Towards an Aspect-Oriented Architecture for Self-Adaptive Frameworks

Eddy Truyen and Wouter Joosen
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{eddy.truyen,wouter.joosen}@cs.kuleuven.be

ABSTRACT
Self-adaptive systems are systems that are able to autonomously adapt to changing circumstances without human intervention. Typically, these systems are often designed as a framework that defines a generic architecture which can be reused across multiple applications. In this paper we study and compare two prominent examples of self-adaptive frameworks. We study the applicability of aspect-oriented programming (AOP) to see where and how AOP technology can provide an interesting alternative for implementing parts of the architecture of self-adaptive frameworks. Subsequently, we present our initial ideas towards an aspect-oriented architecture for self-adaptive frameworks.

1. INTRODUCTION
In this paper we study existing self-adaptive frameworks to see where and how AOP can be applied in the architecture of self-adaptive systems. We use the well-accepted reference model of autonomic computing [9] as the foundation for studying self-adaptive frameworks. A self-adaptive system typically implements a control loop that consists of several parts (see Figure 1): monitoring, analyzing, planning and execution. These parts share and use system-specific adaptation knowledge about when, where and how to adapt the system. This adaptation knowledge is typically specified by developers or administrators as high-level adaptation policies following the well-known Event-Condition-Action format.

A self-adaptive framework offers reusable support for building self-adaptive systems by offering a basic infrastructure that can be reused across multiple applications. The framework is mapped to system-specific probes and effectors via an intermediate translation layer (not shown in Figure 1).

In this paper we study and compare two prominent examples of self-adaptive frameworks. These are the Rainbow framework [6] and the Autonomic Management Engine (AME) [9]. We study the applicability of aspect-oriented programming (AOP) to see where and how AOP technology can provide an interesting alternative for implementing parts of the architecture of these self-adaptive frameworks. Based on this, we present our initial ideas towards an aspect-oriented architecture for self-adaptive frameworks.

The rest of this paper is structured as follows. We first present an overview of the Rainbow framework and the AME engine in section 2. This section also compares these two self-adaptive frameworks by identifying their commonalities and differences. Subsequently section 3 discusses where and how an aspect-oriented design might offer an interesting alternative to the original framework-based design. Section 4 thereafter proposes our preliminary aspect-oriented architecture for building self-adaptive frameworks. Finally, section 5 summarizes our findings and discusses future work.

2. STUDY OF SELF-ADAPTIVE ARCHITECTURES
In this section we look at two prominent examples of self-adaptive frameworks in the literature, namely the Rainbow framework and the Autonomic Management Engine.

2.1 The Rainbow framework
The goal of the Rainbow framework is to offer a generic architecture for building self-adaptive systems such that the various components can be reused across a family of systems. The architecture of the Rainbow framework consists of three layers (see Figure 2): a system-specific infrastructure layer, an architectural layer, and a translation layer.

The system-specific infrastructure layer offers low-level probes for measuring all kinds of properties such as response-time of connections and loads of servers, effectors for performing change, and other infrastructural services such as resource discovery services.
At the architectural layer, the Rainbow framework includes in its run-time system an architectural model of the executing system. This architectural model typically represents the executing system as a set of components, connectors, properties (attached to components and connectors) and constraints (to restrict the components, connectors and properties within certain well-defined configurations). The model manager component gives access to this architectural model. Gauges will then aggregate events from the probes and update the architectural model. The constraint evaluator will periodically evaluate the constraints of the architectural model. In case of constraint violations, adaptations will be triggered. The adaptation engine will then determine the course of action depending on the circumstances.

The translation layer is responsible for bridging the abstraction gap between the system layer and the architectural layer. For example, to translate an architectural level change operator to a system-specific effector mechanism.

In order to apply the framework to a specific system, the framework must be populated with specific adaptation knowledge about the system. This includes the architectural style of the system (i.e. the component and connector types and properties of these), the rules for evaluating constraints, the adaptation strategies and finally the specific action operators that can be performed on the system’s elements. The authors show that reuse of this system-specific adaptation knowledge across a family of systems is also possible. The extent to which this reuse is possible depends on whether these systems share the same architectural style (e.g. client-server) and system concerns to be achieved (e.g. performance, availability).

2.2 The Autonomic Management Engine

The goal of the autonomic management engine (AME) [9] is to offer a generic architecture and complete toolkit for adding a self-adaptive control loop to existing applications. Events are represented as first-class entities and contain all kinds of relevant data such as the reporting component, the affected component, the situation of the event. Events are generated by the applications themselves or are extracted from existing logs.

The architecture of the AME is depicted in Figure 3. The AME is built around a message bus that distributes events to the interested components: event dispatcher, action manager, analyzer, aggregator. The way each component handles the event is configured in the resource model. This resource model contains the system-specific adaptation knowledge for a specific application.

The Analyzer component is responsible for monitoring the application and issues indication events to the message bus if needed. The analyzer executes the decision algorithms as defined by the resource model. The decision algorithms gather information using service objects which represent the relevant resources of the executing system.

The Aggregator component aggregates the indication events so that if an indication event has occurred a certain number of times (in consecutive cycles), an aggregated indication event is issued to the message bus. It is possible to configure holes so that, for example, an aggregated indication event is triggered if the indication event occurs two out of three times (one hole) [9].

The Action Manager calls the Action Launcher component when an aggregated event is received from the message bus. The Action Launcher component in turn uses the Service Objects to effectively perform the change.

Service Objects make use of Common Information model (CIM) classes that are used for monitoring and effecting a particular resource [9]. A CIM class offers three kinds of reflective operations: ENUM, which allows the enumeration of all instances of a particular resource, GET for querying properties of a specific resource instance, and INVOKE for manipulating the resource in a particular way by invoking a method that performs a reconfiguration action. CIM’s are declaratively defined using the Managed Object Format (MOF) language which is an IDL for defining the various CIM methods and connecting them to a specific implementation classes (called ITL classes) of the underlying appli-
ction. These ITL classes define hooks for plugging in specific classes of the underlying application and thus act as some sort of framework interface. CIM’s are managed by the Common Information Model Object Manager component (CIMOM). Service Objects may also use standard shell commands (i.e. basic operating system services) for monitoring and manipulating resources.

2.3 Comparison

There is a clear overlap between the architectures of the two frameworks. Figure 4 shows a mapping between the architectural components of both frameworks.

<table>
<thead>
<tr>
<th>Rainbow</th>
<th>AME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model manager</td>
<td>Service manager</td>
</tr>
<tr>
<td>Adaptation engine</td>
<td>Analyzer</td>
</tr>
<tr>
<td>Gauges</td>
<td>Can be implemented in the ILT classes</td>
</tr>
<tr>
<td>Constraint evaluator</td>
<td>Decision algorithms</td>
</tr>
<tr>
<td>n/a</td>
<td>Aggregator</td>
</tr>
<tr>
<td>Adaptation engine</td>
<td>Action manager</td>
</tr>
<tr>
<td>Translation infrastructure</td>
<td>CIM classes generated from MOF specs</td>
</tr>
<tr>
<td></td>
<td>ILT classes defining hooks for utility classes</td>
</tr>
<tr>
<td>Effectors</td>
<td>Utility classes</td>
</tr>
<tr>
<td>Probes</td>
<td>Shell commands</td>
</tr>
</tbody>
</table>

Figure 4: The table presents a mapping from architectural components of the Rainbow framework to architectural components of the Autonomic Management Engine (AME).

There are some interesting commonalities and differences to mention. Both frameworks support monitoring, analysis and execution. Typically, in both frameworks, access to the underlying system through the model manager/service objects is highly reflective in order to make these components generic. For example, for getting a numeric property of a particular resource, the AME offers a generic operation in the form of getNumProperty(Resource Type, Resource Identity, PropertyName).

The largest difference between the two frameworks is the way the frameworks make abstraction of the underlying application: AME takes a resource-centric approach, while the Rainbow framework takes an architecture-centric approach. In AME, the application is modelled as a set of service objects that each represent a certain resource. In Rainbow, the application is represented using an overall architectural style.

Another difference is that the AME framework also supports planning whereas the Rainbow framework does not. In AME, the Analyzer component will detect the occurrence of problematic situations and trigger an indication event to indicate that a problem has occurred, while the Aggregator component really keeps a historical view on the problem in development and decides when corresponding action must be taken. Planning when to adapt thus requires analysis of sequences of events and must take into account the history of previous execution traces.

Rainbow additionally supports gauges that collect and aggregate information from the underlying system probes. This is not explicitly supported in AME although this functionality can be implemented as part of the ILT classes.

3. APPLICABILITY ANALYSIS OF AOP

In both frameworks, the self-adaptive control loop seems already well separated from the underlying application by means of a translation infrastructure. In fact, we could not find any significant crosscutting concern (related to self-adaptation) in the design of these frameworks. This indicates that using AOP just for modularizing crosscutting concerns does not imply much improvements to the design of existing self-adaptive frameworks. Early work from Yang [20] proposed an aspect-oriented approach for modularly integrating existing applications with a rule engine that implements the self-adaptive control loop. However, the authors did not continue this work on AOP has been dropped in their further research on self-adaptive systems.

The goal of this section is to explore other usage scenarios of, and advantages manifested by AOP. For different parts of the self-adaptive architecture we explore where and how aspect-oriented programming can lead to a better design.

Translation infrastructure. The translation layer that maps the architectural layer to a lower-level system is typically implemented using program transformation or using framework specialization.

By means of program transformation, it is easy to take multiple artifacts from several different domain-specific languages and other non-programming artifacts [12]. Another advantage is that the translation layer infrastructure is quite reusable across multiple applications as the program transformers can be parameterized with application-specific mappings. It has been argued however that program transformation also entails several disadvantages with respect to composability, scalability, understandability [12].

In the other approach, frameworks offer hooks that have to be specialized towards the underlying application. For example in the AME framework, these hooks are represented by the ILT interfaces. These hooks are typically filled in by classes that wrap around existing classes of the system. It is well-known however that in the pure object-oriented programming model, wrappers lead to various problems such as object identity mismatch and problems with bi-directional dependencies [10].

Aspect-oriented programming languages such as CaesarJ [11] support the on-demand remodularization of an existing application in order to fit it into a separate view of architectural abstractions. In this sense, aspect-oriented programming has a high value for representing architectural models and mapping these to various applications.

Monitoring and Planning. The Aggregator component in the AME framework is typically implemented as a user library that contains functionality for aggregating events. Event-based AOP [5] languages already support this functionality through well-defined language constructs. Using event-based AOP, pointcuts can be written that match with a historical sequence of events. This ability seems an interesting way for subsuming a user library, especially if the latter is cumbersome and difficult to use. Various researchers have studied and created language constructs for history-based pointcuts. These include tracematches [4] and tracechecks [16]. Also the experimental ALPCHA programming language supports history-based pointcuts [2].
In particular the work of tracematches [4] presents various examples of how complex monitoring patterns can be concisely expressed as regular expressions over sequences of events, each of which are defined as pointcuts. The proposed language construct has been fully implemented as a language extension of Aspect/J implemented in the AspectBench Compiler suite [1]. In the remainder of this paper, we will use the tracematches construct to illustrate our ideas.

**Event brokering.** Instead of using the message bus in AME, AOP can be used as underlying communication mechanism. As indicated by Matti Hiltunen [8], event-based communication can be simulated by AOP in various ways. Either by means of naming conventions or annotations. More recent work from Rajan [13] explicitly supports a dedicated language construct for the notion of event types as explicit joinpoints.

**Summary.** The above analysis indicates that some parts of the control loop can be implemented using aspects. In particular, aggregation of events can be implemented by means of event-based AOP, whereas an translation infrastructure can be implemented using the concept of on-demand remedialization. Finally, aspects could directly implement adaptation policies in a concise and easy-to-understand manner, depending on the ability to create a domain-specific or concern-specific language on top of the base aspect-oriented programming language.

4. **TOWARDS AN ASPECT-ORIENTED ARCHITECTURE**

In this section we present our initial ideas about an aspect-oriented architecture for self-adaptive frameworks. Figure 5 gives an overview of the architecture. The architecture consists of three parts:

- A self-adaptive interface that specifies all abstractions that are needed to formulate and execute the system-specific adaptation policies.
- An application-specific binding that connects a particular application to the self-adaptive interface.
- A set of reusable aspects where we distinguish between aggregator aspects and adaptation aspects.

![Figure 5: Aspect-oriented self-adaptive architecture](image)

We outline how our aspect-oriented architecture is implemented in the context of the Caesar.J programming language. Our running example for illustrative purposes is based on an existing case study from the Rainbow project [6] that we slightly have adapted.

We would like to stress that the presented design has not been validated in a running proof-of-concept. The examples in Caesar.J are only for illustrative purposes. In fact, as will be clear below, the presented concepts may require language constructs that are not supported in the Caesar.J language, but that are supported in event-based AO programming languages with support for history-based pointcuts.

4.1 **Self-adaptive interface**

The self-adaptive interface provides an abstraction layer that reflects only information relevant for the self-adaptive framework. The self-adaptive interface is similar to the architectural model of the Rainbow framework in that it specifies all that is needed to formulate system-specific adaptation policies. Similar to the Rainbow framework, the self-adaptive interface views an application as a set of components and connectors and both component and connectors can have properties and adaptation operators attached to them. The properties describe the kind of information that is needed to determine when and what change is necessary. The operators facilitate the basic primitives that are needed to implement various adaptation strategies.

We propose to extend Rainbow’s approach by also declaring various event types in the self-adaptive interface. These event types represent semantic information about problematic situations that may require change. For example, a sophisticated failure detection component can throw a specific...
event type to indicate a specific failure. Note that we distinguish between two categories of events, as inspired by AME: indication events represent the occurrence of constraint violation or any other problematic situation which does not require immediate change but maybe in the future. An adaptive event is triggered to communicate that adaptation is required now. Finally, the self-adaptive interface optionally also declares various operations that represent common infrastructure services such as resource discovery services.

What to put exactly in the self-adaptive interface for a particular application depends on (1) the architectural style of the application (e.g. client-server) and (2) the system concerns to be achieved (e.g. performance, availability). This is an essential element that we take from the Rainbow approach.

Consider the following concrete application, a web-based client-server system: “The system consists of a set of clients that each make stateless requests to one of separate server groups. Clients connected to a server group send requests to the group’s shared request queue, and servers that belong to the group grab requests from the queue. The system concerns focus primarily on performance and availability. Specifically, the response time and availability as experienced by the clients. A queuing theory analysis of the system identifies that the server load and available bandwidth are two properties that affect the response time” [6]. We extend this case study by also including server failures as an important type of event. Failures are important information because not immediately dealing with a server failure may seriously affect the availability and responsiveness of the overall server group.

Based on this information, the developer defines the self-adaptive interface for the system. For example, the self-adaptive interface can consist of the following elements:

- Components and Connectors: Server, Client, ServerGroup, Link,
- Properties: Server.load, Client.responseTime, ServerGroup.load, Link.bandwidth,
- Adaptation Operators: Client.move(), ServerGroup.addServer(),
- Adaptive Event types: ServerFailure, LinkResponseFailure,
- indication event types: ResponseTimeExceeded,
- Infrastructural operators: findBestServer().

In the aspect-oriented programming language CaesarJ, the self-adaptive interface can be implemented as an abstract top-level Caesar class. This class declares various nested classes each of which introduce a particular component or connector type, or event type:

```
public abstract class SelfAdaptiveInterface{
    public abstract class Component{}
    public abstract class Connector{}
    public abstract class AdaptiveEvent {}
    public abstract class IndicationEvent {}
}
```

The concrete self-adaptive interface for the client-server architectural style as introduced above is then implemented as follows in CaesarJ:

```
public abstract class AdaptiveClientServer extends SelfAdaptiveInterface{
    public abstract class Server extends Component{
        public abstract int load();
    }
    public abstract class Client extends Component{
        public abstract int responseTime();
        public abstract void move(ServerGroup from, ServerGroup to);
        public abstract Link getLink();
    }
    public abstract class ServerGroup extends Component{
        public abstract void addServer(Server server);
        public abstract int load();
    }
    public abstract class Link extends Connector{
        public abstract int bandwidth();
        public abstract Client client();
        public abstract ServerGroup serverGroup();
    }
    public class ServerFailure extends AdaptiveEvent{
        ServerGroup failedServerGrp();
        Client[] affectedClients();
    }
    public class ResponseTimeExceeded extends IndicationEvent{...
    }
    public class LinkResponseFailure extends AdaptiveEvent{...
    }
    public abstract Server findBestServer(Client[] c1);
}
```

### 4.2 Application-specific binding

The self-adaptive interface must be bound to a concrete system through the application-specific binding. We subdivide this binding into four parts. (1) It first maps software elements from the underlying system to component and connector types of the architectural model. (2) It determines how to compute the necessary properties of components and connectors by accessing state from the application classes or by using the underlying probes. (3) It connects the adaptation operators to either software elements of the system or by means of a reflective API for implementing the distributed adaptations if necessary. This reflective API might give support for dynamic reconfiguration. (4) Finally, it implements the common infrastructure services such as the facilities for resource discovery. In CaesarJ, an example application-specific binding may look as depicted in Figure 6

---

1 Notice other work such as Eos [14], Classpects [15] and ObjectTeams [7] support similar ways of integrating the self-adaptive interface on top of an existing application
Adaptation policies that are triggered by the occurrence of an adaptive event can be directly implemented as an aspect. For example, the aspect below encodes the following adaptation policy: *when a server crashes in a certain server group, a new server must be added to the server group; this new server must suit the needs of the clients that are currently connected to that server group.*

```java
public cclass FailureAdaptation {
    // Propagation of indication events
    after (ServerFailure f) : execution (ServerFailure.new()) & this(f) {
        Server s = findBestServer(f.affectedClients());
        f.failedServerGroup().addServer(s);
    }
}
```

Note that the adaptive event types and adaptation operators on which the aspect depends are stable abstractions in a well-defined family of applications. More specifically, this family of applications consists of those applications that share the same architectural style (client-server) and that target the same system concerns (responsiveness, availability) [6]. In theory, the above aspect is thus reusable within this family of applications.

**Property-based adaptation policies.** Another category of adaptations are not triggered by the occurrence of adaptive events from underlying system components, but instead are triggered when some property of the system violates a well-defined constraint. Consider for example, the adaptation policy stating that when the response time observed by a particular client exceeds a well-defined threshold, the ResponseTimeExceeded indication event must be signaled. Implementing this adaptation policy requires evaluating the architectural model periodically and if a constraint violation has been found, the corresponding event must be thrown.

At first sight, implementing this adaptation policy as a single aspect is not possible and does not make sense. This is because AOP does not support periodic evaluation of constraints. Furthermore, existing constraint evaluator technology can evaluate constraints very efficiently already.

The way forward is to integrate existing constraint evaluator components into the aspect-oriented architecture. When a violation against the above constraint example is found, this constraint violation can be simply communicated by throwing the appropriate indication event as declared in the self-adaptive interface (i.e., throw a ResponseTimeExceeded event in this case). One open issue is that a typical constraint evaluator component needs a reflective interface for accessing and evaluating the properties of components and connectors. The current self-adaptive interface is however not reflective.

Notice that by means of heuristics it would be possible to use AOP for implementing property-based adaptation policies. For example, based on run-time profiling, one could assume that a certain time interval has elapsed when there have been 5 calls to some logging API [4]. Based on this heuristic, one could assume that the response time for a client request has exceeded when the corresponding response has not been received after 5 calls to the logging API. Then using history-based pointcuts one could concisely express the adaptation policy. For example using tracematches [4] the adaptation policy is expressed (see Figure 7) as a regular expression over three pointcuts that respectively model the occurrence of a client request, server response and log call.

---

**Figure 6:** Application-specific binding of self-adaptive interface to web application.
Aggregating aspects. Many adaptations should not be triggered by one single occurrence of a critical event or one occurrence of a constraint violation. Instead, these are often only indications of a potential need for change. As such, event aggregation is necessary to plan for adaptations. Actions are then only triggered after a certain sequence of indication events has occurred. As already pointed out, event-based AOP languages such as EAOP [5] could be used to match with sequences of events as follows:

Aggregator aspects can then look for certain sequences of indication events by means of pointcuts. When a pointcut fires, the corresponding advice simply throws an adaptive event. In turn, this adaptive event will trigger the appropriate adaptation aspects, as outlined above. The following tracematch states that the LinkResponseFailure event should be thrown after 10 occurrences of the ResponseTimeExceeded event.

4.4 A note on using annotations

We used a naming convention in pointcuts in order to use the aspect weaver as an implicit event broker. The use of such naming conventions could be problematic, however. As pointed out by Hiltunen [8], another way to communicate events is by means of annotations. Currently in Java, however, it is only possible to attach annotations to static program elements such as classes, methods and fields. It is not possible to attach annotations to dynamic elements such as constructor calls or method invocations. If this would be supported, a failure detection component could simply communicate the detection of a component by creating an instance of a fixed class for events, say AdaptiveEvent, and attach semantic information by means of annotations. A work-around that simulates this is the use of anonymous inner classes as follows:

5. SUMMARY AND FUTURE WORK

In this paper we have shown several cases where AOSD programming concepts could be used to implement parts of self-adaptive frameworks. At the level of the (system-independent) architectural layer, event aggregation policies and certain types of high-level adaptation policies can be concisely implemented as aspects themselves. This is due to the abstraction mechanisms provided by state-of-the-art AO programming languages, but also by advances in event-based AO programming languages which allow to write pointcuts that match a sequence of events. It also has been shown that the concept of an abstract architectural model can be easily implemented by means of the AO programming languages which allow to write pointcuts between existing self-adaptive infrastructure (e.g. constraint evaluators, rule engines) and the self-adaptive interface, (2) annotations for dynamic join points, (3) interference between aspects, (4) embedded domain-specific languages in AOP.

Future work involves studying whether the aspect-oriented architecture actually has a cleaner design as compared to the design of the existing self-adaptive frameworks. It is important to (i) investigate open issues and (ii) validate our initial aspect-oriented architecture in a larger case study. In particular we plan to study a case on implementing self-healing as a separate concern in a traffic management application [19].

Open issues to be investigated include: (1) the integration between existing self-adaptive infrastructure (e.g. constraint evaluators, rule engines) and the self-adaptive interface, (2) annotations for dynamic join points, (3) interference between aspects, (4) embedded domain-specific languages in AOP.

Validation of the benefits of AOSD will also be performed in this case study. As indicated above, the Rainbow and AME frameworks already separate the concern of self-adaptation from the specific system or application. Therefore AOSD does not provide much improvement with respect to the modularization of the crosscutting concern self-adaptation. As indicated by other research [3, 12], we believe the largest improvement would be that aspect-oriented programming languages offer a uniform platform for implementing domain-specific languages for encoding system-specific adaptation knowledge.
Acknowledgments
We would like to thank the reviewers for their helpful comments to improve this paper. Also thanks to Frans Sanen for proofreading the paper.

6. REFERENCES