechnick, Sebastian Richly, Sebastian Götz, Claas Wilke, and Uwe Aßmann. Using role-based composition to support unanticipated, dynamic adaptation-smart application grids. *Proceedings of ADAPTIVE*, pages 93–102, 2012.

[26] Kateryna Rybina, Waltenegus Dargie, Rene Schöne, and Somayeh Malakuti. Mutual influence of application- and platform-level adaptations on energy-efficient computing. *Conference Paper*, 04 2006.

[27] Andrew P. Sage and Christopher D. Cuppan. On the systems engineering and management of systems of systems and federations of systems. *Inf. Knowl. Syst. Manag.*, 2(4):325–345, December 2001.

[28] Liping Zhao. Designing application domain models with roles. In *Model Driven Architecture. European MDA Workshops: Foundations and Applications*, Lecture Notes in Computer Science, 2005.

[29] Haibin Zhu and MengChu Zhou. Role-based collaboration and its kernel mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(4):578–589, July 2006.

Confirmation

I confirm, that I created this work independent of foreign contributions, using no other sources than those, that have been indicated in this work.

Dresden, 23.05.2016