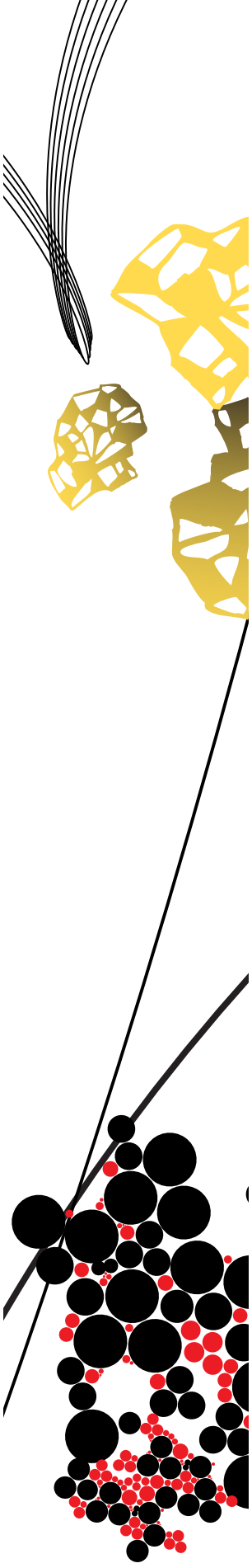

A Framework for Modular Implementation of Domain-Specific Event-Based Applications

Author:
Roel ter Maat
(s0219681)

Master Thesis

Supervisor:
Dr. Somayeh Malakuti

Committee:
Prof. dr. ir. Mehmet Aksit
Dr. Ing. Christoph Bockisch
Ir. Steven te Brinke

March 10, 2014

UNIVERSITY OF TWENTE.

Contents

CHAPTER 1

Introduction

There are various types of application that can be used to implement different types of event processing. Examples of event processing are runtime verification [1], self-adaptive software [2], and traffic monitoring. To implement these types of application, *event producers* are used to indicate a state change. *Event consumers* can perform actions on the event streams and *event processing agents* are used to mediate between the event producers and event consumers.

When implementing event processing applications, a large amount of event producers, event processing agents and event consumers can be used, increasing the complexity. To manage this complexity, this thesis claims that a separation of concerns, a loose coupling between concerns, and understandability of the code is required.

In current advanced programming languages and frameworks, module abstractions, composition operators, and some language extensibility is provided. To evaluate the suitability of these abstractions, a set of requirements for event processing applications is set in chapter 3. A representative set of languages is evaluated with respect to these requirements. Using the results of these evaluations, the shortcomings are identified.

In [3, 4], *event modules* are introduced as linguistic abstractions. These modules support modular representation of events and their reactions. In EventReactor [3, 5, 6] these event modules are implemented. In this thesis, it is explained that event modules can modularize the concerns, and facilitate loose coupling in the compositions.
In chapter 4 a new EventReactor implementation is introduced. This framework improves the composability of event modules by composing them on instance level and supports both implicit and explicit composition of event modules. The modularity is also improved by modularizing both the event consumer and event processing agents. Finally, an extensible set of languages is used to defined the functionality in the event modules, which is used to increase the understandability of the code.

In chapters 5- 7 the implementation details of the event processing concerns are explained. In these chapters, it is described how the criteria are fulfilled and how the framework processes the concerns.

In chapter 8 a set of evolution scenarios is used to show how the designed constructs are useful to fulfill the criteria. This is done by evaluating the modularity and composability of the concerns by removing, adding, and replacing them and counting the amount of changes that are required. Secondly, the possible language extensibility is compared to existing languages and frameworks to show that more extensibility is offered.

CHAPTER 2

Background: Event Processing Applications

In Event Processing Applications (EPA) [7], the flow of the program is influenced, determined, and changed by events. In the section 2.1, the basic components of EPAs are explained. This is followed by section 2.2 in which a case study is described which will be used throughout this thesis. Section 2.3 explains a set of languages that can be adopted for programming EPAs. In section 2.4, the advantages of using EPAs are explained. Finally, the chapter is summarized in section 2.5.

2.1 Core Concepts in Event-Processing Applications

2.1.1 Events

An **event** [7] is something commonly known in the real world as *something that has happened* [7]. This can, for instance, be a phone starting to ring or a pen dropping on the floor. Events like this can prompt a person to react, because he or she is able to detect those events and take actions accordingly, such as answering the phone or picking up the pen. Using events in such a manner and using the ability to react to them allows a person to continue living his life without waiting for the phone to ring in order to react to it. Secondly, other people are able to make a person perform an action by initiating an event, in this case by calling him.

Events can be used in the same manner in software. Events represent a state change of interest during the execution of the software. In this case, an event

may maintain information about the corresponding state change, for example the time of occurrence and the thread of execution in which it has occurred.

There are various ways to represent events in software. A typical way is to represent them as a data structure that defines a set of fields to keep information about the state changes. Events may be typed, where an event type can be used to instantiate events which have the same structure and meaning.

2.1.2 Event Processing Network

In EPAs, three types of entities interact with each other forming an **Event Processing Network** [7], an example of which is shown in Figure 2.1.

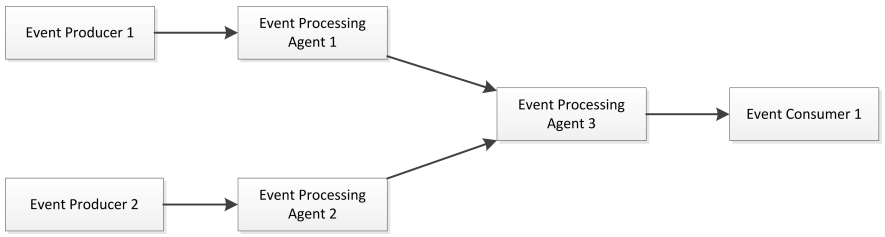

Figure 2.1: Event Processing Network

The first of the three components is the **event producer** [7], which is responsible for the generating and publishing of events. Events can be produced by both hardware and software. For example, a sensor can be used to measure certain values, such as temperature, in the real world. Every newly measured value can be regarded as an event allowing hardware to produce events.
At the software level, various kinds of state changes may be regarded a events. Examples are: occurrence of exceptions, invocation of methods, construction of objects, access to I/O, etc.

When an event is produced, a reaction can take place which is the responsibility of the **event consumer** [7]. The reactions that take place can, for example, be logging or storing of data, checking and reacting to received data, altering the event producer, and so on. Depending on the content, event consumers are also able to become event producers. At this point an event can be produced which will be consumed by another, or perhaps the same, consumer.

To mediate between the event consumer and event producer, **Event Processing Agents** [7] are used. Event processing agents are entities which can both

consume and produce events. They may perform various operations on the events. For example, filtering, routing, modifying and creating new events etc. The produced events can then be sent to either the consumer, or other event processing agents. Unlike the producer and consumer, the event processing agent can only be expressed in software entities.
The processing performed by the agents can be stateless or stateful. With stateless agents, the action which is performed is independent of the previously published events. With stateful agents, previous events influence the action which is performed with new events. An example of stateful agents is a filter which only lets an event pass when a certain amount of events have been published before it.
Using the processing agents to connect the consumers and producers removes a direct coupling between the entities, and the producers have no knowledge of which consumers will react to the event.

2.2 Case Study

Throughout this thesis, a thermostat system is as an illustrative example. In this system, 10 sensors are located outside a house and measure a temperature every 10 minutes. The sensors are event producers and generate an event every time the sensor performs measures a new value.
There are software components used as consumers, which will publish the measured temperature to the weather station. An event processing agent is used between the producer and consumer, filtering for the specific events produced by the outside temperature sensors. This allows other types of sensors to produce events without these events being consumed by the existing consumers. Figure 2.2 shows the basic architecture of the system with the sensors producing events, which are filtered by the event processing agent. When an event is produced by this agent, the event consumer takes the temperature from the event and sends it to the thermostat.

2.3 Implementation Languages for Event Processing Applications

Software engineers face various alternative languages for implementing EPAs In this section three alternatives are discussed, namely Stream-Processing Languages, Object Oriented(OO) programming and Aspect Oriented(AO) programming.

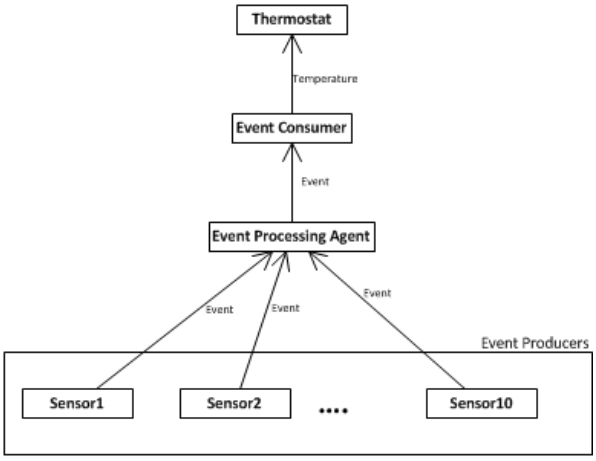

Figure 2.2: Case Study

2.3.1 Stream-Processing Languages

Esper [8] is one of the event stream processing languages currently available. There are multiple implementations for Esper for Java and .Net. This section only focusses on the Java implementation. In Esper, events can be created using a *Plain Old Java Objects* (POJO). In the configuration of the *Complex Event Processing*(CEP) engine, the event needs to be added. Events can be instantiated in the base code and can be sent to the engine. Esper offers a dedicated language, similar to SQL, to filter the event stream and performing various actions on the events.

```
1 select * from EventName.win:length(2)
```

Listing 2.1: Esper EPL Statement

Consider, for example the statement in Listing 2.1. Here, the event is triggered with every second published *EventName* object by defining the window length as 2, where a window contains event stream data which is defined like a table. Event filtering statements are defined in a Java object. Reactions to the filtered events can be defined using Java objects. For this, Esper facilitates defining the Java objects as the listener for the selected events. An example of a listener object is shown in Listing 2.2.

```
1 public class exampleListener implements UpdateListener {
2   public void update(EventBean[] newEvents, EventBean[] oldEvents)
      {
3     EventBean event = newEvents[0];
4     System.out.println("Event Filtered"));
5   }
6 }
```

Listing 2.2: Esper Listener

2.3.2 Object Oriented Programming

In Object Oriented(OO) programming, *objects* are a means to modularize the concern of interest; e.g. event producers, event processing agents. This allows a program to be designed by creating objects for every real world object, creating a more readable and reusable program. Variables and methods are defined within an object, which can be accessed and executed by other objects.

A possible implementation of EPAs can be achieved using the Observer design pattern [9]. Using the Observer pattern, event producers are assigned the role of *subject*, and event processing agents and/or event consumers the role of *observer*. Observers are able to register with subjects, allowing them to be notified of changes when the subject invoke their *notify* method. The invocation of this method is regarded as an event being produced. The arguments set in the invocation are used as the attributes of the event.

Another possibility would be to use the event-delegate mechanism [9]. In this mechanism, event type can be defined using a special type of class. Events can be created by instantiating the correct event type. In order to bind producers and consumers, a delegate is used. A method which matches the signature of a delegate can be assigned to it, allowing multiple event consumers to be bound to a producer. Event can be published by invoking the delegate corresponding to the event.

2.3.3 Aspect Oriented Programming

Aspect Oriented (AO) programming [10] introduces the notion of **aspects** to modularize crosscutting concerns in software. The languages offer dedicated constructs to implements aspect elements.

AO languages offer a couple of basic elements; the following can be related to the EPA concepts:

Join points: Join points represent a state change of interest in the execution of a program. Join points are not explicitly handled in many systems, yet it is an important element that needs to be discussed. The created join points make up the join point model which can be used by the other AO elements. This element can be roughly translated to an event. A join point describes the parts of the program which must publish an event with a specification of the information which the event contains. The join points model can be *point in time* or *region-in-time* [11]. The point-in-time model selects a specific point in the execution of the base program, while the region-in-time model select an interval in the execution. The latter can, for instance, be a method execution at which the join point will include the entire execution of this method.

Pointcuts: A pointcut is a means to select a set of join points of interest. The selection is usually expressed using a query language. Pointcuts can be more complex by making them history based [12], where a pointcuts is activated based on the previous actions and pointcuts.
In implementing EPAs in AO languages, pointcuts can be adopted to implement the event filtering functionality. This, however depends on the expressive power of pointcuts designators. If it is not expressive enough, one has to also define the event filtering functionality using advice code.

Advice: The advice is a piece of code that is bound to one or more pointcuts and is executed when these are activated. Depending on how the concerns of EPA are separated, an advice can be used to implement the functionality of event processing agents and/or event consumers.

One of the currently best known AO languages is AspectJ [13, 14, 15], which is an extension to the Java language. Join points in AspectJ are static and consist of things like method calls, method executions, object instantiations, constructor executions, field references and handler executions [14, 16]. Inside a main *aspect* component a user can define a pointcut by selecting the specific join points.

```
1 public aspect ExampleAspect{
2
3   private pointcut methodCall(): this(Sensor) &&
4     call(void doSomething(String));
5
6   before(): methodCall(){
7     //perform action
8   }
9 }
```

Listing 2.3: Example Aspect

In Listing 2.3, an example aspect is shown. The pointcut is named allowing it to be referenced in both other pointcut, creating more complex pointcuts. In the

aspect, it is defined that, whenever the method *doSomething* with the String parameter is called from a *Sensor* object, the pointcut *methodCall* is activated. The advice, which has been specified to be executed *before* the pointcut, will perform the required actions.

At compile time, the Java program is compiled to Java byte code. The byte code is examined by the AspectJ compiler to find every point in the flow which matches the defined pointcuts. For each matching point in the byte code, a call to the advice code is weaved in automatically.

2.4 Advantages of EPAs

One of the main reason EPAs are used is that it is supposed to facilitates loose-coupling between the event producers and consumers, or a separation of concerns [17]. Both these concerns can be implemented without requiring any knowledge of the function and implementation of the other. Using the events communication is possible between the two. Events consumers can also be implemented without knowing about each other. The loose coupling between the concerns should mean that consumers and producers can be added and removed without having any effect on the other producers and consumers. However, as this thesis shall show, the loose coupling is highly dependant on the core abstractions that are offered by the various languages which are used to program EPAs.

2.5 Summary

This chapter explains the core concepts of Event-Processing Applications. In it, it is claimed that these applications improve the maintainability of software by improving the separation of concerns, extensibility, reusability and understandability. To fully show the background, a list of existing languages and frameworks are explained with their own specific implementation of the core concepts. With these languages and frameworks established, an analysis can be performed examining to what extend the maintainability is achieved.

CHAPTER 3

Evaluation of Event Processing Languages

EPAs, like any other type of application, can implement complex functionalities which may be subject to changes during, or after development. To support this, linguistic mechanisms need to be provided to increase reusability and evolvability. To evaluate whether or not the existing languages and framework fulfill these requirements, this chapter defines three criteria, and evaluates a set of representative languages.
In section 3.1, the three criteria are described. In section 3.2, a representative set of languages is evaluated. In section 3.3, evolution scenarios are applied to the case study to show the shortcomings in existing languages and frameworks. In section 3.4, the results are summarized and a solution is explained.

3.1 Evaluation Criteria

In this section, we define three evaluation criteria: *language extensibility*, *modularity* and *composability*. This is done by examining how the criteria can be improved, and what advantages this provides.

3.1.1 Language Extensibility

The most obvious choice to implement EPAs is the adoption of General Purpose Languages (GPL), which are Turing complete languages.

Because there are situations where people with less programming knowledge need to implement a functionality or a very specific action needs to be performed which can be simplified, the concept of a Domain Specific Language(DSL) is introduced. A DSL is a language created to implement functionalities for specific

application domains. A DSL is usually not Turing complete and contains a limited set of features. Examples of DSLs are HTML, used for the creation of web pages, and SQL, for querying relational databases. Using the DSLs allows for a more readable functionality and makes the code more maintainable for people without a lot of programming knowledge.
When designing a framework a choice usually needs to be made which language is used to define the functionalities. Since the number of existing languages is too large to expect each and every one to be implemented in the framework, the focus lies on whether or not new languages, either DSL or GPL, could be added to a framework and used to implement a functionality.
With an increased language extensibility, a user is able to add new languages necessary for a specific domain. This would increase the understandability and maintainability of the different event consumers and event processing agents.

3.1.2 Modularity

In an EPA various event processing agents and event consumers may be used. To be able to cope with the complexity and to increase usability, a language must provide means to modularize these concerns. This allows them to be implemented without the knowledge of other elements [18]. The module abstraction must provide well-defined input and output interface and an implementation part which encapsulates the functionality [19]. To support referencing the modules they must provide a unique identity.

3.1.3 Composability

To compose the modularized concerns with each other, a language must provide mechanisms to support composing modules at interface level and defining modularized constraints in their language. The composition configuration can occur on three different levels:

- Instance level, where specific instances of modules have a specific configuration.
- Module level composition, where every instance of a module is configured identically.
- Language level composition, where every module written in the same language has the same configuration.

Binding is possible in two different ways:

- Homogeneous, to compose two concerns of the same type.

- Heterogeneously, to compose two concerns of a different type.

3.2 Evaluation

In this section, the shortcomings of existing Object Oriented, Aspect Oriented and Event Processing languages are explained with respect the defined criteria by using the representative languages and frameworks.

3.2.1 Language Extensibility

3.2.1.1 Aspect Oriented Programming

AspectJ, like other AO languages, offers a join point model which include all the events that can be produced. In the case of AspectJ it is a fixed model, meaning join points not included in this model can not easily be used in the definition of a pointcut and workarounds are required to make this possible, increasing the complexity and decreasing the modularity.
The language used to define the pointcut and advice is fixed in AspectJ.
To extend the language extensibility in AO language, composition frameworks are provided allowing multiple DSLs to be used to implement concerns. There are currently various frameworks offering varying types of functionalities. The following list explains the mechanism behind a few of those frameworks and shows a few of their limitations.

- **Awesome** allows users to construct a multi weaver by composing the separate weavers constructed by the user [20]. Components are supplied by the framework which implement the basic functionality of AspectJ like weavers. New weavers can be created by implementing it in these components.
  Awesome supports the AspectJ join point model restricting it to the same problems stated earlier.
  Configuration of compositions in Awesome happens at language level, which results in a lack of precision, because not every module can be configured separately. Composition between the modules happens using a set of pre-defined constraints.

- **XAspects** is a plug-in extending AspectJ [21]. In this framework, aspects defined in DSLs are translated to Java classes and AspectJ aspects. As with Awesome, the AspectJ join point model is used, and thus have the same restrictions. After the first compilation phase the resulting aspects become visible to each other allowing composition to take place between them.

- **Reflex** is a *versatile kernel for the composition of aspects* [22]. These aspects can be defined in varying languages. Plug-ins can be created which translate aspects, written in the desired language, to basic operations understandable by the kernel and meta objects. A limited set of join points is provided.
  composition between aspects is possible using the fixed set of composition instructions. Using these instructions, constraints can be created for a more advance configuration.

3.2.1.2 Object Oriented Programming

Events in the observer pattern are fixed resulting in a tight coupling between the producer and the consumer, because the producer is limited to the specific events of the consumer. New types of events would also require new observers, which makes projects with a large number of event types difficult to manage.
The language used to define event consumers and event processing agents is fixed by the OO language used to implement the pattern.
The event delegate mechanism allows new types to be specified, but limits them to types published within a program. Languages used for event consumers and event processing agents are fixed.

3.2.1.3 Stream Processing Languages

Events in Esper can be defined in various ways including Java objects and XML. The language used to implement event consumers and event processing agents is fixed to Java and SQL respectively.

3.2.2 Modularity

3.2.2.1 Aspect Oriented Programming

AspectJ offers aspects in the form of modules containing pointcut designators to select events. When the expressiveness of the pointcut language is not good enough, not all required event selection can be implemented, meaning the advice needs to contain code to implement it. This results in tangling between the pointcut and the advice.
Aspect modules can be instantiated using a limited amount of strategies, including per object and as a singleton. In the former, each aspect instance is bound to a specific object and the aspect will only handle events published by that object. When changes occur in the amount of event producers being processed by a consumer or event processing agent, problems arise handling this in the implementation.

Singleton instances experience difficulties when implementing stateful event processing where states need to be maintained for a specific group of event producers, requiring workarounds to make this possible.
The advice code can be used to implement stateful event processing. There are other languages which allow history based event selection to be defined in the pointcut [12], yet these are very dependent on the expression power of the pointcuts.

3.2.2.2 Object Oriented Programming

Using the observer pattern results in fixed interfaces with only one type of event being used. For more advanced event processing this could offer some difficulties.
When implementing stateful event processing, other observers need to be used to gather events used to reason.
When implementing complex event consumers and event processing agents, multiple observer are required, each implementing part of the processing. This results in some tight coupling between the observers.

The event-delegate mechanism can also be used to implement the EPA. using this mechanism, event types are defined within their own module and these can be instantiated and published by the base program as events. The event consumer and event processing agents can be defined within their own module by defining a method whose signature matches the delegate. When implementing complex functionalities, the same problems arise as in the observer pattern.

3.2.2.3 Stream Processing Languages

Esper offers primitive statements used as queries for event processing, yet they are not represented as modules, meaning no modularity is supported.

3.2.3 Composability

3.2.3.1 Aspect Oriented Programming

The advice code in AspectJ contains the binding to the pointcut resulting in a tight coupling between the two concerns. Advices are also not named modules, making composition impossible and making it impossible to differentiate between advices initiating a join point which can be caught by other pointcuts using the *adviceexecution* designator. In the former problem, a workaround is required which compose the code in the advice creating a tight coupling between the advices. The latter problem can be worked around by mapping the

join points to the method invocations or executions but would tangle the advice activating the join point and the advice reacting to it.

3.2.3.2 Object Oriented Programming

For observers to process the produced events streams they need to be explicitly bound to the subjects producing them. This create a tight coupling between the concerns and changing the number of subjects or observers the binding need to be redefined.
When observer objects produce events, the object needs to be designated the role of subject. For each target of the event a new implementation of observer pattern can be required, increasing the complexity.
When the amount of observers is changed, a redefinition may be required. Specifically when the order needs to be defined. The Mediator pattern may be used to implement this, but would require a redefinition of the observers.
When subjects provide multiple event streams to be used by an observer, this observer needs to be invoked in all the subjects, creating a tight coupling between the subject and observer, and scattering the invocation across multiple subjects.

The composability of the event-delegate mechanism is very similar to the observer pattern, where there is a tight coupling between the producer and consumer.

3.2.3.3 Stream Processing Languages

Because modularization is not supported interface level composition can also not be supported. The composition of queries is possible using the SQL queries.

3.3 Illustration of Shortcomings in AspectJ

To show the shortcomings this sections uses the case study from section 2.2 to which evolutions are applied. To do this, a set of evaluation criteria and evolution scenarios are defined which are implemented and evaluated to identify areas of improvement.

3.3.1 Evaluation Criteria

In AspectJ, an aspect is considered a module in which a pointcut is the input interface, advice is the implementation, and the set of joinpoints that can be designated within an aspect are the output interface.

In AspectJ, event selection is done by pointcut designators. AspectJ facilitates defining pointcuts in separate aspect modules. Advice is a means to define event reaction. Advice is bound to pointcut, the binding is defined as part of advice. The advice and pointcut will be considered modules in this section, which should be an advantage for AspectJ, and will make comparison easier.

The choice of evolution scenarios, which are used, were selected based on the types of software evolution described in [23]. The types most relevant to this situation are those defined in the business rules, which are *adaptive enhancive*, and *corrective*. This is, because the other evolution scenarios won't create changes which can be used to evaluate the modularity and/or composability. For complete analysis, the main features of modularity have been defined as input, output, functionality and instantiation. These four features have been crosscut with the required types of software evolutions giving 12(4x3) types to be examined, many of which can be combined.
To examine a framework, a set of criteria needs to be defined. When selecting the criteria we specifically want to look at how much existing code needs to be redefined and/or changed to implement these scenarios.

To examine modularity, the evolutions are checked for adding, removing, redefining of modules. In the case of the composability, AspectJ's composition happens within existing modules meaning the redefining of modules should cover that. Some frameworks cover composition in a separate section, requiring a metric which counts changes in composition code. In this case a change in the composition code is more desirable than a change in a module because of the requirement of reducing the amount of redefinition of the modules.

AspectJ can increase the modularity by adding external functionalities defined in classes which can be reused in multiple modules. This is more desirable than duplicate code, but still requires an extra metric.
Finally modularity offers the possibility to loosely couple elements. To examine this, a metric is added counting the amount of strongly coupled concerns.
These requirements lead to the following list of metrics. Since different languages have different sections modularized, this list assumes a module to be one of the EPA concerns. This way, it is possible to properly compare the languages and frameworks.

Added Modules: Number of event consumer and event processing agents added to perform the desired action. In the case of AspectJ, the pointcut and advice aspectmodules are counted and the overall aspect is ignored.
Removed Modules: Amount of modules removed from the implementation. When binding this to a pointcut which is never activated, it will also count as removed.
Redefined Modules: The number of modules where the code is changed. This can either be functionality or AspectJ binding which is changed.
Altered Bindings: In some languages the bindings are handled outside the

modules in a separate script. This metric counts the amount of changes in this script. This type of change is more favourable than a redefined module, since it supports an increase modularity.
Non-modular functionalities added: Number of modules functionalities added not directly defined in a module. E.g. classes which are called from within a module.
Strongly coupled concerns: Counts the number of concerns strongly coupled within one module. For AspectJ, the coupling between the advice and pointcut is ignored.

3.3.2 Evolution Scenarios

With the criteria defined the scenarios could be created. In the following section, six evolution scenarios are defined, using the case study from section 2.2 as a base, which will focus on one or more of the selected types defined in the previous section. These scenarios can be mapped in the following way:

	Input	Functionality	Output	Instantiation
Enhancive	6	1, 2	6	1, 2
Corrective	3	3, 4	3	4
Reductive	6	5	6	5

3.3.2.1 Scenario 1

As an evolution, half of the sensors are moved from the outside to the inside. The indoor temperature was deemed useless for the weather station, yet would be useful for a thermostat to regulate indoor temperature. The values measured by the sensors must be split up to perform different actions. The values of the indoor sensors must be sent to the thermostat, while the outdoor sensors' values must be sent to the weather station. The identifier of the sensors is used to differentiate between the two, where 1 to 5 is used for indoor sensors and 6 to 10 is used for outdoor sensors. This evolution scenario will be used to evaluate the frameworks on their ability of creating more complex event processing agents based on existing agents and the parameterization of instances.

3.3.2.2 Scenario 2

The sensors inside the building are divided over 3 adjacent areas and for each of these areas an average temperature is requested to inform users of area temperatures. Since most rooms have 2 sensors the choice was made to create the

average of the last two values, and publish this to the user.
Secondly, the sensors must be used to measure radiator malfunctions, which needs to be fixed as soon as possible. The radiators are located between the areas, and when 10 continuous values greater than the allowed values (set at 30) is measured in one of the areas near the radiator the maintenance crew needs to be informed. This evolution scenario is useful to test the ability of a framework to have an instance of a module for a specific group of objects. Secondly, it tests the possibility of two aspects performing actions on the both the same and different instances of a functionality.

3.3.2.3 Scenario 3

For the third scenario, it is assumed that a new sensor has been added from an external project. This external project also has an event processing agent and an event consumer responsible for the transmitting of the data to a different weather station.
For the evolution, the old weather station needs to be removed and the outside temperatures must be sent to the new weather station. The reaction, responsible for sending the data to new weather station, only requires the temperature as a value.
This evolution scenario will be used to examine the ability to correct functionalities in existing event consumers and event processing agents. Secondly, it takes a look at correcting input and output of modules.

3.3.2.4 Scenario 4

For some reason the previous programmer switched the IDs in the event processing agents responsible for filtering between inside and outside, meaning the inside temperatures are sent to the weather stations and the outside are sent to the thermostat. In order to fix this, the event processing need to be corrected. This scenario will examine the possibility of correcting functionality and the instantiation of modules.

3.3.2.5 Scenario 5

The following scenario uses the resulting program of evolution scenario 2 as a base.
Because one of the three areas only contains one sensor, the publishing of the aggregate of the last two values has been deemed useless. To solve this, the choice has been made to remove the calculation of the aggregate and publish the temperature of the sensor directly to the user.
This scenario is used to examine the ability to remove modules or instances of

modules from the flow of a program.

3.3.2.6 Scenario 6

The following scenario contains 4 evolutions but to avoid repetition they are combined in one scenario.
This scenario looks at the event processing agents and event consumer of the inside sensors and more specifically the output of the former, the input of the latter, and the binding between the two. For the evolution, variables from both the input and output are added and removed resulting in the 4 (2x2) evolutions.

3.3.3 Implementation of Evolution Scenarios

After defining the criteria and the scenarios, they were implemented in AspectJ and the evaluation criteria are evaluated. Furthermore the results when using both the Observer pattern and Esper are discussed.

3.3.3.1 Base Scenario

Implementing the base scenario in AspectJ an aspect needs to be defined in which joinpoints are selected where sensors store the measured temperature after which the advice is executed.

This results in the AspectJ code shown in Listing 3.1.

```
1 private pointcut filter(TemperatureSensor sensor) : this(sensor) &&
      call(void store(String, double, int));
2
3 before(TemperatureSensor sensor): filter(sensor){
4   //perform action
5 }
```

Listing 3.1: AspectJ Base Scenario Implementation

The advice is not fully implemented since it is not relevant for the evolution scenarios.

In Java, the Observer Pattern can be used where the observers can be created which will react when the sensors notify them that a value is published.

In Esper, a statement needs to be defined to select the events published when the sensors store the values. The reaction defined in a separate class can be bound to statement.

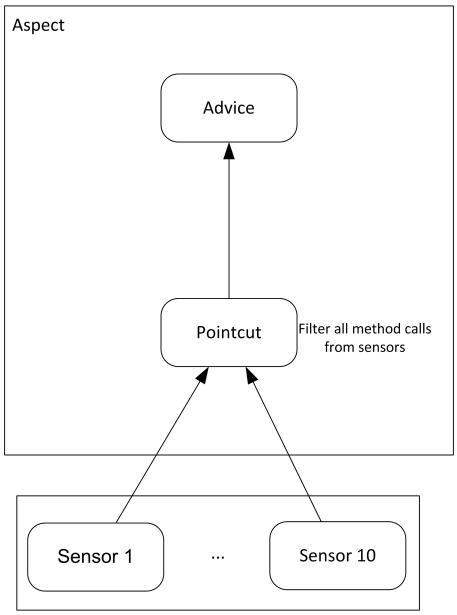

Figure 3.1: Base Project

3.3.3.2 Scenario 1

In this scenario, the original pointcut needs to be extended to filter on both the inside, where the ID is between 1 and 5, and the outside, where the ID is between 6 and 10. For both the filters, a reaction is required resulting in the structure seen in Figure 3.2.

To implement this in AspectJ the original aspect was altered to ensure it would be extensible in other aspects. This is done, by making the original aspect abstract and the pointcut protected. Listing 3.2 shows the resulting AspectJ code.

```
1 public abstract aspect TemperatureFilter {
2   protected pointcut sensorPointcut(TemperatureSensor sensor): this
      (sensor) && call(void store(String, double));
3 }
```

Listing 3.2: Abstract Temperature Aspect

To create a better separation of concern two aspects were created, one for the inside sensors and one for the outside. In both aspects, a pointcut was created extending the original pointcut with a filter to select either the inside or outside sensors. Listing 3.3 shows the aspect for filtering inside sensors, while Listing 3.4 shows the aspect for filtering outside sensors.

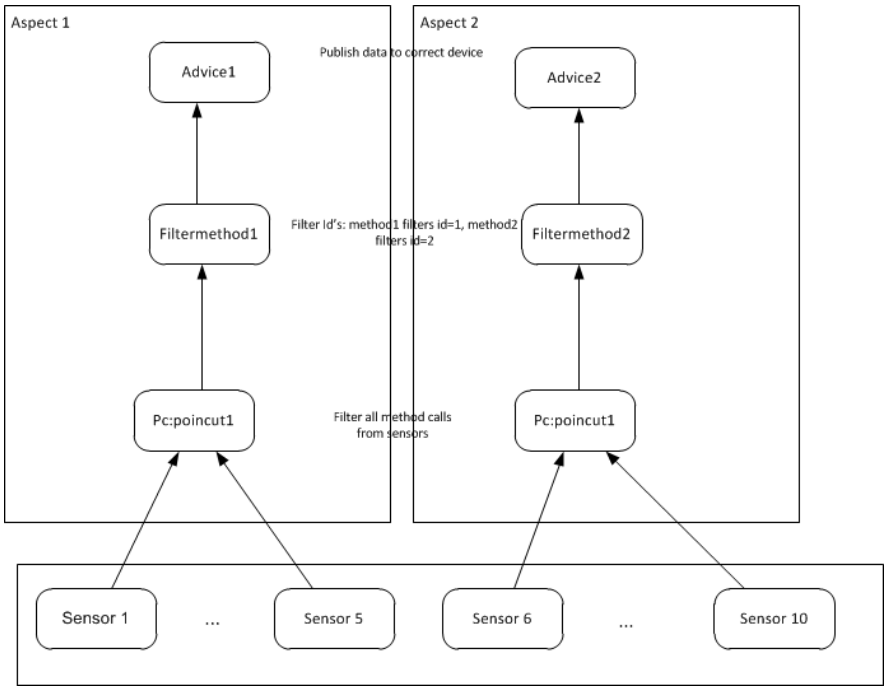

Figure 3.2: Evolution Scenario 1

```
1 public aspect InsideTemperatureFilter extends TemperatureFilter {
2   private pointcut insidePointcut(TemperatureSensor sensor):
      sensorPointcut(sensor) && if(sensor.getId() >0 && sensor.getId
      ()<6);
3
4   before(TemperatureSensor sensor): insidePointcut(sensor){
5     System.out.println("published To thermostat: "+sensor.getName()
      +" with value: "+ sensor.getValue());
6     //perform action
7   }
8 }
```

Listing 3.3: Inside Sensor Filter Aspect

```
1 public aspect OutsideTemperatureFilter extends TemperatureFilter {
2
3   private pointcut outsidePointcut(TemperatureSensor sensor):
      sensorPointcut(sensor) && if(sensor.getId() >5 && sensor.getId
      ()<11);
4
5   before(TemperatureSensor sensor): outsidePointcut(sensor){
6     System.out.println("published To weather station: "+sensor.
      getName()+" with value: "+ sensor.getValue());
7     //perform action
8   }
9 }
```

Listing 3.4: Outside Sensor Filter Aspect

The aspect responsible for the outside sensors needs the advice code from the original aspect, responsible for the publishing to the weather station. This can be copied from the aspect, after which the pointcut needs to be replaced with the pointcut specifically for the outside sensors.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ	3	0	2	0	0	0

The amount of added modules seems logical since two new filters are required and one new reaction, but the filters perform the same action, but on a different ID, meaning this number could be reduced. The redefined modules are not desired. Because of the strong coupling between the filter and the reaction, changing the binding between them requires a redefinition of one of the modules.

To allow this in Java, two new modules need to be created which utilize the original filter. The original filter will need to be removed as an observer. Because of the binding between the filter and the reaction, the original filter will need to be redefined, removing the call to the reaction, which needs to be moved to a new filter. This means, even more changes are required for the Java implementation.

In Esper, the amount of new modules can not be reduced since instantiation or parameterization is not possible in the statements. The binding between the filter and reaction is specified in the filter, just like in Java, and therefore does not have the same problems as AspectJ. The problem with Esper is that the first filter, used to select the right events cannot be bound to other filters. This means the code from the primary filter needs to be moved to both the new filters introducing duplicate code.

3.3.3.3 Scenario 2

For this evolution the filter used for the inside sensors needed to be replaced with three filters for the correct areas. Two functionalities need to be added

and instantiated and composed with the correct filters resulting in the structure seen in Figure 3.3.

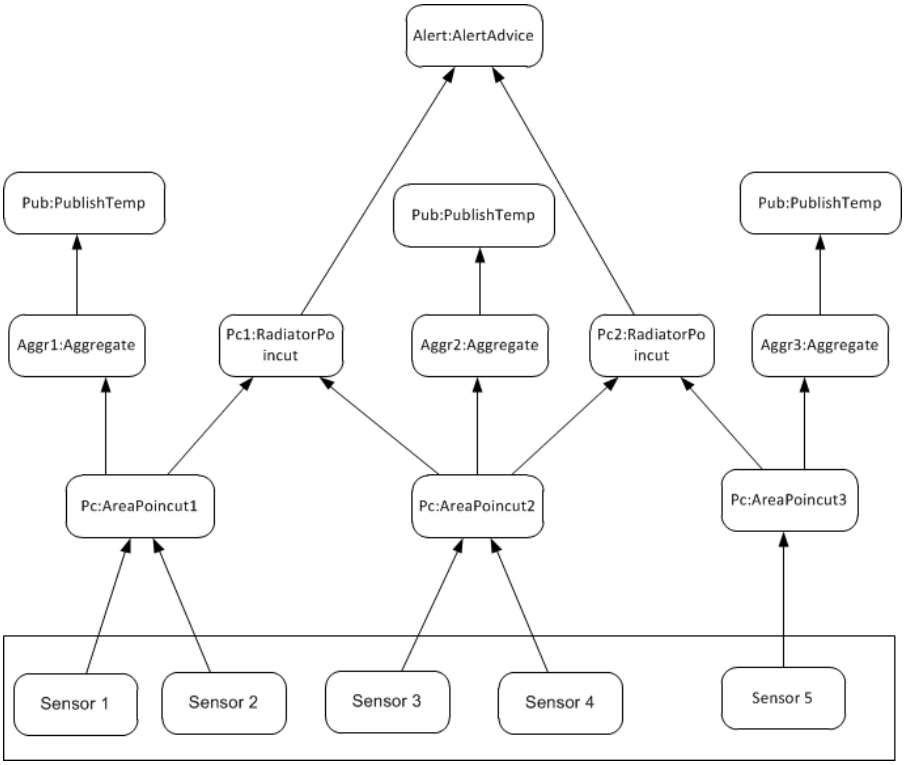

Figure 3.3: Evolution Scenario 2

To reduce the amount of duplicate code for AspectJ, two classes were created to perform aggregate calculation, and temperature measurement. These classes were:

Aggregate: which maintains and returns the average temperature.
RadiatorSensor: Counts the amount of continues values above 30, when it reaches 10 values it will return 'true'.

Since the publishing of the aggregate values is always executed after the calculation of the aggregate, the possibility exists to incorporate this code in the aggregate calculation code. This would reduce the amount of calls in the advice, but would decrease the modularity. Since modularity is something we want to test the code was kept out of the classes. The same counts for the functionalities implementing radiator monitoring and the maintenance notification. Especially since one of these calls will need to be removed from a specific area in a future scenario, preserving modularity will in this case prevent extra work for AspectJ.

To allow different areas to perform an action on the same instance all pointcuts and advices will have be defined within the same aspect. Otherwise instances of a specific type needs to be made public so other aspects can reference it. This would also create a strong coupling between the aspects, which we want to avoid. In the inside aspect, three Aggregate instances and two RadiatorSensor instances are defined, as shown in Listing 3.5.

```
Aggregate agg1 = new Aggregate();
Aggregate agg2 = new Aggregate();
Aggregate agg3 = new Aggregate();
RadiatorSensor radMon1 = new RadiatorSensor();
RadiatorSensor radMon2 = new RadiatorSensor();
```

Listing 3.5: External Class Instantiation

For every area, a pointcut is defined, as shown in Listing 3.6.

```
private pointcut area1(TemperatureSensor sensor): sensorPointcut(
    sensor) && if(sensor.getId()==1 || sensor.getId()==2);
private pointcut area2(TemperatureSensor sensor): sensorPointcut(
    sensor) && if(sensor.getId()==3 || sensor.getId()==4);
private pointcut area3(TemperatureSensor sensor): sensorPointcut(
    sensor) && if(sensor.getId()==5);
```

Listing 3.6: Area Pointcut Definition

After this, two options exist: Either handle all the advices per area, which is shown in Listing 3.7.

```
before(TemperatureSensor sensor): area1(sensor){
  agg1.addValue(sensor.getValue());
  //publish value
  if(radMon2.addValue(sensor.getValue())){
    //alert maintenance
  }
}

before(TemperatureSensor sensor): area2(sensor){
  agg2.addValue(sensor.getValue());
  //publish value
  if(radMon1.addValue(sensor.getValue()) || radMon2.addValue(sensor
    .getValue())){
    //alert maintenance
  }
}

before(TemperatureSensor sensor): area3(sensor){
  agg3.addValue(sensor.getValue());
  //publish value
  if(radMon1.addValue(sensor.getValue())){
    //alert maintenance
  }
}
```

Listing 3.7: Advice Specification per Area

Or handle the radiator monitors as a pointcut, as shown in Listing3.8.

```
@Pointcut("(area1(sensor) || area2(sensor)) && if()")
private boolean radiatorMonitor1(TemperatureSensor sensor){
  return radMon1.addValue(sensor.getValue());
}

@Pointcut("(area2(sensor) || area3(sensor)) && if()")
private boolean radiatorMonitor2(TemperatureSensor sensor){
  return radMon2.addValue(sensor.getValue());
}

before(TemperatureSensor sensor): area1(sensor){
  double aggregate = agg1.addValue(sensor.getValue());
  //publish value
}

before(TemperatureSensor sensor): area2(sensor){
  double aggregate = agg2.addValue(sensor.getValue());
  //publish value
}

before(TemperatureSensor sensor): area3(sensor){
  double aggregate = agg3.addValue(sensor.getValue());
  //publish value
}

before(TemperatureSensor sensor): radiatorMonitor1(sensor) ||
    radiatorMonitor2(sensor){
  //alert maintenance
}
```

Listing 3.8: Separated Pointcut and Advice

The former reduces the amount of modules but increases the redundancy and does not separate the concerns of pointcut and advice. Both options will be examined to see which one is the best.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ per area	6	0	0	0	2	12
AspectJ monitor as pointcut	9	0	0	0	2	3

The problem in AspectJ is that advices are unnamed modules. Composing multiple advices together where the output of one is the input of another would require them to be defined within the same module, or requires utilization of global variable which becomes very dependant on the execution of AspectJ.

In Java, a similar coupling is required, where the reactions must call each other in order to perform multiple reactions after each other.

In Esper, multiple listeners can be added to a statement removing the strongly coupled concerns. The problem is that input and output can not be matched between the listeners, requiring them to be implemented in the same listener.

3.3.3.4 Scenario 3

For this evolution scenario, the filter for the outside sensor needs to be composed with the reaction from the external project, as shown in Figure 3.4. The external

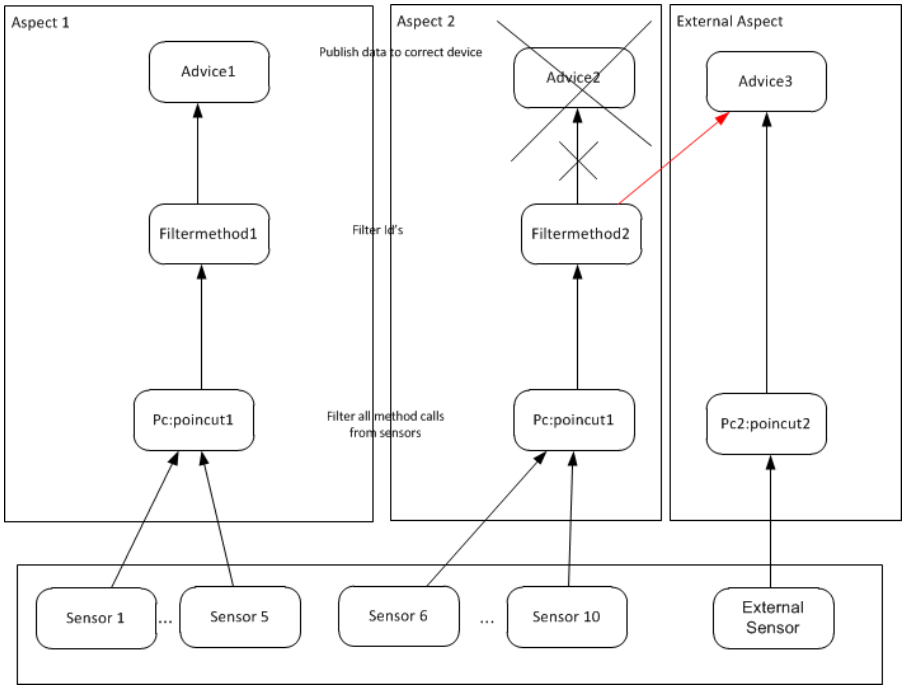

Figure 3.4: Evolution Scenario 3

aspect is shown in Listing 3.9.

```
1 private pointcut externalFilter(double value): this(
      TemperatureSensor2) && args(String, value, int) && call(void
      store(String, double, int));
2
3 before(double value): externalFilter(value) {
4   //Perform external action
5 }
```

Listing 3.9: External Aspect

The problem in AspectJ is that the poincut and filter of the outside sensors and the advice from the external sensor need to be connected, as can be seen in the Figure 3.4. To make this happen some problems arise.
The biggest problem is passing the objects to the advice. Because different types of objects are used it wont be possible to compose the outside pointcut and the external pointcut straight up, because the resulting pointcut needs to

produce the same types as output.

To solve this, the outside filter can be changed by adding the value of the sensor to the pointcut. Now the external and the outside pointcut can be composed by ignoring the sensor in the new pointcut, as shown in Listing 3.10.

```
1 public pointcut outsidePointcut(TemperatureSensor sensor, double
      value): sensorPointcut(sensor) && args(String, value) && if(
      sensor.getId() >5 && sensor.getId()<11);
```

Listing 3.10: Outside Sensor Filter with Added Output

This pointcut can be used in the external advice by composing the external and outside pointcut into one pointcut which can be done in the advice, as shown in Listing 3.11.

```
1 before(double value): externalFilter(value) ||
      OutsideTemperatureFilter.outsidePointcut(TemperatureSensor,
      value) {
2   //Perform external action
3   System.out.println("External value published: "+ value );
4 }
```

Listing 3.11: Pointcut Composition

This does however require another change in the outside pointcut, because it needs to be made public for the external aspect to use it. This can however be avoided by having it public from the start.

If the outside pointcut was also being used by other advices, changing the output becomes a more difficult task since that change will ripple to all modules referencing it. To avoid this, a new pointcut can be created extending the outside pointcut by adding the temperature to the output. Then, the new pointcut can be used for the external aspect while the other modules can keep using the original pointcut. Because the outside pointcut is not, and will not, be used by other modules, this alternative is ignored.

Another solution would be to copy the advice of the external filter and reuse it with the outside filter, as shown in Listing 3.12.

```
1 before(TemperatureSensor sensor): OutsideTemperatureFilter.
      outsideFilter(sensor) {
2   //Perform external action (again)
3 }
```

Listing 3.12: Duplicated External Reaction

This does mean that the temperature normally directly provided by the external filter is no longer available. The advice will need to be changed to use the

temperature defined within the sensor provided by the outside filter. It also creates redundancy because the same code is defined twice. The outside filter will also still need to be made public so it can be referenced by the external aspect.

Another way to handle this is to create a superclass for both sensors. At this point both pointcuts can return that superclass as output and the advice can receive that type as input. This would, however, require a change in the base program and the external pointcut and advice, or the luck or knowledge that all the sensors are already subtypes of one generic superclass. At that point, only the external pointcut and advice need to be changed to alter the input and output, and the way the temperature is retrieved in the advice.

It would also be possible to use the joinpoint as output for the pointcuts and retrieve the class responsible for the activation of the pointcut, e.g. the sensors. This would still require a change in both the outside and the external pointcut to make sure that they output the joinpoint and a change in the advice to retrieve the sensor from the joinpoint. Secondly, because this situation doesn't assume the previously mentioned superclass, the advice has to differentiate between the two different types of sensors to perform the action, increasing the amount of code.

All these different alternatives will be examined for completeness.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ add temperature	0	1	2	0	0	0
AspectJ copied advice	0	1	1	0	0	0
AspectJ sensor superclass	0	1	2	0	1	0
AspectJ joinpoint	0	1	3	0	0	0

In all of the cases in this scenario, AspectJ requires changes to modules to cope with the strong coupling between the modules. The second implementation has a reduced amount of module redefinition but introduces duplicate code.

In Java, less, though still some, redefinition is required, because the filter used for the outside sensors can call the newly added advice and remove the call to the original advice, meaning Java is actually more preferable in this scenario.

The problem in Esper is that input and output are not explicitly used in. Events are used to pass along values. If, in this case, the sensors and event of the external project are of a different type, than the existing project, the code in the reaction which extracts the temperature from the event or sensor will not work for the existing event or sensor. This requires the code to be duplicated and changed to match the existing project.

3.3.3.5 Scenario 4

The fourth scenario requires the advice to be performed after a different pointcut is activated.
In AspectJ, this scenario is solvable in two ways. The first is to change the implementation of the pointcuts to make the filters correct again, as in Listing 3.13.

```
1 private pointcut insidePointcut(TemperatureSensor sensor):
      sensorPointcut(sensor) && if(sensor.getId() >5 && sensor.getId
      ()<11);
2
3 private pointcut outsidePointcut(TemperatureSensor sensor):
      sensorPointcut(sensor) && if(sensor.getId() >0 && sensor.getId
      ()<6);
```

Listing 3.13: Pointcuts with Switched Functionality

The second method is to change the bindings of the pointcuts in the advices to make the outside reaction use the inside filter and vice versa. This does however create the problem that the pointcuts need to be changed to allow them to be referenced in the other aspects.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ change pointcut	0	0	2	0	0	0
AspectJ change advice	0	0	4	0	0	0

This scenario reiterates the point made earlier where instantiation and parameterization can reduce the amount of modules, and when requiring changes like this would reduce the amount of redefined modules.

In Java and Esper, the same situation applies, where both the inside and the outside filter need to be changed.

3.3.3.6 Scenario 5

In this scenario, the functionality aggregating the values for one specific area needs to be removed, composing the filter for the outside sensors to the external reaction as seen in Figure 3.5.
To implement this scenario in AspectJ, the advice module needs to be dissected to remove the part calculating the aggregate. Because it was decided to call methods defined in objects, some modularity has been applied and removing the specific action is rather simple. If this were not the case, some more work would be required. After removing the aggregate code, the call to the publishing needs to be changed to make sure the correct value would be published. The resulting code is shown in Listing 3.14.

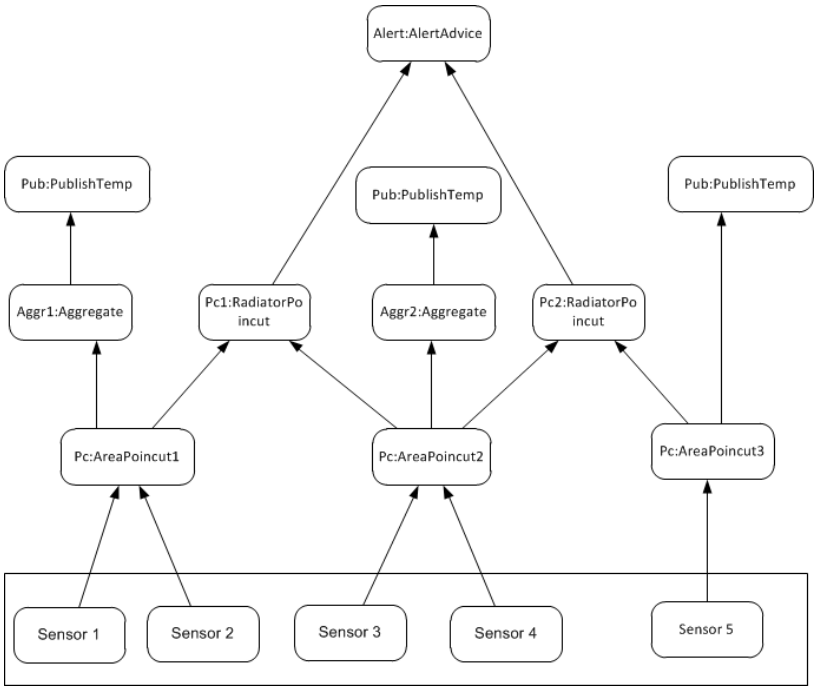

Figure 3.5: Evolution Scenario 5

```
1 before(TemperatureSensor sensor): area3(sensor){
2   double aggregate = sensor.getValue();
3   publAggr.publish(aggregate);
4 }
```

Listing 3.14: Advice with Removed Funtionality

As mentioned in scenario 2, there were two options of publishing the data, where the call to publishing could be incorporated in the aggregate calculation. This option was not chosen because that would reduce modularity and that would truly be shown in this scenario, where one instance of the aggregate-publishing advice would have to be changed. If the two advices would be implemented in one advice module, this scenario would require them to be separated and removed for the third area.

This means modularity can be increased in AspectJ to prevent some work when evolutions are required, but in some scenarios, modules still require change if other modules needs to be removed.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ	0	0	2	0	0	0

In this scenario, the strong coupling between filters and reaction is shown. Because the two reactions are strongly coupled, removing one will result in a change for the other.

In Java, the same coupling problem exists, because the concerns specifies how to continue the flow of Ṙemoving the reactions between the filter and another reactions requires the removal of the reaction and changing the filter to execute a different reaction.

In Esper, the reactions are bound in the same way as in Java, meaning the same actions are required.

3.3.3.7 Scenario 6

For this scenario, both the input of the reaction and the output filter will be changed by adding and removing one of the values.

In AspectJ, the removing a parameter from the input will still allow the same pointcut to be used. It will only require the binding defined in the reaction to be changed to ignore the output from the pointcut by adding the type instead of the name which used to match the input of the advice, as shown in Listing 3.15.

```
1 before(): originalPointcut1() || insidePointcut(TemperatureSensor){
2   //do something
3 }
```

Listing 3.15: Ignored Pointcut Output

A more difficult situation arises when adding an extra input. Because that value will become unbound in the pointcut, an error will be given. For this to work, the pointcut needs to be changed to include the added input as an output value. After this, the input can be matched with the input, as can be seen in Listing 3.16.

```
1 private pointcut addedInsideOutput(TemperatureSensor sensor, String
      string):insidePointcut(sensor) && args(string, double);
```

Listing 3.16: Pointcut with Added Output

This pointcut can then be added to the advice, as shown in Listing 3.17.

```
1 before(TemperatureSensor sensor, String string): originalPointcut2(
      sensor, string) || addedInsideOutput(sensor, string){
2   //do something
3 }
```

Listing 3.17: Bound Pointcut and Advice with Added Input

In the case of removing of output, the same situation arises as the adding of input, which requires a change in the advice module. When adding output to the pointcut, the same situation is used as the removal of input requiring only a change in the binding code in the advice.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
Adding Input	0	0	2	0	0	0
Removing Input	0	0	1	0	0	0
Adding output	0	0	1	0	0	0
Removing output	0	0	1	0	0	0

AspectJ is fairly strict when it comes to input and output of the filter and reaction. Most prominently when adding an input to the reaction the filter both modules require changes. Secondly, the binding of input and output happens within the modules itself, meaning any change immediately means the modules need to be redefined which is not desirable.

In Java, the mapping of input and output is just as strict as AspectJ, since they are simple method call, and happens within the modules, resulting in the same situation as AspectJ.

In Esper, the input in modules is fixed, where the statement and the reaction only receives events, limiting the possibilities.

3.4 Summary

In the previous sections, a set of languages and frameworks were evaluated and shortcomings in fulfilling the criteria from section 3.1 were identified. When looking at the evaluation, some shortcomings which are fairly evident. When evaluating the language extensibility, the more established languages such as AspectJ offer no direct support for language extensibility, and even the plug-in mechanism XAspect does offers language extensibility, but with some limitations. Reflex and Awesome offer a greater language extensibility, but fall short when it comes to the modularity and composability.

The modularity and composability in AspectJ, though possible to some extent, could be improved.
When composing modules, the different frameworks require it to be specified

within one of the module. When a change is required in a filter, or when another filter needs to be used, the modules which are bound to it require a change themselves. Languages which offer a separate section in which the composition can be defined, preserves the separation of concerns and ensures the modules have no knowledge of each other, which is important in modularity.
Advices in AspectJ are unnamed modules and cannot be referenced in other modules. This leads to the problem that when multiple reactions need to take place they either needs to be defined in separate advices or all defined in one. When the output of one reaction is required as the input of another, they will need to be defined in the same advice, or require a construct where multiple advices use the same global variable.
In some scenarios, nearly identical modules were required, or two modules which needed to maintain a separate states. In the existing frameworks little to none direct support is offered for this, increasing duplicate code and .

The scenarios lead to the following list of shortcomings is identified.

- Strong coupling between filters and reactions, requiring lots of redefining of modules when small changes are needed.
- Strict binding of input and output, requiring all values to be mapped.
- Advices are unnamed and cannot be composed outside of the module.
- Duplicate code is increased due to lack of instantiation and parameterization.

With the concerns evaluated, the conclusion can be drawn that no existing framework is able to facilitate the modularity, composability and language extensibility to a full extend. To solve this problem, a new framework is proposed which should be able to allow a bigger support these concerns to a greater degree. This new framework would have the following requirements:

- All filters and reactions are defined within their own, named, module.
- Modules can be instantiated and parameterized at will.
- Modules do not require any knowledge about each other.
- Free composition between filters and reactions.
- Composition between the modules is handles in a separate section.
- New DSLs and GPLs can be added to define the functionalities.

In the next chapters this new framework is designed and implemented, after which the same evaluation methods are applied to it, to ensure the goal is reached.

CHAPTER 4

EventReactor2.0

In the previous chapter, the shortcomings of various languages and frameworks to fulfill the modularity, composability and language extensibility criteria were discussed. To avoid these shortcomings a set of requirements are defined for a framework to implement EPAs. In this chapter the EventReactor2.0 framework is discussed.
In section 4.1, the compilation phase of the framework is explained. In section 4.2, the runtime behaviour of the framework is explained. In section 4.3, the improvements of EventReactor2.0 over EventReactor are explained. In section 4.4, the results are summarized.

4.1 Compile time

Figure 4.1 shows the compiler of EventReactor2.0. To implement and EPA, an application developer has to defined *Specification of Event Types* and *Specification of Events*. These are used to define types of events and a set of events supported by the EPA respectively. *Specification of Event Types* is provided to the component *Event Type Checker* which verifies if the event types are specified using the correct syntax and if language constraints, such as unique names, are adhered to. After verification, the specification is provided to the component *Event Type Compiler*, which compiles the specifications to one *Event Type Java Class* for each defined event type. This class is stored on the file system. For *Specification of Events*, the same type of actions are performed. It is provided to the component *Event Checker* to verify the syntax and the language constraints. After that, the component *Event Compiler* creates one *Event Java Class* for each event, which is stored on the file system. In the application *Base Program*, which is created by the application developer, a generated *Event Java Class* can be instantiated and published to indicate a state change.

In *Specification of Event Modules*, application developers are able to define event

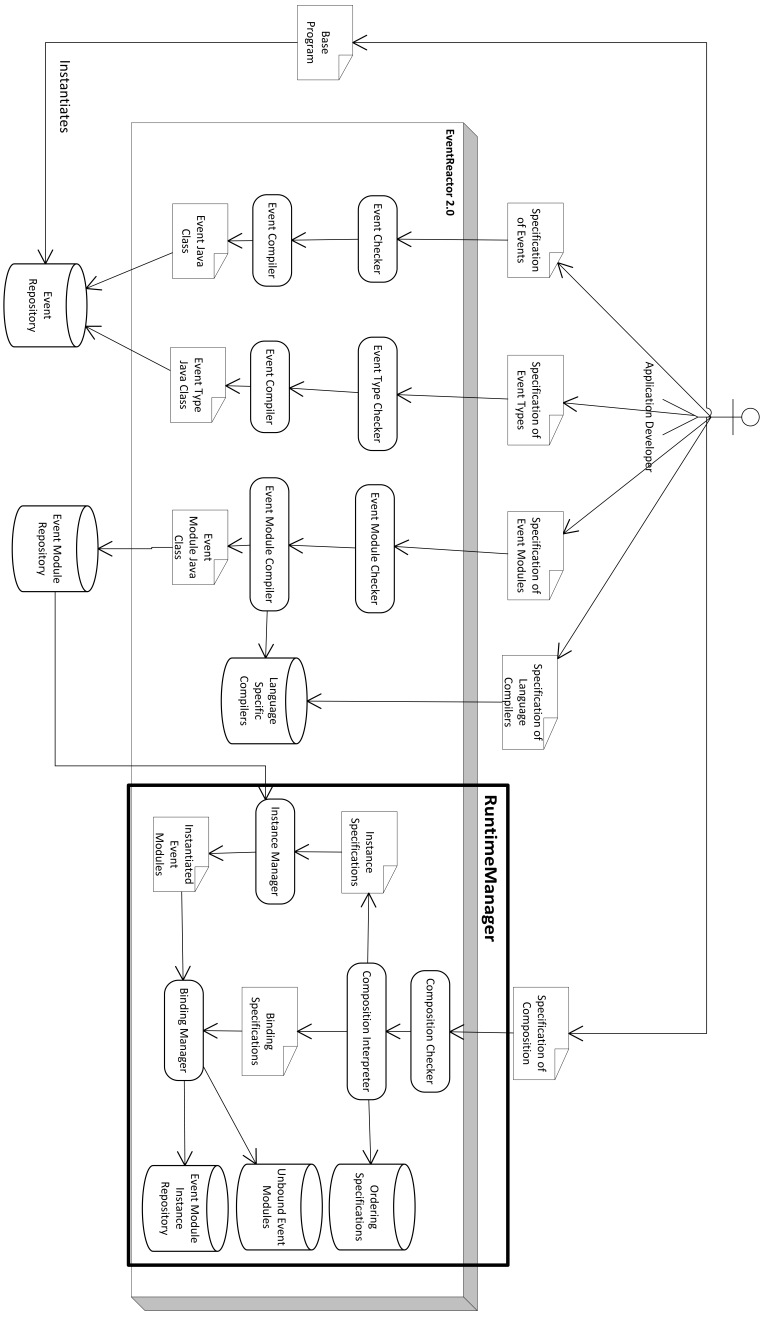

Figure 4.1: Overall Architecture of EventReactor2.0

processing agents and event consumers in an extensible list of GPLs and DSLs. The specification is provided to the component *Event Module Checker*, which verifies whether or not the syntax is correct and the language constraints are adhered to. After verification, the specification is provided to the component *Event Module Compiler*, which is responsible for translating it to one *Event Module Java Class* per specified event module. To do this, the repository *Language Specific Compilers* is used to select the correct compiler based on the language used to define *Specification of Event Modules*. A new *Specification of Language Compiler* can be defined by the application developer and added to the repository *Language Specific Compilers* so new languages to be used to define *Specification of Event Modules*. After *Event Module Java Class* is generated, it is stored in *Event Module Repository* for later use.

The application developer provides *Specification of Composition*. This is input to the component *Composition Checker* to verify the syntax and language constraints. After this, the component *Composition Interpreter* splits it up in three sections: *Specification of Ordering*, *Specification of Instantiation*, and *Specification of Binding*.
Specification of Ordering, in which the order of execution of the modules instances is defined, is directly stored in the repository *Ordering Specifications*. *Specification of Instantiations* describes how many instances of each event module is required and how they're named. This is provided to the component *Instance Manager* which also retrieves the event modules from *Event Module Repository*. Both components are used to create instances of *Event Module Java Class* as specified, to produce a list of *Instantiated Event Modules*. The component *Binding Manager* is provided *Binding Specifications*, in which is described how the event module instances are bound together. *Instantiated Event Modules* are retrieved by *Binding Manager* to create a list of unbound modules, to be stored in the repository *Unbound Event Modules*. This repository contains module instances, which do not have their output bound to the input of another module. These instances are the consumers or event processing agents which are at the start of the event processing network.
The component *Binding Manager* also sets all values in the instances as specified in *Specification of Bindings* as static bindings. All instances are stored in the repository *Event Module Instance Repository*.

The processing of *Specification of Composition* currently happens at runtime. This was done to avoid cluttering the file system with all the event module instances and the other specifications. Secondly, changing a binding, or instantiation will not require the entire compilation phase to be performed. On the other hand however, the interpreting of the binding technically falls under compilation. Secondly, performing the interpreting at runtime means every time the application is executed the entire binding interpreting needs to take place, reducing the performance. As stated, this version will perform the binding at runtime, yet moving this functionality to the compile time phase is understandable and should perhaps be done in later versions.

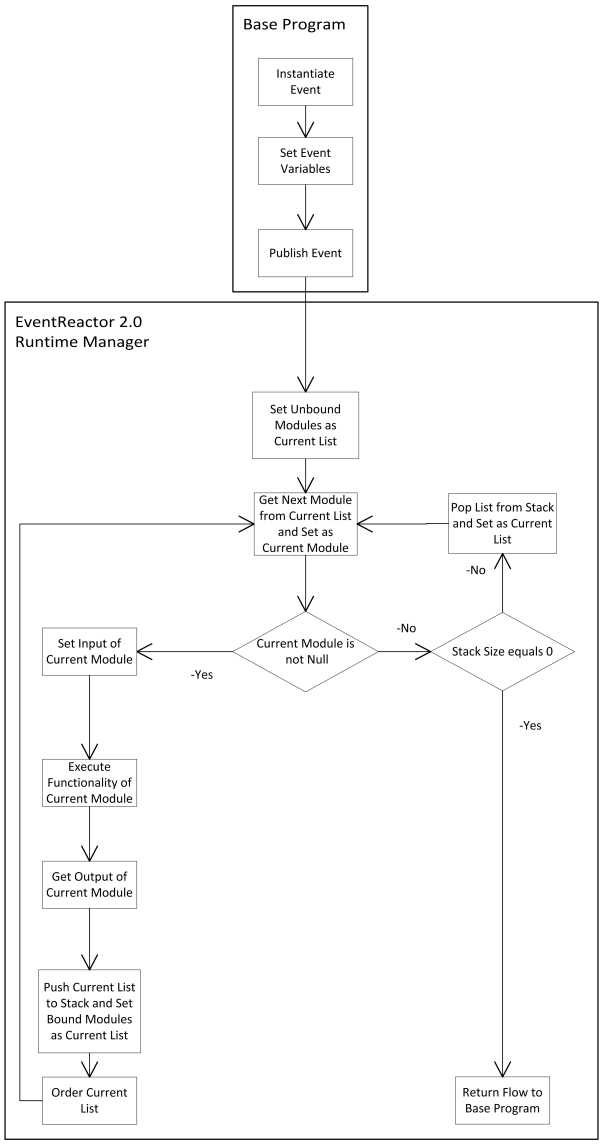

Figure 4.2: Abstract Runtime Architecture

4.2 Runtime

Figure 4.2 show a flowchart of the algorithm which is used at runtime. To start the flow, the *Base Program*, which has been briefly explained at compile time, must instantiate an *Event Java Class*. This event represents a state change of

interest in the program. After this, the application developer must *Set Event Variables*, which can for instance be the temperature measured by the sensor. When all values are set the application developer can *Publish Event*, to notify the EventReactor2.0 framework of the state change.

In the framework, the component *RuntimeManager* is responsible for receiving and processing the published event. After *RuntimeManager* receives the event, the repository *Unbound Event Modules* is used to *Set Unbound Modules as Current List*. The *current list* is the list containing event modules which are currently being traversed to be executed.
From the *current list*, the framework can *Get Next Module from Current List and Set as Current Module*. The *Current Module* is the event module to be executed next. This way, the *current list* is traversed.
Before execution, it is checked whether or not *Current Module is Not Null*. This verifies if there was still an event module in the current list.
If it is not *Null*, the framework can *Set Input of Current Module*. This sets all the input variables based on *Binding Specifications*.

After the correct values have been set *RuntimeManager* can *Execute Functionality of Current Module*. This functionality, expressed in the event module using a GPL or DSL, performs the event processing.
After the functionality is executed, the framework *Get Output of Current Module*. These are the values which are produces by the event module and can be used to bind to the input of other modules using the binding specification.
Using *Binding Specification* the names of the module instances, bound to the current module, are retrieved. Using the repository *Event Module Instance Repository*, the module instances can be retrieved based on their name, resulting in a list of event module instances which should be executed next. The framework can then *Push Current List to Stack and Set Bound Modules as Current List*. The *stack* is filled with lists containing the event modules which still need to be executed, yet are not processed at this moment. By adding the current list to the *stack* and using the bound modules first the execution is performed depth first. Using the repository *Ordering Specifications*, the framework can *Order Current List* to set the order of execution of the created event module list.

The execution of the modules is performed until the *Current Module is Not Null* check fails, meaning no more modules have to be executed for the current list. At this point the framework checks whether the *Stack Size is 0*, which means no more list left to traverse on the *stack*. If this is not the case, the framework can *Pop List from Stack and Set as Current List*, meaning the list traversed before the *current list* will continue its traversal and execution.
If there are no more lists on the *stack*, all necessary modules have been executed and the framework can *Return Flow to Base Program*, meaning the base program can continue its flow.

4.3 Extensions to EventReactor

Since the framework uses EventReactor as a basis, it is possible to examine the improvements offered by this new architecture. The following list explains the advantages offered by EventReactor2.0:

- With respect to the criteria *language extensibility*; EventReactor only DSLs are provided to implement the specific concerns. EventReactor2.0 allows an extensible list of DSLs and GPLs to be used to implement the concerns. New language specific compilers can be plugged into the framework allowing more readable event modules to be defined for specific concerns.
- With respect to the criteria *modularity*: The Prolog expressions in EventReactor can not be modularized, while the different consumer and event processing agent concerns can be modularized using the event module. In these event modules the functionalities can be defined without any knowledge required about other modules.
- with respect to the *composability* concern; module instantiation in EventReactor happens based on the attributes of the events, while EventReactor2.0 support explicit instantiation of the modules, and explicit binding of modules at instance level.

4.4 Summary

In this chapter a framework is designed which should increase the modularity, composability and language extensibility, where the examined languages and frameworks in chapter 3 fall short. The framework allows all concerns to be defined in separate modules, who have no knowledge of each other. In the configuration these modules can be instantiated after which they can be bound. This provides an increased and flexible composability at instance level. New language specific compilers can be added which can be used to define modules in DSLs or GPLs, increasing the language extensibility. With the basic architecture defined the more specific components and functionalities can be designed and implemented.

CHAPTER 5

Specification of Events

The first step in programming EPAs is to define the events which can be published. EventReactor2.0 assumes that events are typed entities, and provides dedicated linguistic abstractions to define event types and events. In section 5.1, the choices made when designing event types and events are explained including the resulting languages. Section 5.2 describes how the two concerns are compiled and how the compiled modules can be used in the base program. Section 5.3 explains some shortcomings currently still existing in the language and how they can be improved. Section 5.4 summarizes the achieved results.

5.1 Design

5.1.1 Event types

Event types are described as *a specification for a set of event objects that have the same semantic intent and same structure* [7]. Events can be seen as instances of event types.
In the following section, it is described how the language extensibility, modularity and composability requirements are fulfilled in the specification of event types.

To fulfill the requirement *language extensibility*, the framework offers a dedicated language to define event types of interest as a data structure encapsulating a set of attributes. This provided the user with the possibility to use an extensible set of event types.
To fulfill the *modularity* requirement, all event types are defined in a separate module. Providing modularized event types facilitates defining a library or reusable event types. The modules have unique names and encapsulate the specification of attributes.
To fulfill the *composability* requirement, event types can be composed with each

other via the inheritance relation. This means that a sub-event types inherits all the attributes defined in the super-event type. This allows multiple event types to have the same properties without having the need to define these properties in each event types, identically to how the events extend the event types.

Event types in the EventReactor2.0 framework, like the event types in EventReactor, all require a standard set of properties. There also exist a small set of event types which are commonly used when designing EPAs. For this reason the choice was made to have a set of standard event types available, identical to those of EventReactor. This set consists of the following types:

- *EventType* is the data structure which functions as the super type for all events. It contains the properties required for all events, which include the publisher, thread and return flow. This can be used by the events to identify where in the program the event is published and this allows a reaction to alter the flow.
- *BaseEvent* is the super data structure for all base events, which are the event published in the base program. This type contains no added attributes.
- *ReactorEvent* is the super data structure for reactor events. These events are published by the event modules, described further in the next chapter, and contain the name of the event module which publishes the event, and a reference to the event originally published which caused the reactor event to be published.
- *MethodBased* events are used to represent state changes concerning method invocations and execution. It contains all specification of the method call, which include the name of the method called, the target object which contains the method which is called, and the arguments of the method called.

Listing 5.1 shows the syntax which has been designed to describe event types.

```
1 import eventtype <EventType>;
2
3 eventtype <name> extends <eventtype> {
4   static:
5     <name>: <type>;
6   dynamic:
7     <name>: <type>;
8 }
```

Listing 5.1: Event Type Syntax

Line 1 shows the import statement which is necessary to allow one or more of the existing event types to be used in the rest of the language. The statement

requires the keyword *import* followed by the name of the event type which is to be imported. This statement can be repeated for all the event types and the statements are separated by a semicolon.
The import statements are added, because of the limitation in Xtext which prevents the super event type to be a selection between a reference to the super event type and a set of strings. By adding the import statements, the super event type is a selection between references, either the import statements or an existing event type.

Line 3 shows the beginning of an event type definition in a code block. The block is started with the *eventtype* keyword after which the name of the event type is defined, which is used to reference it. The name used to identify the event type must be unique.
Every event type has to inherit from another event type which is specified after the *extends* keyword. The extended event type can either be an earlier defined event type, or one of the imported event types.

Within an event type two types of variables can be defined. Line 4 shows the start of the code block in which *dynamic* variables are defined. The variables are set at runtime and are dependent on the base programming producing the events. Examples of these are the temperatures which are measured by a sensor. The dynamic variables code block is started with the *dynamic* keyword followed by a colon. Within the code block multiple variables can be defined, which is done by specifying the unique name, followed by a colon after which the object type is stated. The variable names must also be unique compared to the variables defined in the parent event types. The types are currently limited to a subset of the Java types, being String and Integer. Multiple variables can be define in a block and they are separated using the semicolon.
The second type of variables are the *static* variables which are described in line 6. Static variables are assigned a value at compile time, and every event which is an instance of this event type can be assigned a value to this variable which cannot be changed. These variables can, for instance, be used to specify where the created event needs to be published by including values which specify the point in the flow of the program. The static code block is specified with the *static* keyword followed by a colon. Within this code block variables can be defined identical to dynamic variables.
The code blocks, in which static and dynamic variables are described, can be created in any order, occur multiple times and can be omitted altogether.

5.1.2 Events

Events in EPAs indicate a state change. To design the event mechanism, a user will have to be able to identify where in the base program an event occurs and what the properties of the event are.

With respect to *language extensibility*, the EventReactor2.0 framework offers a dedicated language in which users can define new events similarly to the extensible event types. This is facilitates an extensible set of events, each of which can be used for specific state changes.
To fulfill the *modularity* requirement, each event needs to be defined within its own separate module. Similar to the event types, this facilitates defining a library of reusable events and requires the event to be name, ensuring they can be referenced and instantiated in the base program.
To fulfill the *composability* requirement, events are instance of event types. Events themselves only serve one purpose, which is to indicate a state change. For this reason further composition of events is not desired.

Listing 5.2 shows the syntax in which events can be defined.

```
1 event <name> instanceof <eventtype> {
2   <name> : <value>;
3 }
```

Listing 5.2: Event Syntax

Line 1 shows the beginning of the code block in which the event is specified. The code block is started with the *event* keyword and is followed by the unique name of the event. An event is an instance of an event type which is indicated by the *instanceof* keyword followed by a reference to an existing event type. Enclosed in curly brackets static variable, which have been defined in a parent event type, can be assigned a value as can be seen in line 2. A variable is assigned a value by referencing the name of the static variable, which has been defined in the event types this event is an instance off, or one of its parent types, followed by a colon and the value it must be assigned. Multiple variables can be assigned a value and the statements are separated using semicolons. A variable can only be assigned a value once per event.

5.2 Implementation Details

To use the defined events and event types in the framework, they need to be compiled and published in a base program. This section describes the necessary steps to perform these actions.

5.2.1 Event Compilation

For this purpose an Xtext project is created, as can be seen in appendix A, which implements the syntax and compilation of both the languages.

Each event and event type module is translated to a Java class. By doing this the event type name or event name and the parent event type can be directly used as the class name and super class respectively.
Within the classes maps are maintained. In the event types, a map is used to store the dynamic variables, while the static variables are stored in a map in the event. In the constructor of the events, the variables are assigned the static values. The map containing the dynamic variables is defined in the *EventType* class ensuring every event and event type can reference it through inheritance. For every dynamic variable, a getter and setter method is defined within the event type.
To support easy access to the name of the event type or the event, an extra method is added which returns the name of the object.

In the previous section four basic event types have been described. For it to be used at compile time there need to be Java classes corresponding to these. In Figure 5.1 the classes and the relation between them is defined. For each dynamic variable in the event types a getter and setter method is created. In the *EventType* class the method *getEventTypeName* is created which will return the name of the *eventTypeName* variable. This variable is set in each sub type to match the name of the event type.

In the base program, the created Xtext project must be added as a dependency, to allow automatic compilation of the events and event types to Java classes. At this point the user is able to instantiate the generated classes in the base program and publish them to the Runtime manager as shown in Figure 4.2.

At compile time, the framework is informed of the defined event types and events. The flow of this is shown in Figure 5.2. The files containing the specifications of the events and event types are passed to the *EventHandler* class. This class uses the *EventInterpreter* to translate the file to an abstract syntax tree connecting the events and event types using the existing functionalities of the Xtext project. The abstract syntax tree is then transformed to a list of event types and events which are stored locally. At this point, the repositories, which have been defined by the user, can be added to the framework at which point all the stored events and event types can be used to populate the repository allowing tagging to take place.

5.2.2 Example

As an example, an event type is created as can be seen in Listing 5.3.

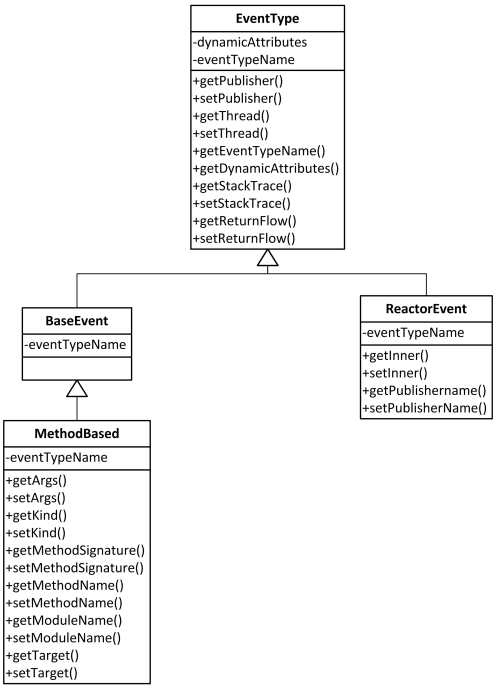

Figure 5.1: Event Type Class Diagram

```
1 import eventtype MethodBased
2
3 eventtype SensorEventType extends MethodBased{
4     dynamic:
5         Value: String;
6     static:
7         purpose: String;
8 }
```

Listing 5.3: Example Event Type

This event type is used for all events which are produced when a sensor produce a value after which a method is executed. It is called *SensorEventType* and extends the *MethodBased* event type, meaning it can contain information of methods executed. Within the event type a dynamic variable called *value* is created which can be assigned the sensed value at runtime. A static variable called *purpose* is also used meaning all events which implement this event type can be assigned a purpose which is used in all instances of that event.

In Listing 5.4 and example event is defined.

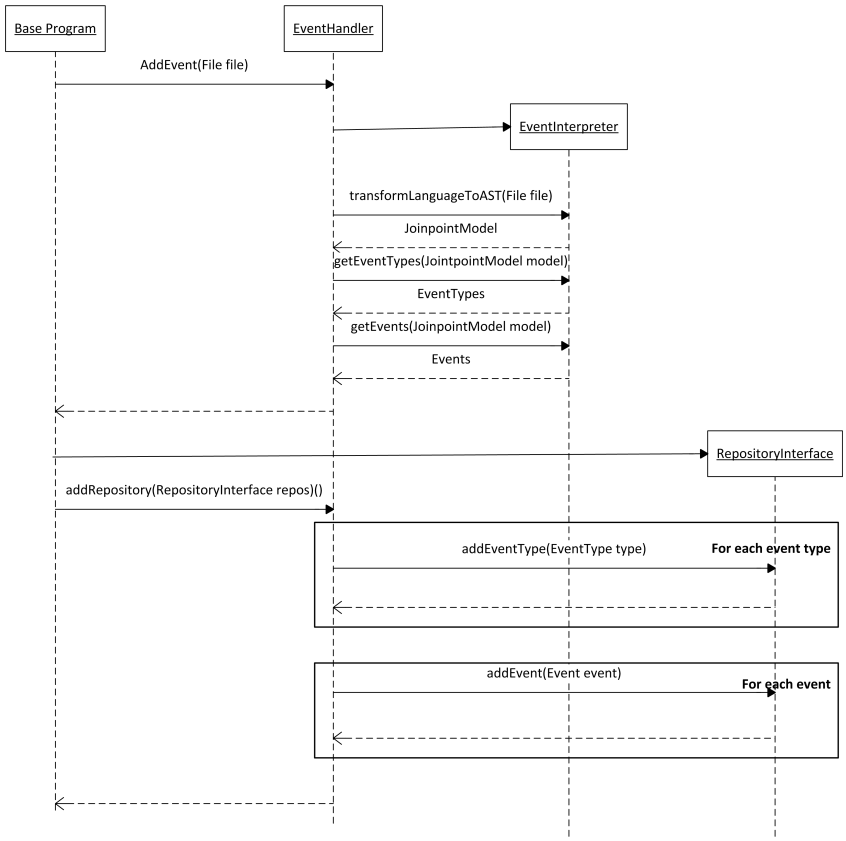

Figure 5.2: Event Handling in EventReactor 2.0

```
event TemperatureSensorEvent instanceof SensorEventType{
        purpose: "Used for temperature sensors";
}
```

Listing 5.4: Example Event

In this example a new event is created, called *TemperatureSensorEvent* which inherits from the previously defined *SensorEventType* event type example. In the event the static variable *purpose* which has been defined in the event type is assigned a value between the quotations. This value will be assigned to the *purpose* variable in every *TemperatureSensorEvent* which is published, and can not be changed.
The example event type and event can be compiled using the defined compilation. Listing 5.5 shows the resulting class from the defined *SensorEventType* event type.

```
1  public class SensorEventType extends MethodBased{
2
3    String eventTypeName="SensorEventType";
4
5    public String getValue(){
6      return (String)dynamicAttributes.get("Value");
7    }
8
9    public void setValue(String Value){
10     dynamicAttributes.put("Value", Value);
11   }
12 }
```

Listing 5.5: Compiled Event Type

The name of the class has been set as the name of the event type, which is *SensorEventType*, and extends the *MethodBased* event type. The variable representing the event type's name is set to the name defined in the language. For the dynamic variable *Value*, getters and setter are created. In those methods the name of the variable is used to store or retrieve the value to or from the map of dynamic variables.

The example event, also described in the previous section, can also be compiled. Listing 5.6 shows the resulting code.

```
1  public class TemperatureSensorEvent extends SensorEventType{
2
3      private static final Map<String, String> attributes = createMap
       ();
4
5      public static Map<String, String> createMap(){
6        Map<String, String> attributes = new HashMap<String, String
       >();
7        attributes.put("purpose", "Used for temperature sensors");
8        return attributes;
9      }
10
11     public Map<String, String> getStaticAttributes(){
12       return attributes;
13     }
14
15     @Override
16     public String getEventName(){
17       return "TemperatureSensorEvent";
18     }
19
20 }
```

Listing 5.6: Compiled Event

The class which is generated will have the same name as the defined event, which is *TemperatureSensorEvent*, and inherits from the earlier compiled *Sen-*

sorEventType event type. In the class, the static attributes are defined in a map. At instantiation the method *createMap* is executed which sets all the values as defined in the event. A method is added which returns the static attributes. Finally a method is defined which returns the name of the event.

With these compiled classes the class diagram is extended for this specific EPA as can be seen in Figure 5.3.

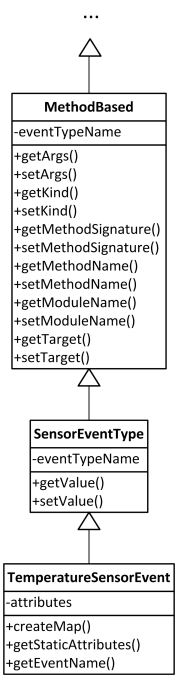

Figure 5.3: Compiled Events Class Diagram

In the base program, events can now be published, and example of which can be seen in Listing 5.7.

```
1  TemperatureSensorEvent event = new TemperatureSensorEvent();
2  event.setValue(this.value});
3  ...
4  //Set all variables.
5  try {
6    RuntimeManager.getInstance().publishEvent(event);
7  } catch (EventException e) {
8    e.printStackTrace();
9  } catch (FocalRuntimeException e) {
10   e.printStackTrace();
11 }
```

Listing 5.7: Event Publishing

In order to do this, the specific event, in this case the *TemperatureSensorEvent*, needs to be instantiated at the point in the base program where the state is changed. At this point, the dynamic variable *Value* can be set to the desired value. Finally, the *RuntimeManager* is retrieved and the instance of the event is published to it.

5.3 Future Work

In the current design of the event and event type mechanism, there are still room for improvement for among others the extensibility and usability of the concerns. In this section, some possible improvements are described and it is discussed why this would be useful and how this could be implemented

5.3.1 Language Extensibility

There is currently no extensibility in the languages used to define events and event types. In future implementations of the framework, it might be possible to allow new language to be added to perform these actions.
To support this type of extensibility, the user must define compilers for these languages, which will translate the events and event types to Java classes, and be able to add these to the framework. In the event types and events language, it must be indicated which language is used in order to select the correct compiler.

5.3.2 Usage of Java types

As described in the language design, the static and dynamic variables can currently only be defined using a subset of Java which only includes String and Integers. Though this allows the framework to be tested to a certain extend, when this framework is to be used for more comprehensive EPAs the use of all Java types should be supported. In order to do this, the biggest obstacles are firstly changing the XText project in which the syntax is defined to be aware of and recognize all possible types, and secondly knowing which packages need to be imported in the Java class to support the use of these types.

5.3.3 Automatic Publishing

As described in the compilation section, events need to be manually instantiated, assigned the values and published to the runtime manager. In AO languages such as AspectJ, the join points are automatically detected by the compiler in

the base program, allowing the Java byte code to be altered with a call to the aspect. This reduces the amount of work for the users and means the base code does not need to be changed in order to perform a reaction. For events which inherit from the *MethodBased* event type, the same sort of mechanism could be used, where the definition of the event can be used in order identify where in the base code it needs to be published. This requires the dynamic attributes of the *MethodBased* event type to become static, allowing them to be assigned a value in the *Specification of Events*. The compiler will then have to traverse the byte code of the base program to identify which method calls match the attributes set in the event. At that point, the byte code can be changed and an instantiation and publishing of the event can be inserted. To make it more usable, the compiler should identify which dynamic attributes are required, and will set them in the event allowing the flow of the base program to be altered.

5.3.4 Extensible Base Language

In the future, the framework should be able to be used for base programs written in a different base language. To support this, the event types and events should be compiled to one of a extensible set of base languages. To facilitate this, the mechanism needs to be changed to contain a set of compilers which can added or removed when necessary. When compiling, the correct compiler needs to be selected based on the chose of the user, or automatically selected after identifying the base program.

5.4 Summary

This chapter proposes an event language which allows user-defined events and event types. The events can be instantiated and published at runtime to indicate a state change. Even though the language can still be extended, an extensible list of modularized and composable events and event types is offered to the user.

CHAPTER 6

Specification of Event Modules

The EventReactor2.0 framework offers event modules as a means to modularity implement event processing agents and event consumers. This chapter describes the linguistic constructs which have been created to facilitate this. In section 6.1, the language is designed and the syntax and semantics are explained. In section 6.2, explains how extensible languages can be used to define the actions. In section 6.3, the mechanism used to compile and executed the language is described. In section 6.4, possible improvements to the language is described. In section 6.5, the results are summarized.

6.1 Design

In the EventReactor2.0 framework, the event processing agent and event consumer functionalities are defined in *event modules*. These modules are assigned a unique name and have well-defined interfaces that encapsulate the functionality [19]. By providing this structure to implement the event processing agents and event consumers, the *modularity* requirement is fulfilled.
To fulfill the *language extensibility* requirement, the section in which the functionality is defined can be programmed using an extensible list of DSLs and GPLs.
To fulfill the *composability* requirement, the event modules are not required to explicitly refer to each other. Composition between event modules happens at interface level and is done using the composition script explained in chapter 7.

With the requirements, a language was created. The syntax of this language is shown in Listing 6.1.

```
1  eventmodule <name>{
2
3    input{
4      (<type> <name>)*;
5    }
6
7    local{
8      (<type> <name>)*;
9    }
10
11   functionality language=<language>{
12     <Statement>*
13   }
14
15   output{
16     <type> <name>;
17   }
18
19 }
```

Listing 6.1: Specification of an Event Module

In line 1, the *module* keyword is used, together with a user-defined name, to define the module and encapsulate the components. Within this module, four separate sections are defined. In line 3, the *input* keyword is used to start the code block in which the variables are defined which can be assigned a value in the composition. In line 4, a variable can be defined by specifying an object types followed by the name which will be used within this module. In the input, zero or more variables can be defined and they are separated by the semicolon. In line 7, the second code block is started indicated with the *local* keyword. It contains a set of variables which are used to maintain a state. The variables are defined in a similar fashion as the input, yet these variables can not be assigned a value in the composition. In this section, zero or more variables can be defined identical to the input. The local section is not required and can be omitted. Variables which are defined in the input and local section must have a unique name.

In line 11, a new code block is started using the he *functionality* keyword. In this block, the event processing agent and event consumer functionalities can be defined. To specify which compiler has to be used, a language needs to be specified. This can be done after the language keyword, and should be available in the extensible list of languages. The implementation of the functionality is dependent on the language which is chosen, but a Java and Prolog example are discussed after this.

The final code block, in line 15, starts with the *output* keyword and contains the list of variables which will be produced as output. Within the section, a list of variables is defined in the same way as the input and local sections. The variables which are defined in the output must have already been created earlier

in the module.

6.2 Functionality Languages

The EventReactor2.0 framework currently support the Prolog and Java language to implement the functionality section of event modules. In this section, the syntax and semantics of these languages are explained, as well as possibilities of adding more languages, where SQL is used as an example.

6.2.1 Prolog

In Prolog, rules can be defined to filter the events of interest based on the static and/or dynamic attributes or events. First, a set of rules is explained which are provided in the current version of EventReactor2.0:

isChildEvent(X, Y): Y is event type of X, or the event type of X is a direct or indirect child type of Y

isChildType(X, Y): Y is the event type of X, or the event type of X is a direct or indirect child type of Y

isEventWithName(X,Y): Y is the name of event X

isEventTypeWithName(X,Y): Y is the name of event type X

hasAttribute(X, Y): Y is name of attribute of event X or of direct or indirect event type of event X

hasDynamicAttributeWithValue(X, Y, Z): Y is the name of an attribute of the direct of indirect event type of X, which has the value Z.

hasStaticAttributeWithValue(X, Y, Z): Y is the name of an attribute of event X, which has the value Z.

hasStaticAttribute(X, Y): Y is the name of an attribute of event X.

hasDynamicAttribute(X, Y): Y is the name of an attribute of the direct or indirect event type of event X

hasDynamicAttributeWithNameAndType(X, Y, Z): Y is the name of an attribute of the direct or indirect even type of X, and is of the type Z.

To conform to the rules of a module, instead of returning a true or a false, a event processing agent needs to return an event if the processing was successful and *null* if it was unsuccessful.

```
1 <eventX> = {<eventY>|<rule>(<var>,...), (<rule>(<var>,...))*};
```

Listing 6.2: Prolog Filter Statement

Listing 6.2 shows the statement in which users can define the rule. In this statement, *eventX* will be set to either *eventY*, when the query returns *true*, or *null*, when it returns *false*. *eventY* contains an event to be checked with the query, and can also be *eventX*. Due to the nature of Prolog, the event which is used within the filter needs to start with a capital letter. Because the filter will be translated to a rule in Prolog using the module name as the name of the rule, the module can currently only be named starting with a lowercase letter.

```
1  eventmodule temperatureSensorEventFilter{
2      input{
3          EventType Et;
4      }
5
6      functionality language=prolog{
7          Et = {Et| isEventWithName(Et,'TemperatureSensorEvent'),
       hasDynamicAttributeWithValue(Ev, 'value', '10')};
8      }
9
10     output{
11         EventType Et;
12     }
13 }
```

Listing 6.3: Prolog Example Module

Listing 6.3 shows a module using the Prolog language to filter events. In line 3, an input is defined which specifies that the modules receives the published event. Within this module, the event will be referred to by the name given in the input, in this case *Et*. For this module, a state does not need to be maintained, meaning the local variables can be omitted
In line 6, the functionality is defined in the Prolog language. Line 7 contains the one Prolog statement, which checks whether the published event is of the type *TemperatureSensorEvent*, and whether the *value* attribute, defined within the event, has the value 10. If this is the case, the variable *Et* maintains its value, otherwise it is assigned *null*. Line 10 start the definition of the output in which the event type *Et* is produced as output.

6.2.2 Java

Along with Prolog, the EventReactor2.0 framework currently also support the Java language to implement the functionality of event modules.
Because it is a GPL, it is able to perform all types of consumer actions and because of that is sufficient for the first version of this framework.
Listing 6.4 shows the syntax of the functionality using the Java language.

```
1 `` (<java statement>;)* ``
```

Listing 6.4: Java Functionality Syntax

What is noticeable is the need for the double grave accent(``) delimiters. This is required, because the Xtext editor and compiler need to be able to differentiate between the base module language and the Java language. This solution does not influence the functionality or performance, but should be improved at a later time.
Regular Java statements can be used between the delimiters.

To improve the usability of the languages used in the functionality, import statements have been added to the base event module. This allows objects to be used within the functionality without having to reference the entire package name every time.

```
1 import <packagename>;
```

Listing 6.5: Import Syntax

Listing 6.5 shows the syntax which is identical to the import statements in Java classes.

```
eventmodule tempSensorReaction{
    input{
        EventType Et;
        String name;
    }

    local{
        int couter;
    }

    functionality language=java{
        `
        counter++;
        System.out.println("Sensor "+ name +" measured value '10'");
        System.out.println("This value has been measured "+counter+" times");
        `;
    }

    output{
    }

}
```

Listing 6.6: Example Java Module

Listing 6.6 shows an example event module which is implemented using the Java language. In this case the module receives as an input the event, which is named *Et*, and the String named *name* which will contain the name of the reaction. For this module, a state is maintained counting the amount of times this reaction has been executed. This is done by adding the *local* section in which the integer *counter* is defined. In the part *functionality*, specified with the *java* language name, the counter is raised by one every time it is executed. After this, two lines are printed informing the user about the action which has just occurred.

6.2.3 SQL

In the current implementation of the EventReactor2.0 framework, both Prolog and Java are supported, but new languages can be added. In this section, it is explained how the SQL language can be used.

```
<event> = <SQL Statement>
```

Listing 6.7: SQL Functionality Syntax

Listing 6.7 shows an example syntax should be able to perform the basic actions. When an event gets published it must be stored in a database. When the event module is executed the defined query is executed to see whether or not an event

in the database matches.
The SQL statement has been abbreviated for this thesis, since the syntax is of no real importance for this thesis and can become elaborate.

```
1  eventmodule temperatureSensorEventFilter{
2      input{
3          EventType Et;
4      }
5
6      functionality language=sql{
7          Et = SELECT * FROM Events WHERE Events.name='
       TemperatureSensorEvent' AND DynamicValues.value='10';
8      }
9
10     output{
11         EventType Et;
12     }
13 }
```

Listing 6.8: Example SQL Module

Listing 6.8 shows an example module using the SQL language, which should perform the same action as the previously defined example Prolog module, as can be seen in Listing 6.3.

6.3 Implementation Details

The EventReactor2.0 framework must be provided the event modules in order to compile them and use them at runtime. In this section, the compilation of both the base module and the functionality languages is discussed and how the framework works internally to make that happen. It also contains a description how new functionality languages can be added to the framework.

6.3.1 Event Module Compiler

At compile time, the framework must be provided the event modules defined by the users. Figure 6.1 shows the specific flow of the framework to support such actions and is a more detailed description of the abstract design in Figure 4.1. In this diagram, the Xtext and Java compiler functionalities have been abbreviated to avoid clutter, since these require a fair amount of method calls and the names are not very descriptive.
The event modules are added to the framework by creating an instance of the *EventModuleCompiler* class and passing it all the files containing event module specifications. After collecting all the event modules, the *Compile* method is called. The base module Xtext project, shown in Appendix B, is used for each

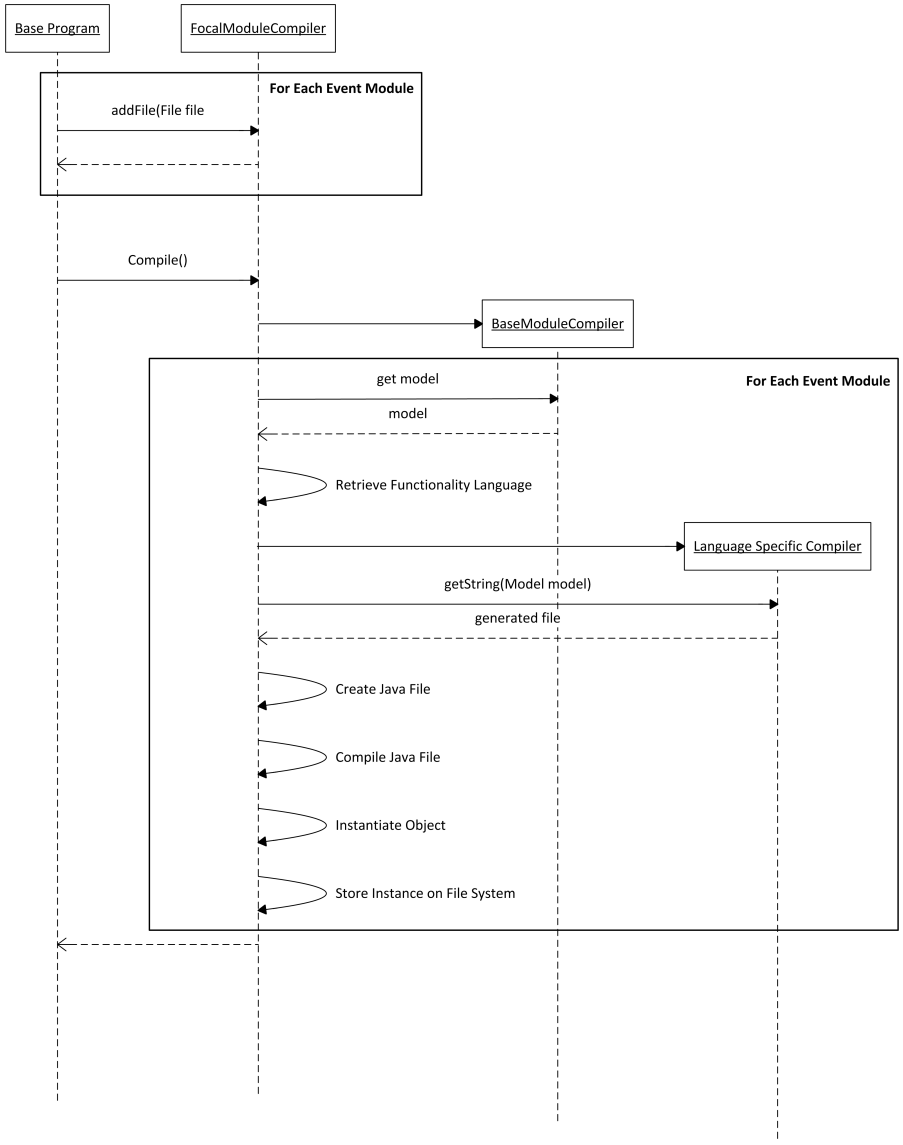

Figure 6.1: Module Compilation Process

event module to verify and interpret the basic components of the event module. At this point, the name used to identify the language of the functionality is retrieved and is used to select the correct Xtext project. Using the compiler of the correct language specific compiler, the event module is compiled to Java code.

To avoid cluttering the base project, the Java classes, representing the event modules, are directly compiled to Java byte code using the Java compiler. To support events to be tagged at compile time, the classes are instantiated after compilation. Finally, the instances of all event modules are stored on the file system. This avoids the need to instantiate the classes again at runtime. At runtime, these instances are retrieved from the file system at which point they can be executed.

As shown in chapter 4, a repositories can be provided allowing the extensible functionality languages to query their specific data store. As mentioned in chapter 5, the events and event types are stored in these repositories. For the event types, the name, super event type, and both the static and dynamic variables names and their types are stored. For the events, the name, the name of the event type it is an instance of, and the static variable names and their values are stored. At this point, the queries defined in the event modules can be used to select events of interest based on their static properties, or *tagging*. These tagged events are stored on the file system and can be retrieved at runtime. By performing the query at compile time and tagging the events of interest, the execution of these same queries can be avoided at runtime increasing the performance.
Because the tagging of events only uses the static structure of the events and event types, queries which rely on the dynamic attributes of an event or event type can not be used for tagging. For this case, tags can be marked as either static or dynamic. Static tags are used to indicate the query only uses the static structure of an event when it is tagged, while dynamic tags indicate that the static parts of the event match the query, yet still require the dynamic variables to be checked at runtime.

When an event is published to the runtime manager, it is added to the relevant repositories. After this, the event modules are executed according to the composition script.

6.3.2 Generated and Base Classes

In the EventReactor2.0 framework, event modules are compiled to Java classes. To support this, an abstract super-class was created called *Module*. In Figure 6.2, a class diagram is shown of the module and the created sub-class. The abstract class is responsible for three variables:

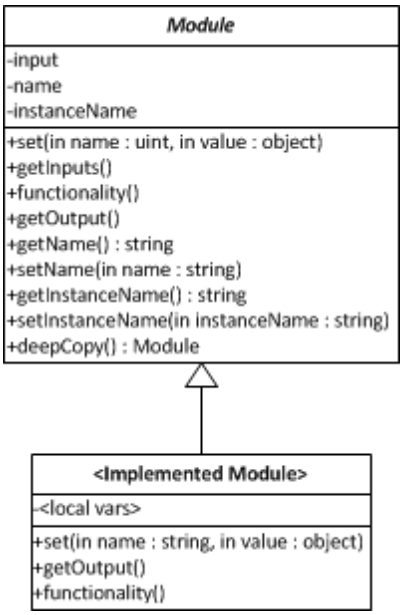

Figure 6.2: Module Class Diagram

- **input** is a map containing the name of all the input variables and the types they have been assigned. This is used for type matching when binding input and output.
- The **name** is the name of the module assigned as defined earlier in this chapter.
- The **instanceName** is the name given to the exact instance of this instance. This is discussed further in the next chapter.

Besides getters and setters for the variables, there have been three abstract methods which are defined:

- The **set** method takes as an input the name of a variable, which has to be defined in the input of a module, and the value which it has to be assigned. In the method the names of the variables maintained in the class are compared to the given name, when the correct variable is found the value is assigned.
- The **functionality** method described the actions, defined in the functionality section of the module, in Java.
- The **getOutput** method collects the values of all the variables which are defined in the output section of the module and returns them in the form of map containing the variables names as well for reference.

Finally, a *deepCopy* method is provided which creates an identical copy of the instance and all the instances it references, to make sure the copied instance does not reference the exact same variables. This method is required, because instances of the compiled event modules are stored on the file system, and are created of the compiled event modules

When compiling the module, first the base compiler creates a Java class which is named identical to the name of the module and inherits from the abstract Module class. For each of the variables defined in both the *input* and the *local* section, a private variable gets created with the defined name and type. The *set* methods are implemented by creating code comparing all the input variables names to the provided name and setting the value when they match. In the *Output* method, all the variables defined in the output are added to a map with as a key the name of the variable, which is returned.

For the repositories, the *RepositoryInterface* is provided which must be implemented. For this repository, a few basic methods need to be implemented in order to support querying.

- The **addEventType** and **addEvent** methods, as the name suggests, add the basic structure of both events and event types to the database. This allows them to be used for tagging purposes.

- The **resetDatabase** method removes the temporarily added event which was published, and makes sure the repository can be used for the next published event without interference.

- The **findEvents** mis used to actually perform the query and must return either the found query, or *null*.

- The **tagEvents** takes as an input a query and uses the earlier store events and event types to tag events which currently match the query. Secondly, it can indicates whether or not the tag was static or dynamic. When it is statically tagged, the query does not need to be executed again when a matching event is passed. This method is not required, but offers some extra functionalities.

- The **backup** method stores the current repository on the file system, to allow tagging to take place at compile time and still support the use of the repository at run time.

- The **restoreDatabase** retrieves the repository from the file system and restores it for use at run time.

6.3.3 Compilation of Functionality Languages

For the functionality section, a compiler has been created for both the Java and Prolog implementation in their respective Xtext project.

For the Prolog language, a repository is created using the *tuProlog*[x] engine. For this repository, two files are created implementing two sets of the rules defined in section 6.2. In the first set, the rules are directly implemented as defined, while the second set the dynamic variables are ignored. The latter is used for tagging events while the former is used when executing a query at runtime.
At compile time, the repository is provided with the user defined events and event types. The repository creates a file and populates it with the static information of these event types and events. For the event type, the following facts are used, with examples given the events and event types specified in chapter 5:

- **eventtype(X)**: Where X is the name of the event type, which results in **eventtype('SensorEventType')**.
- **dynamicattribute(X, Y, Z)**: Where X is the name of the dynamic variable, Y is the type of the variable, and Z is the name of the event type. This results in **dynamicattribute('Value', 'String', 'SensorEventType')**.
- **staticattribute(X, Y, Z)**: Where X is the name of the dynamic variable, Y is the type of the variable, and Z is the name of the event type. This results in **staticattribute('purpose', 'String', 'SensorEventType')**.
- **supertype(X, Y)**: Where X is the parent event type and Y is the child event type, which results in **supertype('MethodBased', 'SensorEventType')**

For the event, the following facts are used:

- **event(X)**: Where X is the name of the event, resulting in **event('TemperatureSensorEvent')**.
- **type(X, Y)**: Where X is the event type of which the event Y is an instance. This results in **type('SensorEventType','TemperatureSensorEvent')**.
- **attribute(X, Y, Z)**: Where X is the variable name of the static variable, Y is the value assigned to it, and Z is the name of the event. This results in **attribute('purpose', 'Used for temperature sensors', 'TemperatureSensorEvent')**.

When compiling the Prolog event modules, the query defined by the user is verified by the Prolog Xtext project from appendix C. In the constructor of the generated Java class, the defined query is added as a new rule to the repository, using the event module name as the rule name. When the rule is added, the repository is instructed to tag all relevant events. In the repository, the newly added rules and the previously stored events and event types are used to find events of interest. These are tagged in the repository using the **tag(X, Y, Z)** fact, where X is the name of the rule, Y is the name of the event which is tagged, and Z indicates whether or not the tag is dynamic.

In the functionality method, code is added which calls the repository using the published event and the name of the event module. In the repository, the name of the event module is used to retrieve all the events tagged at compile time, which are compared to the published event. If the types match, two options exists: either the tag is static and the event, which is provided, is returned. If the tag is dynamic, the dynamic attributes of the event are stored in a file using the following facts:

- **attribute(X, Y, Z)**: Where X is the name of the attribute, Y is the value assigned to the variable, and Z is the name of the Event
- **type(X, Y, Z)**: Where X is the name of the variable, Y is the type of the value and Z is the name of the event.

Using these facts with the already defined static facts of the events and event types, and the set of dynamic rules, the query is executed again. If the published event is returned when performing this query, this event is returned to the module. If nothing is returned, or if no tag was found with the query, *null* is returned.

The Java Xtext project, shown in appendix D, contains a similar compiler. In this compiler, the code defined in the functionality of a Java event type is directly copied to the functionality method of the Java class.

6.3.4 Extending the Functionality Language

To add new functionality languages, the user must create a new Xtext project in order to define the syntax, checker and compiler of the language. In the new project, the Xtext project of the base module, defined in appendix B, must be added as a dependency in the plug-in. Secondly, the syntax must be extended with the syntax of the base module after which the first rule must be defined as a reference to the first rule of the base module syntax. At this point, the syntaxes are identical and in the *languageStatement* rule, the syntax of the new

language can be implemented.

In the compiler, the compiler of the base module is extended after which the abstract methods of the base module are implemented in the new compiler. The compiled code can be added to both the constructor and the functionality of the class.

When the compiler has been defined, the framework needs to be notified of the new language. In the *EventModuleCompiler* class, a compiler map is defined mapping the name of the language, which must match the name defined in the *functionality* section of the module, to the specific compiler. Secondly, the injector map is defined mapping the language to a specific injector of the language, allowing code to be inserted. Both these maps need to be extended to include the compiler and injector of the new Xtext project.

For some language, new repositories are required which can be added by implementing the *RepositoryInterface*. At compile time, the repository must be provided to the framework to be used as described earlier in the internal design.

6.4 Future Work

The designed modules are currently able to perform all the required actions, yet should be improved for the future version to increase usability and remove limitations. The following list describes some of the improvements that can be implemented.

- The syntax used to defined the module still faces some limitations due to Xtext or the mechanisms behind it. Examples of this are the requirement of delimiters when using the Java language, or the requirement of Prolog modules having to start with a lower case letter. The former means a change in the Xtext compiler while the latter requires the mechanism behind which could translate every modules first letter to lower case.

- Similar to the events, described in the previous chapter, the types used to define variables is currently restricted to a subset of Java. To increase the usability all Java types should be usable when defining the variables.

- The event type object used as input type can be extended to increase functionality. By using the events or event types defined by the user, filtering can already take place at the input. If, for instance, the user specified that as an input the *temperatureSensorFilter* is required, only those specific events can be used as input for that module.

6.5 Summary

The language designed in this chapter has been specifically created to increase modularity and language extensibility. The modularity is increased by having every event consumer and event processing agent defined in a separate module without any knowledge of each other. The module, which is named, contains both an input and an output allowing them to be composed to create more complex functionalities. The functionality within a module can be defined using an extensible set of GPLs and DSLs which increases the language extensibility. There are currently two languages implemented for the functionality, with Prolog for event processing agents, and Java for both event processing agents and event consumers. There are still some parts of the design which can be improved. These parts are discussed in this chapter, but these must be implemented in a later version.

CHAPTER 7

Specification of Compositions

In the EventReactor2.0 framework, the event modules can be composed allowing the construction of the event processing network. The framework offers a dedicated language to define the composition. In section 7.1, this language is designed and the syntax and semantics are explained. In section 7.2, describe the details of the framework and explains what actions are performed in order to compose the event modules. In section 7.3, possible improvements are discussed. In section 7.4, the results are summarized.

7.1 Design

Event modules provide the functionality to process the events of interest. Event modules can be composed with each other, allowing users to construct an event processing network.
The composition between event modules takes place at the level of interfaces. This is done by binding the input and output interface, provided by the event modules, to each other.
When binding the input and the output interface, the EventReactor2.0 framework allows for variables to be ignored on both ends. This allows event modules with varying amount of variables in their interface to be bound.

In the event modules, composition constraints can be defined and modularized. This facilitates implementing the constraints in the language suitable for each specific problem, and flexibly changing the constraints.
The framework also provides a mechanism to influence the order of execution of event modules when two or more need to be processed on a shared event.
Event modules can be instantiated multiple times, with each of the instances explicitly configurable. This is a concept adopted from the OO paradigm and allows each instance to maintain its own state, increasing the reusability of the event modules.

Listing 7.1 shows the language which can be used to specify the composition of event modules.

```
1  modules {
2      (<instance-name> : <module-name>;)*
3  }
4
5  bindings {
6      bind((<instance-name1>,)? <instance-name2>){
7          ((<value>|<variable-name>) -> <variable-name>;)*
8      }
9  }
10
11 ordering {
12         (<instance-name>;)*
13 }
```

Listing 7.1: Composition Specification Language

In line 1, the instantiation code block is started using the *modules* keyword. Enclosed in curly brackets, the various instances can be defined. Line 2 shows how instances can be created. First, a unique instance name is defined followed by a colon and the name of the module which is to be instantiated. Event modules can be instantiated multiple times and each instantiation is separated by a semicolon.
Line 5-9 shows the specification of binding for event modules. In this code block, multiple bindings can be defined, each of which contained within its own code block. Line 6 shows the start of this code block using the *bind* keyword, which is followed by brackets in which one or two event modules instances can be defined, if necessary, separated by a comma. If one event module instance is defined, the static values can be assigned to the selected instance. If two event modules instances are defined, the two instances are connected for the event processing network.
In line 7, the binding details are defined, where input variables of an event module can be bound to either a value or an output variables of another event module. A static binding is created by defining the static value followed by an arrow ($->$) and the name of the input variable which is assigned the value. For the dynamic binding the syntax is similar, but instead of the static value an output variable has to be referenced. Multiple bindings can be defined and must be separated by a semicolon. Input variables of an event modules van be bound multiple times, but only the binding defined last will be used.

In line 11, a code block is created starting with the *ordering* keyword, which is used to specify the order of execution. Enclosed in curly brackets, instances of event modules can be referenced separated by semicolons. The instance which is referenced earliest will be performed earlier when multiple instance from the list have to be executed at the same time. If multiple modules have to be executed and none of them is mentioned in the ordering, the order in which the bindings are defined are used to set the order. If only a subset of the event

modules, which have to be executed, are set in the ordering, those mentioned will be executed first using the order set, after which the rest is executed in the order in which the bindings are declared.

Listing 7.2 shows an example composition script for the event modules defined in Listings 6.6 and 6.3.

```
1  modules{
2      tempFilter1:temperatureSensorEventFilter;
3      tempReaction1:tempSensorReaction;
4      tempReaction2:tempSensorReaction;
5  }
6
7  bindings{
8      bind(tempReaction1){
9          "Reaction 1"->name;
10     }
11
12     bind(tempReaction2){
13         "Reaction 2"->name
14     }
15
16     bind(tempFilter1, tempReaction1){
17         Et->Et;
18     }
19
20     bind(tempFilter1, tempReaction2){
21         Et->Et;
22     }
23 }
24
25 ordering{
26     tempReaction2;
27     tempReaction1;
28 }
```

Listing 7.2: Example Composition Language

In line 2 of Listing 7.2, one instance of the *temperatureSensorEventFilter* event module is created and is named *tempFilter1*. In line 3 and 4, two instances of the *tempSensorReaction* event modules are created, which are named *tempReaction1* and *tempReaction2*.
In line 8 and 12, two static bindings are defined, one for each of the *tempSensorReaction* instances, in which the names are set. In line 16 and line 20, bindings are defined where the *temperatureSensorEventFilter* instance is bound to both the *tempSensorReaction* instances, in both cases binding the *Et* output variable to the *Et* input variable.
In line 26, the order of execution is specified to ensure the *tempReaction2* instance is performed before the *tempReaction1* instance. If this ordering was removed, the order in which they are bound is used to define order of execution, which in this case would be the reverse of the specified order.

Using the example composition script, 3 instances are created at the start of execution. In the *tempReaction1* and *tempReaction2* instances, the *name* input variable are set as "Reaction 1" and "Reaction 2" respectively.
At a given point in time, the base program publishes a *TemperatureSensorEvent* with the value of *value* set at 10. At this point the unbound modules are selected, which is the *tempFilter1* instance. Using the event, the functionality is executed. After this, the input variable *Et*, of both bound modules *tempReaction1* and *tempReaction2*, is assigned the value the output variable *Et* provided by the *tempFilter1*. As specified in the ordering, first the functionality of *tempReaction2* is executed, after which the functionality of *tempReaction1* is executed. This results in the output shown in Listing 7.3.

```
1 Sensor Reaction 2 measured value '10'
2 The value has been measured 1 times
3 Sensor Reaction 1 measured value '10'
4 The value has been measured 1 times
```

Listing 7.3: Output of Composition Example

If the same event type is published again the same output is produced, safe for the number representing the amount of times the functionality is executed, which will increase by 1 every time. If any other type of event, or the *TemperatureSensorEvent* with a different value for the input variable *value* is published, the filter does not succeed and none of the reactions are performed.

7.2 Implementation Details

To compose the event modules at runtime, the framework needs to interpret the composition script, perform the functionalities of the event modules in the correct order, and bind the set the input of event modules using the output of another event module.

At runtime, two actions are used to compose the event modules. The first is performed at the start of execution, where the composition script is provided to the framework. Secondly, and more often, an event is published which triggers the framework to execute the event modules. Both of these actions are currently handled by the *RuntimeManager* as explained in Figure 4.2.

For the first action, which can be seen in Figure 7.1, the *RuntimeManager* is instantiated which in instantiates the *ModuleManager*. In the constructor of the *ModuleManager*, the instances of the event modules, which were stored on the file system as described in section 6.3, are retrieved.
At this point, the user supplies the composition specifications to the *RuntimeManager*. Using the Xtext project, which can be seen in appendix E, an abstract syntax tree is generated.

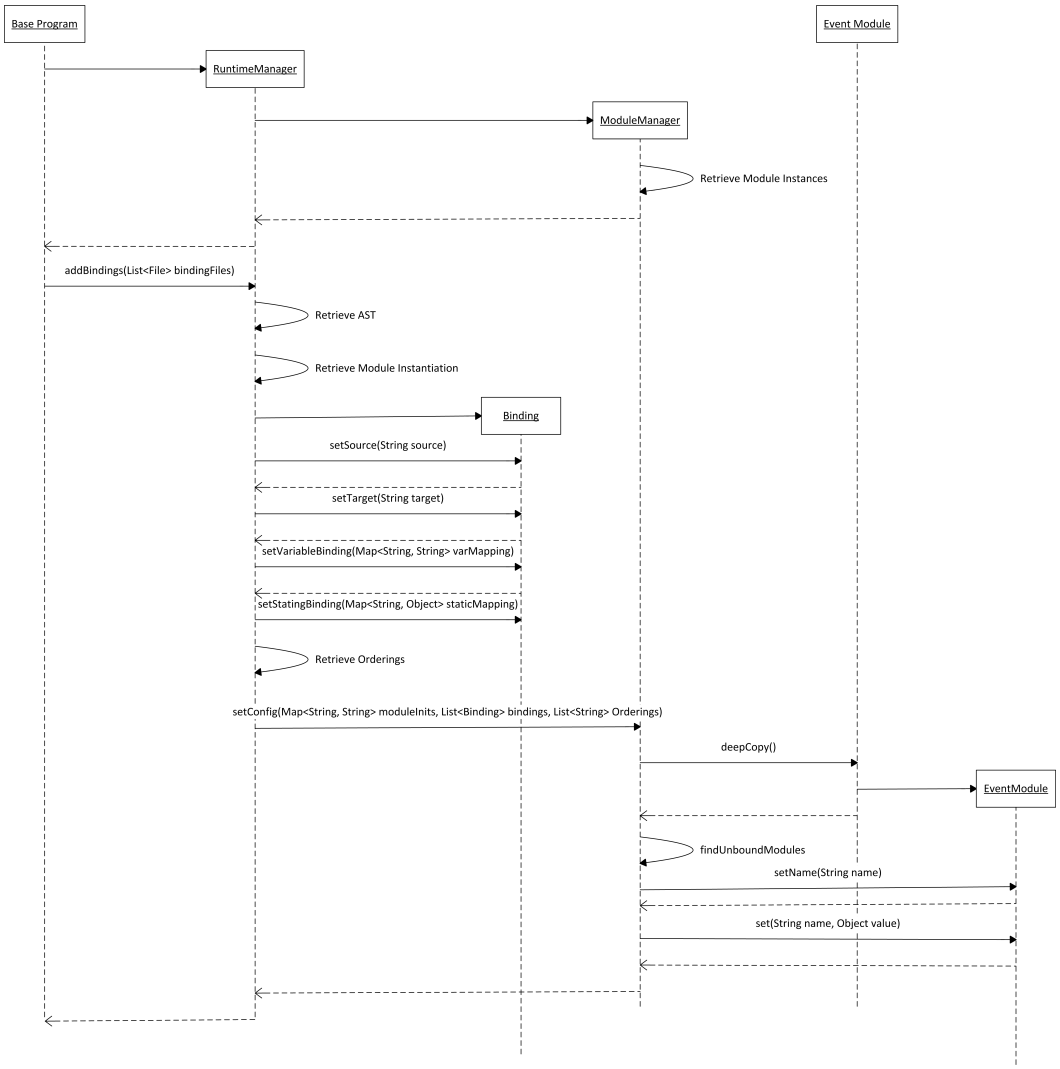

Figure 7.1: Initializing Composition

By examining the abstract syntax tree, the three sections are retrieved. First, the instantiation specifications are retrieved. In a map, both the name of the instance and the name of the event module are stored.
For each binding specification, a *Binding* object is instantiated and stored in a list. In the binding object, the *target* is assigned the name of the instance who's input will be bound. The *source*, if specifying a dynamic binding, will be assigned the name of the instance whos output will be bound to an input. When a static binding is specified, the static bindings are set in the object, which is a map containing variable names and the value assigned to them. When a dynamic binding is specified, the dynamic attributes are set in the object, which is a map containing names of the output variables of the source instance and the names of the input variables of the target instance.
The *orderings* are finally retrieved from the abstract syntax tree and stored in a list, ensuring the order is maintained.

The maps and lists created from the composition specification is sent to the *ModuleManager*. Based on the list of instantiations, each module which needs to be instantiated is retrieved from the earlier defined list of event modules after which the *deepCopy* method is called, creating a new instance of this event module. In each event module instance the name is set as specified.
After all specified event modules are instantiated, the bindings are evaluated to search for all unbound modules. This is done by examining all bindings in search for event module instances which are not set as the target in any dynamic binding. The unbound modules are stored in a list.

Using the static bindings, each instance is assigned the specified valued.

The process at runtime, which can been seen in Figure 7.2, allows the base program to instantiate and publish the events as defined in section 5.2. The *RuntimeManager* receives the event and calls the *ModuleManager* to handle the published event. In the *ModuleManager*, all unbound modules are retrieved. The input of each event module is analyzed in search for the event type input. When this value is selected, it is set as the published event and the functionality is executed. After finalizing the functionality, the output is retrieved from the event module and from the list of *Binding* objects all instances are selected which have the instance which was just executed as its source. Using the dynamic binding specification from the *Binding* object, the output is bound to the input of the bound modules, after which they are executed. This process continues until no more bound modules exist, after which the flow is returned to the base program.

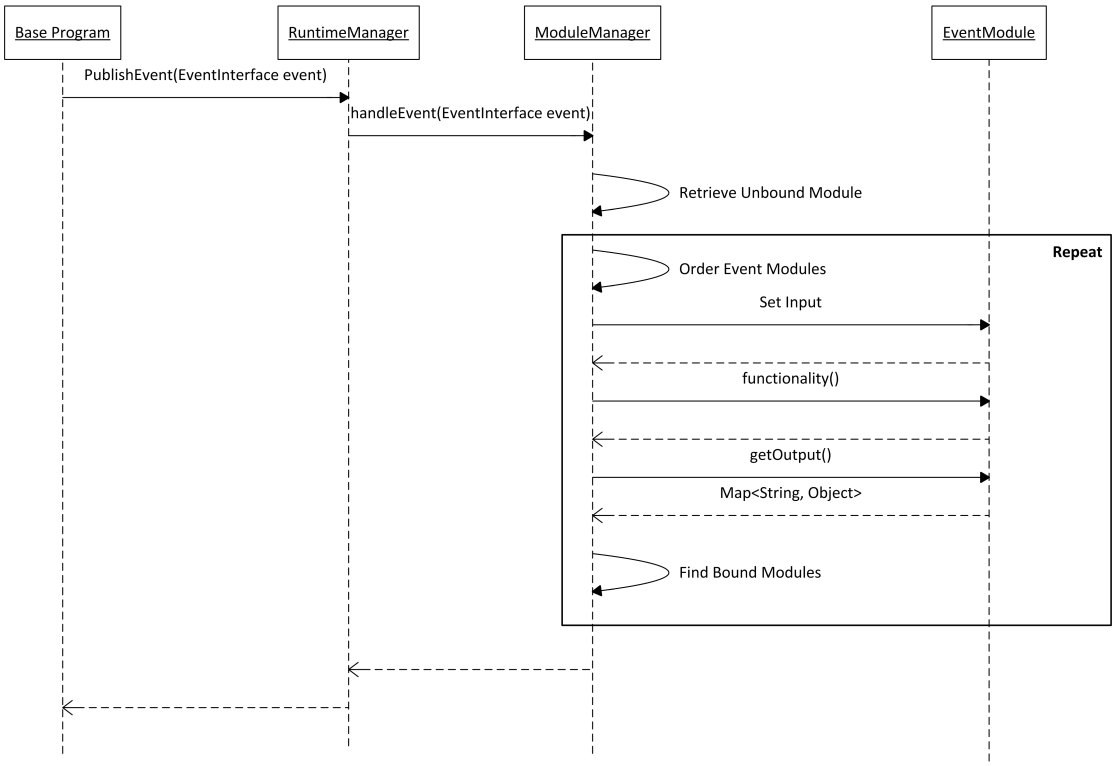

Figure 7.2: Event Publishing and Event Module Execution

7.3 Future Work

The binding language is fairly straightforward and should be able to perform all the required actions. In the future, however, it may be logical to move the mechanism, which interprets the composition language, instantiates the modules and stores all the values, to the compile time phase. This would mean a larger amount of data needing to be stored on the file system, but would reduce the need to perform all these actions every time the program is executed, which would in turn improve the performance.

7.4 Summary

In this chapter, a language and mechanism was created to specify the composition between modules. The modules which have been created, using the language defined in the previous chapter, can be instantiated and bound to-

gether using the input and the output of those modules. The binding happens at instance level to ensure full control over the composition. This supports a free homogeneous and heterogeneous composition between the modules. The explicit binding of the input and output increases the reusability of the modules. Finally, ordering is supported to have direct control over the order of execution and ensuring that the desired actions are performed at the point in the flow of the program.

CHAPTER 8

Evaluation of EventReactor2.0

In this thesis, it is claimed that the linguistic constructs offer by EventReactor2.0 creates an increased modularity and composability when creating EPAs. Furthermore, language extensibility is added, which supports the creation of EPAs using languages best suited for the problems they try to solve. In this chapter the added value of EventReactor2.0 is illustrated by implementing a set of evolution scenarios, and by discussing the details of the language extensibility.
In section 8.1, the evolution scenarios, described in chapter 3, are implemented and evaluated. In section 8.2, the language extensibility of EventReactor2.0 is discussed. In section 8.3 the results are summed up and a conclusion is drawn about whether or not the goals are reached.

8.1 Evolution of Modularity and Composability

In section 3.2, multiple evolution scenarios are created for the case study. These evolution scenarios are implemented for AspectJ and discussed for other paradigms to evaluate the modularity and composability of existing languages and frameworks. To illustrate that EventReactor2.0 has an improved modularity and composability, these evolution scenarios are also implemented for the EventReactor2.0 framework and the same metrics are examined.

8.1.1 Base Scenario

In EventReactor2.0, the implementation of the base scenario is fairly similar to design of the event processing network shown in Figure 2.1. First, an event module is defined, shown in Listing 8.1, which must filter for the *store* method call by a temperature sensors. For these scenarios, instances of the *MethodBased* event type are used, called *MethodCallEvent*. The event is published when a

TemperatureSensor class calls the *store* method.

```
eventmodule temperatureEventFilter{
    input{
        EventType Et;
    }

    functionality language=prolog{
        EventType t = {Et| hasDynamicAttributeWithNameAndType(Ev, '
    publisher', 'TemperatureSensor'), hasDynamicAttributeWithValue(
    Ev, 'methodname', 'store')};
    }

    output{
        EventType t;
    }
}
```

Listing 8.1: Base Scenario Event Processing Agent

A second event module is created, shown in Listing 8.2, which performs the publishing to the weather station.

```
eventmodule outsideTemperatureReaction{
    input{
        EventType Et;
    }

    functionality language=java{
        `
        //perform action
         `;
    }

    output{
    }
}
```

Listing 8.2: Base Scenario Event Consumer

In the composition script, shown in Listing 8.3, one instance of each event module is created after which they are bound together.

```
modules{
    tef1:temperatureEventFilter;
    tr2:outsideTemperatureReaction;
}

bindings {
    bind(tef1, tr2){
        t->Et;
    }
}

ordering{ }
```

Listing 8.3: Base Scenario Composition Script

8.1.2 Evolution Scenario 1:

In the first scenario, an extra reaction has to be added which needs to be performed for all sensors located indoors, while the reaction of the base scenario only needs to be performed for the outdoor sensors.
To do this, an extra filter is created which will filter between inside and outside sensors. Using the parameterization of event modules, the filter shown in Listing 8.4 can be used to filter for both situations.

```
eventmodule temperatureIdFilter{
    input{
        EventType Et;
        int min;
        int max;
    }

    functionality language=java{
        `
            MethodCallEvent event = (MethodCallEvent) Et;
            int id = ((TemperatureSensor)Et.getPublisher()).getId()
    ;
            if(id< min || id>max){
                Et=null;
            }
        `;
    }

    output{
        EventType Et;
    }

}
```

Listing 8.4: Event Module Filtering on Sensor ID's

Using the composition script, two instances are created of this event module and assigned a minimum and maximum value as can been seen in Listing 8.5. In

this case, *tif1* will filter for all IDs between one and five, i.e. the inside sensors, while *tif2* filters between six and ten, i.e. the outside sensors.
A new event module to implement the second reaction, called *insideReaction*, is created similar to the reaction defined in base scenario. Secondly the previously defined reaction has been renamed *outsideReaction* to improve the understandability for this scenario. The latter action is not required and therefore not taken into the metrics.

Listing 8.5 shows the complete composition script. Instead of the event filter being bound to the reaction directly, it is now bound to both the indoor and outdoor filter module. Both of these modules are in their turn bound to their respective reaction.

```
modules{
    tef1:temperatureEventFilter;

    tif1:temperatureIdFilter;
    tif2:temperatureIdFilter;

    tr1:insideTemperatureReaction;
    tr2:outsideTemperatureReaction;

}

bindings {
    bind(tif1){
        1->min;
        5->max;
    }

    bind(tif2){
        6->min;
        10->max;
    }

    bind(tef1, tif1){
        t->Et;
    }

    bind(tef1, tif2){
        t->Et;
    }

    bind(tif1, tr1){
        Et->Et;
    }

    bind(tif2, tr2){
        Et->Et;
    }
}
```

Listing 8.5: Composition Script Using Indoor and Outdoor Filters

In the following table, the measured values are shown combined with the values of the AspectJ implementation.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
EventReactor2.0	2	0	0	2	0	0
AspectJ	3	0	2	0	0	0

Because EventReactor2.0 allows named modules, which can be instantiated multiple times, and parametrized interfaces, the event module used to filter between the sensors can be used for both situations. AspectJ lacks this and needs two modules to define both filters. If parts of an AspectJ module needs to be reused, which is in this case the pointcut, the module needed to be redefined. Because the binding between a pointcut and an advice is tangled within an advice, binding an advice to a different pointcut requires the advice to be changed. In EventReactor2.0 the composition script can change the binding at interface level without modifying a module. The same tangling exists for AspectJ when composing pointcuts. In AspectJ, extending one pointcut with, in this case, ID filtering requires the second pointcut to reference the first one in its definition. In EventReactor2.0, the binding script can be used to perform this same actions without the tangling.

8.1.3 Evolution Scenario 2:

For the second evolution scenario, average temperatures need to be calculated and published for specific areas, and temperatures need to be monitored over a multiple of areas.
To implement this evolution using the EventReactor2.0 framework, the event module used to filter sensors in an area can actually be implemented by using the filter created in the previous evolution scenario. This way, only instantiation is required to define areas as can be seen in Listing 8.6.

```
modules{
    area1:temperatureIdFilter;
    area2:temperatureIdFilter;
    area3:temperatureIdFilter;
}

bindings{
    bind(area1){
        1->min;
        2->max;
    }

    bind(area2){
        3->min;
        4->max;
    }

    bind(area1){
        5->min;
        5->max;
    }
}
```

Listing 8.6: Composition Script for Area Filters

Four event modules have been defined. *aggregateReaction* calculates and maintains the aggregate temperature, *aggregatePublishing* is publishes the aggregate value to the interested party, *monitorFilter* monitors the values of the radiators and is successful when the maximum temperature has been reached multiple times, and *RadiatorReaction* contacts maintenance.

With the areas the two different functionalities are created by composing the correct event modules together. In Listing 8.7 each area is composed with their specific instance of the *aggregateReaction* module. Each of these module is then bound to the same instance of the *aggregatePublishing* module.

```
modules{
    aggr1:aggregateReaction;
    aggr2:aggregateReaction;
    aggr3:aggregateReaction;

    aggrPubl:aggregatePublishing;
}

bindings{
    bind(area1, aggr1){
        Et->et;
    }

    bind(area2, aggr2){
        Et->et;
    }

    bind(area3, aggr3){
        Et->et;
    }

    bind(aggr1, aggrPubl){
        value->value;
    }

    bind(area2, aggrPubl){
        value->value;
    }

    bind(area3, aggrPubl){
        value->value;
    }

}
```

Listing 8.7: Composition Script for Aggregate Calculation and Publishing

In Listing 8.8, the composition is shown to implement the radiator monitoring. Two instance of the *monitorFilter* module are created, one responsible for area 1 and 2, the second responsible for area 2 and 3. Both instance are bound to the same instance of the *RadiatorReaction* event module.

```
modules{
    radMon1:monitorFilter;
    radMon2:monitorFilter;
    radReact:RadiatorReaction;
}

bindings{

    bind(area1, radMon1){
        Et->er;
    }
    bind(area2, radMon1){
        Et->er;
    }

    bind(area2, radMon2){
        Et->er;
    }

    bind(area3, radMon2){
        Et->er;
    }

    bind(radMon1, radReact){
        Et->Et;
    }

    bind(radMon2, radReact){
        Et->Et;
    }
}
```

Listing 8.8: Composition Script for Temperature Monitoring and Publishing

The following table shows the metrics of this implementation together with the previously generated examples.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ per area	6	0	0	0	2	12
AspectJ monitor as pointcut	9	0	0	0	2	3
EventReactor2.0	4	0	0	0	0	0

The AspectJ implementation is possible in two different ways. By handling all the actions per area the amount of modules is reduced but it greatly increases the strongly coupled concerns since multiple actions get performed in each advice module. Because modularity is one of the main focuses of this analysis separating the advices over more modules is the preferred solution.

Comparing that solution to the EventReactor2.0 implementation, the advantage of the instantiatable modules with programmable interfaces is shown, allowing the EventReactor2.0 implementation to use a lot less event modules. AspectJ also suffers from a tangling between two pointcuts where the radiator is monitored. The area pointcuts needs to be referenced in the pointcut which monitors

the radiator, where EventReactor2.0 uses the binding script to bind them at interface level.

Instantiation and binding at interface level is also used in EventReactor2.0 to allow different filter modules and reaction modules to be bound to the same instance of another module without that being defined within that module. AspectJ requires the two external classes to maintain an instance which can be referred to within other modules.

Because EventReactor2.0 allows the event reaction code to be defined modularly, the amount of strongly coupled concerns is reduced to zero, while AspectJ still has those concerns within the reaction.

8.1.4 Evolution Scenario 3:

In the third evolution scenario, an external sensor, with filter and reaction is added. The outside reaction, which was created in the first evolution scenario, must now be replaced with the external reaction. For this scenario, it assumed three event modules are inserted from the external project. First, a filter selecting all storage calls from the external temperature sensor, which is shown in Listing 8.9.

```
1  eventmodule externalFilter{
2      input{
3          EventType Et;
4      }
5
6      functionality language=prolog{
7          EventType t;
8          t = {Et| hasDynamicAttributeWithNameAndType(Ev, 'publisher
       ', 'TemperatureSensor2'), hasDynamicAttributeWithValue(Ev, '
       methodname', 'store')};
9      }
10
11     output{
12         EventType t;
13     }
14 }
```

Listing 8.9: External Event Module for Filtering

Secondly, a module used to retrieve the temperature and output it, which is shown in Listing 8.10.

```
eventmodule externalValueRetriever{
    input{
        EventType Et;
    }

    functionality language=java{
        `
        MethodCallEvent event = (MethodCallEvent) Et;
        TemperatureSensor2 sensor = (TemperatureSensor2)event.
    getPublisher();
        value = ""+sensor.getValue();
         `;
    }

    output{
        String value;
    }
}
```

Listing 8.10: External Event Module for Retrieving Temperature

And finally, Listing 8.11 shows the event module responsible for publishing to the external weather station.

```
eventmodule externalReaction{
    input{
        String value;
    }

    functionality language=java{
        `
        double aDouble = Double.parseDouble(value);
        //Perform external action
        System.out.println("External value published: "+ aDouble);
         `;
    }

    output{

    }
}
```

Listing 8.11: External Event Module for Publishing

In the bindings, all modules have been instantiated and bound to each other according to the specification.

To implement the evolution, a new event module is created similar to the *externalValueRetriever* event module which takes the value measured by the sensor and produces it as output. This way the current filter can be indirectly bound to the external reaction without encountering problems with matching input and output types.
In the composition script, shown in Listing 8.12, the added module is bound

to the external reaction, and the binding from the outside filter to the outside reaction needs to be removed, together with the instantiation of this reaction to avoid making it unbound.

```
modules{
    extReact:externalReaction;
    extFilter:externalFilter;
    valueRetr:valueRetriever;
    extVal:externalValueRetriever;
}

bindings{
    bind(extFilter,extVal){
        t->Et;
    }

    bind(extVal, extReact){
        value->value;
    }

    bind(tif2,valueRetr){
        Et->Et;
    }

    bind(valueRetr, extReact){
        value->value;
    }
}
```

Listing 8.12: Composition Script for Aggregate Calculation and Publishing

The current filter can also be changed by producing an extra output, allowing the filter to be directly bound to the external reaction. In the following table, both options are compared with the AspectJ results from chapter 3.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
EventReactor2.0 Added Temperature	1	1	0	1	0	0
EventReactor2.0 Added Event Module	0	1	2	2	0	0
AspectJ add temperature	0	1	2	0	0	0
AspectJ copied advice	0	1	1	0	0	0
AspectJ sensor superclass	0	1	2	0	1	0
AspectJ joinpoint	0	1	3	0	0	0

In AspectJ, the first two solution seems the most logical ones. Using the joinpoints as input and output is the worst solution in AspectJ, because all involved modules need to be redefined to allow this to happen. To differentiate between the first two scenarios, it is required to look beyond the metrics. Creating a copy of the advice requires only one module to be created/modified. The problem is that the code is identical to the original and thus creates a redundancy. Adding a temperature to the output of the filter requires changes to 2 modules, but allows the external reaction to be reused. Because modularity is of importance, reusing the reaction is deemed more important and therefore selected as the best solution. Possible future scenarios which require change in the external reaction will only require one module to be changed instead of two if the other was selected.

In EventReactor2.0, the best solution is fairly clear. Using the event as input and output is the better idea when starting from scratch, but in this scenario, where the external modules have already been defined, changing only the outside filter is the better solution.

Comparing both best solutions with each other there is no clear better solution. The main advantage EventReactor2.0 offers, is the binding at interface level. This leaves the module untouched and improves the modularity, while AspectJ tangles the binding between the pointcut and advice which requires a change in the advice to bind the changed pointcut to it.

8.1.5 Evolution Scenario 4:

In the fourth scenario, the IDs, used to differentiate between the inside and outside filters, must be switched.
Implementing this in EventReactor2.0 is possible in two ways. The first one, shown in Listing 8.13, changes the values which are statically bound to the instances of the filters. The second way is changing the binding between the filters and the reactions. Though they result in the same amount of changes, the latter is a better representation in showing a change in the functionality, since only changing the values of variable is less likely to happen than overall changes in the flow of the functionality.

```
1 bind(tif1){
2     6->min;
3     10->max;
4 }
5
6 bind(tif2){
7     1->min;
8     5->max;
9 }
```

Listing 8.13: Altered Static Binding

The following table shows the metric of both the EventReactor2.0 and AspectJ implementations.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
EventReactor2.0	0	0	0	2	0	0
AspectJ change pointcut	0	0	2	0	0	0
AspectJ change advice	0	0	4	0	0	0

The choice between the AspectJ alternatives is fairly easy and changing the implementation of the pointcuts has therefore been chose as the best solution.

Allowing binding at interface level and allowing static values to be bound al-

lows EventReactor2.0 to only require two changes to the binding, while AspectJ requires changes to be applied directly inside the module, which is less desirable.

8.1.6 Evolution Scenario 5:

For the fifth scenario, the second scenario is used and changed to remove part of the functionality for only one of the areas. Implementing this in EventReactor2.0 first requires the temperature to be retrieved from the event so it can be used as input for the *aggregatePublishing* event module. This is already discussed in the previous scenarios where it was decided the better option is to create a new module which retrieves this value.
In the composition script, shown in Listing 8.14, an instance of the event module responsible for retrieving the value from the event is created and replaces the event module which calculates the aggregate. In this case the latter instance needs to be removed to ensure it not remain unbound.

```
1  modules{
2      ...
3      valueRetr:valueRetriever;
4
5  }
6
7  bindings{
8      ...
9      bind(area3, valueRetr){
10         Et->Et;
11     }
12
13     bind(valueRetr, aggrPubl){
14         value->aggr;
15     }
16     ...
17 }
```

Listing 8.14: Composition Script

The following table shows the resulting metrics and the AspectJ metrics.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
EventReactor2.0	1	0	0	2	0	0
AspectJ	0	0	2	0	0	0

Because the initial strong coupling between the different reactions in AspectJ, the amount of redefinition, required in the modules, is a bit higher than in EventReactor2.0. It also becomes more complex to separate the modules because of the strong coupling.
In EventReactor2.0, the main action is changing the binding script to replace the aggregate calculation with the temperature retrieval module for the specific area. Adding a new module, which retrieves this temperature value, will not

influence the other modules. Because event modules in EventReactor2.0 are modularized the composition script can be used to remove parts of the functionality without it influencing any other event module.

8.1.7 Evolution Scenario 6:

In the sixth scenario, both the input and output are changed by adding and removing variables. In EventReactor2.0, not all output has to be bound to the input and vice versa. For the module with the added input, the original filter can be used only ignoring the binding for that parameter. This allows modules to be bound without changing or creating any modules. When removing an input the binding must also be changed to remove references to the input. The resulting composition is shown in Listing 8.15

```
1  modules{
2      ...
3      ext1:externalReaction1;
4      ext2:externalReaction2;
5  }
6
7  bindings{
8      bind(tif1 , ext1){
9      }
10
11     bind(tif1 , ext2){
12         Et->Et;
13     }
14     ...
15 }
```

Listing 8.15: Composition Script

The same situation arises when output of a module is added or removed. EventReactor2.0 will allow any number of input to be bound to any number of output, although how the actual module handles certain values not being bound, and not being assigned a value, depends on what the user has defined as the actions in the module.

In the following table, all situations are examined and compared to the AspectJ implementation.

	Added Modules	Removed Modules	Redefined Modules	Altered Bindings	Non-modular Funct. Added	#Strongly couple concerns
AspectJ Adding Input	0	0	2	0	0	0
EventReactor2.0 Adding Input	0	0	0	0	0	0
AspectJ Removing Input	0	0	1	0	0	0
EventReactor2.0 Removing Input	0	0	0	1	0	0
AspectJ Adding output	0	0	1	0	0	0
EventReactor2.0 Adding Output	0	0	0	0	0	0
AspectJ Removing output	0	0	1	0	0	0
EventReactor2.0 Removing Output	0	0	0	1	0	0

In AspectJ, the advice always needs to be redefined no matter what evolution

happens. It either needs to make sure the value is no longer used in the advice, or needs to remove the reference in the binding. The worst case scenario is when adding input to the advice. In this case, the values need to be added to both the binding in the advice as the pointcut.
The EventReactor2.0 binding script allows both input and output to be ignored. Only when removing input or output which is already bound, the script needs to be changed to remove these bindings.
Especially when a lot of advices use the same pointcut in AspectJ, one advice requiring a new value to be added to the input can ripple through to all the other reactions using the same pointcut.

8.2 Evaluation of Language Extensibility

As defined in chapter 3.2, it is difficult to measure language extensibility and the most viable method is simply examining which languages can be used to implement the various concerns of an EPA, and whether or not new DSLs or GPLs can be added to implement these concerns.

In EventReactor2.0, there are three concerns which are defined using a language. For the event and event type language a fixed language is used in which data structures of the events and event types can be defined. The language that was used allows for extensibility, in that it allows users to define new events and event types. Language extensibility is not supported in the current implementation, but can be added in a later version.

For the event modules, in which event processing agents and event reactor can be implemented, new DSLs and GPLs can be added. This allows each functionality to be implemented using a language best suited for the situation. In the event module only the *functionality* section is implemented using an extensible language, while the rest of the concern is defined using a fixed language. This means the event module is able to support both event processing agents and event reactors.

The final concerns, being the composition script, does not support extensible languages, since it must only provide basic functionalities which include a mapping between modules. The composition script also aims at improving the extensibility by allowing instantiation, instance level binding and ordering.

When comparing these results to the existing languages and frameworks, especially the extensible languages which can be used in the event modules provide a great improvement. The frameworks which support this type of language extensibility do exist but fall short when comparing the extensibility of events and event types and extensibility of composition.

8.3 Summary

When examining the results of the evolution scenarios, small improvements can already be measured even thought the scale of the examples is fairly small. When increasing the size and complexity of the EPAs, the gained advantages will only increase with it, since it is not limited to the size of the program. The language extensibility which is offered is also an improvement over the existing frameworks in that new DSLs and GPLs can be added to implement event processing agents and event consumers, while also offering extensibility for events and composition. Both these results prove that the modularity, composability and language extensibility has increased with the use of the EventReactor2.0 framework, and should be researched even further to examine the practicality in a real world environment.

CHAPTER 9

Conclusion

There are currently numerous amounts of application which can utilize event processing of some sort. Because these can become very complex, extensible linguistic mechanisms are required to modularize and compose the concerns.
There are quite a few programming languages that can be used to implement event processing. In this thesis, these languages are evaluated using a set of criteria; i.e. language extensibility, modularity and composability.
The languages extensibility criteria examines the possibility of adding new domain specific or general purpose languages and using them to define the event processing concerns.
The modularity criteria is used to examine to what extend the concerns are defined in separate modules. This requires a uniquely named entity without knowledge of other modules, and a well defined input, output and an encapsulated functionality.
The composability criteria is used to examine what types of concerns can be composed with each other. In more detail, it checks at what level composition is possible and to what extend it can be configured. Using these criteria, Object Oriented, Aspect Oriented and Event Processing languages are evaluated and their shortcomings discussed. This is done by using a set of evolution scenarios and applying it to AspectJ and discussing the solutions for Java and Esper. For the language extensibility, the languages and frameworks were examined to see to what extend new languages can be added.

In the EventReactor framework, *events* and *event types* can be defined. Secondly it offers *event modules*. In an event module a *selector*, defined in Prolog, which is used to select event of interest are composed with *reactorchains*. In a reactorchain multiple *reactors*, which define the reactions, are composed.

Like other event processing applications, this thesis identified that EventReactor suffers from tight coupling between concerns. The composition between the concerns is defined within one of the concerns and specification is limited. The languages used to define the concerns are fixed and limited to one per concern.

To solve this problem EventReactor2.0 is introduced, which allows an extensible list of DSLs and GPLs to be used to implement some concerns. It also provides the event module to implement a modularized event consumer and event processing agent. Finally, EventReactor2.0 support explicit instantiation of event modules and binding can happen both explicit and implicit.

Using the evolution scenarios it is proven that, by defining each concern in its own named module, with a well defined input, output and functionality section with out any knowledge of other modules, the modularity criteria is fulfilled. By allowing modules to be composed on instance level, binding them at interfaces-level both explicitly and implicitly, and supporting configurable ordering, the composability criteria is fulfilled.
The functionality language within the event modules can be defined using an extensible list of DSLs and GPLs. This helps fulfill the language extensibility criteria.

The results show that the designed framework is already an improvement over existing languages and frameworks. The next step would be to research the usability in larger and more complex projects. Secondly, there are still some improvements which, when implemented, should increase the usability of the framework and improve the selected criteria even more. These improvements should be implemented in the next version of the framework.

Bibliography

[1] S. Malakuti, C. Bockisch, and M. Aksit, "Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software," in *2009 20th International Symposium on Software Reliability Engineering*. IEEE, Nov. 2009, pp. 31–40. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5362079

[2] M. Salehie and L. Tahvildari, "Self-adaptive software," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 1–42, May 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1516533.1516538

[3] S. Malakuti, "Achieving naturalness in runtime enforcement," Enschede, The Netherlands, Sep. 2011. [Online]. Available: http://doc.utwente.nl/78019/ http://purl.org/utwente/doi/10.3990/1.9789036532464

[4] ——, "Complex Event Processing with Event Modules," *Reactivity, Events and Modularity, co-located with SPLASH 2013, USA*, 2013. [Online]. Available: http://soft.vub.ac.be/REM13/papers/rem20130_submission_4.pdf

[5] S. Malakuti and M. Aksit, "Event-based Modularization of Reactive Systems," *2Concurrent Objects and Beyond, LNCS, 2013(to appear)*.

[6] S. Malakuti, "Event Modules: Modularizing Domain-Specific Crosscutting R.V. Concerns," *ACM Transactions on Aspect-Oriented Software Development, Special Issue on Runtime Verification and Analysis, 2013(to appear)*.

[7] O. Etzion and P. Niblett, *Event Processing in Action*. Stamford, CT, USA: Manning Publications Company, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894960

[8] "Esper - Complex Event Processing." [Online]. Available: http://esper.codehaus.org/

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison Wesley Publishing Company, 1995.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," no. June, pp. 220–242, 1997. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0053381

[11] H. Masuhara, Y. Endoh, and A. Yonezawa, "A Fine-Grained Join Point Model for More Reusable Aspects," in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2006, no. Aplas, ch. 8, pp. 131–147.

[12] C. Allan, J. Tibble, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam, "Adding trace matching with free variables to AspectJ," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*. New York, New York, USA: ACM Press, 2005, p. 345. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1094811.1094839

[13] "The AspectJ Project." [Online]. Available: http://eclipse.org/aspectj/

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with ASPECTJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, Oct. 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=383845.383858

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," *ECOOP 2001 Object-Oriented Programming*, vol. 2072, pp. 327–354, Jun. 2001. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-45337-7_18 http://link.springer.com/10.1007/3-540-45337-7

[16] "Join Points and Pointcuts." [Online]. Available: http://www.eclipse.org/aspectj/doc/next/progguide/language-joinPoints.html

[17] W. L. Hürsch and C. V. Lopes, "Separation of Concerns," in *Technical report NU-CCS-95-03*. Boston, USA: Northeastern University, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5223

[18] D. Parnas, P. Clements, and D. Weiss, "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, pp. 259–266, Mar. 1985. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702002

[19] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison Wesley, 2003.

[20] S. Kojarski and D. Lorenz, "Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions,"

ACM SIGPLAN Notices, pp. 515–534, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1297065

[21] M. Shonle, K. Lieberherr, and A. Shah, "XAspects: An Extensible System for Domain-Specific Aspect Languages," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03*. New York, New York, USA: ACM Press, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=949349 http://portal.acm.org/citation.cfm?doid=949344.949349

[22] E. Tanter and J. Noy, "A Versatile Kernal for Multi-Language AOP," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, R. Glück and M. Lowry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3676, ch. 13, pp. 173–188. [Online]. Available: http://link.springer.com/10.1007/11561347

[23] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, Jan. 2001. [Online]. Available: http://doi.wiley.com/10.1002/smr.220

Appendices

APPENDIX A

Event and Event Type Language

A.1 Xtext

```
grammar nl.utwente.er2.event.EventLanguage with org.eclipse.xtext.common.Terminals

generate eventLanguage "http://www.utwente.nl/er2/event/EventLanguage"

EventModel:
    ((eventTypes += EventTypeDecl)|(events+=Event))*
;

EventTypeDecl:
    ImportEvent|EventType
;

ImportEvent:
    'import event' name=ID
;

EventType:
    'eventtype' name=ID 'extends' superEvent = [EventTypeDecl] '{'
        (staticcontext += StaticContext  | dynamicContext += DynamicContext  )*
    '}'
;

StaticContext:
    'static' ':' (attribute += Attribute ';')*
;

DynamicContext:
    'dynamic' ':' (attribute += Attribute ';')*
;

Attribute:
    name = ID ':' type = Type
;
```

```
34
35 Type:
36     type = ID
37 ;
38
39 Event:
40     'event' name=ID 'instanceof' type = [EventTypeDecl] '{'
41         (attributes += AttributeInit ';')*
42     '}'
43
44 ;
45
46 AttributeInit:
47     name = ID ':' value = Value
48 ;
49
50 Value:
51     value = STRING |INT
52 ;
```

A.2 Xtend

```
1  class EventLanguageGenerator implements IGenerator {
2
3      override void doGenerate(Resource resource, IFileSystemAccess
       fsa) {
4          for(e: resource.allContents.toIterable.filter(typeof(
       EventType))) {
5              fsa.generateFile(
6                e.name.toString() + ".java",
7                e.compile)
8          }
9          for(e: resource.allContents.toIterable.filter(typeof(Event)
       )) {
10             fsa.generateFile(
11               e.name.toString() + ".java",
12               e.compile)
13         }
14
15
16     }
17
18     def compile(EventType it) '''
19         import nl.utwente.er2.util.EventTypeInterface;
20         import nl.utwente.er2.util.baseevent.*;
21         import java.util.HashMap;
22         import java.util.Map;
23
24         public class name IF superEvent != nullextends
       superEvent.name ENDIF implements EventTypeInterface{
25
26
```

```
27          @Override
28          public String getEventTypeName(){
29              return "name";
30          }
31
32
33          FOR stat:staticcontext
34          stat.compile
35          ENDFOR
36          FOR dyn:dynamicContext
37          dyn.compile
38          ENDFOR
39          }
40      '''
41
42      def compile(StaticContext it) '''
43
44      '''
45
46      def compile(DynamicContext it) '''
47          FOR attr: attribute
48
49              public attr.type.type getattr.name(){
50                  return (attr.type.type)dynamicAttributes.get("
        attr.name");
51              }
52
53              public void setattr.name(attr.type.type attr.name
        ){
54                  dynamicAttributes.put("attr.name", attr.name);
55              }
56          ENDFOR
57      '''
58
59      def compile(Event it) '''
60
61          import nl.utwente.er2.util.EventInterface;
62          import nl.utwente.er2.util.baseevent.*;
63          import java.util.HashMap;
64          import java.util.Map;
65
66          public class name extends type.name implements
        EventInterface{
67
68              private static final Map<String, String> attributes =
        createMap();
69
70              public static Map<String, String> createMap(){
71                  Map<String, String> attributes = new HashMap<String
        , String>();
72                  FOR attr: attributes
73                  attributes.put("attr.name", "attr.value.value")
        ;
74                  ENDFOR
75                  return attributes;
76              }
77
```

```
78
79
80              @Override
81              public Map<String, Object> getDynamicAttributes(){
82                  return dynamicAttributes;
83              }
84
85              @Override
86              public Map<String, String> getStaticAttributes(){
87                  return attributes;
88              }
89
90
91              @Override
92              public String getEventName(){
93                  return "name";
94              }
95
96          }
97      ,,,
98  }
```

APPENDIX B

Event Module Language

B.1 Xtext

```
grammar nl.utwente.er2.basemodule.BaseModule with org.eclipse.xtext
    .common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate baseModule "http://www.utwente.nl/er2/basemodule/
    BaseModule"

Model:
    (imports +=Import)*
    'eventmodule' name=ID '{' input=Input locVars=LocalVars?
    functionality=Functionality output=Output '}'
;

terminal STR: '\'' ('a'..'z'|'A'..'Z'|'0'..'9')+ '\'';
terminal CAPITAL : ('A'..'Z');
terminal ID          : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z
    '|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING :
            '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') |
    !('\\'|'"') )* '"' |
            "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') |
    !('\\'|"'") )* "'";
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT     : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS          : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
terminal OPERATOR: ('+'|'-'|'*'|'/'|'=='|'>'|'<'|'!=');

Import:
    'import' packageName=STRING
;

Input:
```

```
31      'input' '{'
32          (vars += VariableInstantiation ';')*
33
34      '}'
35  ;
36
37  LocalVars:
38      'local' '{'
39          (vars += VariableInstantiation ';')*
40      '}'
41
42  ;
43
44  VariableInstantiation:
45      type=Type name=ID
46  ;
47
48  Boolean:
49      'true'|'false'
50  ;
51
52  Type:
53      ('String'|'int'|'double'|'Boolean'|'EventType')
54  ;
55
56  Output:
57      'output' '{'
58          ((type=Type refs+=VarReference) ';')*
59      '}'
60  ;
61
62
63  VarReference:
64      name=[VariableInstantiation]
65  ;
66
67  Functionality:
68      'functionality' 'language' '=' language=Language '{'
69          (stats += Statement ';')*
70      '}'
71  ;
72
73  Language:
74      name=ID
75  ;
76
77  Statement:
78      LanguageStatement
79  ;
80
81  LanguageStatement:
82      //Empty statement to be implemented in language specific
        compilers
83      value='<<>>'
84  ;
```

B.2 Xtend

```
abstract class AbstractBaseModuleGenerator implements IGenerator {

    public var filename = "";

    override void doGenerate(Resource resource, IFileSystemAccess
    fsa) {

    }

    def CharSequence getString(Model model){
        filename = model.name;
        return (getImports + model.compileModel);
    }

    def compileModel(Model it) '''
    import java.util.List;
    import java.util.ArrayList;
    import nl.utwente.er2.module.model.Module;
    import nl.utwente.er2.repository.RepositoryInterface;
    import nl.utwente.er2.util.EventInterface;
    import nl.utwente.er2.util.baseevent.EventType;
    import java.util.HashMap;
    import java.util.Map;

    FOR imp: imports
            import imp.packageName;
    ENDFOR



    public class name extends Module {

        private EventInterface temp = null;
        IF locVars != null
            FOR varDecl: locVars.vars
                private varDecl.type varDecl.name;
            ENDFOR
        ENDIF

        IF input != null
        FOR varDecl: input.vars
            private varDecl.type varDecl.name;
        ENDFOR

        ENDIF
        FOR stat: functionality.stats
            IF stat instanceof VariableInstantiation
                var varDecl = stat as VariableInstantiation
                private varDecl.type varDecl.name;
            ENDIF
        ENDFOR


        public name(){
```

```
54              self = this;
55              name = "name";
56
57          IF input != null
58
59          FOR varDecl: input.vars
60              input.put("varDecl.name", varDecl.type.class);
61          ENDFOR
62
63          ENDIF
64
65              functionality.compileConstructorFunct
66          }
67          input.compile
68          functionality.compileFunctionality
69          output.compileOutput
70
71          public void set(String name, Object value){
72              input.compileSetter()
73          }
74      }
75      '''
76
77      def String compileSetter(Input it){
78          var returnValue = "";
79          if(it != null){
80
81          for(VariableInstantiation decl:vars){
82              returnValue = returnValue + "if(name.equals(\""+ decl.
    name + "\")){
83                  this."+decl.name+" = ("+decl.type+") value;
84              }";
85          }
86          }
87          return returnValue;
88      }
89
90      def String getImports()
91
92      def compile(Input it)'''
93
94      '''
95
96      def compileOutput(Output it) '''
97          @Override
98          public Map<String, Object> getOutput(){
99
100             Map<String, Object> returnList = new HashMap<String,
    Object>();
101
102         IF it != null
103
104             FOR ref:refs
105                 ref.compileVarReference
106             ENDFOR
107
108         ENDIF
```

```
109             return returnList;
110         }
111     '''
112
113     def compileVarReference(VarReference it)'''
114         returnList.put("name.name", name.name);
115     '''
116
117     def compileFunctionality(Functionality it) '''
118         public void functionality(){
119             FOR stat:stats stat.compileStatement ENDFOR
120         }
121     '''
122
123     def compileStatement(Statement it){
124         var value = "";
125         if(it instanceof LanguageStatement){
126             var langStat = it as LanguageStatement;
127             value = value + compileLanguageStatement(langStat);
128         }
129         return value;
130
131     }
132
133     def String compileLanguageStatement(LanguageStatement it)
134
135     def compileVariableInstantiation(VariableInstantiation it) {
136         var returnValue = "";
137
138         return returnValue;
139     }
140
141     def String compileConstructorFunct(Functionality it)
142
143 }
```

APPENDIX C

Prolog Language

C.1 Xtext

```
grammar nl.utwente.er2.language.prolog.Prolog with nl.utwente.er2.
    basemodule.BaseModule

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate prolog "http://www.utwente.nl/er2/language/prolog/Prolog"

Mod: model=Model;

Language:
    name='prolog'
;

LanguageStatement:
    VariableInstantiation|AssignmentStatement | ConditionStatement
;

AssignmentStatement:
    variable=VarReference '=' vall=Expression
;

ConditionStatement:
    'if' '(' condition = Expression ')' '{'
        (ifStatements += Statement ';')+
    '}'('else' '{'(elseStatements += Statement ';')+ '}')?
;

Pointcut:
    '{'variable=VarReference '|' (rules += Rule ',')* rules +=
    Rule '}'
;

Rule:
(rule= 'isChildEvent' '(' vars+=Variable ',' vars+=Variable ')')|
(rule= 'isChildType' '('vars+=Variable','vars+=Variable') ')|
(rule= 'isEventWithName' '('vars+=Variable',' vars+=Variable') ')|
```

```
(rule= 'isEventTypeWithName' '('vars+=Variable',' vars+=Variable')')|
(rule= 'hasAttribute' '('vars+=Variable',' vars+=Variable')')|
(rule= 'hasDynamicAttributeWithValue' '('vars+=Variable',' vars+=Variable',' vars+=Variable')')|
(rule= 'hasStaticAttributeWithValue' '('vars+=Variable',' vars+=Variable',' vars+=Variable')')|
(rule= 'hasStaticAttribute' '('vars+=Variable',' vars+=Variable')')|
(rule= 'hasDynamicAttribute' '('vars+=Variable ',' vars+=Variable')')|
(rule= 'hasDynamicAttributeWithNameAndType' '('vars+=Variable',' vars+=Variable',' vars+=Variable')')
;


Variable:
    (ID|STR)
;

Expression:
    lhSide=Value (operator=OPERATOR rhside=Expression)?
;

Value:
    ((primitive=PrimitiveValue)|(complex=ComplexValue))
;

PrimitiveValue:
    STRING|Boolean|INT
;

ComplexValue:
    VarReference|Pointcut
;

VarReference:
    name=[VariableInstantiation]
;

VariableInstantiation:
    type=Type name=ID ('=' val1=Expression)?
;
```

C.2 Xtend

```
class PrologGenerator extends AbstractBaseModuleGenerator
    implements IGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess
    fsa) {
```

```
4         for(e: resource.allContents.toIterable.filter(typeof(Model)
  )) {
5             filename=e.name.toString();
6             fsa.generateFile(
7               e.name.toString() + ".java",
8               e.compileModel)
9         }
10    }
11
12    override getImports() {
13        return 'import nl.utwente.er2.repository.PrologRepository;'
14    }
15
16    override compileLanguageStatement(LanguageStatement it){
17        var value = "";
18        if(it instanceof VariableInstantiation){
19            var VariableInstantiation varInst = it as
  VariableInstantiation;
20            value = value+compileVariableInstantiation(varInst);
21        }else if(it instanceof AssignmentStatement){
22            var AssignmentStatement assStat = it as
  AssignmentStatement;
23            value = value + compileAssignmentStatement(assStat);
24        }
25        return value;
26    }
27
28    def compileAssignmentStatement(AssignmentStatement it){
29        var returnValue = "";
30        if(vall.lhSide.complex instanceof Pointcut){
31         var p= vall.lhSide.complex as ComplexValue
32         returnValue = returnValue + compilePointcut(p);
33        }
34        returnValue = returnValue + variable.name.name+ "="+
  compileExpression(vall)+";";
35        return returnValue;
36    }
37
38    override compileConstructorFunct(Functionality it) {
39        var value = "";
40        for(Statement stat:stats){
41            if(stat instanceof VariableInstantiation){
42                var varInst = stat as VariableInstantiation;
43                if(varInst.vall != null && varInst.vall.lhSide.
  complex instanceof nl.utwente.er2.basemodule.baseModule.
  Pointcut){
44                    var p = varInst.vall.lhSide.complex as Pointcut
45                    value = value + compileConstructorPointcut(p);
46                }
47            }else if(stat instanceof AssignmentStatement){
48                var assStat = stat as AssignmentStatement;
49                if(assStat.vall.lhSide.complex instanceof nl.
  utwente.er2.basemodule.baseModule.Pointcut){
50                    var p = assStat.vall.lhSide.complex as Pointcut
51                    value = value + compileConstructorPointcut(p);
52                }
53            }
```

```
54          }
55          return value;
56      }
57
58      def compileVariableInstantiation(VariableInstantiation it) {
59          var returnValue = "";
60          if(vall != null){
61              if(vall.lhSide.complex instanceof Pointcut){
62               var p= vall.lhSide.complex as ComplexValue
63               returnValue = returnValue + compilePointcut(p);
64              }
65              returnValue = returnValue + name+ "="+
    compileExpression(vall) +";";
66          }else{
67              returnValue = returnValue + name+ "="+ "new "+ type +"
    ()" +";";
68          }
69          return returnValue;
70      }
71
72      def compileConstructorPointcut(Pointcut it)'''
73          try{
74          PrologRepository.getInstance().tagEvents("filename", "
    variable.name.name", "compileConstructorRules");
75          }catch(nl.utwente.er2.repository.exception.QueryException e
    ){
76              e.printStackTrace();
77          }
78      '''
79
80      def String compileConstructorRules(Pointcut it){
81          var returnValue = ""
82          var i=0;
83          while(i<rules.size()){
84              returnValue = returnValue + rules.get(i).rule + "(";
85              var j =0;
86              while(j<rules.get(i).vars.size()){
87                  returnValue = returnValue +rules.get(i).vars.get(j)
    ;
88                  if(j!=rules.get(i).vars.size()-1){
89                      returnValue = returnValue +", ";
90                  }
91                  j = j+1
92              }
93
94              returnValue = returnValue + ")"
95              if(i!=rules.size()-1){
96                      returnValue = returnValue +", ";
97              }
98              i = i+1;
99          }
100
101         return returnValue;
102     }
103
104
105
```

```
106     def compilePointcut(ComplexValue it) '''
107         var pc = it as Pointcut
108         temp = PrologRepository.getInstance().checkEventByPointcut
        ((EventInterface) pc.variable.name.name, "filename");
109     '''
110
111     def compileExpression(Expression it) '''
112         IF lhSide.complex instanceof Pointcut(EventType)
        tempELSEIF lhSide.complex instanceof VarReference var vr =
        lhSide as VarReference vr.name.name ELSE lhSide.
        primitive ENDIF IF operator != null operator rhside.
        compileExpression ENDIF
113     '''
114
115 }
```

APPENDIX D

Java Language

D.1 Xtext

```
grammar nl.utwente.er2.language.java.Java with nl.utwente.er2.
    basemodule.BaseModule

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate java "http://www.utwente.nl/er2/language/java/Java"

Mode: model=Model;

Language:
    name='java'
;

LanguageStatement:
    text = TEXT_BLOCK
;

terminal TEXT_BLOCK :  '``' -> '``'
;
```

D.2 Xtend

```
class JavaGenerator extends AbstractBaseModuleGenerator implements
    IGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess
    fsa) {
        for(e: resource.allContents.toIterable.filter(typeof(Model)
    )) {
            filename=e.name.toString();
            fsa.generateFile(
              e.name.toString() + ".java",
```

```
8                 e.compileModel)
9             }
10        }
11
12        override getImports() {
13            return '';
14        }
15
16        override compileConstructorFunct(Functionality it) {
17            return '';
18        }
19
20        override compileLanguageStatement(LanguageStatement it) {
21            var temp = it as nl.utwente.er2.language.java.java.
      LanguageStatement;
22            var text = temp.text;
23            value = text.substring(1, text.length-1);
24            return value;
25        }
26
27    }
```

APPENDIX E

Composition Specification Language

E.1 Xtext

```
grammar nl.utwente.er2.composition.Composition with org.eclipse.
    xtext.common.Terminals

generate composition "http://www.utwente.nl/er2/composition/
    Composition"

Model:
    'modules' '{'
    (modules+=ModuleInit)*
    '}'

    'bindings' ' {'
        (mappings+=Mapping)*
    '}'

    'ordering' '{'
        (order+=[ModuleInit] ';')*
    '}'
;

ModuleInit:
    name=ID ':' type=ID ';'
;

Mapping:
    'bind' '('(source=[ModuleInit] ',')? target=[ModuleInit] ')'
    '{'
        (moduleMappings +=varMapping|staticMapping+=StaticMapping)*
    '}'
;

StaticMapping:
    value=Literal '->' target=ID ';'
```

```
31 ;
32
33 Literal:
34     STRING|INT
35 ;
36
37 varMapping:
38     source=ID '->' target=ID ';'
39 ;
```