TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211XX

TUD-FI16-01-Februar 2016

Somayeh Malakuti
Software Technology group

An Overview of Event-based Facades for Modular Composition and Coordination of Multiple Applications

Technische Universität Dresden
Nutzerberatung CD
01062 Dresden
Germany
http://tu-dresden.de/cd

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

An Overview of Event-based Facades for Modular Composition and Coordination of Multiple Applications

Somayeh Malakuti[*]
Software Technology group
Technical University of Dresden, Germany
somayeh.malakuti@tu-dresden.de

Abstract

Complex software systems are usually developed as systems of systems (SoS's) in which multiple constituent applications are composed and coordinated to fulfill desired system-level requirements. The constituent applications must be augmented with suitable coordination-specific interfaces, through which they can participate in coordinated interactions. Such interfaces as well as coordination rules have a crosscutting nature. Therefore, to increase the reusability of the applications and to increase the comprehensibility of SoS's, suitable mechanisms are required to modularize the coordination rules and interfaces from the constituent applications. We introduce a new abstraction named as *architectural event modules (AEMs)*, which facilitate defining constituent applications and desired coordination rules as modules of SoS's. AEMs augment the constituent applications with event-based facades to let them participate in coordinated interactions. We introduce the EventArch language in which the concept of AEMs is implemented, and illustrate its suitability using a case study.

1 Introduction

As the complexity of today's software systems increases, they are usually developed as systems of systems (SoS's) in which multiple applications are composed together to fulfill desired SoS-level requirements [5]. However, since the constituent applications are developed independently, there might be some undesirable (implicit) interactions among them, which prevent the system-level requirements to be fulfilled. Therefore, it is necessary to coordinate the interactions of the constituent applications towards meeting the system-level requirements.

[*]This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence cfaed in the Collaborative Research Center 912.

Applications may be reused individually and/or as the constituent of various SoS's. In the latter case, different kinds of coordination rules may be applied to them depending on the requirements of the SoS's. This implies that the reusability of the applications must be preserved when they are composed into one SoS. In addition, due to the inherent complexity of SoS's, we require means to increase the abstraction level of architectural specifications so that the comprehensibility of the architectural specifications increases.

To these aims, we claim that an architectural description language (ADL) must fulfill the following requirements. a) It must facilitate applying coordination rules based on different patterns such as centralized, peer-to-peer and hybrid. b) It must offer suitable abstractions to modularize crosscutting coordination rules from the corresponding constituent applications, so that the reusability and evolvability of both parties increase. c) The ADL must offer means to define and modularize crosscutting coordination-specific interfaces for the constituent applications. d) The ADL must be able to cope with the language heterogeneity of the constituent applications.

Although a large body of research exist in various areas such as aspect-oriented (AO) software development, publish/subscribe systems, system modeling languages and coordination languages, we discuss that there is no solution that fulfills all of these requirements. We therefore introduce a new kind of component named as *architectural event modules (AEMs)* as a solution.

AEMs are means to represent constituent applications and their coordination rules as modules of SoS's. An AEM modularly augments a constituent application with event-based interfaces, which are facades to let the application participate in coordinated interactions. Primitive AEMs can be adopted to modularly apply coordination rules based on the centralized pattern. A Composite AEM provides a two-level facade for a constituent application, using which coordination rules can modularly be applied based on the peer-to-peer pattern.

We initially introduced AEMs in our previous report [25] with following two characteristics. Firstly, they had a primitive structure, meaning that the interfaces of AEMs were stateless primitive predicates over event attributes. Secondly, AEMs defined only one provided/required interface for an application. Consequently, AEMs could only be used for defining only one coordinator, which could only be applied based on the centralized pattern. We lift these constraints in this report by supporting multiple interfaces as well as AEMs with composite structure, respectively.

This report introduces the EventArch[1] ADL in which the concept of AEMs is implemented, and illustrates its suitability using a case study in the domain of energy optimization. This report also covers the compiler of EventArch, and explains its runtime event processing semantics.

This report is organized as follows: Section 2 explains our illustrative case study. Section 3 outlines a set of requirements that must be fulfilled in compos-

[1]We named the language as EventReactor in our previous work [25]. Due to the name overlap with the EventReactor programming language [24], we renamed the language to EventArch.

ing multiple applications. Section 4 identifies the shortcomings of the current languages and frameworks in fulfilling these requirements. Section 5 explains the concept of AEMs, and Section 6 shows their usage. Section 7 discusses the EventArch language via our illustrative case study. Section 8 discusses the compiler and the event processing semantics of EventArch. Section 9 discusses the suitability of AEMs in improving the modularity of implementations. Finally Sections 11 and 12 outlines future work and conclusion.

2 Illustrative Case Study

Assume for example that we have two energy optimization techniques, which operate at the level of application software [13] and virtual machines (VMs). They are referred to as adaptive software and load balancer, respectively.

Figure 1 abstractly shows the components of the adaptive software, which are executed within a VM in a server. There are multiple implementations of each *Application* component, which offer the same functionality but with different qualities of service such as energy consumption and performance. They require different CPU frequency, network bandwidth and RAM to offer their services. When a user issues a request to the application, the request triggers the *Optimizers* components, which analyze the availability of the hardware resources on the server. They may select the best implementation of each *Application* component that offers a better performance and energy consumption based on the available resources. After these steps, the user's request is served by the *Application* components.

As the figure shows, the load balancer also consists of multiple components, which monitor and analyze the load of the server at predefined intervals to detect whether the server is underutilized or over-utilized. If so, they plan for migrating some VMs to another server, and perform the migration to balance the load of the server.

The adaptive software and the load balancer are developed and being used independently. Nevertheless, we would like to compose them with each other into one SoS to improve the overall energy consumption of the system via their joint execution. This may result in various kinds of unforeseen interplays among them [23, 38].

Suitable rules must be defined to coordinate such interplays towards fulfilling desired system-level requirements. The focus of this report is not on specific coordination rules; as examples we adopt the following simplified rules:

Rule (1): If the load balancer detects that the server is underutilized, plenty of resources are available to serve users' requests. Therefore, we postpone the migration until the current request is served by the adaptive software. To reduce the required time and energy for the migration, no request will be served until the migration terminates. This coordination rule will be implemented by the so-called *StateCoordinator*.

Rule (2): If the load balancer detects that the server is over-utilized, we have limited resources to serve users' requests properly. Therefore, the mi-

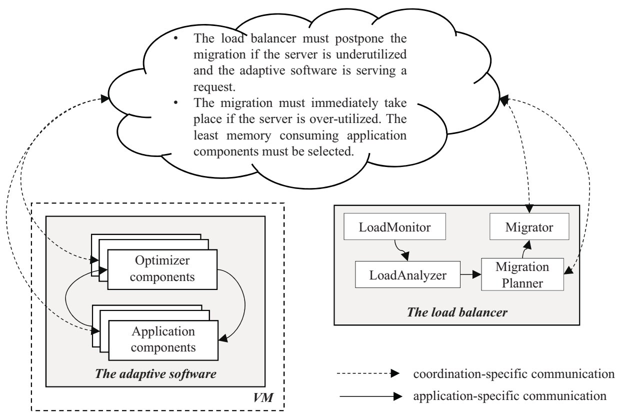
• The load balancer must postpone the migration if the server is underutilized and the adaptive software is serving a request.
• The migration must immediately take place if the server is over-utilized. The least memory consuming application components must be selected.
Optimizer components
Application components
The adaptive software
VM
LoadMonitor
Migrator
LoadAnalyzer
Migration Planner
The load balancer
coordination-specific communication
application-specific communication

Figure 1: The adaptive software and the load balancer

gration must immediately start; nevertheless we switch to the least memory consuming *Application* components to reduce the required time and energy for the migration. This coordination rule will be implemented by the so-called *SwitchCoordinator*.

3 Requirements

We particularly seek for solutions to increase the resuability of constituent applications when they are reused individually and/or as the constituent of various SoS's. In addition, due to the inherent complexity of SoS's, we seek for means to increase the comprehensibility of the architectural specifications. In the following, we outline a set of requirements that we believe an ADL must fulfill with these regards:

REQ (1)–Supporting various coordination patterns: Constituent applications execute in parallel and may be distributed across network. Various patterns may be adopted to organize the constituent applications and their coordinators. In this report, we focus on three widely-known patterns; i.e. centralized, peer-to-peer and hybrid. In the centralized case, coordinators are distinct entities that are separated from the constituent applications and mediate between them. The peer-to-peer case improves the scalability of SoS's by letting each constituent application have its share of coordination rules to directly communicate with other constituent applications. The hybrid case is a combination of these two.

The choice of a suitable pattern depends on the requirements of SoS's, and may vary over the time. Therefore, an ADL must offer suitable abstractions to specify desired coordination patterns, and to flexibly change them when needed.

REQ (2)–Modularizing and composing coordination rules: Coordination is inherently a *crosscutting* concern because it is related to the interactions of multiple entities (e.g. constituent applications in SoS's). The need for separating crosscutting concerns, including coordination logic, has been widely studied in the aspect-oriented community [3]: If crosscutting concerns are not modularized, their implementation scatters across and tangles with other components in applications. Consequently, the complexity of the applications increases, and ripple modification effect may be experienced in the applications if the concerns evolve.

Since applications may be reused individually and/or as the constituent of various SoS's, we claim that an ADL must offer suitable abstractions to modularize coordination rules and to compose them with corresponding constituent applications.

REQ (3)–Specifying and modularizing coordination-specific interfaces: Regardless of whether applications are used as standalone software or as the constituents of an SoS, they already have various public interfaces through which their functional services can be accessed. However, such interfaces may not define the necessary information for coordinating the applications within an SoS.

Therefore, an ADL must offer means to define coordination-specific interfaces for constituent applications. Such interfaces must specify the operational states of the applications at which they must be coordinated, the information that must be gathered about these operational states, and the flow of control among the applications.

Coordination-specific interfaces may refer to the information that is available within multiple application components. This means that these interfaces may also have a *crosscutting* nature. An example is shown in Figure 1 for our illustrative case study, where we need to gather current execution state of multiple components of the adaptive software to apply the coordination rules to this software.

Besides being crosscutting, coordination-specific interfaces vary depending on the adopted coordination rules and the requirements of a specific SoS. Therefore, we claim that an ADL must also facilitate modularizing the description of coordination-specific interfaces from the core functionality of the constituent applications. Such a separation helps to increase the reusability of the applications, besides preventing unnecessary information to be exposed from them.

REQ (4)–Coping with the heterogeneity of implementation languages: Heterogeneity of constituent applications is one of the key characteristics of SoS's; in this report, we focus on the language heterogeneity of the applications. Since constituent applications are independently developed by different teams with different skills and preferences, they may be implemented using different languages and techniques. For example, the adaptive software of our illustrative case has been implemented in Java based on the OSGi component model, and the load balancer has been implemented in C++. Besides, coordination rules may be implemented in a language differently from the constituent applications. Therefore, an ADL must enable us to define the architecture of SoS's abstractly from the implementation languages of the constituent applica-

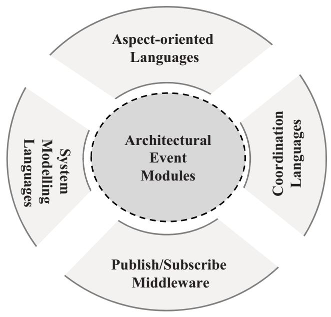
Aspect-oriented Languages
System Modelling Languages
Architectural Event Modules
Coordination Languages
Publish/Subscribe Middleware

Figure 2: The related research areas

tions. This would help to increase the comprehensibility and reusability of the architectural specifications by eliminating irrelevant implementation details.

4 State of the Art

Figure 2 shows various research areas that are relevant to this report. AO programming/architectural languages are relevant because of our requirements for modularizing crosscutting concerns. System modeling languages are relevant because of their support for defining the structure and behavior of large-scale software. Publish/subscribe systems are important for their support in composing multiple applications, which are possibly developed in different languages, in a loosely coupled manner. Coordination languages are relevant because of their support for defining coordination rules and patterns.

Our evaluation reveals that there is no solution that fulfills all of the requirements outlined in previous section. The details of our evaluation are explained below.

4.1 AO Programming/Architectural Languages

AO languages facilitate modularizing crosscutting concerns via aspects, which are applied to the so-called base programs through a join point and pointcut designation mechanism. In our case, constituent applications get the role of base programs to which coordination aspects must be applied.

It has been claimed that to preserve information hiding in base programs and to control the impacts of aspects on them, the interface of the base programs to the aspects must explicitly be defined [41, 4, 40]. The requirement REQ (3) is in line with this claim. The proposals in [4, 40] extend the interface of the base programs with a set of pointcuts that denote internal semantic join points to which the aspects can be applied. In these approaches, interface specifications scatter across and tangles with the base programs. Consequently, the reusability of the applications reduces when they must be adopted as the constituent of various SoS's.

In XPIs [41], AspectJ aspects are adopted to define and modularize crosscutting interfaces for the base programs. Such interfaces define a set of pointcuts that are visible to the aspects. XPIs and the other mentioned proposals are supported at the level of programming languages, and are limited to modularize crosscutting concerns within one application developed in one language. However, we require modularizing the concerns that crosscut multiple applications, which are executed in parallel and are possibly developed in different languages.

Various proposals exist to unify the notion of aspects and ordinary objects/components [43, 42, 35, 26]. These proposals are also at the level of programming languages and fall short of addressing the other requirements outlined in Section 3.

We introduced *object-level event modules* as means to modularize domain-specific concerns, which may crosscut multiple components that are implemented in different languages [24]. As the successor of *composition filters* [3], object-level event modules can also cope with heterogeneity of implementation languages. However, both object-level event modules and composition filters fall short of fulfilling REQ (1)–(3). Firstly, there is no support for specifying and modularizing crosscutting coordination-specific interfaces; instead, events must explicitly be announced from applications. Such code scatters across the applications, and reduces their reusability if they must be adopted as the constituent of various SoS's. Secondly, object-level event modules and composition filters can be best adopted for implementing centralized coordinators within one application.

Various ADLs are proposed to modularize crosscutting concerns at the architectural level [11, 33, 32, 34]. These languages offer abstractions to define aspectual components, architectural pointcuts, and the composition of the application components with the aspectual components. Adopted from the conventional AO programming languages, the ADLs do not offer means to define and modularize crosscutting coordination interfaces for applications. As a result, REQ (3) is not fulfilled in these languages. Moreover, these ADLs do not fulfill REQ (4).

Several AO languages and middleware have been proposed to facilitate modularizing the crosscutting concerns that are distributed on multiple hosts [29, 20, 28, 27]. The main focus of these approaches is on defining remote pointcut and remote advice, which can be evaluated and executed on different hosts. Only [27] aims at defining dedicated interfaces from application components to aspects; however, it does not support specifying and modularizing crosscutting interfaces. Except for [20] that supports .Net applications, the other approaches are limited to support Java-based applications; hence, they fall short of fulfilling REQ (4).

4.2 System Modeling Languages

SysML [2] is defined as a dialect of UML, which supports the specification, analysis, design, verification and validation of a broad range of systems and SoS's. CML [44] is a dedicated language for building and analyzing the models of SoS's, in which constituents are modelled as processes and theit reactive behavior is defined via CSP. Communicating Structures [18] are other means of defining the structure and behavior of SoS's. They adopt a C++ based library and an object-oriented core environment for the modeling and analysis of SoSs.

All these proposals have a root in object-oriented design, without addressing the need for modular representations of crosscutting concerns as discussed in REQ (2) and REQ (3).

The need for extending constituent applications with SoS-specific interfaces is also studied in [17], where the so-called Relied Upon Interfaces are introduced as a solution. This proposal, however, is very abstract without addressing the crosscutting nature of such interfaces as in REQ (2), and needs for modularizing such interfaces and composing them with application developed in different languages as in REQ (3) and (4).

4.3 Publish/Subscribe Middleware

Event-based architecture and publish/subscribe paradigm are adopted for developing applications that must remain loosely coupled while interacting with each other based on the one-to-many and many-to-one styles of communication. Several academic and commercial proposals exist for such middleware [36, 8, 9, 15, 6]. The main focus of these proposals is on, among others, offering basic abstractions for defining, notifying and filtering events in an efficient and reliable manner. These proposals also have a root in object-oriented design, in which event-based ports are defined for components to let them interact in a loosely coupled and asynchronous manner. These proposals do not address the need for modular representations of crosscutting concerns as discussed in REQ (2) and REQ (3). Consequently, programmers have to directly identify coordination points in the applications, and add necessary communication and coordination code there. Such modifications scatter across and tangle with the core functionality of the applications. Consequently, the reusability of the applications decreases. Moreover, coordination-specific information and interactions will only be available at programming level, which makes it difficult to trace them back to the architectural level.

4.4 Coordination Languages

Coordination languages can be classified as data-driven and control-driven [30]. In data-driven languages, a coordinator or coordinated application is responsible for manipulating data as well as for coordinating either itself and/or other applications. Linda [12] and Linda-like languages [31, 30] are in this category of coordination languages. The separation of coordination rules from coordinated

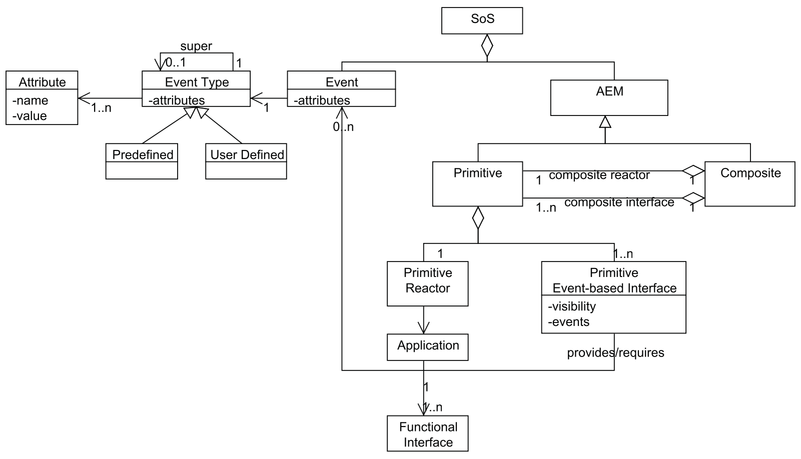

Figure 3: The meta-model of AEMs

applications is not enforced at the syntactic level by data-driven coordination languages. Consequently, the coordination rules may scatter across and tangle with the core functionality of the coordinated applications. One may adopt AO languages to modularize crosscutting coordination-specific code. We have explained the shortcomings of AO languages in Section 4.

In control-driven languages, coordinated applications are seen as black boxes with clearly defined input/output interfaces. Hence, these languages facilitate a clear separation between coordinators and the coordinated applications. Examples of such languages are the ADLs that offer dedicated entities as connectors to glue multiple processes/components together [39, 22, 21, 7]. These proposals have a root in procedural or object-oriented design, without addressing the need for modular representation of crosscutting concerns as discussed in REQ (2) and REQ (3).

5 Architectural Event Modules (AEMs)

In this report, we extend the concept of AEMs to overcome the shortcomings outlined in the previous section. As the meta-model in Figure 3 shows, we assume that an SoS consists of a set of events, and a set of AEMs that communicate with each other via events.

Events are means to represent state changes of interest in AEMs and/or in the environment. They are typed; an event type defines a set of attributes using which events convey necessary information about the state changes. Event types may be predefined or user-defined based on the requirements of SoS's.

An AEM may have a primitive or composite structure. A primitive AEM has one primitive reactor, and one or more primitive event-based interfaces. The reactor part represents the application whose behavior must be coordinated, and/or the application that implements a coordination rule. The interfaces specify the events that must be exchanged among the primitive AEMs for the purpose of coordination. The visibility of the interfaces can be private or public, meaning that the events are only visible inside a composite AEM or are visible to all AEMs in the system, respectively.

A composite AEM consists of a composite reactor and one or more composite interfaces, which are defined in terms of primitive AEMs. The composite interfaces define coordination rules; the composite reactor encapsulates an application that must be coordinated. Composite AEMs allow of implementing the peer-to-peer pattern such that each peer defines its share of coordination rules in a modular way.

Our new model of AEMs has the following benefits over our previous proposal in [25]. Firstly, composite AEMs facilitate modular implementations of the peer-to-peer pattern. Secondly, supporting multiple interfaces for primitive AEMs let them participate in multiple coordinated interactions, which are applied based on centralized, peer-to-peer or hybrid patterns.Thirdly, composite AEMs provide two levels of facades for applications. The first level is defined by primitive event-based interfaces that let the applications participate in coordinated interactions. The second level is defined via the composite interfaces of the composite AEMs, which express peer-to-peer coordination rules. Such a separation increases the reusability of the applications and coordination rules, and is a means to impose information (event) hiding within the composite AEMs.

6 Example Usages of AEMs

Figure 4 schematically show the use of AEMs to modularly apply our coordination rules to the adaptive software and the load balancer in a centralized manner. Here, we define two AEMs named as *WrappedAdaptiveSoftware* and *WrappedLoadBalancer*, which respectively augment the adaptive software and the load balancer with two event-based interfaces. These interfaces let the applications interact with our two coordinators, which are also modularly defined via separate AEMs. In this figure, the coordinators are applied based on the centralized pattern.

Figure 5 shows the peer-to-peer case. As the left part of the figure shows, two coordination rules are defined via two separate primitive AEMs named as *SwitchCoordinatorLBPeer* and *StateCoordinatorLBPeer*. We define a primitive AEM named as *WrappedLoadBalancer* that wraps the load balancer with two interfaces to let it locally interact with two coordinators. These primitive modules are grouped together via the composite AEM *CoordinatedLoadBalancer*, in which *SwitchCoordinatorLBPeer* and *StateCoordinatorLBPeer* are the composite interfaces, and *WrappedLoadBalancer* is the composite reactor.

The division of coordination rules between the peers depends on the complexity of the rules and the design of software. In our case study, we decided to localize the logic of *SwitchCoordinatorLBPeer* into one module, and locate it

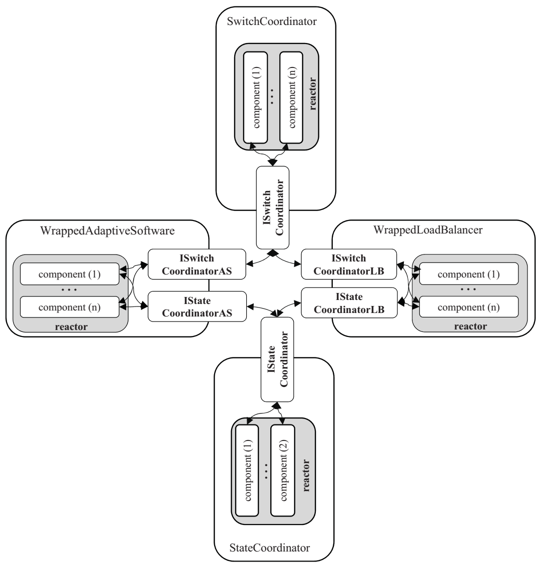

Figure 4: An example usage of AEMs for the centralized pattern

within *CoordinatedLoadBalancer*. This coordinator only commands the adaptive software to switch to the least memory consuming component. In the right part of Figure 5, *CoordinatedAdaptiveSofwtare* defines another peer, which has only one composite interface named as *StateCoordinatorASPeer*. The composite reactor is defined by *WrappedAdaptiveSoftware*, which has one interface to locally communicate with *StateCoordinatorASPeer*, and one interface to communicate with *SwitchCoordinatorLBPeer*.

It is also possible to achieve the hybrid pattern via applying some coordinators based on the centralized pattern and some based on the peer-to-peer pattern.

7 The EventArch Language

We implemented the concept of AEMs in the EventArch language [2]. The abstract syntax of this language is presented in Appendix A. In the following, we make use of our illustrative example to explain how primitive and composite AEMs can be adopted for modular coordination of multiple applications.

7.1 The Specification of the Event Types

The first step towards defining AEMs is to specify the events that must be exchanged among them. In EventArch, events are typed entities; an event

[2]https://github.com/eventbasedmodules/EventArch

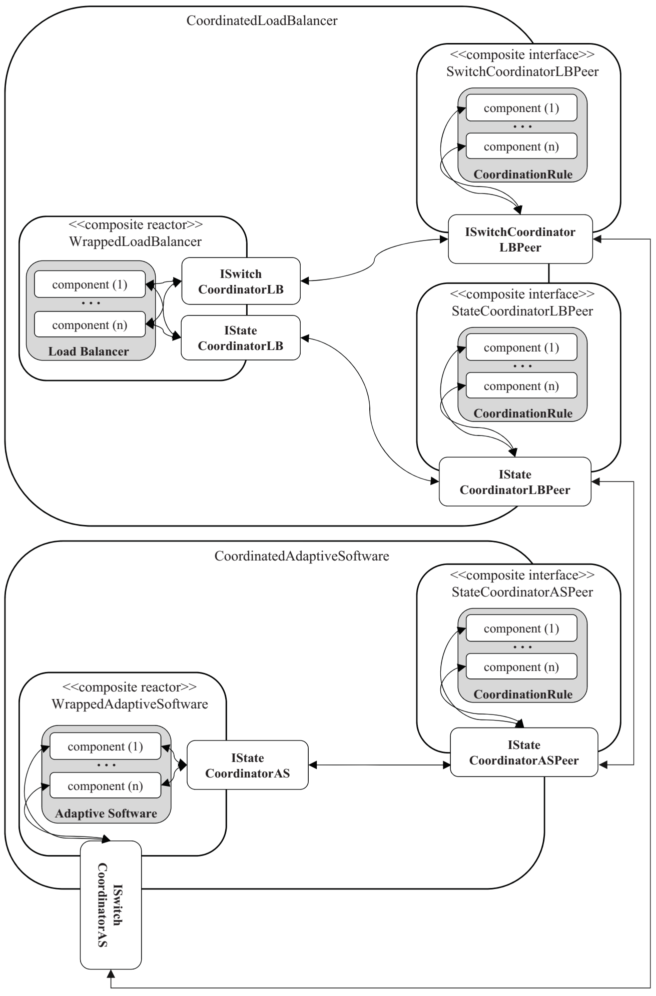

Figure 5: Peer-to-peer coordination via composite AEMs

type is a data structure that defines a set of attributes for the events. Listing 1 defines the event types for our illustrative example. `BaseEventType` is the super type for other event types. It defines the attributes `publisher`, `target` and `timestamp`. There are two other event types named as `ApplicationStarted` and `ApplicationTerminated`, which are used to indicate that the execution of an application has started and terminated, respectively..

For our case study we define the following three event types. The event type `CoordinationCommand` is to represent the commands that are sent by the coordinators to the adaptive software and the load balancer. The events of the type `StateRequest` are used to ask for the current execution state of the

```
eventtype BaseEventType { String publisher; String target; Long timestamp; }
eventtype ApplicationStarted extends BaseEventType{ String name;}
eventtype ApplicationTerminated extends BaseEventType{ String name;}
...
eventtype CoordinationCommand extends BaseEventType { String command; }
eventtype StateRequest extends BaseEventType {}
eventtype ConstituentState extends BaseEventType { String applicationState; String serverLoad; }
```

Listing 1: The specification of event types

adaptive software and the load balancer. The event type `ConstituentState` represents the current execution state of the adaptive software and the load balancer via its attribute `applicationState`. The attribute `serverLoad` will be used to represent the current load of the server.

7.2 The Specification of Centralized Coordinators

The constituent applications usually offer some functional interfaces through which their services are accessed. There are many proposals to specify such interfaces [6]; this topic is out of the scope of this report.

As the next step we would like to augment our constituent applications with event-based interfaces. As Figure 4, since we have two coordinators, we define two separate interfaces for each application to let it communicate with the corresponding coordinator.

Line 1 of Listing 2 defines `WrappedLoadbalancer`, which wraps a C++ reactor application named as `LoadBalancer` via two event-based interfaces named as `ISwitchCoordinatorLB` and `IStateCoordinatorLB`. The reactor and interfaces are bound to each other via the operator `<->`.

Lines 3–22 define the interface `ISwitchCoordinatorLB` that specifies the events exchanged between the `WrappedLoadBalancer` and `SwitchCoordinator`. The `required` part specifies the events of interest that must be selected; the selection criteria is defined as Boolean expressions over event attributes. Here, we select the events of the type `CoordinationCommand`, which are published by the module `SwitchCoordinator`. These events are represented in the interface via the variable `e_CoordinationCMD`.

Lines 8–21 of Listing 2 specify the events that are published by the load balancer application, and the necessary synchronization conditions for them. Currently four kinds of state changes in the execution of applications can be published as events. These are before invocation, before execution, after invocation and after execution of methods. We adopt a set of pointcut designators that are supported by most AO languages to refer to the state changes of interest. This facilitates binding an event to multiple state changes in multiple application components.

The event `e_StartedLB` of the type `ConstituentState` is published before the execution of the method `plan` in the class `MigrationPlanner` when the server is over-utilized. Lines 11–13 show that before publishing an event, its attributes can be initialized. Here, we assign the current state of the server to

```
 1 WrappedloadBalancer[CPP]:= {ISwitchCoordinatorLB,IStateCoordinatorLB}<->{'LoadBalancer'}
 2
 3 interface ISwitchCoordinatorLB{
 4   requires {
 5     CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' &&
 6                      E.publisher== 'SwitchCoordinator' && E.target == 'WrappedLoadBalancer'}
 7   }
 8   provides {
 9     ConstituentState e_StartedLB := before execution (void org.loadbalancer.MigrationPlanner.plan(String load))
10         if (load =='OverUtilized') {
11           serverLoad = load;
12           applicationState = 'StartExecuting';
13           target = 'SwitchCoordinator';
14         }
15     wait when (e_StartedLB) until (e_CoordinationCMD){
16         switch (e_CoordinationCMD.command){
17           case 'proceed': proceed;
18           case 'suspend': suspend;
19         }
20     }
21   }
22 }
23 interface IStateCoordinatorLB{
24   requires {
25     CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' && E.publisher == 'StateCoordinator' &&
26                             E.target == 'WrappedLoadBalancer'}
27   }
28   provides {
29     ConstituentState e_StartedLB := before execution (void org.MigrationPlanner.plan(String load))
30           if (load =='UnderUtilized') {
31               serverLoad = load;
32               applicationState = 'StartExecuting';
33               target = 'StateCoordinator';
34             }
35     wait when (e_StartedLB) until (e_CoordinationCMD){
36       switch (e_CoordinationCMD.command){
37           case 'proceed': proceed;
38           case 'suspend': suspend;
39         }
40     }
41     ConstituentState e_EndedLB := after execution (void org.Migrator.migrate(..)){
42         applicationState = 'EndExecuting';
43         target = 'StateCoordinator';
44         serverLoad = 'UnderUtilized';
45     }
46   }
47 }
```

Listing 2: The specifications of the load balancer for the centralized pattern

the attribute `serverLoad`, and specify `SwitchCoordinator` as the recipient of the event. The value `'StartExecuting'` is assigned to `applicationState` to indicate that the load balancer wants to plan for migration.

Events are by default published in a non-blocking way. However, it may be needed to block the execution of wrapped application after publishing an event until a specific response event is received. For example, after the event `e_StartedLB` is published, the execution of the load balancer must block until `SwitchCoordinator` informs it that the adaptive software has switched to the least memory consuming component. Such conditions can be expressed via the `wait when...until` expressions.

Line 15 specifies that after publishing `e_StartedLB`, the execution of the load balancer is blocked until we receive the event `e_CoordinationCMD`. Currently, hree kinds of actions can be performed upon receiving an event in the `wait when...until` expressions: `retry` means that the execution of the application must resume by re-publishing its last event; `proceed` means that the execution must resume; `suspend` means that the execution must stay blocked until the specified response event arrives and causes the execution to resume.

```
 1 WrappedAdaptiveSoftware[Java]:= {ISwitchCoordinatorAS,IStateCoordinatorAS} <−>{'AdaptiveSoftware'}
 2
 3 interface ISwitchCoordinatorAS{
 4   requires {
 5     CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' &&
 6                        E.publisher == 'SwitchCoordinator' && E.target == 'WrappedAdaptiveSoftware'}
 7     on (e_CoordinationCMD) {invoke ('org.application.Optimizer', 'reconfigure',e_CoordinationCMD); }
 8   }
 9   provides {
10     ConstituentState e_EndSwitch := after execution (static void org.Optimizer.reconfigure(..)) { applicationState = 'SwitchEnd'; }
11   }
12 }
13 interface IStateCoordinatorAS{
14   requires {
15     CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' &&
16                      E.publisher == 'StateCoordinator' && E.target =='WrappedAdaptiveSoftware'}
17   }
18   provides {
19     ConstituentState e_StartedAS := before execution (void org.ApplicationComponent.execute(..)){
20               applicationState = 'StartExecuting';
21               target = 'StateCoordinator';
22     }
23     wait when (e_StartedAS) until (e_CoordinationCMD){
24       switch (e_CoordinationCMD.command){
25           case 'proceed': proceed;
26           case 'suspend': suspend;
27       }
28     }
29     ConstituentState e_EndedAS := after execution (void org.ApplicationComponent.execute(..)){
30               applicationState = 'EndExecuting';
31               target = 'StateCoordinator';
32     }
33   }
34 }
```

Listing 3: The specifications of the adaptive software for the centralized pattern

Lines 17–18 show that if the event `e_CoordinationCMD` arrives and has the command `proceed`, the execution of the application must proceed as normal. If the command is `suspend`, the execution of the application remains blocked until an `e_CoordinationCMD` is received that has the command `proceed`.

Lines 23–47 of Listing 2 define the interface `IStateCoordinatorLB`. The corresponding coordinator makes sure that if the server is underutilized, the load balancer waits until the adaptive software finishes serving the current request. To let the load balancer participate in this coordination, the state change before the execution of the method `plan` when the server is underutilized is announced via the event `e_StartedLB`. The execution of the load balancer is blocked until the event `e_CoordinationCMD` arrives and commands the load balancer to proceed with its execution. The event `e_EndedLB` is published after the migration finishes to inform the coordinator.

Likewise, Listing 3 defines the module `WrappedAdaptiveSoftware`, which augments the Java application named as `AdaptiveSoftware` with two event-based interfaces. The EventArch language makes use of pointcut expressions to receive events from wrapped applications. It may also be necessary to send events to the wrapped applications; for example, to send an event to the adaptive software that it must switch to the least memory consuming component. EventArch offers the `on` expressions for this matter.

The interface in lines 5–6 selects the events sent by `SwitchCoordinator`. The expression in line 7 specifies that after the event is selected, the method `reconfigure` on the `Optimizer` component must be invoked, and the event

```
 1 SwitchCoordinator[StateMachine] := {ISwitchCoordinator} <-> {
 2     initial state Start {
 3       during:
 4         on (ISwitchCoordinator.e_StartedLB) -> WaitForSwitch;
 5         on (ISwitchCoordinator.e_TerminatedAS) -> WaitForRestart;
 6       exit:
 7         send ISwitchCoordinator.e_CoordinationCMD = new CoordinationCommand(){
 8           command = 'suspend'; target = 'WrappedLoadBalancer';
 9         };
10     }
11     state WaitForSwitch {
12       entry:
13          send ISwitchCoordinator.e_CoordinationCMD = new CoordinationCommand() {
14           target = 'WrappedAdaptiveSoftware'; command = 'switch'; };
15       during:
16           on (ISwitchCoordinator.e_EndedSwitch) ->Start;
17           on (ISwitchCoordinator.e_TerminatedAS) -> WaitForRestart;
18       exit:
19         send ISwitchCoordinator.e_CoordinationCMD = new CoordinationCommand(){command = 'proceed';  target = 'WrappedLoadBalancer';};
20     }
21     state WaitForRestart {
22       during:
23         on (ISwitchCoordinator.e_InitializedAS) -> Start;
24         on (ISwitchCoordinator.e_StartedLB){ send ISwitchCoordinator.e_CoordinationCMD = new CoordinationCommand(){
25                                   command = 'proceed';target = 'WrappedLoadBalancer';};
26         }
27     }
28   }
29 interface ISwitchCoordinator{
30     requires {
31     ConstituentState e_StartedLB= {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedLoadBalancer' &&
32                           E.applicationState == 'StartExecuting' && E.serverLoad == 'OverUtilized'}
33     ConstituentState e_EndedSwitch = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedAdaptiveSoftware' &&
34                            E.applicationState =='SwitchEnd'}
35     ApplicationTerminated e_TerminatedAS = {E | E instanceof 'ApplicationTerminated' && E.name == 'WrappedAdaptiveSoftware'}
36     ApplicationStarted e_InitializedAS = {E | E instanceof 'ApplicationStarted' && E.name == 'WrappedAdaptiveSoftware'}
37   }
38   provides { CoordinationCommand e_CoordinationCMD;}
39 }
```

Listing 4: The specification of switch coordinator for the centralized pattern

must be provided to it as an argument. The interface `IStateCoordinatorAS` is defined in a similar way as `IStateCoordinatorLB`.

Lines 1–28 of Listing 4 defines `SwitchCoordinator`, which has an interface to interact with `WrappedLoadBalancer` and `WrappedAdaptiveSoftware`. This interface selects the event indicating that the load balancer wants to plan for the migration, and the event indicating that the adaptive software has finished switching to the least memory consuming component. In addition, it selects the event indicating that the execution of the adaptive software has terminated. As the provided interface, the events of the type `CoordinationCommand` are published to the load balancer and the adaptive software.

Coordination rules may be defined via Java or C++ applications, which are represented via primitive AEMs in a similar way as other applications. Alternatively, the coordination rules can be defined in a declarative way as a state machine; an example is shown in lines 2–27 of Listing 4. The state machine language supports defining states, transitions and the actions that can be performed upon the entrance to a state, exit from a state, and during the activation of a state.

This state machine starts executing in its initial state, where it responds to receive the event `e_StartedLB` or `e_TerminatedAS`. The event `e_StartedLB` results in a transition to the state `WaitForSwitch`. Upon the entrance to this state, an event of the type `CoordinationCommand` is prepared and is sent to

`WrappedAdaptiveSoftware` to command the adaptive software to switch to the least memory consuming component. In the state `WaitForSwitch`, the coordinator waits for the arrival of the event `e_EndedSwitch` from the adaptive software. This event results in a transition to the state `Start`. The code in the `exit` part, which executes before the transition takes place, publishes an event to the load balancer to indicate that it can proceed with the migration.

If the adaptive software is terminated, the load balancer may wait forever to receive a command to proceed its execution. To prevent this case, the state machine also considers the case that the adaptive software is terminated. When it receives the event `e_TerminatedAS` in the state `Start`, it takes a transition to the state `WaitForRestart` where the load balancer is commanded to proceed its execution.

Listing 5 shows an implementation of `StateCoordinator`, which ensures the load balancer and adaptive software are mutually executed when the server is underutilized. To this aim, whenever the load balancer starts executing, a transition is take to the state `RunLB`. While in this state, if the adaptive software wants to start executing, a transition is taken to the state `SuspendAS` to suspend its execution. The same logic is defined for suspending the load balancer when the adaptive software is executed. For the sake of simplicity, we eliminated the cases that the execution of these applications terminates.

```
StateCoordinator[StateMachine] := {IStateCoordinator} <−> {
  initial state Start {
        during:
          on (IStateCoordinator.e_StartedAS) {
               send IStateCoordinator.e_CoordinationCMD = new CoordinationCommand(){
                  command = 'proceed'; target ='WrappedAdaptiveSoftware'; };
          } −> RunAS;
          on (IStateCoordinator.e_StartedLB_UnderUtilized) {
               send IStateCoordinator.e_CoordinationCMD = new CoordinationCommand(){
                  command = 'proceed'; target ='WrappedLoadBalancer'; };
          }−> RunLB;
    }
    state RunAS {
        during:
          on (IStateCoordinator.e_StartedLB_UnderUtilized) {
             send IStateCoordinator.e_CoordinationCMD = new CoordinationCommand(){
                  command = 'suspend'; target ='WrappedLoadBalancer'; };
          } −>SuspendLB;
          on (IStateCoordinator.e_EndedAS) −> Start;
    }
    state SuspendLB {
        during:
          on (IStateCoordinator.e_EndedAS) {
               send IStateCoordinator.e_CoordinationCMD = new CoordinationCommand(){
                  command = 'proceed'; target ='WrappedLoadBalancer'; };
          } −>RunLB;
    }
    state RunLB{
        during:
          on (IStateCoordinator.e_StartedAS){
               send IStateCoordinator.e_CoordinationCMD = new CoordinationCommand(){
                  command = 'suspend'; target ='WrappedAdaptiveSoftware'; };
          } −> SuspendAS;
          on (IStateCoordinator.e_EndedLB_UnderUtilized) −> Start;
    }
    state SuspendAS{
      during:
        on (IStateCoordinator.e_EndedLB_UnderUtilized) {
               send IStateCoordinator.e_CoordinationCMD = new CoordinationCommand(){
                  command = 'proceed'; target ='WrappedAdaptiveSoftware'; };
        } −> RunAS;
     }
}

interface IStateCoordinator{
  requires {
    ConstituentState e_StartedLB_UnderUtilized = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedLoadBalancer'
                                      && E.applicationState == 'StartExecuting' && E.serverLoad =='UnderUtilized'}
    ConstituentState e_EndedLB_UnderUtilized = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedLoadBalancer' &&
                                       E.applicationState == 'EndExecuting' && E.serverLoad == 'UnderUtilized'}
    ConstituentState e_StartedAS = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedAdaptiveSoftware' &&
                                       E.applicationState == 'StartExecuting'}
    ConstituentState e_EndedAS = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedAdaptiveSoftware' &&
                                       E.applicationState == 'EndExecuting'}
  }
  provides { CoordinationCommand e_CoordinationCMD;}
}
```

Listing 5: The specification of the state coordinator for the centralized pattern

7.3 The Specification of Peer-to-Peer Coordinators

In this section, we would like to adjust the previous specifications and support the peer-to-peer coordination pattern. As the first step we would like to define the composite module *CoordinatedLoadBalancer* whose structure is shown in Figure 5. Listing 6 defines this module, in which `SwitchCoordinatorLBPeer` and `StateCoordinatorASPeer` are composite interfaces, and the composite reactor is the `WrappedLoadBalancer` module.

Line 3 of Listing 6 defines the AEM named as `WrappedLoadbalancer`, which wraps a C++ application named as `LoadBalancer` via two primitive interfaces `ISwitchCoordinatorLB` and `IStateCoordinatorLB`. Lines 5–43 define these interfaces with the `private` visibility. This means that if the corresponding primitive AEM is defined as a member of a composite AEM, the events specified in these interfaces will only be visible within that composite AEM. In our case this means that the event are only visible inside the composite AEM `CoordinatedLoadBalancer`.

It is worth mentioning that if the keyword `private` is used for entire interface, all the events specified in the interface will get this visibility. Alternatively, one can specify certain events that are required and/or provided to be private.

```
CoordinatedLoadBalancer[Composite]:= {SwitchCoordinatorLBPeer, StateCoordinatorLBPeer} <−> {WrappedLoadBalancer}

WrappedLoadBalancer[CPP]:= {ISwitchCoordinatorLB,IStateCoordinatorLB} <−>{'LoadBalancer'}

private interface ISwitchCoordinatorLB{
  requires {
    CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' && E.publisher== 'SwitchCoordinatorLBPeer' }
  }
  provides {
    ConstituentState e_StartedLB := before execution (void org.MigrationPlanner.plan(String load))
        if (load =='OverUtilized') {
          serverLoad = load;
          applicationState = 'StartExecuting';
          target = 'SwitchCoordinatorLBPeer';
        }
    wait when (e_StartedLB) until (e_CoordinationCMD){
        switch (e_CoordinationCMD.command){
          case 'proceed': proceed;
          case 'suspend': suspend;
        }
    }
  }
}
private interface IStateCoordinatorLB{
  requires {
    CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' && E.publisher == 'StateCoordinatorLBPeer'}
  }
  provides {
    ConstituentState e_StartedLB := before execution (void org.MigrationPlanner.plan(String load))
            if (load =='UnderUtilized') {
              serverLoad = load;
              applicationState = 'StartExecuting';
              target = 'StateCoordinatorLBPeer';
            }
    wait when (e_StartedLB) until (e_CoordinationCMD){
      switch (e_CoordinationCMD.command){
          case 'proceed': proceed;
          case 'suspend': suspend;
        }
    }
    ConstituentState e_EndedLB := after execution (void org.Migrator.migrate(..)){applicationState = 'EndExecuting';}
  }
}
```

Listing 6: The specifications of the load balancer for the peer-to-peer pattern

Listing 7 shows the specification of `CoordinatedAdaptiveSoftware` and its elements, which are schematically shown in Figure 5. An attentive reader may notice that the specifications are largely similar to the centralized case, except that the name of modules and the visibility of events had be tailored.

The specification of the `SwitchCoordinatorLBPeer` and its interface is defined in Listing 8. As Figure 5 shows, this module must communicate with `WrappedLoadBalancer` that is encapsulated within `CoordinatedLoadBalancer`, and with the module `WrappedAdaptiveSoftware`. To control the visibility of the events that are published by `SwitchCoordinatorLBPeer`, lines 39 and 40 define two events with two different visibility. Accordingly, the state machine code in lines 2–27 is tailored to publish these events.

```
CoordinatedAdaptiveSoftware[Composite]:= {StateCoordinatorASPeer} <−> {WrappedAdaptiveSoftware}

WrappedAdaptiveSoftware[Java]:= {ISwitchCoordinatorAS,IStateCoordinatorAS} <−>{'AdaptiveSoftware'}

interface ISwitchCoordinatorAS{
      requires {
    CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' && E.publisher == 'SwitchCoordinatorLBPeer' &&
                                    E.target == 'WrappedAdaptiveSoftware'}
    on (e_CoordinationCMD) {invoke ('org.application.Optimizer', 'reconfigure',e_CoordinationCMD); }
  }
  provides {
    ConstituentState e_EndedSwitch := after execution (static void org.Optimizer.reconfigure(..)) { applicationState = 'SwitchEnd'; }
  }
}
private interface IStateCoordinatorAS{
  requires {
    CoordinationCommand e_CoordinationCMD = {E | E instanceof 'CoordinationCommand' && E.publisher == 'StateCoordinatorASPeer'}
  }
  provides {
    ConstituentState e_StartedAS := before execution (void org.ApplicationComponent.execute(..)){
          applicationState = 'StartExecuting';
          target = 'StateCoordinatorASPeer';
    }
    wait when (e_StartedAS) until (e_CoordinationCMD){
      switch (e_CoordinationCMD.command){
            case 'proceed': proceed;
            case 'suspend': suspend;
          }
    }
    ConstituentState e_EndedAS := after execution (void org.ApplicationComponent.execute(..)){
          applicationState = 'EndExecuting';
          target = 'StateCoordinatorASPeer';
    }
  }
}
```

Listing 7: The specifications of the adaptive software for the peer-to-peer pattern

```
SwitchCoordinatorLBPeer[StateMachine] := {ISwitchCoordinatorLBPeer} <−> {
  initial state Start {
        during:
          on (ISwitchCoordinatorLBPeer.e_StartedLB) −> WaitForSwitch;
          on (ISwitchCoordinatorLBPeer.e_TerminatedAS) −> WaitForRestart;
        exit:
          send ISwitchCoordinatorLBPeer.e_LB_CoordinationCMD = new CoordinationCommand(){
            command = 'suspend'; target = 'WrappedLoadBalancer';};
    }
    state WaitForSwitch {
      entry:
           send ISwitchCoordinatorLBPeer.e_AS_CoordinationCMD = new CoordinationCommand() {
            target = 'WrappedAdaptiveSoftware'; command = 'switch'; };
      during:
            on (ISwitchCoordinatorLBPeer.e_EndedSwitch) −>Start;
            on (ISwitchCoordinatorLBPeer.e_TerminatedAS) −> WaitForRestart;
      exit:
          send ISwitchCoordinatorLBPeer.e_LB_CoordinationCMD = new CoordinationCommand(){
            command = 'proceed'; target = 'WrappedLoadBalancer';};
    }
    state WaitForRestart {
      during:
          on (ISwitchCoordinatorLBPeer.e_InitializedAS) −> Start;
          on (ISwitchCoordinatorLBPeer.e_StartedLB){ send ISwitchCoordinatorLBPeer.e_LB_CoordinationCMD = new CoordinationCommand(){
                                    command = 'proceed';target = 'WrappedLoadBalancer';};
          }
    }
  }
interface ISwitchCoordinatorLBPeer{
  requires {
    private ConstituentState e_StartedLB= {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedLoadBalancer' &&
                                    E.applicationState == 'StartExecuting' && E.serverLoad == 'OverUtilized'}
    ConstituentState e_EndedSwitch = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedAdaptiveSoftware' &&
                                 E.applicationState =='SwitchEnd'}
    ApplicationTerminated e_TerminatedAS = {E | E instanceof 'ApplicationTerminated' && E.name == 'WrappedAdaptiveSoftware'}
    ApplicationStarted e_InitializedAS = {E | E instanceof 'ApplicationStarted' && E.name == 'WrappedAdaptiveSoftware'}
  }
  provides {
    private CoordinationCommand e_LB_CoordinationCMD;
    CoordinationCommand e_AS_CoordinationCMD;
  }
}
```

Listing 8: The specification of switch coordinator for the peer-to-peer pattern

To ensure the mutual execution of the load balancer and the adaptive software in the peer-to-peer case, we need to adopt an algorithm that is suitable for distributed processes [19]. This is different from the centralized case presented in Listing 5, in which the knowledge about the execution states of the load balancer and the adaptive software is localized in the `StateCoordinator` module. In the following, we make use of the Lamport algorithm for implementing distributed mutual execution.

In the Lamport algorithm, every process maintains a queue of pending requests for entering critical section, in which the requests are sorted based on their timestamps. A requesting process pushes its request in its own queue, and sends the request to other processes. It then waits for the replies from all other processes. If all replies have been received and the its request is at the head of its queue, the process enters the critical section. Upon exiting the critical section, the process removes its request from the queue and sends a release message to other processes. Every other process, which receives a request, pushes the request in its own request queue and replies with a timestamp. After receiving the release message, these processes remove the corresponding request from their own request queue.

To implement this algorithm, we need to define one or more event types representing lock requests, release and reply. For the sake of readability, we define three new event types as shown in Listing 9. Alternatively, we could define one event type and distinguish among three cases via event attributes.

```
eventtype LockRequest extends BaseEventType { }
eventtype LockReply extends BaseEventType { }
eventtype LockRelease extends BaseEventType { }
```

Listing 9: The specification of event types for the Lamport algorithm

The coordinator `StateCoordinatorLBPeer` is defined as shown in Listing 10. Here, `pending_requests` is defined of the type `PriorityEventQueue` to keep the list of requests for entering the critical section, i.e. executing the load balancer application. The predefined type `PriorityEventQueue` orders the events based on their timestamps.

While in the state `Idle`, if the event `e_StartedLB_UnderUtilized` arrives, an event representing a lock request is created, inserted in `pending_requests` and is published. Afterwards, an event is sent to the module `WrappedLoadBalancer` to suspend its execution, and a transition is taken to the state `Waiting`. In this state, a reply from `StateCoordinatorASPeer` is expected, which implements the same state machine for the adaptive software. If a reply arrives, and the request sent by `StateCoordinatorLBPeer` is at the top of the queue, a transition is taken to the state `Running`. Otherwise, the state machine remains in the state `Waiting` until a release message is received from `StateCoordinatorASPeer`.

Each state of the state machine also contains code to send reply messages to `StateCoordinatorASPeer`. In addition, we consider the case when the execution of the load balancer terminates, for example due to unchecked runtime exceptions. As shown in the state `AlwaysGrant`, in such a case we always grant the lock to the adaptive software.

In this example, we have only two processes that must be coordinated; therefore, we specify their name as the target and publisher of events. Nevertheless, it is possible to leave these fields empty or use wildcard characters instead of explicit names.

```
StateCoordinatorLBPeer[StateMachine] := {IStateCoordinatorLBPeer} <−> {
  PriorityQueue <BaseEventType> pending_requests;
  initial state Idle {
      during:
        on (IStateCoordinatorLBPeer.e_StartedLB_UnderUtilized){
          IStateCoordinatorLBPeer.e_LockRequestByLB = new LockRequest(){ target = 'StateCoordinatorASPeer';};
          pending_requests.add(IStateCoordinatorLBPeer.e_LockRequestByLB );
          send IStateCoordinatorLBPeer.e_LockRequestByLB;
          send IStateCoordinatorLBPeer.e_LB_CoordinationCMD = new CoordinationCommand(){
              command = 'suspend'; target = 'WrappedLoadBalancer'; };
        } −> Waiting;
        on(IStateCoordinatorLBPeer.e_LockRequestByAS){
          pending_requests.add(IStateCoordinatorLBPeer.e_LockRequestByAS);
          send IStateCoordinatorLBPeer.e_LockReplyByLB = new LockReply(){ target = 'StateCoordinatorASPeer'; };
        }
        on (IStateCoordinatorLBPeer.e_LockReleaseByAS){ pending_requests.remove (); }
        on (IStateCoordinatorLBPeer.e_TerminatedLB)−> AlwaysGrant;
  }
  state AlwaysGrant {
      during:
         on(IStateCoordinatorLBPeer.e_LockRequestByAS){
            pending_requests.add(IStateCoordinatorLBPeer.e_LockRequestByAS);
          send IStateCoordinatorLBPeer.e_LockReplyByLB = new LockReply(){ target = 'StateCoordinatorASPeer'; };
        }
        on (IStateCoordinatorLBPeer.e_LockReleaseByAS){ pending_requests.remove (); }
        on(IStateCoordinatorLBPeer.e_InitiatedLB) −> Idle;
  }
  state Waiting {
      during:
        on(IStateCoordinatorLBPeer.e_LockReplyByAS)[pending_requests.peek().get('publisher')=='StateCoordinatorLBPeer']
                                   −> Running;
        on(IStateCoordinatorLBPeer.e_LockReleaseByAS){
          pending_requests.remove ();
          pending_requests.peek().get('publisher')=='StateCoordinatorLBPeer' −> Running;
        }
          on(IStateCoordinatorLBPeer.e_LockRequestByAS){
            pending_requests.add(IStateCoordinatorLBPeer.e_LockRequestByAS);
          send IStateCoordinatorLBPeer.e_LockReplyByLB = new LockReply(){ target = 'StateCoordinatorASPeer'; };
        }
  }
  state Running {
      entry:
        send IStateCoordinatorLBPeer.e_LB_CoordinationCMD = new CoordinationCommand(){
            command = 'proceed'; target = 'WrappedLoadBalancer'; };
      during:
        on (IStateCoordinatorLBPeer.e_EndedLB) −> Idle;
        on(IStateCoordinatorLBPeer.e_LockRequestByAS){
          pending_requests.add(IStateCoordinatorLBPeer.e_LockRequestByAS);
          send IStateCoordinatorLBPeer.e_LockReplyByLB = new LockReply(){ target = 'StateCoordinatorASPeer'; };
        }
      exit:
        pending_requests.remove ();
        send IStateCoordinatorLBPeer.e_LockReleaseLB = new LockRelease(){ target = 'StateCoordinatorASPeer';};
  }
}
interface IStateCoordinatorLBPeer {
  requires {
    private ConstituentState e_StartedLB_UnderUtilized = {E | E instanceof 'ConstituentState' && E.publisher == 'WrappedLoadBalancer' &&
                                    E.applicationState == 'StartExecuting' && E.serverLoad == 'UnderUtilized'}
    private ConstituentState e_EndedLB = {E | E instanceof 'ConstituentState' && E.applicationState == 'EndExecuting'}
    LockRequest e_LockRequestByAS = { E | E instanceof 'LockRequest' && E.publisher == 'StateCoordinatorASPeer'}
    LockRelease e_LockReleaseByAS = { E | E instanceof 'LockRelease' && E.publisher == 'StateCoordinatorASPeer'}
    LockReply e_LockReplyByAS = { E | E instanceof 'LockReply' && E.publisher == 'StateCoordinatorASPeer'}
    ApplicationTerminated e_TerminatedLB = { E | E instanceof 'ApplicationTerminated' && E.publisher == 'WrappedLoadBalancer'}
    ApplicationStarted e_InitiatedLB = { E | E instanceof 'ApplicationStarted' && E.publisher == 'WrappedLoadBalancer'}
  }
  provides {
    private CoordinationCommand e_LB_CoordinationCMD;
    LockRequest e_LockRequestByLB;
    LockReply e_LockReplyByLB;
    LockRelease e_LockReleaseLB;
  }
}
```

Listing 10: The specifications of the state coordinator for the peer-to-peer pattern

8 Implementation of Compiler

8.1 The Compiler of EventArch

To extend applications with event-based interfaces(facades), we face several design decisions, such as: a) How to extract events from applications? b) How to send events to applications? c) How to integrate event-based interfaces with the conventional functional interfaces of applications? d) How to process an event if there are `on` and `wait when...until` expressions defined for the event? e) What if an application is no longer available to exchange events?

In this section, we explain our answers to these questions by providing more details about the compiler of the EventArch language and its event processing semantics.

The compiler receives a configuration file as input, in which the name and the path of the actual application components that are wrapped by AEMs are provided. As shown in Listing 11, the name and path of the specification files are defined in the configuration file. In addition, the following information is specified in for each AEM: a) The name of the application, which is referred to in the reactor part of the corresponding event module, b) the main method of the application, which can be invoked to start executing the application, and c) the path of the application files that are wrapped by the event module.

```
<config>
        <specifications>
          <specification path="./" file="eventtypes.er"/>
          <specification path="./" file="eventmodules.er"/>
        </specifications>
        <applications>
      <application name = "AdaptiveSoftware" mainclass="Runner" method="main">
        <files>
              <entry path = "./application/" file="adaptivesoftware.jar"/>
              <entry path = "./application/" file="AppComponents.jar"/>
            </files>
      </application>
        ...
    </applications>
</config>
```

Listing 11: The specification of configuration file

The compiler also receives the specifications as input, and generates executable code for the AEMs. Figure 6 abstractly shows the set of generated classes and their runtime interactions. Each AEM is executed in a separate process as shown by the gray rectangle. The processes make use of Java Message Service (JMS) [37] to exchange events among each other. There is a logical clock to synchronize the processes and to keep track of the timestamp of events. The *Runtime Manager* object is the core part of EventArch runtime environment, which receives JMS messages and translates them to event. It also receives the events representing that an AEM started or finished its execution. These events are of the predefined types *ApplicationStarted* and *ApplicationTerminated*, and are provided to other AEMs.

For each AEM, the compiler generates the so-called *AEM Executor* object, which has a local event queue to maintain the JMS messages that are sent to the AEM. The events are inserted in this queue by *Runtime Manager*. The

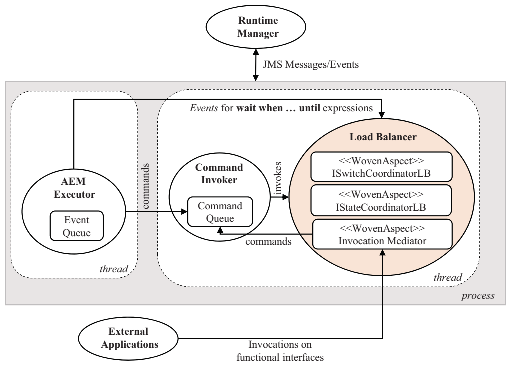
Runtime Manager
JMS Messages/Events
Events for wait when ... until expressions
Load Balancer
<<WovenAspect>> ISwitchCoordinatorLB
<<WovenAspect>> IStateCoordinatorLB
<<WovenAspect>> Invocation Mediator
AEM Executor
Event Queue
commands
Command Invoker
Command Queue
invokes
commands
thread
thread
process
External Applications
Invocations on functional interfaces

Figure 6: The runtime view of primitive AEMs

Command design pattern [10] is adopted to let the wrapped applications receive and process events. If an event matches any selector of the AEM and if there is at least one *on* expression for that event, the method specified by the *on* expression must be invoked to process the event. To this aim, *AEM Executor* translates the event to a command, and stores it in the command queue of the *Command Invoker* object.

Command Invoker is also generated by the EventArch compiler, and is executed in the same thread as the wrapped application; it processes the commands by invoking the method specified in the *on* expression.

As shown by the white rectangles, the compiler also generates an aspect in AspectJ or AspectC++ for each interface, depending on the language mentioned in the specifications. The aspect implements the functionality to publish the events that are specified as the provided interface of the corresponding AEM. If there is a *wait when . . . until* expression for an event, the aspect also implements body of the *wait when . . . until* expression.

Listing 12 shows an excerpt of the AspectJ code generated for the interface `IStateCoordinatorAS` of Listing 3 . In the constructor of the aspect, necessary initialization to work with JMS is performed. The event expression `e_StartedAS` in lines 19–22 of Listing 3 is translated to a pointcut and advice in the aspect. The code for publishing the event is defined in lines 6–9 of the advice code; an instance of the class `ConstituentState` is created to represent the event, its attributes are initialized, and the event is published to the runtime manager of EventArch by invoking the method `publish`. The specification in Listing 3 defines a `wait when ...until` expression for the event, which is translated to the code in lines 11–25 of the advice. Here, information about the required events is retrieved in the list `waits`. If this list is not empty, and there is any event in queue that matches any of the required events, the body of the `wait when ...until` is executed.

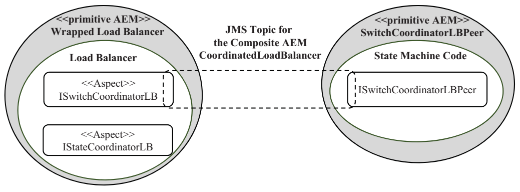

Figure 7: The runtime view of composite AEMs

```
public aspect IStateCoordinatorASWrappedAdaptiveSoftware{
    //initializations ...
  }
 pointcut e_StartedASPC(): execution (void org.ApplicationComponent.execute(..));
 void around(): e_StartedASPC() {
          ConstituentState e_StartedAS = new ConstituentState();
          e_StartedAS.applicationState= "StartExecuting";
          e_StartedAS.target= "StateCoordinator";
          EventReactor.publish(e_StartedAS);
          BaseEventType waits = module.waitOn(e_StartedAS);
       if(waits != null){
          boolean proceedexe = false;
          while(!proceedexe){
                 BaseEventType ev = queue.retrieve (waits);
                 if (ev == null) continue;
                 if (ev.get("command") != null &&
                      ev.get("command") == CoordinationCommands.proceed){
                          proceedexe=true; break;
                      }
                 if (ev.get("command") != null &&
                      ev.get("command") == CoordinationCommands.suspend)) continue;
               if (ev.get("command") != null &&
                      ev.get("command") == CoordinationCommands.retry))
                      EventReactor.send(e_StartedAS);
          }//while
          if (proceedexe) proceed(p);
        }//if
  }

}
```

Listing 12: The generated aspect for *WrappedAdaptiveSoftware*

In addition to our event-based interfaces, the wrapped application offers functional interfaces to let its services be invoked by other applications. Such invocations must also be translated to commands, which are sent to *Command Invoker*. The EventArch compiler generates the so-called *Invocation Mediator* aspect, which intercepts the invocations to the functional services, translates them to commands, and stores them in the command queue of *Command Invoker*.

Composite AEMs facilitate reducing the visibility of the events between its interfaces and reactor. We make use of JMS topics to implement this feature; in the topic-based communication, event consumers register interest in receiving messages (events) on a particular message topic. As shown in Figure 7, each composite AEM is translated to a message topic with the same name, which are used by its sub-modules to communicate with each other. To impose information hiding within a composite AEM, composite reactors can only communicate with this topic, whereas composite interfaces can communicate with other topics to be able to send events to external AEMs.

8.2 The Event Processing Semantics of EventArch

Algorithm 1 abstractly represents the event processing performed by *AEM Executor*. As the initial activity, it starts the corresponding *Command Invoker* and the wrapped application. As long as the application is not terminated, for example due to unchecked exceptions, it processes the events in a FIFO manner. If *event* matches any required interface the AEM, *AEM Executor* first checks whether there is any *wait when ... until* expression currently waiting for *event*. This means that the thread executing the wrapped application is blocked to receive *event*. In this case, *AEM Executor* forwards the event to the corresponding aspect in the wrapped application. Afterwards, for each *on* expression defined for *event*, it creates a command and inserts the command in the queue of *Command Invoker*.

Algorithm 1 Event selection and forwarding

```
1: procedure EventSelectionForwarding
2:   start the $CommandInvoker$ entity and the wrapped application
3:   publish and event of the type $ApplicationStarted$
4:   while the application is alive do
5:     let $event$ be the event at the front of the event queue
6:     if $event$ and its visibility matches any required interface then
7:       let $wait$ be the current wait expression for $event$
8:       if $wait$ <> null then
9:         forward $event$ to the corresponding $wait$ expression
10:      for $expr$ in the selected $on$ expressions do
11:        let $command$ be a new command for $event$
12:        insert $command$ in the command queue
13:  publish an event of the type $ApplicationTerminated$
```

Algorithm 2 abstractly represents the event processing performed by *Command Invoker* and the aspects woven into the wrapped application. *Command Invoker* processes the commands in its queue in a FIFO manner by invoking the corresponding method on the wrapped application. Since *Command Invoker* and the wrapped application are executed in one thread, *Command Invoker* will be blocked until the execution of *command* terminates by the wrapped application.

While processing *command*, the pointcut expressions in the event-based interfaces may match a join point, which results in publishing *newEvent* to *Runtime Manager*. After publishing *newEvent*, the aspect checks whether there is any *wait when ... until* expression defined for *newEvent*. If it is the case, it enters a busy-loop until it receives an event from *AEM Executor* to exit the loop. Alternatively, the busy-loop terminates if no event is received after the specified threshold T, which can be specified by the users. The execution of the method continues normally afterwards.

Algorithm 2 Event processing in applications

```
 1: procedure EVENTPROCESSING
 2:   while the application is alive do
 3:     let command be the command at the front of the queue
 4:     start processing command by invoking the method
 5:     while the method is not terminated do
 6:       if new event is produced then
 7:         let newEvent be the new event
 8:         publish newEvent as a private or public event
 9:         if any wait expression exists for newEvent then
10:           while True or Timeout do
11:             let inputEvent be the event sent by AEM Executor
12:             if inputEvent.command == Proceed then
13:               break
14:             if inputEvent.command == Suspend then
15:               continue
16:             if inputEvent.command == Retry then
17:               publish newEvent with a new timestamp
18:       continue processing command
```

9 Discussions

AEMs and their implementation in the EventArch language fulfill the requirements outlined in Section 3 in the following ways.

REQ (1)–Supporting various coordination patterns: Adopting an event-based communication mechanism helps to keep AEMs loosely coupled to each other. As a result, the architecture of an SoS can flexibly be changed by switching between different patterns.The degree to which the specifications must be tailored depends on the implementation of coordination rules and their modularization. For example, the implementation of the switch coordinator in Listing 8 is fairly similar to the one in Listing 4 except that the visibility of events is adjusted. The same is valid for the specification of primitive AEMs, which required us to adjust the visibility of their interfaces for the centralized and peer-to-peer cases. As shown in Listings 5 and 10, we may need to largely adjust a module if its logic differs based on the adopted coordination pattern.

REQ (2)-Modularizing and composing coordination rules: Constituent applications and coordination rules can uniformly be defined as the modules of SoS's via AEMs. This uniformity increases the compositionality of SoS's because the coordination rules can be treated as normal applications and can be composed further with other AEMs.

Currently, there are two main trends in supporting aspects at the architectural level: packaging the aspect with the corresponding base component [34], or supporting them as standalone components [33]. Our proposal supports the first case via composite AEMs, which define crosscutting coordination rules via their composite interfaces. The latter case is supported by separating coordination rules, and applying them based on the centralized pattern.

REQ (3)-Specifying and modularizing coordination-specific interfaces: We consider events as the basic abstractions to represent the state changes of interest in the system. Unlike most AO languages that fix the set of supported join points, we allow new events be defined depending on application requirements. Events can be sent to specific AEMs, or be broadcast to all modules. Events can be collected from multiple AEMs; event queries can be defined based on the contents of events as well as their type.

As Listing 6 shows, the specification of primitive interfaces is modularized from the actual application components. The reusability of both primitive interfaces and the wrapped applications increases due to this separation. For example, an application can be reused as the constituent of different SoS's by defining new events and mapping them to the desired state changes in the application. Naturally, changes in the signature of the methods in applications impact the interfaces; this is known as the "fragile pointcut problem", which is studied in the aspect-oriented community [16]. Nevertheless, as long as the changes do not impact event names or event attributes, they will not affect the specification of other AEMs.

Since EventArch makes use of existing AO languages in its back-end, the expression power of its pointcut definition language is influenced by the adopted AO languages. To let applications participate within the context of an SoS, we assume that they must expose some information publicly. The join point model of current AO languages support public interfaces of classes, which we consider it sufficient for our purpose.

REQ (4)-Supporting heterogeneity in implementation languages: In the back-end of EventArch, we make use of JMS publish/subscribe middleware; in addition, inspired from XPIs, we adopt aspects to define crosscutting interfaces. Where one could directly uses these languages and middleware to composed multiple applications with each other, the EventArch language facilitates defining the architecture of SoS's at a higher level of abstraction, in a declarative way and independently from the implementation languages of wrapped applications. These characteristics pave the way to perform various kinds of analysis on the specifications of SoS's; for example, to reason about the impacts of coordination rules on the control flow of constituent applications.

The EventArch language currently supports applications developed in the Java or C++ languages. New languages can be supported by plugging suitable AO compilers in the EventArch compiler so that the necessary aspect code can be generated from the specifications of primitive interfaces.

10 Positioning at the Related Work

As Figure 2 shows, AEMs combine some features of the previously-evaluated techniques towards fulfilling the requirements outlined in Section 3. Inspired from the system modeling languages, constituent applications and their coordination rules are uniformly represented via AEMs as the modules of an SoS. Events are means to represent the state changes of interest in constituent applications and/or in coordination rules; hence, they are analogous to join points in AO languages. AEMs are analogous to aspects; the required interface, the reactor and the provided interface of AEMs are analogous to pointcut designators, advice code and the join points within aspects, respectively.

Higher-level AEMs can select the events that are published by multiple lower-level AEMs. Therefore, higher-level modules can be adopted to modularize the concerns that crosscut multiple lower-level modules; e.g. coordination rules. In this case, the event-based interfaces of the lower-level modules define crosscutting coordination-specific interfaces for the wrapped applications. AEMs also adhere to the publish/subscribe paradigm because modules are decoupled from each other, and make use of an event-based mechanism to communicate with each other. An event may be published to a specific module, or it may be broadcast to all available modules.

In our previous work [24], we introduced *object-level event modules* as means to modularize domain-specific concerns, which may crosscut multiple components that are implemented in different languages. Object-level event modules and architectural event modules have the concept of 'event modules' in common: a module is defined as a group of required and provided events that are processed by reactors. As for objects in OO and aspects in AO, which can be supported at programming language and architectural levels, the concept of event modules can also be supported at both levels. Our proposal for object-level event modules and its implementation in the EventReactor language is at the level of programming languages.

To support the requirements outlined in this paper, the syntax, semantics and compiler of the EventArch language is largely different from the EventReactor language in the following ways: 1) Three languages Java, C++ and state machines are supported, 2) A complex event-based interface specification language is offered, which is modularized from base applications, 3) Legacy applications are extended with new interfaces to let them participate in coordinated interactions, 4) EventArch supports distributed applications, and 5) EventArch facilitates unify representation of base applications and coordination rules as architectural event modules.

Composite AEMs differ from existing composite components [32] in the following ways. Firstly, they have pure event-based interfaces, which help to support one-to-one, one-to-many and many-to-many styles of communication. Secondly, they are means to define two levels of facades for applications. The first level is a means to define language-independent event-based interfaces for the applications to let them participate in a coordinated interaction. The second level is a means to modulary define complex coordination rules for the applications. Such facades increase the reusability of modules and paves the way to flexibly change the architecture of an SoS.

Primitive and composite AEMs differ from other proposals for (modular) event-driven architectures [9, 36, 21] in the following ways: a) augmenting applications with modularized event-based interfaces, b) supporting applications implemented in languages, and c) declarative languages for defining AEMs and coordination rules.

11 Future Work

Integrating coordination-specific interfaces with base applications: The noticeable challenge in developing AEMs was to integrate coordination-specific interfaces with existing functionality of the applications. This issue can be generalized further: To reuse an application as the constituent of an SoS, we require means to represent the SoS-specific behavior of the applications, understand the interplays between such behavior and the base functionality of the applications, provide means to integrate these two with each other in a modular way, and validate the correctness of the integrated behavior. These issues open many interesting research challenges in defining the architecture of SoS's.

In this report, we mainly offered means to *modularly* extend applications with SoS's specific coordination rules. Currently, the SoS-specific behavior of applications is defined via event-based interfaces of AEMs, and the Command pattern and the *Interceptor Aspect* are adopted to integrate the SoS-specific behavior with the base functionality of the applications. In future, we would like to study the suitability of this solution in different applications such as robotic applications.

Heterogeneous kinds of constituents: SoS's may consist of various kinds of constituents, such as applications, end users, hardware devices, and sensors. Although this report only focused on applications as the constituents of SoS's, we claim that our approach can be generalized to support other kinds of constituents too. Events are suitable means to abstract necessary information about the behavior of such constituents, and EventArch is open-ended with new types of events that can be published from different sources. The events that are published to EventArch can be referred to and processed by AEMs in a similar way as application-specific events. A preliminary example is the events *ApplicationStarted* and *ApplicationTerminated*, which are in principle generated by OS or VM.

Cloud-based AEMs: The event-based interfaces of AEMs let them be loosely coupled and communicate with each other in an asynchronous manner. This gives us the flexibility to explore different deployment options (e.g. on the cloud) for AEMs. We would like to study the suitability of AEMs for modular and efficient could-based SoS's in future. We will particularly study means to extend EventArch with the middleware that facilitate efficient and reliable event/data stream processing on the cloud [14].

Different types of SoS: In the literature, five types of SoS's are identified, which differ from each other based on the degree to which constituents receive control from an SoS [1]. Our proposal for centralized coordination pattern is suitable for the so-called directed SoS, in which constituents can operate independently, but within the SoS they accept some central management to ensure that SoS-level goals are met. Our proposal for the peer-to-peer pattern may be suitable for the so-called acknowledged or coordinated SoS, where such a centralized management is not desirable. We would like to extend our study along this line, and evaluate the suitability of AEMs for various types of SoS's.

12 Conclusions

Due to the inherent complexity of today's software systems, they are usually developed as SoS's in which existing applications are reused as the constituents. This raises the need for coordinating the interplays of the applications toward fulfilling desired system-level requirements. To keep the applications reusable, we require means to define necessary coordination rules and coordination-specific interfaces in a modular way.

This report outlined four requirements that an ADL must fulfill for a modular definition of coordination-specific interfaces and coordination rules. We evaluated a large set of related work, and discussed that they fall short of fulfilling these requirements. We proposed AEMs as a special kind of component to uniformly and modularly represent constituent applications and desired coordination rules in SoS's. AEMs define two layers of event-based facades for the applications, which enable applying coordination rules based on the centralized, peer-to-peer and/or hybrid patterns.

References

[1] COMPASS Project. http://www.compass-research.eu/.

[2] System Modeling Language. http://www.sysml.org/.

[3] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In *the Workshop on Object-Based Distributed Programming*. Springer-Verlag, 1994.

[4] J. Aldrich. Open Modules: Modular Reasoning About Advice. In *Proceedings of ECOOP'05*. Springer-Verlag, 2005.

[5] J. Boardman and B. Sauser. System of Systems - the meaning of of. In *Proceedings of SoSE '06*. IEEE Press, 2006.

[6] J. Boldt. The Common Object Request Broker: Architecture and Specification. Technical report, OMG, 1995.

[7] C. Chen and J. Purtilo. Configuration-level Programming of Distributed Applications Using Implicit Invocation. In *Proceedings of TENCON '94*, 1994.

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[9] L. Fiege, M. Mezini, G. Mhl, and A. Buchmann. Engineering Event-Based Systems with Scopes. In *Proceedings of ECOOP '02*. Springer Berlin Heidelberg, 2002.

[10] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[11] A. Garcia, C. Chavez, T. Batista, C. Santanna, U. Kulesza, A. Rashid, and C. Lucena. On the Modular Representation of Architectural Aspects. In *Software Architecture*, volume 4344. Springer Berlin Heidelberg, 2006.

[12] D. Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, Jan. 1985.

[13] S. Gotz, C. Wilke, S. Richly, and U. Abmann. Approximating Quality Contracts for Energy Auto-tuning Software. In *Proceedings of GREENS '12*, 2012.

[14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, 2012.

[15] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service. Technical report, Sun Microsystems, April 2002.

[16] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the Evolution of Aspect-oriented Software with Model-based Pointcuts. In *Proceedings of ECOOP '06*. Springer-Verlag, 2006.

[17] H. Kopetz, B. Fromel, and O. Hoftberger. Direct versus stigmergic information flow in systems-of-systems. In *Proceedings of SoSE '15*. IEEE Press, 2015.

[18] V. Kotov. Communicating Structures for Modeling Large-scale Systems. In *Simulation '98*. IEEE Press, 1998.

[19] A. D. Kshemkalyani and C. Singhal. *Distributed Computing Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

[20] B. Lagaisse and W. Joosen. True and transparent distributed composition of aspect-components. In *Proceedings of Middleware'06*. Springer-Verlag, 2006.

[21] D. C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Technical report, 1996.

[22] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *Software Engineering Journal*, 8:73–82, 1993.

[23] S. Malakuti. Detecting Emergent Interference in Integration of Multiple Self-Adaptive Systems. In *Proceedings of SESoS ' 14*. ACM, 2014.

[24] S. Malakuti and M. Aksit. Event Modules - Modularizing Domain-Specific Crosscutting RV Concerns. *T. Aspect-Oriented Software Development*, pages 27–69, 2014.

[25] S. Malakuti and M. Zia. Adopting Architectural Event Modules for Modular Coordination of Multiple Applications. Technical report, Technical University of Dresden, 2015.

[26] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proceedings of AOSD '03*. ACM Press, 2003.

[27] R. Mondjar, P. Garca-Lpez, C. Pairot, and L. Pamies-Juarez. Damon: A Distributed {AOP} Middleware for Large-scale Scenarios. *Information and Software Technology*, 54(3):317 – 330, 2012.

[28] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly Distributed AOP Using AWED. In *Proceedings of AOSD '06*. ACM, 2006.

[29] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *Proceedings of AOSD '04*. ACM, 2004.

[30] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *ADVANCES IN COMPUTERS*, pages 329–400. Academic Press, 1998.

[31] E. Parallel and E. G. Wilson. Linda-Like Systems and Their Implementation, 1991.

[32] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A Model for Developing Component-Based and Aspect-Oriented Systems. In *Proceedings of Software Composition*, volume 4089. Springer Berlin Heidelberg, 2006.

[33] M. Pinto, L. Fuentes, and J. M. Troya. Specifying aspect-oriented architectures in AO-ADL. *Information and Software Technology*, 53(11):1165 – 1182, 2011.

[34] J. Prez, N. Ali, J. Cars, and I. Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In *Proceedings of CBSE '06*. Springer Berlin Heidelberg, 2006.

[35] H. Rajan and K. Sullivan. Eos: Instance-Level Aspects for Integrated System Design. In *Proceedings of ESEC/FSE-11*. ACM Press, 2003.

[36] C. Rathfelder, B. Klatt, K. Sachs, and S. Kounev. Modeling Event-based Communication in Component-based Software Architectures for Performance Predictions. *Software Systems Modeling*, 13(4), 2014.

[37] M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java Message Service*. O'Reilly Media, 2009.

[38] K. Rybina, W. Dargie, R. Schoene, and S. Malakuti. Mutual Influence of Application- and Platform-Level Adaptations on Energy-Efficient Computing. In *Proceedings of PDP '15*. IEEE Press, 2015.

[39] I. Sommerville and G. Dean. PCL: a Language for Modelling Evolving System Architectures. *Software Engineering Journal*, 11(2):111–121, Mar 1996.

[40] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Trans. Softw. Eng. Methodol.*, 20:1:1–1:43, July 2010.

[41] K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular Aspect-oriented Design with XPIs. *ACM Trans. Softw. Eng. Methodol.*, 20(2):5:1–5:42, Sept. 2010.

[42] D. Suvée, B. De Fraine, and W. Vanderperren. A Symmetric and Unified Approach Towards Combining Aspect-oriented and Component-based Software Development. In *Proceedings of CBSE'06*. Springer-Verlag, 2006.

[43] D. Suvée, W. Vanderperren, D. Wagelaar, and V. Jonckers. There Are No Aspects. *Electron. Notes Theor. Comput. Sci.*, 114:153–174, Jan. 2005.

[44] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: A Formal Modelling Language for Systems of Systems. In *Proceedings of SoSE '12*. IEEE Press, 2012.

A Appendix: Abstract Syntax of EventArch

- <SoS> ::= (<EventType> | <PrimitiveAEM> | <CompositeAEM>)*

- <EventType> ::= <EventTypeName> (‘**extends’** <EventTypeName>)? (<AttributeType> <AttributeName>)*
- <AttributeType> ::= … <!-- a valid type in the language -->

- <PrimitiveAEM> ::= <ModuleName> <PrimitiveInterface>+ <PrimitiveReactor>

- <PrimitiveReactor> ::= <ApplicationLiteral> | <StateMachine>
- <ApplicationLiteral> ::= … <!– a string literal referring to the application name in the configuration file -->
- <PrimitiveInterface> ::= (**‘private’**)? <InterfaceName> <RequiredEvents> <ProvidedEvents>

- <RequiredEvents> ::= (<SelectorExpression> | <OnExpression>)*
- <SelectorExpression> ::= (**‘private’**)? <EventTypeName> <SelectorName> <EventQuery>
- <EventQuery> ::= … <!-- Boolean expressions over even attributes -->
- <OnExpression> ::= ‘**on**’ <SelectorName> ‘**invoke**’ <Class> <Method> <Arguments>
- <Class> ::= … <!-- a string literal denoting a class name -->
- <Method> ::= … <!-- a string literal denoting a method name -->
- <Arguments> ::= … <!-- a string literal or <SelectorName> -->

- <ProvidedEvents> ::= (<EventDef> | <WaitExpression>)*
- <EventDef> ::= (**‘private’**)? <EventTypeName> <EventName> (<PointcutExpression> (‘**&&**’ | ‘||’) <PointcutExpression>)*
  <ConditionalExpression>? <Initialization>)?
- <PointcutExpression> ::= (‘**before**’ | ‘**after**’) (‘**execution**’ | ‘**call**’) <MethodPattern>
- <ConditionalExpression> ::= … <!-- conditional statement over method arguments -->
- <Initialization> ::= … <!--initializing the attributes of the event before publishing -->
- <WaitExpression> ::= ‘**wait when**’ <EventName> ‘**until**’ <SelectorName>+ <Condition> <ControlCommand>
- <MethodPattern> ::= … <!-- a language-independent specification of methods signature -->
- <Condition> ::= … <!-- conditional statements over the attributes of <SelectorName> -->
- <ControlCommand> ::= **‘retry’** | **‘proceed’** | **‘suspend’**

- <CompositeAEM> ::= <ModuleName> <CompositeInterface>+ <CompositeReactor>
- <CompositeInterface> ::= <ModuleName> <!-- refers to a primitive AEM; the primitive module can only be part of one composite AEM -->
- <CompositeReactor> ::= <ModuleName> <!-- refers to a primitive AEM; the primitive module can only be part of one composite AEM -->

- <StateMachine> ::= <StateVariable>* <InitialState> <State>*
- <StateVariable> ::= <VariableType> <VariableName>
- <VariableType> ::= <!-- a valid type in the language -->

- <InitialState> ::= **‘initial’** <State>
- <State> ::=   **'state'** <StateName> (**'entry'** (<Action>)+)?
  (**'during'** (<StateMachineOnExpression>)+)?
  (**'exit'** (<Action>)+)?
- <Action> ::= <Print> | <VariableManipulation> | <ConditionalTransition> | <EventCreation> | <EventPublishing>
- <StateMachineOnExpression> ::= **'on'** <SelectorName> <TransitionCondition>? (<Action> <Transition>?)?
- <Print> ::=‘**print**’ … <!-- a string literal -->
- <VariableManipulation> ::= … <!– changing variable values -->
- <ConditionalTransition> ::= <TransitionCondition> <Transition>
- <EventCreation> ::= <EventName> **'=' 'new'** <TypeName> (<Initialization>)?
- <EventPublishing> ::= **'send'** <EventName> (**'=' 'new'** <TypeName> (<Initialization>)?)?
- <TransitionCondition> ::= … <!– Boolean expressions over state variables -->
- <Transition> ::= **'->‘ <StateName>**

- <!– all names in the rules are defined via valid identifiers in the language>