TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211XX

TUD-FI15-02-Juli 2015

Somayeh Malakuti, Mariam Zia
Software Technology group

Adopting Architectural Event Modules for Modular Coordination of Multiple Applications

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

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Adopting Architectural Event Modules for Modular Coordination of Multiple Applications

Somayeh Malakuti and Mariam Zia
Software Technology group
Technical University of Dresden, Germany
somayeh.malakuti,mariam.zia@tu-dresden.de

Abstract

Nowadays, large-scale software systems consist of multiple applications, which interact with each other to fulfill desired system-level requirements. It is usually required to coordinate the interactions of the constituent applications to ensure that the system-level requirements are fulfilled. In this paper, we outline a set of requirements that must be fulfilled to facilitate the modular composition of multiple applications. We introduce the concept of **architectural event modules**, which are abstractions to represent constituent applications and their coordination logic in a modular and uniform way. We explain the implementation of this concept in the EventReactor language, and define their formal semantics in processing events using the UPPAAL toolset. We illustrate the suitability of architectural event modules in achieving modularity and loose coupling in the composition of multiple applications by means of a case study in the domain of energy-efficient computing.

1 Introduction

Nowadays, large-scale software systems consist of multiple applications, which interact with each other towards fulfilling desired system-level requirements [17, 16]. 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 [10, 15, 27]. Therefore, it is necessary to constrain and coordinate the interactions of the constituent applications.

Constituent applications and their coordination logic can be seen as the modules of a complex software system. To be able to effectively compose the constituent applications with each other and to coordinate their interactions, we claim that a module system that fulfills the following requirements is needed: a) specifying and modularizing the so-called crosscutting coordination-specific interfaces for constituent applications; b) modularizing coordination logic from the constituent applications; c) supporting the distribution of the constituent

applications and their coordination logic; d) uniform representation of the constituent applications and their coordination logic, so that the coordination logic can further be composed with other concerns in the software; and e) supporting heterogeneity in the implementation language of the constituent applications and the coordination logic.

We evaluate a large set of programming, architectural and coordination languages with respect to these requirements, and outline their shortcomings. We introduce the concept of **architectural event modules**, which are means to represent constituent applications and their coordination logic in a modular and uniform way. Architectural event modules have event-based interfaces; events are means to define the flow of control and data among the applications and the modules implementing the coordination logic. Consequently, loose coupling is achieved in the compositions.

We explain the EventReactor language, which offers a set of declarative abstractions to program architectural event modules. The suitability of this language in achieving modularity and loose coupling in the composition of multiple applications is illustrated by means of a case study in the domain of energy-efficient computing. We define the formal semantics of architectural event modules using the UPPAAL [3] toolset.

This paper is organized as follows: Section 2 explains our illustrative case study. Section 3 outlines a set of requirements that must be fulfilled in composing multiple applications. Section 4 identifies the shortcomings of the current languages and frameworks in fulfilling these requirements. Section 5 explains the concept of architectural event modules. Section 6 discusses the EventReactor language, and the implementation of our illustrative case study in this language. Section 7 depicts the formal semantics of architectural event modules. Section 8 discusses the suitability of architectural event modules in improving modularity of implementations. Section 9 explains related work, and finally Section 10 outlines conclusion and future work.

2 Illustrative Example

Assume for example that we have two energy optimization techniques, which operate at the level of application software [12] and virtual machines (VM). These two are referred to as adaptive software and load balancer in this paper, respectively.

The components of the adaptive software are shown in Figure 1 within the VM (1). There are multiple implementations for 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. To support runtime energy optimization, four components named as *Monitor*, *Analyzer*, *Planner* and *Configurator* are defined.

When a user issues a request to the application, the request triggers the *Monitor* component. The *Monitor* and *Analyzer* components observe and ana-

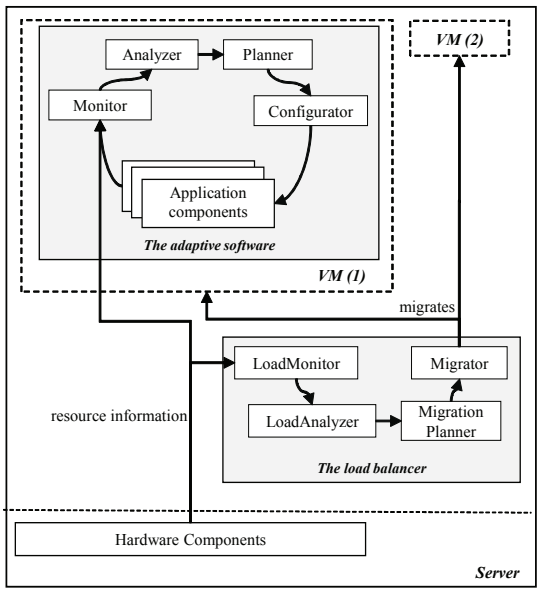

Figure 1: The adaptive software and the load balancer

lyze the availability of the hardware resources on the server, respectively. The *Planner* component selects the best implementation of each *Application* component, which offers the best performance and energy consumption based on the available resources. The *Configurator* component reconfigures the application to execute with the selected implementations. After the reconfiguration, the user's request is served by the *Application* components.

As Figure 1 shows, the load balancer also consists of four components, which are executed sequentially. These components monitor and analyze the load of the server at predefined intervals, plan for migrating some VMs to another server, and perform the migration for balancing the load of the server.

If the adaptive software and the load balancer are composed as one software system, various kinds of interplays may emerge as the result of such composition [27, 41]. For example, if during the reconfiguration of the application components, VM (1) is migrated to a new server, the selection performed by the *Planner* component is no longer valid, because the set of available resources on the target server differs from the source server. Besides, the required time for the migration adds to the required time for serving the user's request. This may violate the quality of the service that the adaptive software has guaranteed to give to the user.

To avoid such cases, we would like to coordinate the adaptive software and the load balancer in the following way to ensure their mutual execution. If a user's request is being served by the adaptive software, and the load balancer components want to start executing at their regular intervals, their execution

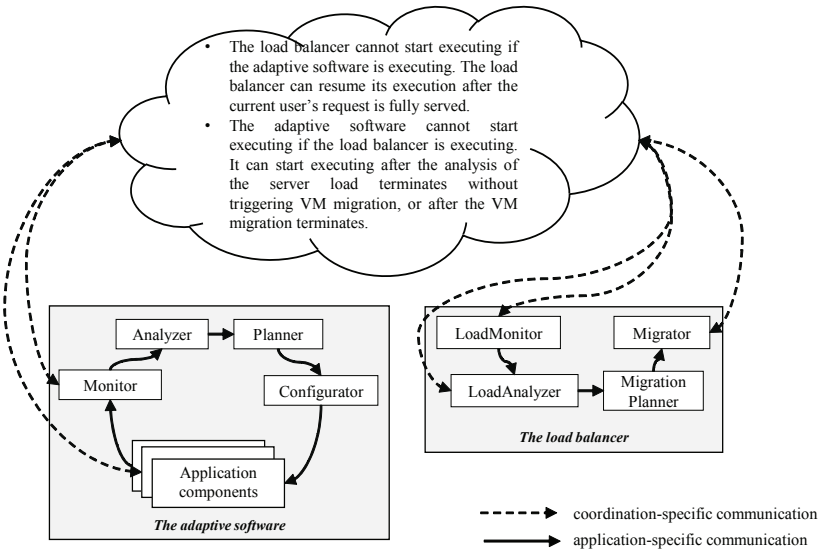

Figure 2: Coordinating the adaptive software and the load balancer

must be suspended until the adaptive software finishes serving the user's request. Likewise, if the load balancer components are already executing, the adaptive software must not start serving new users' requests[1].

3 Requirements

Constituent applications can be regarded as the modules of a software system, which interact with each other to fulfill desired system-level requirements. To be able to effectively compose the constituent applications with each other and to coordinate their interactions, we claim that a module system is required that fulfills the following requirements:

Specifying and modularizing crosscutting coordination-specific interfaces: Constituent applications must provide the so-called coordination-specific interfaces, so that they can be composed and coordinated within the context of a software system. 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 and information among the applications. Depending on the adopted coordination logic, it may be necessary to gather the necessary information from multiple application components. This implies that coordination-specific interfaces may *crosscut* multiple application components.

[1]Other coordination logic may also be considered; the focus of this paper is not on specific logic.

An example is shown in Figure 2 for our illustrative case study. Here, before the *Monitor* component of the adaptive software starts executing, we would like to check whether the load balancer has started executing at its regular intervals. If so, the execution of *Monitor* must be suspended until the *LoadAnalyzer* or *Migrator* components finish executing. Similarly, before the *LoadMonitor* component starts executing, we would like to check whether the adaptive software is already executing, and suspend the execution of *LoadMonitor* if needed. For each of these cases, we need to gather information about the execution states of multiple components.

Regardless of whether applications are used independently or are reused as the constituents of a software system, 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 when they are reused as the constituents of software systems. Besides, functional interfaces are usually fixed; whereas, coordination-specific interfaces may vary depending on the adopted coordination logic and the requirements of a specific software system.

Therefore, we claim that a module system must offer suitable means to define and modularize desired crosscutting coordination-specific interfaces, and to extend constituent applications with these interfaces while preserving the reusability of the constituent applications.

Modularizing inter-application crosscutting concerns: Coordination is inherently a crosscutting concern because it is related to the interactions of multiple entities (e.g. constituent applications in software systems).

The need for separating crosscutting concerns, including coordination logic, has been widely studied by the aspect-oriented (AO) community [20, 4]. 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 (crosscutting) concerns evolve.

Therefore, we claim that a module system must offer suitable abstractions to modularize coordination logic from constituent applications.

Supporting distribution of concerns: Constituent applications may execute in parallel with each other, and may be distributed in different hosts. Therefore, a module system must offer means to support distributed execution of the constituent applications and their coordination logic.

Uniform representation of concerns: It is generally accepted that adopting uniform module abstractions to represent base and crosscutting concerns increases the compositionality of applications. This is mainly because the crosscutting concerns can further be composed with other base and/or crosscutting concerns [39, 46, 45]. Within the context of complex software systems, it is also desirable to represent constituent applications and their coordination logic using uniform module abstractions, so that the coordination logic can further be composed with other concerns in the software systems, if needed.

Supporting heterogeneity in implementation languages: Since constituent applications are independently developed by different teams with dif-

ferent 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 logic may also be implemented in a language differently from the constituent applications. These imply that a module system must be able to cope with such heterogeneity in the definition and composition of modules.

4 Problem Statement

Due to the inherent crosscutting nature of coordination logic, we evaluate the suitability of current aspect-oriented (AO) techniques with respect to the requirements outlined in the previous section. In the AO terminology [19], join points refer to well-defined places in the structure or execution flow of the so-called base program. Pointcut designators are means to select the join points of interest. Advice is otherwise crosscutting code, which is bound to the pointcut designators and is executed when a specified join point is activated in the base program. In most AO languages, aspects are dedicated modules to group a set of correlated pointcut designators and advice code.

Conceptually, in an AO composition of multiple constituent applications into one software system, the constituent applications are base programs to which coordination aspects are applied. Several advances have been made in the AO research community, which to some extent address the requirements outlined in the previous section. However, to the best of our knowledge, there is no solution that meets all of the outlined requirements. In the following, we explain the shortcomings of the existing proposals in more detail.

Specifying and modularizing crosscutting coordination-specific interfaces: It has been claimed that to preserve information hiding in base programs and to control the impacts of aspects on the base programs, the interface of the base programs to the aspects must explicitly be defined [44, 5, 14, 21, 43]. The requirement for defining coordination-specific interfaces for constituent applications is in line with this claim.

In XPIs [44], AspectJ aspects are adopted to define and modularize crosscutting interfaces for base objects. Such interfaces define a set of pointcuts that are visible to aspects. In addition, they define pre- and post-conditions that must be fulfilled when the aspects are applied to the base objects. As a result, the set of join points that are visible to the aspects can be restricted, syntactic changes in the join points can be hidden from the aspects, and the changes made by the aspects can be controlled. Although the set of visible join points can be restricted, the set of contextual information that is exposed by the join points is still fixed by the join point model of AspectJ. Consequently, unnecessary information may be exposed, and workarounds must be provided to expose necessary contextual information that is not supported by the language.

Open Modules [5] export pointcuts as part of their interface to denote internal semantic events to which aspects can be applied. Unlike XPIs, interface

specifications scatter across application components. Consequently, programmers have to modify the application components and extend them with necessary pointcut specifications. This reduces the reusability of the components if they must be adopted as the constituent of various software systems. Explicit join point types [14], Aspect Aware Interfaces [21] and IIIA [43] are other proposals, which extend base objects with specification of pointcuts; consequently they suffer from the same shortcoming as Open Modules.

XPIs and other mentioned proposals suffer from the following shortcomings too. First, they distinguish between base and aspect modules. Second, they are limited to support intra-application crosscutting concerns; thus fall short of modularizing inter-application coordination logic. Last but not least, they cannot cope with the heterogeneity of implementation languages in software systems.

Supporting distribution of concerns: Several AO languages and middleware have been proposed to facilitate modularizing crosscutting concerns that are distributed on multiple hosts [34, 24, 33, 32]. The main focus of these approaches is on defining remote pointcut and remote advice, which can be evaluated and executed on different hosts.

DJCutter [34] is an extension to AspectJ that supports remote pointcut and advice. DyMAC [24] is a .Net-based AO middleware, which offers an aspect-component model to support complex distributed compositions by means of advanced remote pointcuts, transparent remote advice and distributed instantiation scopes for aspects. AWED [33] offers remote pointcut constructors, which are more general than previous approaches via supporting remote sequences. Damon [32] is Java-based middleware, which supports dynamic weaving of remote aspects, meta-level aspects, and an event-based communication model to facilitate asynchronous and synchronous executions of advice.

With respect to the requirements outlined in Section 3, only Damon aims at separating interface specifications from the actual application code. However, the offered language is limited to specify method names and interface names, and falls short of modularizing crosscutting interfaces. Except for DyMAC, the evaluated middleware solutions are limited to support Java-based applications; DyMAC is limited to .Net-based applications. Only DyMAC unifies the notion of aspects and ordinary application components.

Uniform representation of concerns: Various proposals exist to unify the notion of aspects and ordinary objects/components [46, 45, 39, 31]. These proposals, however, fall short of addressing the other requirements outlined in Section 3.

Supporting heterogeneity in implementation languages: Compose* [8] is a language- and platform-independent AO language, which supports the Java, C and .Net languages. Through supporting the so-called filter types, Compose* is extensible with new domain-specific languages to implement advice code [30]. An aspect instance can only be applied to one application object. Consequently, Compose* cannot be adopted to implement crosscutting coordination logic, which must be applied to multiple objects that are executed in

parallel and are implemented in different languages. The other requirements outlined in Section 3 are not also fulfilled by Compose*.

In our previous work, we introduced object-level event modules to modularize domain-specific concerns, which may crosscut multiple objects that are possibly implemented in different languages. This concept is implemented in the EventReactor language [29, 28]. Through supporting events as means to represent the state changes of interest in base programs, EventReactor is open-ended to support different kinds of join points and pointcut designators.

Object-level event modules and their implementation in EventReactor fall short of fulfilling the requirements outlined in Section 3. Firstly, there is no support for crosscutting coordination-specific interfaces; instead, events must explicitly be announced from application components. Such code scatters across the application components, and reduces the reusability of the applications if they must be adopted as the constituent of various software systems. Second, object-level event modules can only be adopted to modularize intra-application crosscutting concerns. Last but not least, object-level event modules and application components are distinct types of modules.

5 Architectural Event Modules

In this paper, we introduce the concept of **architectural event modules**, which are means to represent constituent applications and inter-application level crosscutting concerns uniformly as the modules of software systems.

We adopt **events** as the basic abstractions to define the interfaces of applications, and to convey necessary contextual information among the applications. An event represents a state change of interest in the execution of an application. Different **attributes** can be defined for the events to represent necessary contextual information.

As Figure 3 shows, an **architectural event module** has an event-based **required interface**, an implementation termed as **reactor**, and an event-based **provided interface**. An architectural event module is a wrapper around an existing application, which extends the application with event-based interfaces to communicate with other architectural event modules, and to coordinate their interactions. In this case, the application components form the reactor part of the architectural event module. These components are the actual producer and/or consumer of the events.

The required interface specifies the events of interest for the wrapped application, as well as the instructions to forward the events to the corresponding application components to process them. This interface has crosscutting nature from both receiver and recipient sides; it may select events from multiple publishers, and may forward them to multiple application components.

The provided interface specifies the events that are published by the application components. By default, these events are published in a non-blocking way. If the execution of the architectural event module must be blocked until a specific response event arrives, blocking conditions can also be expressed in the

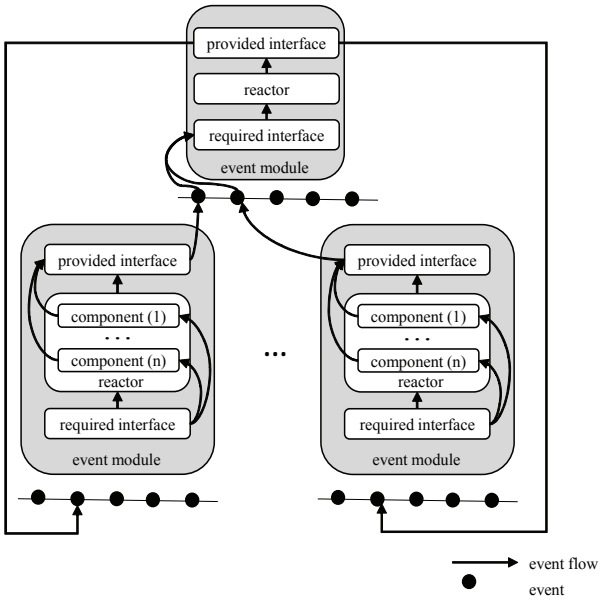

Figure 3: Architectural event modules

interface. The provided interface also has crosscutting nature because events may be published by multiple application components.

As Figure 3 shows, the events published by architectural event modules can be selected further by other architectural event modules. This facilitates forming a hierarchy, in which the modules at the higher levels represent the concerns that crosscut the modules at the lower levels.

Architectural event modules are executed in parallel. Each module is associated with a local event queue, which stores the events that are targeted at it and/or are broadcast to all modules.

There is an analogy between architectural event modules and aspects/objects. Events are analogous to join points; the required interface of an architectural event modules is analogous to pointcut designators; the reactor part is analogous to advice; the provided interface is analogous to the set of join points that can be designated within an aspect. Since multiple events can be selected via the required interface of architectural event module, and such events may be published by various publishers, architectural event modules can be adopted to modularize the concerns that crosscut multiple publishers. In addition, architectural event modules can uniformly be adopted to represent base concerns. The event-based interfaces of architectural event modules are means to modularly extend wrapped applications with necessary coordination-specific interfaces.

6 The EventReactor Language

In our previous work [29, 28], we introduced the EventReactor language, which implements the concept of object-level event modules. The shortcomings of object-level event modules in composing multiple applications are explained in Section 4. In this paper, we extend EventReactor to support architectural event modules. Due to space limit, we only explain this language by means of our illustrative example.

6.1 An Implementation of the Illustrative Case Study

For our illustrative example, we can wrap the adaptive software and the load balancer via two architectural event modules, and define another architectural event module at the higher level to define the coordination logic. In the following, we explain the implementation step by step.

6.1.1 The Event Types

The first step towards defining architectural event modules is to define the events that must be exchanged among them. In EventReactor, events are typed entities; an event 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 a predefined type in EventReactor, which is the super type for other event types. It defines the attributes `publisher`, `target` and `timestamp` for events.

`MethodBased` is another predefined event type to represent method-based state changes in applications. Since `MethodBased` extends `BaseEventType`, it inherits the attributes `publisher`, `target` and `timestamp`.

```
1 eventtype BaseEventType{ Object publisher; Object target; Long timestamp;}
2 eventtype MethodBased extends BaseEventType{
3   org.eventreactor.core.builtin.types.MethodBasedEvents kind;
4   String signature; String module; Object[] args;}
5 eventtype ConstituentState extends MethodBased {org.example.types.State state;}
6 eventtype ControlCommand extends BaseEventType{
7                         org.example.types.Command command;}
```

Listing 1: The specification of event types

The event type `MethodBased` defines the attribute `kind` to represent the kind of a state change that must be regarded as an event. The supported state changes are before invocation, before execution, after invocation, and after execution of a method, as it is supported by the current AO languages [19]. This event type also defines the attributes `signature`, `module` and `args` to represent the signature of the method, the module in which the method is defined, and the arguments of the method, respectively.

The event type `ConstituentState` is a specialization of `MethodBased` for our case study, which represents the current execution state of the adaptive software

and the load balancer via its attribute `state`. The event type `ControlCommand` is to represent the control commands for coordinating the adaptive software and the load balancer.

6.1.2 The Architectural Event Modules

As the next step, we represent the adaptive software and the load balancer as architectural event modules. Listing 2 represents the specification of these two architectural event modules.

The `em_adaptiveSW` module specifies the events represented by `e_controlSW` as its required interface, and publishes the events `e_app_start` and `e_app_end` of the type `ConstituentState` as its provided interface. These events indicate that the execution of the adaptive software has started and finished, respectively. This event module is a wrapper around the Java application named as `AdaptiveSoftware`.

Likewise, `em_loadBalancer` is defined as a wrapper around the C++ application named as `LoadBalancer`. This event module receives the control events specified by `e_controlLB` as its required interface, and publishes the events `e_LB_start` and `e_LB_end`. These events indicate that the execution of the load balancer has started and finished, respectively.

```
1 eventmodules
2     em_adaptiveSW [Java] := {e_controlSW} <- {"AdaptiveSoftware"}
3                                          -> {ConstituentState e_app_start,e_app_end;}
4     em_loadBalancer [C++]:= {e_controlLB} <- {"LoadBalancer"}
5                                          -> {ConstituentState e_LB_start,e_LB_end;}
```

Listing 2: The specification of architectural event modules

6.1.3 The Coordination-specific Interfaces

Until now we have represented the applications via two architectural event modules, and defined the events that are processed and published by them. These events are processed and published by the wrapped components of these applications. As the next step, we must define how these events are received and published by these components. In EventReactor this information can be defined separately from the wrapped components and the event modules.

Lines 1–21 of Listing 3 define the coordination-specific interface for the event module `em_adaptiveSW`. Lines 2–4 specify the set of events that form the required interface of the module. The expression in line 3 specifies that the events of the type `ControlCommand`, which are targeted at `em_adaptiveSW` are of interest, and are represented in the specification via the variable `e_controlSW`. Since many events can be published in a system, and the events can be broadcast to all architectural event modules, one can use logical predicates to select the events of interest based on the attributes of the events.

Lines 5–19 of Listing 3 define the provided interface of the module. Currently four kinds of state changes in the execution of application components

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 [19] to select these state changes. This facilitates binding an event to multiple state changes in multiple application components.

```
1  interface em_adaptiveSW{
2   requires {
3    e_controlSW= {E | E.type == "ControlCommand" && E.target == "em_adaptiveSW"};
4   }
5  provides{
6    event e_app_start:= before execution (void Monitor.monitor(..)) ||
7                        execution (void AppComponent*.execute(..)) {
8              e_app_start.publisher ="em_adaptiveSW";
9              e_app_start.state = org.example.types.State.StartExecuting;
10   }
11     event e_app_end:= after execution (void AppComponent*.execute(..)){
12             e_app_end.publisher = "em_adaptiveSW";
13             e_app_end.state = org.example.types.State.EndExecuting;
14   }
15     wait when (e_app_start) until e_controlSW {
16       (e_controlSW.command == org.example.types.Command.Retry) retry;
17       (e_controlSW.command == org.example.types.Command.Proceed) proceed;
18       (e_controlSW.command== org.example.types.Command.Suspend) suspend;
19     }
20  }
21 }
22 interface em_loadBalancer{
23  requires {
24   e_controlLB= {E | E.type == "ControlCommand" && E.target == "em_loadBalancer"};
25  }
26 provides{
27     event e_LB_start:= before execution (void LoadMonitor.monitor(..)){
28             e_LB_start.publisher = "em_loadBalancer";
29             e_LB_start.state = org.example.types.State.StartExecuting;
30   }
31   event e_LB_end:= after execution (void Migrator.migrate()) ||
32                    execution (void LoadAnalyzer.terminateWithoutMigration(..)){
33             e_LB_end.publisher = "em_loadBalancer";
34             e_LB_end.state = org.example.types.State.EndExecuting;
35   }
36     wait when (e_LB_start) until e_controlLB {...}
37  }
38 }
```

Listing 3: The specification of interfaces

For example, lines 6–7 state that the event `e_app_start` is published before the execution of the method `monitor` in the class `Monitor`, or the method `execute` in classes whose name starts with `AppComponent`.

The attributes of the events can also be initialized with the desired values. For example in lines 8–9, `'em_adaptiveSW'` is specified as the unique identifier of the event publisher, and `StartExecuting` is specified as the execution state of the publisher.

Likewise, lines 11–14 state that the event `e_app_end` is published after the execution of the method `execute` in the classes whose name starts with `AppComponent`. Besides, `'em_adaptiveSW'` and `EndExecuting` are specified as the unique identifier of the event publisher and the execution state of the publisher, respectively.

Events are published by architectural event modules in a non-blocking way. However, it may be needed to block the execution of an event module (wrapped application) after publishing an event, until a specific response event is received by the event module in its required interface. This can be expressed via the `wait when...until` expressions, which relate the provided events to the expected response event.

Line 15 specifies that after publishing `e_app_start`, the execution of the event module `em_adaptiveSW` (the execution of the wrapped application) must be blocked until `e_controlSW` is received. Currently, three kinds of actions can be performed upon receiving an event in `wait on...until` expressions: `retry` means that the execution of the architectural event module 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.

Lines 16–18 show that if the event `e_controlSW`, arrives and has the command `Retry`, the execution of the event module resumes with re-publishing the last event that was published by the event module, i.e. `e_app_start`. If the command is `Proceed`, the execution of the event module proceeds as normal. If the command is `Suspend`, the execution of the event module remains blocked until an `e_controlSW` is received, which has the command `Retry` or `Proceed`.

Lines 22–38 define the interface of `em_loadBalancer`. Line 24 specifies that the events of the type `ControlCommand` targeted at `em_loadBalancer` are of interest.

Lines 26–37 define the provided interface of the module. Here, the events `e_LB_start` and `e_LB_end` are mapped to the state change before and after the execution of the procedures `monitor` and `migrate` in the classes `LoadMonitor` and `Migrator`, respectively. The execution of the event module blocks after publishing the event `e_LB_start` until the event `e_controlLB` arrives. The condition to process this event is similar to the one explained for `em_adaptiveSW`.

6.1.4 The Coordination Logic

The next step is to define our desired coordination logic using an architectural event module. As Listing 4 shows, `em_coordinator` reacts to the events `e_constituentstate`, and publishes events of the type `ControlCommand`.

As for the reactor part, we implemented the coordination logic explained in Section 2 as a Java program. This program implements a state machine, which reacts to the events published by `em_adaptiveSW` and `em_loadBalancer`, maintains the current execution state of these two modules, and publishes control commands to ensure their mutual execution.

Lines 5–10 of Listing 4 specify the interface of the module `em_coordinator`. It selects events of the type `ConstituentState`, and upon arrival of an event it invokes the method `coordinate` on the class `Coordinator` to process the event.

```
1  eventmodules
2   em_coordinator[Java] := {e_constituentstate} <- {"Coordinator"}
3                               -> {ControlCommand e_command};
4  ...
5  interface em_coordinator {
6   requires {
7     e_constituentstate = {E| E.type == "ConstituentState"};
8     on (e_constituentstate){ invoke("Coordinator", "coordinate", e_constituentstate); }
9   }
10 }
```

Listing 4: The specification of coordinator

```
1  eventmodules
2    em_coordinator [Statechart] := {input} <- {
3    initial state Start: (input && input.publisher == 'em_adaptiveSW'){
4                   ControlCommand e_command = new ControlCommand();
5                   e_command.command = org.example.types.Command.Proceed;
6                   e_command.target ='em_adaptiveSW'; publish(e_command);}->SuspendVM;
7        (input && input.publisher == 'em_loadBalancer'
8         && input.state != org.example.types.State.EndExecuting)
9             {/*send Proceed command*/...} -> SuspendApp;
10     state SuspendVM: (input && input.publisher == 'em_loadBalancer'){
11                  ControlCommand e_command = new ControlCommand();
12                  e_command.command = org.example.types.Command.Suspend;
13                  e_command.target = 'em_loadBalancer';publish(e_command);};
14                (input && input.publisher == 'em_adaptiveSW'
15                        && input.state == org.example.types.State.EndExecuting){
16                  ControlCommand e_command =  new ControlCommand();
17                  e_command.command = org.example.types.Command.Retry;
18                  e_command.target = 'em_loadBalancer'; publish(e_command);}-> Start;
19      state SuspendApp: (input && input.publisher == 'em_adaptiveSW'){
20                  ControlCommand e_command = new ControlCommand();
21                  e_command.command = org.example.types.Command.Suspend;
22                  e_command.target='em_adaptiveSW'; publish(e_command);};
23              (input && input.publisher == 'em_loadBalancer'
24                      && input.state == org.example.types.State.EndExecuting){
25                ControlCommand e_command = new ControlCommand();
26                e_command.command = org.example.types.Command.Retry;
27                e_command.target = 'em_adaptiveSW'; publish(e_command);}-> Start;
28   } -> {ControlCommand e_command;}
29
30 interface em_coordinator{
31  requires {input= {E| E.type == "ConstituentState"}; }
32 }
```

Listing 5: The specification of coordinator as a state machine

One may notice that the interface specification in lines 5–10 does not specify how the events `e_command` are published from within the coordinator application.

This is because the coordinator is especially developed for this example, and publishing coordination events is part of its main functionality. Therefore, the events are explicitly published from within this application in the same way as explained in [29]. This is unlike the adaptive software and the load balancer, which are legacy applications extended with event-based interfaces.

EventReactor also supports state machines as a language for expressing the coordination logic. Listing 5 shows a state machine which reacts to the events published by `em_adaptiveSW` and `em_loadBalancer`, maintains the current execution state of these two modules, and publishes control commands to ensure their mutual execution.

Line 31 of Listing 5 specifies the interface of the module `em_coordinator`, which selects events of the type `ConstituentState`. Lines 3–27 define a state machine, which is initially in the state `Start`. If an event is received from `em_adaptiveSW`, a transition takes place to the state `SuspendVM` and an event is published to `em_adaptiveSW` indicating that the execution of the event module must proceed. Likewise, if an event is received from `em_loadBalancer`, a transition takes place to the state `SuspendApp`.

In the state `SuspendVM`, if an event is received from `em_loadBalancer`, a control event is published to inform the corresponding event module that its execution must be suspended. In this state, if an event is received from `em_adaptiveSW` indicating that the execution of a video transcoder request is finished, a control event is produced for `em_loadBalancer` indicating that its execution must be resumed, and a transition is taken to the state `Start`. The other states and transitions are defined in a similar way.

6.2 The EventReactor Compiler

Briefly explained, each event type is translated to a Java class, whose instances represent events. To support architectural event modules, the compiler also receives a configuration file as input; as shown in Listing 6, the name and path of the specification files are defined in the configuration file.

In addition, the following information is specified in for each architectural event module: a) The name of the application, which is referred to in the reactor part of the corresponding event module, b) the IP address of the machine on which the event module is deployed, c) the main method of the application, which can be invoked to start executing the application, and d) the path of the application files that are wrapped by the event module.

Architectural event modules are executed in separate processes and make use of Java Message Service (JMS) [40] to exchange events among each other. There is a logical clock to synchronize the processes and to keep track of the time stamp of events. Each module is associated with a local event queue, which maintains the events that are explicitly targeted to the module. Each event that is published without a specific target is broadcast to all modules by storing the event in the local event queue of each module.

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

Listing 6: The specification of configuration file

Each architectural event module, which wraps an existing application, is translated to a class in the language mentioned in the specifications (see Listing 2); currently Java and C++ are supported. This class maintains necessary meta-data about the event module, such as its name, the list of selectors in its required interface, the list of events in its provided interface. Besides, it implements the functionality to start executing the event module and its corresponding application.

The compiler generates an aspect in AspectJ or AspectC++ for each interface specification, depending on the language mentioned in the specifications. The aspect implements the functionality to publish events specified as the provided interface of the corresponding architectural event module. If there is a `wait when ...until` expression for an event, the aspect also implements body of the `wait when ...until` expression.

Listing 7 shows an excerpt of the AspectJ code generated for the event module `em_adaptiveSW`. In the constructor of the aspect, necessary initialization to work with JMS is performed, an instance of the Java class `em_adaptiveSWClass` and the corresponding event queue is retrieved. The class `em_adaptiveSWClass` is generated by the EventReactor compiler to keep the necessary meta-data about the event module.

The event expression `e_app_start` in lines 6–7 of Listing 3 is translated to a pointcut and advice in the aspect. The code for publishing the event is defined in lines 9–12 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 EventReactor 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 13–30 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.

```
public aspect em_adaptiveSWAspect{ em_adaptiveSWAspect(){
    //initialization ...
          eventmodule = em_adaptiveSWClass.getInstance();
          queue = eventmodule.getQueue();
  }
 pointcut e_app_startPC(): execution (void Monitor.monitor(..)) ||
             execution (void AppComponent*.execute(..));
 void around(): e_app_startPC() {
         ConstituentState e_app_start = new ConstituentState();
         e_app_start.publisher = "em_adaptiveSW";
         e_app_start.state = org.example.types.State.StartExecuting;
         EventReactor.publish(e_app_start);
         BaseEventType waits = eventmodule.waitOn(e_app_start);
       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") == ControlCommands.proceed){
                           proceedexe=true; break;
                     }
                 if (ev.get("command") != null &&
                     ev.get("command") == ControlCommands.suspend)) continue;
               if (ev.get("command") != null &&
                     ev.get("command") == ControlCommands.retry))
                     EventReactor.send(e_app_start);
           }//while
           if (proceedexe) proceed(p);
         }//if
  }

}
```

Listing 7: The generated aspect for *em_adaptiveSW*

7 Formal Event Processing Semantics

As an intuitive example assume that a user issues a request to the adaptive application. In our example, this request results in invoking the method `monitor` in the class `Monitor`. Starting the execution of this method leads to publishing the event `e_app_start` for the event module `em_adaptiveSW`.

Since no target is specified for this event, it is propagated to all event modules; but `em_coordinator` is the only one whose required interface selects the event. Consequently, the specified state machine starts executing, takes a transition to the state `SuspendVM` and publishes a command event to the event module `em_adaptiveSW`. While `em_coordinator` is in the state `suspendVM`, if the event module `em_loadBalancer` attempts to migrate a VM, the event `e_LB_start` will be published, and will be received by `em_coordinator`. As a result, this event module informs `em_adaptiveLB` that the migration must be suspended until the application finishes serving the user's request.

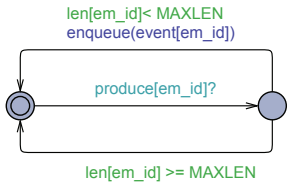

Figure 4: A model for inserting events in the queue

In the following, we adopt the UPPAAL [3] simulation and model checking toolset to formally represent and simulate the semantics of architectural event modules in processing events. UPPAAL facilitates modular modelling of software behavior using separate automata, which are executed concurrently. In UPPAAL, each automata may have a set of local variables and functions, and may be instantiated multiple times similar to classes. Automata communicate via shared variables and channel expressions such as *c!* and *c?*. The channel expression *c!* in an automaton is comparable to an asynchronous method invocation; this invocation is received by the expression *c?* in another automaton.

In our approach, a software system is composed of multiple architectural event modules, which are executed in parallel. Each architectural event module is associated with a local event queue; the operations to put events in the queue, and to retrieve them are also executed in parallel. To model these in UPPAAL, we define three templates, named as *putqueue*, *getqueue* and *eventmodule*. There can be multiple instances of these automata to represent multiple architectural event modules in a system. These instances and their corresponding variables and channels are distinguished by their unique index represented via *em_id*.

We define the data structure *BaseEventType* to represent event types, and the global arrays *event*, *p_event*, *result*, *specific* of this type. For each event module, *event* maintains information about the event that must be inserted in its queue; *p_event* maintains information about the event that is published by the event module; *result* maintains information about the event that is retrieved from the queue by the event module; and *specific* maintains information about the specific event that event module waits for in its *wait when. . . until* expression. We also have defined a set of helper C-style functions, which are not shown due to the space limit.

Figure 4 shows an automaton for inserting the event shown via *event[em_id]* in the local event queue of an architectural event module indexed by *em_id]*. Here, when a request comes with the channel *produce[em_id]?* and if the local event queue is not full, the event is inserted in the local event queue via invoking the function *enqueue*.

Figure 5 shows an automaton for retrieivg events from the local event queue of the architectural event module indexed by *em_id*. Here, when a request comes with the channel *consumeFirst[em_id]?*, the event at the front of the queue is retrieved, if the queue is not empty. If a request comes with the channel *consumeSpecific[em_id]?*, the first event whose timestamp is greater than the last

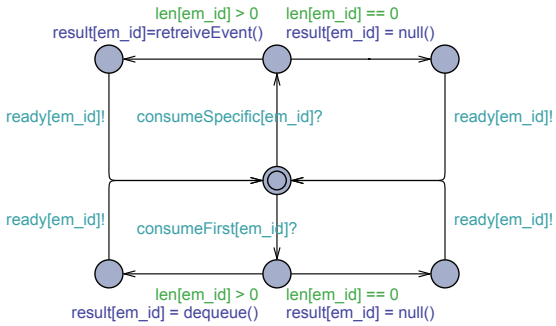

Figure 5: A model for retrieving events from the queue

produced event by the module is retrieved. This is for retrieving input events that are refereed to in the *wait when... until* expressions. The termination of event retrieval is announced via the channel *ready[em_id]!*.

Figure 6 shows the template for architectural event modules; individual instances of this template are distinguished by the variable *em_id*. As this figure shows, there are two ways to execute an event module: by receiving a specified event, or by explicitly invoking the main method of the wrapped application. In our example, the `em_coordinator` is executed in the former way, and the other two architectural event modules are executed in the latter way.

In the following, we first continue with the former case. In the location *Start*, if *eventbased[em_id] == true*, the event module requests for consuming an event via the channel *consumeFirst[em_id]?*, which is responded to by the event queue of the event module. The event module keeps waiting until there is one event to process, *result[em_id].type == 0*. Upon the availability of an event, *result[em_id].type != 0*, via the expressions *matchesInterface(result[em_id])* and *on(result[em_id])* it is checked whether the event matches a required interface of the event module and there is at least one *on* expression defined for the event. If it is not the case, a transition takes place to the location *Retrieving* to ignore the event and to continue with the next event in the queue. Otherwise, a transition takes place to the location *Processing*.

In this location, the number of *on* expressions is retrieved and stored in *onNum[em_id]*. For each expression, the corresponding application method is invoked to process the event; this is modelled by taking a transition to the location *AppProcess* and setting the clock variable c to *0*. A time-consuming operation to process an event is modeled via the invariant $c < TIME$ in the location *AppProcessing*. The event processing operation may terminate at some point without publishing an event, *eventProduction() == false*; consequently, a transition takes place to *Processing* to execute the next *on* expression. If there is no *on* expression, a transition takes place to *Retrieving* to process the next event in the queue.

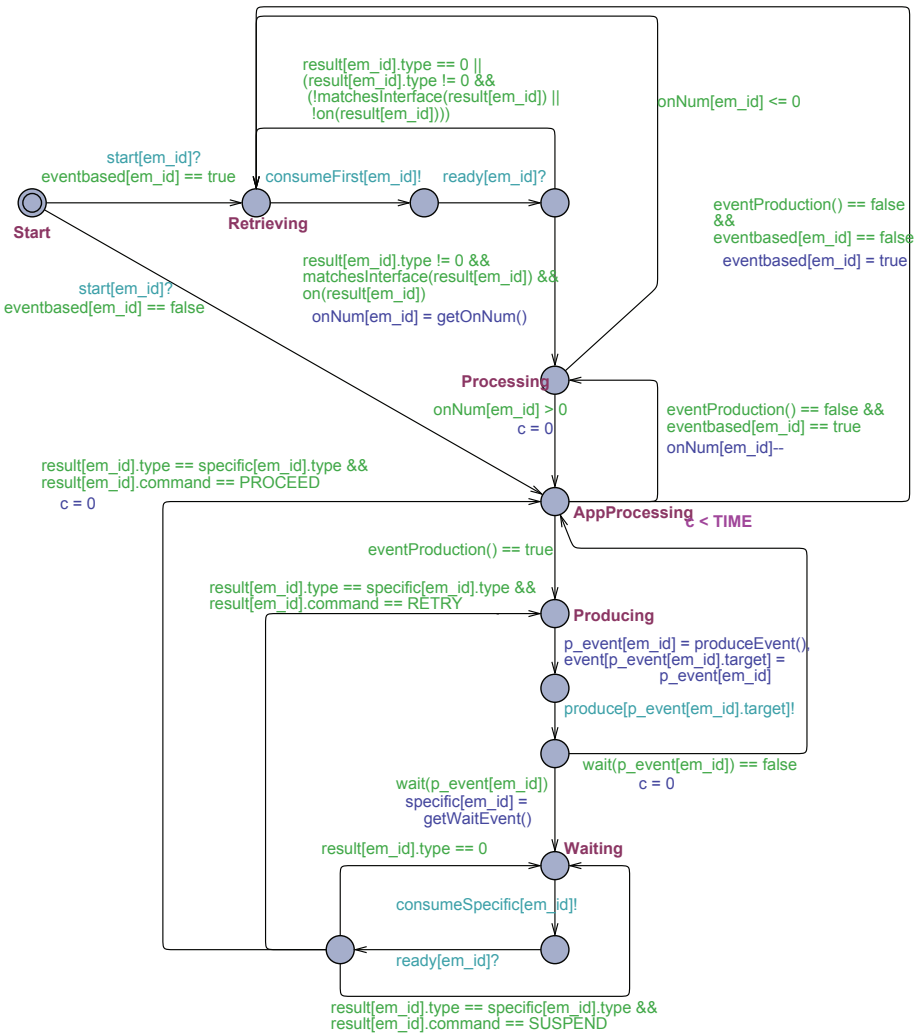

Figure 6: Event processing semantics of architectural event modules

The transition labeled as *eventProduction() == true* in the location *AppProcessing* indicates that events specified in the provided interface of the event module may be published while processing an input event. In the model, an event is produced via invoking the function *produceEvent* and storing the event information in the variable *p_event[em_id]*. The expression *event[p_event[em_id].target] = p_event[em_id]* stores the produced event in the variable *event* of the target event module; the request to insert the event in the queue of the target

event module is issued via the channel *produce[p_event[em_id].target]!*, which is responded to by the event queue.

After publishing an event, if there is no *wait when... until* expression for the event, *wait(p_event[em_id]) == false*, a transition takes place to *AppProcessing* to continue with processing the input event. Otherwise, information about the event that is required to be received is retrieved, stored in *specific[em_id]*, and a transition takes place to the location *Waiting*. The automaton stays in this location until the specified event is available, and is retrieved from the corresponding queue via the channels *consumeSpecific[em_id]!* and *ready[em_id]?*.

If the retrieved event requests for *SUSPEND*, a transition takes place to the location *Waiting*, until the requests for *RETRY* or *PROCEED* arrives. If the retrieved event requests to *PROCEED*, a transition takes place to the location *Processing*, so that the event processing continues by the event module. If the retrieved event requests for *RETRY*, a transition takes place to the location *Producing*, in which the same event is produced and is inserted in the queue of the target event module.

If the execution of an event module starts by explicitly invoking the main method of the wrapped application, i.e. *eventbased[em_id] == false* in the location *Start*, a transition takes place to the location *AppProcessing* to execute the main method; this indicates that during the execution of this method, new events may be published by the method. Whenever the execution of this method terminates, *eventbased[em_id] == false && eventProduction() == false*, a transition takes place to the location *Retrieving* to continue the execution by processing events in the queue, if any.

8 Discussions

In this section, we explain how architectural event modules and their implementation in EventReactor fulfill the requirements outlined in Section 3.

Specifying and modularizing crosscutting coordination-specific interfaces: As shown in Listing 2, architectural event modules facilitate representing existing applications as modules of a software system. We consider events as the basic abstractions to represent the state changes of interest in the constituent applications, and to abstract necessary information about the state changes. Unlike most AO languages that fix the set of supported join points and join point contexts, the sets of necessary events and event attributes are not fixed, and can be programmed depending on application requirements.

As Listing 3 shows, the interface specifications have crosscutting nature because they need to gather events from multiple components. Pointcut designators are adopted to facilitate gathering events from multiple components. As Listing 3 shows, the interface specifications can also be modularized from the actual implementation of components. Listing 7 shows that necessary code to compose these interfaces with the components is automatically generated, providing that suitable (AO) compiler exist to map the specified events to state changes in the components.

The reusability of both interface specifications and their corresponding applications increases due to the separation and modularization of interfaces. For example, an application can be reused individually and/or as a constituent of different software systems; new events can be defined in the interfaces, and can be mapped to the state changes before/after execution/invocation of methods 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 [18]. Nevertheless, as long as the changes do not impact event names or event attributes, they will not affect the specification of other architectural event modules.

Modularizing inter-application crosscutting concerns: Listing 4 shows desired coordination logic can be programmed like an ordinary application, and can be modularized via architectural event modules.

Supporting distribution of concerns: Architectural event modules are executed in separate processes. Events may be targeted at a specific architectural event module, or they can be broadcast to all. As shown in Listing 2 and 4, loose coupling can be achieved among multiple architectural event modules via their required interface, which selects the events of interest based on the attributes of the events. Loose coupling is increased further by interface specifications, which abstract from the actual application components and their state changes.

Uniform representation of concerns: Constituent applications and inter-application crosscutting concerns can be modularized via architectural event modules. This uniformity in the modular representation of concerns increases the compositionality of software systems. For example, the implementation of coordination logic can be treated as a normal application, which can be composed further with other architectural event modules.

Supporting heterogeneity in implementation languages: As explained in Section 6.2, EventReactor can currently support applications developed in Java or C++. The EventReactor language is nevertheless extensible with new languages.

9 Related Work

We evaluated a large set of AO languages in Section 4. In the following, we study other work that is related to our proposal.

9.1 Systems of Systems

A system of system (SoS) is a large-scale concurrent and distributed system whose constituents are complex systems [17, 16, 10, 15]. There are various forms of SoS [1]; directed SoS with whitebox constituent systems is a special form in which the constituent systems can be executed independently, but within the SoS they accept some central coordination to ensure that the goals of the SoS

are fulfilled. Our approach for integrating multiple applications can be regarded as a means for developing directed SoS with white-box constituent applications.

Developing suitable communication techniques, middleware, modeling languages and verification techniques are active research directions in engineering SoS's. Systems Modeling Language (SysML) [2] is a general-purpose modeling language that supports specification, analysis, design, verification and validation of SoS's. Within the context of the COMPASS project [1], methods and tools to support developers in building models of SoS's and in analysing SoS-level properties of these models. Communicating Structures [22] are other modeling techniques in which hierarchical structures that represent SoS's in a uniform, systematic way as composition of a small number of basic system objects.

Our proposal can be considered complementary to these, which mainly focus on modeling and model-driven development of SoS's. Our focus is on high-level module systems and languages to compose multiple constituent applications, in which the applications are modularly extended with crosscutting interfaces, and coordination among them is defined modularly. Such language abstractions of EventReactor pave the way to perform various kinds of analysis on the implementation of SoS's in future.

9.2 Architectural Description Languages (ADLs)

Architectural event modules can also be regarded as abstractions to represent the architecture of software systems that are composed of multiple applications.

Prisma [38] is an aspect-oriented ADL, which supports interfaces, aspects, components and connectors as architectural types. Aspects are explicitly bound to components of specific types, and can be adopted to define various kinds of crosscutting concerns such as coordination and distribution. DAOP-ADL [37] is a component- and aspect-based language to specify the architecture of an application in terms of components, aspects and a set of plug-compatibility rules between them. Rapide [25] is an event-based ADL, in which component behavior is represented by explicit event sequences; events are ordered with respect to two criteria, time and causality. Components are assembled into an architecture via connections. The behavior of a connection can be specified as the correlation between the events that are received and published by the connection.

In contrary to these approaches that offer separate architectural types to represent components, connectors and aspects (if supported at all), architectural event modules are uniform abstractions to represent constituent applications as well as crosscutting concerns such as coordination logic. Unlike these ADLs, our proposal facilitates defining and modularizing crosscutting interfaces for the constituent applications, and to enables augmenting the applications that are possibly developed in different languages with such interfaces.

9.3 Publish/Subscribe Systems

The publish/subscribe paradigm is accepted as a suitable paradigm to develop applications that require one-to-many and many-to-one style of communication [9]. There are various commercial middleware that offer low-level abstractions to facilitate implementing publish/subscribe systems; examples are CORBA [6] and JMS [13]. As explained in Section 6.2, the back-end of EventReactor makes use of JMS to facilitate remote communication between distributed applications.

One could directly adopt existing publish/subscribe middleware to integrate multiple applications with each other, and to implement the necessary coordination logic for them. This, however, requires programmers to directly modify the applications and tailor them. In our approach, we raise the abstraction level of composition specifications by offering high-level languages and uniform module abstractions to compose multiple applications with each other. Legacy applications can be extended with modularized coordination-specific interfaces; new applications such as the ones implementing desired coordination logic can also be supported. The specifications of modules and their interfaces are declarative. This paves the way to perform various checks on the compositions in future.

9.4 Coordination Languages

Several coordination languages are proposed in the literature, whose aim is to integrate a number of possibly heterogeneous components together to form a single software system that can execute on and take advantage of parallel and distributed systems [35].

Coordination languages can be classified as data-driven and control-driven [35]. In data-driven languages, the state of the computation is defined in terms of both the data being received/sent and the actual configuration of coordinated applications. In other words, a coordinator or coordinated application is responsible for manipulating data as well as for coordinating either itself and/or other applications. Linda [11] and Linda-like languages [36, 35] are in this category of coordination languages.

The separation of coordination logic from coordinated applications is not enforced at the syntactic level by data-driven coordination languages. Consequently, the coordination logic may scatter across and tangle with the core functionality of the coordinated applications. One may adopt existing modularization mechanisms such as AO to modularize crosscutting coordination-specific code. We have explained the shortcomings of AO languages in detail 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 [42, 23, 26, 25]. Our proposal also falls into this category.

These languages offer a set of primitives to define the connectors to explicitly bind components together. Unlike these languages, architectural event modules are not explicitly bound to each other; via adopting event-based communication and event quantification, they are loosely coupled to each other. Consequently, software reuse and evolution are eased. Besides, our proposal does not distinguish between constituent applications and connectors, and facilitates modularizing crosscutting coordination-specific interfaces from the application components.

The closest to our proposal is Polylith [7], which also aims at extending applications with event-based interfaces to facilitate their composition and coordination. In Polylith, applications can generate and react to specific event types. This is unlike our approach where events of interest can be specified via logic queries over event attributes. Besides, Polylith does not facilitate modularizing crosscutting coordination-specific interfaces from application components.

10 Conclusion and Future work

Nowadays, it is becoming inevitable to develop complex software systems as the composition of multiple applications, which are developed independently, evolve in due time, and their composition may cause various kinds of undesirable interactions to emerge. We proposed architectural event modules as means to represent constituent applications and necessary coordination logic as modules of of complex software systems.

In EventReactor, the specifications of event modules and interfaces are declarative. In future, we would like to perform various analysis on such declarative specifications; for example, analyzing the changes that happens in the execution flow of an application when it becomes the constituent of a software system, or detecting conflicting coordination commands when multiple coordinators are present.

Currently, we assume that the wrapped applications are single-threaded; consequently *on* and *wait when* expressions are not executed in parallel. As future work, we would like to extend the EventReactor language to define architectural event modules for multi-threaded applications.

In this paper, we focused on applications as the constituents of software systems. However, systems of systems may also consist of other types entities such as end users, hardware devices, and sensors. We consider events as the standard means to abstract necessary information about the behavior of such entities, and claim that our approach to wrap existing entities as event-based modules can be generalized to cover these kinds of entities too. In future, we would like to illustrate the suitability of our proposals for such cases. Last but not least, we would like to apply our proposal to large case studies and evaluate the modularity and coupling of the implementations using quantitative software metrics.

Acknowledgement

This work is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

References

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

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

[3] UPPAAL. http://www.uppaal.org/.

[4] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of the Workshop on Object-Based Distributed Programming*, ECOOP '93. Springer-Verlag, 1994.

[5] Jonathan Aldrich. Open Modules: Modular Reasoning About Advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 144–168, Berlin, Heidelberg, 2005. Springer-Verlag.

[6] Juergen Boldt. The Common Object Request Broker: Architecture and Specification. Technical report, Object Management Group, July 1995.

[7] Chen Chen and J.M. Purtilo. Configuration-level Programming of Distributed Applications Using Implicit Invocation. In *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 43–49 vol.1, Aug 1994.

[8] Compose*. http://composestar.sourceforge.net/.

[9] Opher Etzion and Peter Niblett. *Event Processing in Action.* Manning, 2010.

[10] Jochen Fromm. Types and Forms of Emergence.

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

[12] S. Gotz, C. Wilke, S. Richly, and U. Abmann. Approximating Quality Contracts for Energy Auto-tuning Software. In *International Workshop on Green and Sustainable Software (GREENS)*, 2012.

[13] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java Message Service. Technical report, Sun Microsystems, April 2002.

[14] Kevin Hoffman and Patrick Eugster. Cooperative Aspect-Oriented Programming. *Sci. Comput. Program.*, 74:333–354, March 2009.

[15] O. Thomas Holland. Taxonomy for the Modeling and Simulation of Emergent Behavior Systems. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*. Society for Computer Simulation International, 2007.

[16] Mo Jamshidi. *System of Systems Engineering: Innovations for the Twenty-First Century*. Wiley, 2008.

[17] Mo Jamshidi. *Systems of Systems Engineering: Principles and Applications*. CRC Press, 2008.

[18] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-oriented Software with Model-based Pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06. Springer-Verlag, 2006.

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

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[21] Gregor Kiczales and Mira Mezini. Aspect-oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.

[22] V.E. Kotov. Communicating structures for modeling large-scale systems. In *Simulation Conference Proceedings, 1998. Winter*, volume 2, pages 1571–1577 vol.2, Dec 1998.

[23] J. Kramer, J. Magee, and A. Finkelstein. A Constructive Approach to the Design of Distributed Systems. In *Building Distributed Systems, IEE Colloquium on*, pages 3/1–315, Nov 1990.

[24] Bert Lagaisse and Wouter Joosen. True and transparent distributed composition of aspect-components. In *Proceedings of Middleware'06*, pages 42–61, Berlin, Heidelberg, 2006. Springer-Verlag.

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

[26] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring Parallel and Distributed Programs. *Software Engineering Journal*, 8:73–82, 1993.

[27] Somayeh Malakuti. Detecting Emergent Interference in Integration of Multiple Self-Adaptive Systems. In *the 2nd Workshop on Software Engineering for Systems of Systems*, ECSAW '14, New York, NY, USA, 2014. ACM.

[28] Somayeh Malakuti and Mehmet Aksit. Event-Based Modularization of Reactive Systems. In *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 367–407. Springer Berlin Heidelberg, 2014.

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

[30] Somayeh Malakuti, Mehmet Aksit, and Christoph Bockisch. Distribution Transparency in Runtime Enforcement. In *Proceedings of the 2011 IEEE/FTRA International Conference on Advanced Software Engineering*. IEEE Press, 2011.

[31] Mira Mezini and Klaus. Ostermann. Conquering Aspects with Caesar. In *Proceedings of the 2nd Aspect-Oriented Software Development*. ACM Press, 2003.

[32] Rubn Mondjar, Pedro Garca-Lpez, Carles Pairot, and Lluis Pamies-Juarez. Damon: A Distributed {AOP} Middleware for Large-scale Scenarios. *Information and Software Technology*, 54(3):317 – 330, 2012.

[33] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly Distributed AOP Using AWED. In *Proceedings of the 5th Aspect-Orieneted Software Development*, pages 51–62, New York, NY, USA, 2006. ACM.

[34] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *Proceedings of the 3rd Aspect-Oriented Software Development*, pages 7–15, New York, NY, USA, 2004. ACM.

[35] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. In *ADVANCES IN COMPUTERS*, pages 329–400. Academic Press, 1998.

[36] Edinburgh Parallel and Edited Greg Wilson. Linda-Like Systems and Their Implementation, 1991.

[37] Mónica Pinto, Lidia Fuentes, and Jose María Troya. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-based Development. In *Proceedings of Generative Programming: Concepts and Experiences*, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[38] Jennifer Prez, Nour Ali, JoseA. Cars, and Isidro Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In Ian Gorton, GeorgeT. Heineman, Ivica Crnkovi, HeinzW. Schmidt, JudithA. Stafford, Clemens Szyperski, and Kurt Wallnau, editors,

Component-Based Software Engineering, volume 4063 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin Heidelberg, 2006.

[39] Hridesh Rajan and Kevin Sullivan. Eos: Instance-Level Aspects for Integrated System Design. In *Proceedings of the 9th European Software Engineering Conference*, ESEC/FSE-11. ACM Press, 2003.

[40] Mark Richards, Richard Monson-Haefel, and David A Chappell. *Java Message Service*. O'Reilly Media, 2009.

[41] Kateryna Rybina, Waltenegus Dargie, Rene Schoene, and Somayeh Malakuti. Mutual Influence of Application- and Platform-Level Adaptations on Energy-Efficient Computing. In *Proceedings of the 23rd Parallel, Distributed, and Network-Based Processing*, PDP '15, 2015.

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

[43] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Trans. Softw. Eng. Methodol.*, 20:1:1–1:43, July 2010.

[44] Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular Aspect-oriented Design with XPIs. *ACM Trans. Softw. Eng. Methodol.*, 20(2):5:1–5:42, September 2010.

[45] Davy Suvée, Bruno De Fraine, and Wim Vanderperren. A Symmetric and Unified Approach Towards Combining Aspect-oriented and Component-based Software Development. In *Proceedings of the 9th International Conference on Component-Based Software Engineering*, CBSE'06, pages 114–122, Berlin, Heidelberg, 2006. Springer-Verlag.

[46] Davy Suvée, Wim Vanderperren, Dennis Wagelaar, and Viviane Jonckers. There Are No Aspects. *Electron. Notes Theor. Comput. Sci.*, 114:153–174, January 2005.