Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study

Filip Krikava, Philippe Collet, Romain Rouvoy

Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study

Filip Křikava
University Lille 1 / LIFL
Inria Lille - Nord Europe
Lille, France
filip.krikava@inria.fr

Philippe Collet
Université Nice Sophia Antipolis
I3S - CNRS UMR 7271
Nice, France
philippe.collet@unice.fr

Romain Rouvoy
University Lille 1 / LIFL
Inria Lille - Nord Europe
Lille, France
romain.rouvoy@inria.fr

ABSTRACT

Control theory provides solid foundations for developing reliable and scalable feedback control for software systems. In particular, there exist well-established feedback controllers for common problems, however, the state-of-the-art approaches for designing self-adaptive software systems do not primarily support classical feedback control and its integration is usually engineered manually.

In this paper, we revisit the *Znn.com* case study and we present a new implementation based on classical feedback controllers. We show how these controllers can be easily integrated into software systems through control theory centric architecture models and domain-specific modeling support. We also provide an assessment of the resulting self-adaptive characteristics, quality attributes and limitations.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

self-adaptive software systems; feedback control loop; architecture models; domain-specific modeling; control theory

1. INTRODUCTION

Feedback control is a central element of control theory that is often envisaged as one of the viable solutions for self-adaptive software systems engineering [8, 30, 34]. Control theory provides solid foundations and a systematic approach for developing reliable and scalable adaptation mechanisms, *controllers*, which drive the system adaptation [3]. However, integrating these controllers into comprehensive self-adaptive software systems remains challenging [8, 10]. This requires selecting the appropriate target system outputs and control inputs—*i.e.* the necessary interfaces exposing the target system state and management operations as *touchpoints*—devising the actual controller design, and finally building the software architecture that integrates the controller into the target system [20].

There are already several approaches trying to address the many challenges in engineering self-adaptive software systems (*cf.* surveys in Salehie and Tahvildari [34] or Villegas *et al.* [37]). They mostly aim at reducing the implementation using a specific type of adaptation mechanism (*e.g.* utility theory in Rainbow [17], requirement models in Zanshin [5], RDF control objectives specification in DYNAMICO [36], or rule-based languages in StarMX [6])[1]. While these mechanisms are all relevant, the support for classical feedback controllers from control theory as well as the integration activities (connection between the adaptation mechanism and the target system) have received less attention. As a matter of example, well-established feedback controllers for common and recurring problems (*e.g. Quality of Service* (QoS) management [2, 19], overload control [16] or performance guarantees [1, 3, 4]), keep being manually tuned and integrated into target systems. This requires extensive handcrafting of non-trivial code, which gives rise to significant accidental complexities, particularly in the case of distributed systems or complex control schemes.

In this paper, we revisit the *Znn.com* case study and present a new solution that is based on classical feedback controllers implemented using our previously proposed control theory centric architecture models [26]. Concretely, we show an integration of a content delivery adaptation using an existing and proven control algorithm [2, 3].

Our architecture model is based on a technologically agnostic domain-specific modeling language for defining *Feedback Control Loops* (FCLs) through hierarchically organized networks of adaptive elements representing the FCL processes (*i.e.* monitoring, decision-making and reconfiguration). This model is statically typed and supports composition, distribution and reflection, thereby enabling coordination and composition of multiple distributed FCLs using control schemes. The use of domain-specific modeling techniques raises the level of abstraction at which the FCL architectures are defined, and support is provided for automated implementation code synthesis and verification.

The application to *Znn.com* enables us to demonstrate the model capabilities to progressively refine adaptation mechanisms, going from local to distributed and adaptive control. Furthermore, we show how it can be used for the system identification design phase.

The remainder of this paper is organized as follows. Section 2 describes the *Znn.com* case study. Section 3 presents our control theory centric architecture models. Section 4 illustrates their application on the *Znn.com* case study demonstrating progressively better solutions, with local then distributed control, adaptive control and help in system identification. Section 5 assesses the self-adaptive characteristics, quality attributes and limitations of the approach. Finally, Section 6 reviews related work and Section 7 concludes the paper.

[1] All these examples are taken for the recent SEAMS venues.

2. THE ZNN.COM CASE STUDY

The *Znn.com* news service [12] is one of the exemplar case studies proposed by the *Software Engineering for Adaptive and Self-Managing Systems* (SEAMS) research community. We chose it because it models a relevant, well-known problem and has been already implemented by different other approaches (*cf.* Section 6). Thus, it serves as a reference to the community. *Znn.com* is a model of a real news service provider, such as *cnn.com*. It is web-based N-tier client-server system that serves multimedia news content to its customers. The control objective is then to make the *Znn.com* system serve the content within reasonable response time and quality in the event of traffic spikes caused by highly popular news events.

Within the frame of this case study, several adaptation challenges have been identified including *content adaptation* (*e.g.* serving reduced content quality), *service differentiation* (*e.g.* prioritizing premium customers), and in the case of distributed deployment, *infrastructure adaptation* (*e.g.* increasing the size of the server pool serving the news content). In Section 4, we show how these challenges can be addressed using our approach, introduced in the next section.

3. CONTROL THEORY CENTRIC ARCHITECTURE MODELS

This section outlines our approach for integrating adaptation mechanisms into software systems through control theory centric architecture models. A detailed description is provided in Krikava [25].

3.1 Principles and Design Decisions

Generality. In order to be applicable to a wide range of software systems and adaptation properties, the approach should be both technology- and domain-agnostic. *Visibility.* Following best practices [31], the FCLs, their processes and interactions should be made explicit at design time as well as at runtime. *Composability.* The FCLs should be created by composing fine-grained elements representing the FCL processes. This should help the resulting FCLs to be clearly structured, easier to implement and to reason about. It should also help to foster reuse of existing FCLs and FCL elements across multiple adaptation scenarios.

In order to meet these requirements we structure the approach around a domain-specific modeling language that is based on an actor-oriented design. The key advantage of using domain-specific modeling is in the possibility to raise the level of abstraction at which the FCLs are described, allowing to use directly the problem domain concepts, *i.e.* the concepts from control theory. Moreover, domain-specific models are particularly suitable to automated analysis and synthesis of implementation code [24]. Since FCLs are inherently concurrent and concurrent programming is known to be difficult [27], we choose to use an actor-oriented design [21] for our model. The FCL processes are represented as message-passing actors that encapsulate their state and behavior. This allows one to implement these processes without worrying about thread safety, which greatly simplifies code [27]. The actor model is also known to be highly scalable [18], supports distribution computation, and is easily usable as there exist several high-performing actor libraries[2].

To better illustrate our approach, we use the Apache overload control FCL (*cf.* Figure 1) from Hellerstain *et al.* [20, §4.6.2][3], which can be considered as a simple adaptation mechanism for the *Znn.com* case study. It adjusts the maximum number of connections to be processed simultaneously (*MC*) based on the difference between reference (MEM^*) and measured memory usage (*MEM*).

[2] `http://bit.ly/1f41vHw`

[3] For simplicity we only use the case with one controller.

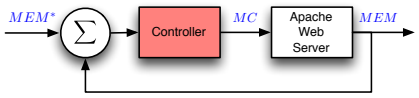


Figure 1: A block diagram of Apache overload control FCL [20]

3.2 Feedback Control Definition Language

Our domain-specific modeling language, called *Feedback Control Definition Language* (FCDL) [26], is grounded on an actor-oriented component meta-model representing abstractions of FCLs. The components are actor-like entities called *Adaptive Elements* (AE). An architecture is created by assembling and connecting AEs into hierarchically composed networks that form closed FCLs.

Syntax. An AE exposes a well-defined interface that abstracts its internal state and behavior. It defines properties together with input and output ports through which AEs exchange data. The ports and properties data values are statically typed. In order to further improve AE reuse the language supports parametric polymorphism over the data types.

The communication can be both data-driven (*push*) and demand-driven (*pull*). Once an AE receives a message, it executes its associated behavior. The result of the execution may or may not be sent further to the connected downstream elements that in turn cause them to react and so forth. An AE can be *passive*, *i.e.* triggered by receiving a message, or *active*, *i.e.* triggered by events of interests (*e.g.* a file modification). We distinguish the following AEs as the key components of a FCL: *a sensor* (collecting raw information about the state of the target system and its environment), *an effector* (carrying out changes on the target system using provided management operations), *a processor* (processing and analyzing incoming data both in the monitoring and reconfiguration parts), and *a controller* (special case of a passive processor that is directly responsible for the decision making). FCDL also allows to construct *composite* components from both atomic AEs and from other composites. A composite defines the instances of AEs its contains and the connections among their ports. Furthermore, it can also define ports, which are used to promote ports of the contained elements. A composite is also the primary unit of deployment.

Figure 2 shows an FCDL composite `ApacheLoadControl` implementing the example FCL from Figure 1. The figure uses an informal FCDL graphical notation. Its purpose is to provide an intuitive and expressive visual representation of the model that can be easily sketched by hand. A formal textual syntax is presented further in Section 4.

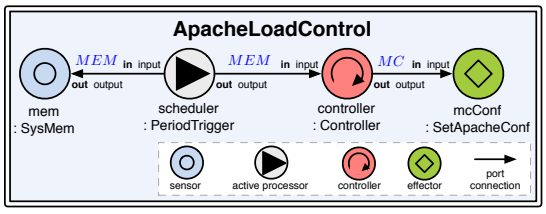


Figure 2: A FCDL model of Apache overload control FCL

The `PeriodicTrigger` is an active processor. It periodically pulls memory utilization (*MEM*) from `SysMem` sensors and in turn pushes the value to the `Controller` that computes a new *MC* configuration to be applied by the `SetApacheConf` effector. The MEM^* value is modeled as a property of the controller.

Conceptually, each AE can be seen as a target system itself, and as such it can provide sensors and effectors enabling the AE to be introspected and modified. The *provided sensors* and *provided effectors* are essentially AEs touchpoints making them reflective and thereby enabling them to be adaptable. This is a crucial feature that permits one to hierarchically organize multiple feedback control loop in an uniform way and therefore realize complex control schemes from elementary building blocks.

Semantics. The execution semantics is based on the Ptolemy [13] push-pull model of computation [40]. The message communication originates in ports. A port can be configured in one of the three modes: *push*, *pull* or *agnostic*, in which case the exact mode is resolved during element instantiation according to the connected ports. The model is restricted to allow only the same port-mode combinations. The reason is that connecting a push output to a pull input indirectly implies using a queue and analogically the opposite requires to use a scheduler. This is intended to be explicitly modeled in the architecture in order to properly define the storage and the trigger mechanisms.

An AE can execute different behaviors depending on what port or combination of ports caused its activation. To precise this, we adapt a notion of *Interaction Contracts* (IC) that defines the AE allowed interactions [9]. Concretely, an IC specifies what port interactions activates an AE, what input ports it might pull during its execution, and over which output ports it will push results. For example, the IC associated with `PeriodicTrigger` is $\langle self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output}?) \rangle$. It denotes an interaction caused by *self* activation where `input` port might be pulled during the element execution and conditionally data might be pushed to the `output` port. By using ICs, we can assert certain architectural properties, such as consistency, determinacy, and completeness. The AE interactions are clearly made explicit in its interface and therefore amenable to automatized analysis and verification. Finally, an IC denotes the type of the associate activation function allowing the generated code to be both *prescriptive* (guiding the developer) and *restrictive* (limiting the developer to what the architecture allows). For example, the Listing 1 shows the Java code generated for the `PeriodicTrigger`.

```
// polymorphic AE - allowing to pull/push any data type
public class PeriodicTrigger<T> extends AdaptiveElement {
  public void init() {} // initialization
  public void destroy() {} // termination
  protected void activate(long self, Pull<T> input, Push<T> output) {}
}
```

Listing 1: Example of the PeriodicTrigger AE generated Java class

4. APPLICATION TO ZNN.COM

This section demonstrates how some of the *Znn.com* challenges can be addressed by control theory based controllers using FCDL.

4.1 Local Content Delivery Adaptation

One of the adaptation opportunities in the *Znn.com* case study is content adaptation, whereby in the case of high server utilization, the content quality is reduced (*e.g.* degraded image quality). The web server content delivery adaptation is well studied by Abdelzaher *et al.* [1–3], providing a control theoretic approach, which we integrate to *Znn.com* using FCDL.

The aim of the adaptation is to maintain web server load at a certain pre-set value preventing both its underutilization and its overload. The content of the web server is pre-processed and stored in M content trees where each one offers the same content, but of a different quality and size. For example, let us take two trees `/full_content` and `/degraded_content`. At runtime, a given URL request, *e.g.* `photo.jpg`, is served from either `/full_content/photo.jpg` or `/degraded_content/photo.jpg` depending on the current load of the server. Since the resource utilization is proportional to the size of the content delivered, offering the content from the degraded tree helps to reduce the server load.

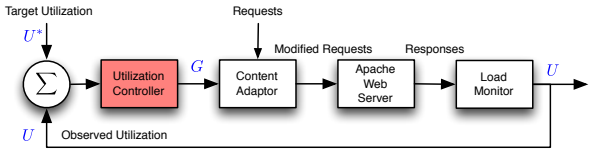


Figure 3: Block diagram of the adaptation scenario [2]

Controller Design. Figure 3 depicts the block diagram of the proposed control. The *Load Monitor* is responsible for quantifying server utilization U. It periodically measures request rate R and delivered bandwidth W. These measurements are then translated into a single value, U. Since service time of a request constitutes of a fixed overhead and a data-size dependent overhead, using algebraic manipulations, the utilization from the request rate and delivered bandwidth is derived as

$$U = aR + bW = a\frac{\sum r}{t} + b\frac{\sum w}{t} \tag{1}$$

where a and b are platform constants derived by server profiling (details in Abdelzaher *et al.* [2, 3]), $\sum r$ and $\sum w$ are the number of request and the amount of bytes sent over some period of time t, respectively. The *Utilization Controller* is a *Proportional* (P) controller[4], which based on the difference between the target utilization U^* (set by a system administrator) and the observed utilization U, computes the severity of the adaptation action G

$$G = G + K_p E = G + K_p(U^* - U) \tag{2}$$

where K_p is the proportional gain of the controller determined using standard control analytic techniques [2,3]. This value is used by the *Content Adaptor* to select which content tree should be used for the URL rewriting. The achieved degradation spectrum ranges from $G = M$, servicing all requests using the highest quality content tree to $G = 0$ in which case all requests are rejected. Shall $G < 0$ then $G = 0$ and similarly shall $G > M$ then $G = M$.

For brevity's sake, we only show a control delivery adaptation, however, the controller can be easily extended to support also QoS management control, such as service differentiation [2, §3.2].

Architecture. Figure 4 shows one possible FCDL implementation of the adaptation scenario. It is derived from the block diagram depicted in Figure 3.

The *decision-making* part from the block diagram) is responsible for mapping the current system utilization characteristics U into the abstract parameter G controlling which content tree should be used by the web server. In FCDL, we implement a general proportional controller in AE `PController`. Once a new utilization value U is pushed, it computes G using (2) and pushes the result out. The proportional gain K_p and the reference value U^* are represented as a AE properties.

The *monitoring part* is responsible for computing the system utilization U from request rate R and bandwidth W. Both values can be obtained from Apache access log file. We create an active sensor, `FileTailer`, that activates every time the content of a file changes and sends the new lines over its push output port. It is connected to the `AccessLogParser` that extracts the number of requests

[4] An alternative proportional integral controller is provided in Abdelzaher *et al.* [3].

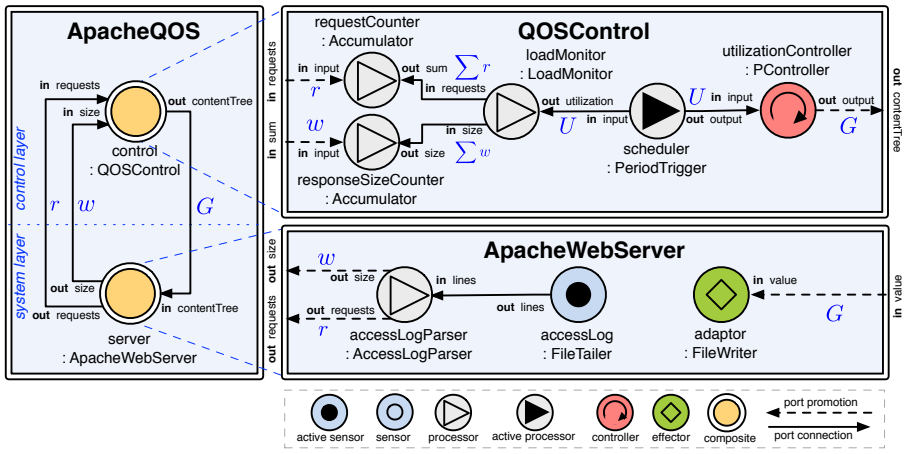


Figure 4: Apache content delivery control

r and the size of the responses w and pushes the values through the corresponding `requests` and `size` ports. Consequently, this increments the values of two connected counters `requestCounter` and `responseSizeCounter`, implemented as passive processors that accumulate the sum of all received values. To compute utilization U, the sum of requests $\sum r$ and response size $\sum w$ has to be converted to request rate R and bandwidth W—*i.e.*, the number of request and sent bytes over certain time period t. One way of doing this is by adding a `PeriodicTrigger`, an active processor that every t milliseconds pulls data from its pull input port and in turn pushes the received value to its output port. Essentially, it is a scheduler that acts as a mediator between the two connected AEs. In this scenario, it is responsible for the timing of the FCL execution. By pulling data from its input port, it activates the `LoadMonitor` processor that fetches the corresponding sums of requests $\sum r$ and response sizes $\sum w$; converts them to request rate R and bandwidth W; and finally computes U using (1). The resulting utilization is then forwarded by the scheduler to the `UtilizationController`.

In the *reconfiguration part*, the `FileWriter` updates the web server URL rewrite rules so that the newly computed content tree is used to serve the upcoming requests.

To demonstrate composition, the presented elements are assembled into three composites `ApacheQOS`, `QOSControl` and `ApacheWebServer`, representing the main composite that will be deployed, the control, and the target system, respectively. This makes a clear separation of concerns and easy to switch from Apache to another web server.

Implementation. FCDL models are implemented in a domain-specific language called *Extended Feedback Control Definition Language* (XFCDL). It is a textual DSL for authoring FCDL models that further supports modularization and AE implementation using a Java-like expression language. The language is close to Java and it uses some of its concepts, such as modularization (packages and imports), type system, and naming conventions.

The architecture consists in defining AE types that participate in the FCLs. Listing 2 shows an excerpt[5] of the `PeriodicTrigger` AE. Line 1 defines a new active polymorphic processor type with data type parameter T. Lines 2–4 declare ports including the implicit self port in order to specify its data type, followed by a property definition on line 7 and an IC on line 8.

Next, in order to assemble a FCL, we need to connect the AEs together. This is done by creating a composite (*cf.* Listing 3) in which we define all the AEs to be used (lines 4 and 7) and specify the data-flow by connecting their ports (line 12).

```
1 active processor PeriodicTrigger<T> {
2   push in port output: T
3   pull in port input: T
4   self port selfport: long // self port for self-activation
5
6   property initialPeriod: Duration = 10.seconds
7
8   act activate(selfport; input; output?)
9 }
```

Listing 2: XFCDL implementation of `PeriodicTrigger` AE

```
1  composite ApacheQOS {
2    property targetUtilization: double // U*
3
4    feature scheduler = new PeriodicTrigger<Double> {
5      initialPeriod = 30.seconds // property specification
6    }
7    feature utilizationController = new PController {
8      reference = this.targetUtilization // ref composite property
9      // ...
10   }
11   // ...
12   connect scheduler.output to utilizationController.input
13 }
```

Listing 3: XFCDL implementation of `ApacheQOS` composite

XFCDL further allows to specify the implementation of a AE interaction contracts directly using Xbase[6], a statically typed Java-like expression language that supports lambda expressions, type inference, and Java interoperability. For example, the implementation of `PController` is shown in Listing 4.

4.2 Distributed Content Delivery Adaptation

So far we have focused on the case of a single server instance. In the following text, we extend the self-adaptation capabilities to a distributed scenario in which *Znn.com* is deployed on a pool of replicated servers with a load balancer distributing requests among them. For brevity, for the next FCLs we only provide controller design and architecture sections leaving the implementation code available in the companion website.

Controller Design. The distributed deployment of *Znn.com* consists of a server pool S with n servers and one load balancer. Each server S_i runs locally the previously developed `ApacheQOS` FCL

[5]The complete XFCDL code is available from the companion website `http://fikovnik.github.io/Actress/SEAMS14.html`

[6]`http://bit.ly/1mr36bt`

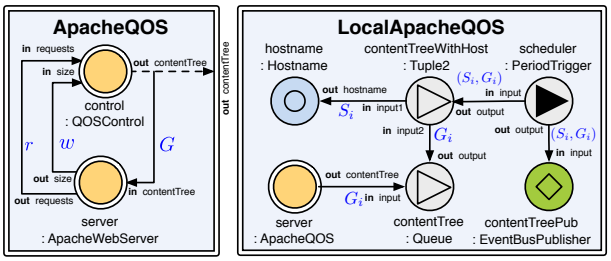
```
controller PController {
  in push port input: double
  out push port output: double
  property Kp: double // proportional gain
  property reference: double // reference input
  property loBnd: double // lower bound
  property upBnd: double // upper bound

  act activate(input; ; output)

  implementation xbase {
    var U = reference // new variable
    // implementation of the 'act activate(input; ; output)'
    act activate {
       val E = reference - input // computes the error
       U = U + Kp * E // computes new output
       if (U < loBnd) U = loBnd; if (U > upBnd) U = upBnd // corr. bounds
       U // returns the result
    }
}
```

Listing 4: XFCDL implementation of `PController` AE

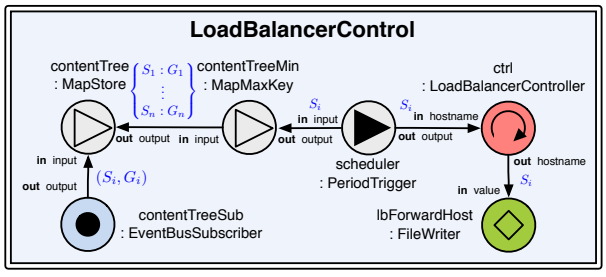
computing its target content tree G_i. In order to maintain the highest QoS, the load balancer dynamically schedules the arriving requests to a server $s \in S$ that provides the least degraded content:

$$\texttt{content\_tree}(s) = \texttt{max}(\texttt{content\_tree}(S)) \tag{3}$$

Architecture. Figure 5 depicts the FCL architecture representing the distributed control. It consists of two composites: the `LocalApacheQOS` (*cf.* Figure 5a) that runs at each of the server S_i encapsulating the local `ApacheQOS` FCL, and the `LoadBalancerControl` (*cf.* Figure 5b) that runs on the load balancer controlling the scheduler using the above equation (3).



(a) Composites running at each of the servers



(b) Composite running at the load balancer

Figure 5: Distributed QoS Management Control FCLs

The *monitoring part* is responsible for collecting the content tree status of all the participating servers. One way of doing that is to use a distributed publish/subscribe event bus into which each of the servers can push its newly computed content tree. An advantage of using an event bus is that it does not need to know *a priori* all the participating servers. In FCDL this is facilitated by two AEs: the publisher (`EventBusPublisher`) that sends data pushed over its input port and the subscriber (`EventBusSubscriber`) that pushes the published data over its output port. Concretely, in this FCL the data we want to publish are the key-value tuples of servers S_i (server hostname) with their corresponding content trees G_i. To obtain G_i, an output port has to be added to the `ApacheQOS` to which the `QOSControl` content tree output is promoted so that the value is available to the outside (*cf.* Figure 5a). The server hostname (S_i) is provided by the `Hostname` sensor. Finally, to create a key-value entry, we use the `Tuple2` processor that creates a tuple from its two inputs and pushes them using the `EventBusPublisher`. The pushed (S_i, G_i) entries are received by the `EventBusSubscriber` in the `LoadBalancerControl` composite. They are aggregated using the `MapStore` AE, which is a map storage built from the received key-value pairs. The content of the map can be observed through the pull output port.

The *decision-making part* is responsible for selecting the server with the highest G_i. The `LoadBalancerController` periodically requests the collected content tree status from `MapStore`, extracts the entry with the highest value using `MapMaxKey` and pushes the corresponding hostname.

In the *reconfiguration part*, the `FileWriter` updates the load balancer scheduling so that the newly computed server is used to handle the upcoming requests.

4.3 Distributed Resource Management

Deploying services like *Znn.com* into a cloud infrastructure is becoming mainstream. They enable services to scale elastically upon demand. This brings a new adaptation challenge in determining the appropriate number of servers based on the service utilization and available budget. In this section, we address it by developing a controller that dynamically scales *Znn.com* in a cloud based on the above content delivery adaptation.

Control Design. Similarly to the content delivery adaptation (*cf.* Section 4.1), we design a proportional controller that, based on the difference between the target content tree G^* and the observed average content tree used in the server pool $\overline{G}$, computes the corresponding number of servers n.

$$n = n + K_p E = n + K_p(G^* - \overline{G}) \tag{4}$$

The resulting value should be further constrained by a cost function representing the available budget for server resources, for example using the controllers higher bounds.

Architecture. Figure 6 depicts the resulting architecture model. The *monitoring part* is the same as in the `LoadBalancerControl` FCL. The *decision-making part* periodically computes the global average content tree $\overline{G}$ value using the `MapAvgValue` based on which it calculates the corresponding number of servers n using (4). The *reconfiguration part* adjusts the server size pool using the `ClusterManager` effector through a corresponding cloud API.

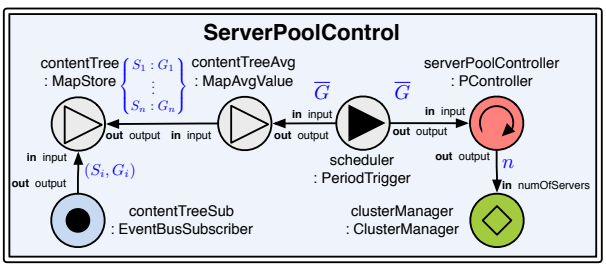


Figure 6: Dynamic Resource Management FCL

4.4 System Identification

The control theory based controllers are driven by models that estimate the dynamics of the system. While mechanical and electrical systems are governed by a number of physical laws expressible as mathematical relationships, for computing systems few of these laws exist [20]. Therefore, controllers usually employ statistical, *"black box"* models that quantify the relationship between the system inputs (*e.g.* content tree) and system outputs (*e.g.* utilization). This modeling activity usually involves two steps: (1) experimental runs to collect data, and (2) model construction based on the data. Control theory provides established methods for the model construction. However, there is no support in the data collection step and engineers have to do it in ad hoc manner.

An experimental run consists of observing the effect of control inputs on the measured outputs. In FCDL, this can be greatly facilitated by designing an open loop architecture in which target system touchpoints are used to set control inputs and observe/log corresponding system outputs. Figure 7 shows an architecture model for tuning the local content delivery adaptation controller from Section 4.1.

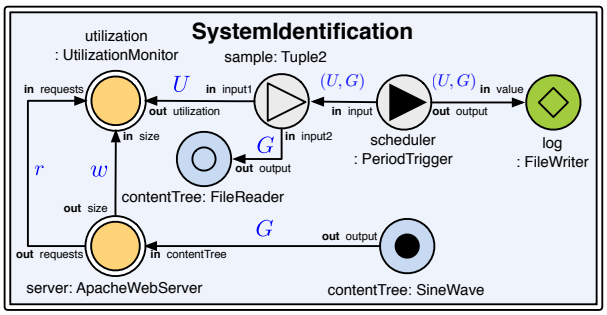


Figure 7: Apache content delivery control. The `UtilizationMontor` is a composite containing the `requestCounter`, `responseSizeCounter` and `loadMonitor` elements from Figure 5a.

The aim of this open loop is to exercise the system on range of input values G and observe its output U while executing a given workload. Instead of connecting a controller output into the `ApacheWebServer` content tree input, we connect it directly to a value generator, which in this case is a discrete sine wave (in order to consider saturation limits of the input space [20]). This corresponds to an open loop control in which the control input is set directly without considering any system output. For the data collection we reuse most of the AE developed for the FCL. It simply outputs a pair for utilization and content tree values that are periodically observed. The resulting log file can be then loaded into modeling tools, such as MATLAB[7].

4.5 Adaptive Control

A primary reason for the feedback controller proliferation in industry is their ability to tolerate a certain degree of modeling inaccuracy caused by workload deviation from the system identification conditions [4]. Their robustness can be further improved by developing adaptive controllers, *i.e.* controllers that automatically adjust their parameters based on a system identification that is performed on-line [7]. The adaptive control can eliminate the need for system profiling and controller tuning [28], and therefore significantly improve FCL portability to new load conditions and platform resource capacities.

In the previous section, we have shown an example of the FCDL support for system identification. This, coupled with the model reflection, allows one to develop adaptive controllers. Figure 8 depicts an architecture of adaptive control combined to a local content delivery adaptation FCL (*cf.* Section 4.1).

The aim is to perform an online profiling of the target system, *i.e.* the utilization U and the content tree G, and based on the measured values, to estimate the controller parameters, *i.e.* the proportional gain K_p. To do so, we first need to extend the `PController` with a provided effector that will allow to change the K_p property at runtime. Next, we reuse the part of the architecture developed for the system identification and we create the `AdaptiveController` that is responsible for the parameter estimation. A possible approach is shown by Lu *et al.* [28]. Finally, the corresponding elements are encapsulated into a new composite `AdaptiveControl` that can be simply placed on the top of the FCL we developed in Section 4.1.

Adaptive control is just one example of the possibilities of the FCDL reflection, which can also be used to design adaptive monitoring, or to organize multiple FCLs using various control schemes, such as hierarchical control. For example, we can extend the `PeriodicTrigger` with a provided effector that allows to change the triggering period at runtime so another FCL can control the loop scheduling.

5. ASSESSMENT AND DISCUSSION

In this section we assess our approach by discussing the properties and self-adaptation characteristics of control mechanisms that can be implemented using FCDL including its quality attributes and limitations.

5.1 Implementation

For a domain-specific modeling approach to be effective, it has to be associated with software development tools that automate tasks, such as model construction and implementation code synthesis [24]. To facilitate the development using FCDL, we have implemented a prototype of an EMF[8] based modeling environment called ACTRESS [26]. It provides support for FCDL modeling, verification and source code generation together with runtime platform.

The modeling support is based on the XFCDL DSL that is implemented using Xtext[9]. From the FCDL model, the code generator synthesizes an executable control system for a concrete runtime platform. Currently, ACTRESS targets Akka[10], a scalable and lightweight framework and runtime for actor-based applications on the Java platform. Since the FCDL model is already an actor-oriented model, the source code transformation is rather straightforward, essentially turning each AE type into a Java class, essentially turning each AE type into a Java class (*cf.* Listing 1), which is used as a delegate called by the actor runtime. Finally, the verification support automatically checks FCDL consistency including user-defined constraints (*e.g.* architecture bad smells, such as AE overlaps) expressed in either OCL or Xbase. The abstractions realized in the support also brings the possibility to use external model checking. For example ACTRESS provides a FCDL transformation into Promela models to verify some temporal properties using the SPIN model checker [22].

5.2 Properties

The main properties considered for the design of our approach were *Generality*, *Visibility*, and *Composability* (*cf.* Section 3). The generality is addressed by using a fine-grained FCL decomposition which should be usable in any domain for various adaptation property. FCDL is built as a technologically-agnostic model and the Java based ACTRESS implementation provides only one tech-

[7] http://www.mathworks.fr/products/matlab

[8] `http://www.eclipse.org/modeling/emf`

[9] A software language engineering framework *cf.* `http://www.eclipse.org/Xtext`

[10] `http://akka.io`

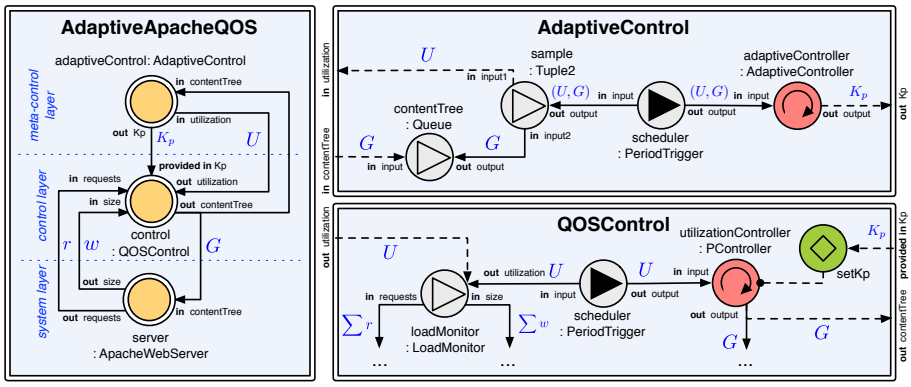


Figure 8: Adaptive Control for Apache content delivery controller

nological solution (see Section 5.3). Generality is also obtained in adaptation scenarios, as they are captured at a conceptual level using the problem domain concepts, rather than the implementation concepts. The visibility property is tackled by having all the FCL processes represented as first-class entities with explicit interactions that are precisely guided by interaction contracts. Finally, we have shown that FCLs are composed from clearly structured fine-grained AEs using ICs to guide AE implementations (cf. Section 4.1). The composition mechanisms make naturally possible to compose rather than to program FCLs.

Self-adaptation criteria and quality attributes have already been proposed to examine the strengths and weaknesses of different approaches. For the StarMX framework [6], the authors proposed a number of them, which were synthesized from other studies and adjusted for evaluation of self-adaptive software engineering approaches. They were provided to be used as a basis for evaluating and comparing different engineering approaches. While the proposed set might not be complete and is rather qualitative, we use it as a part of the FCDL and ACTRESS evaluation, as according to us, it gives a reasonable well-organized basis. Further criteria and quality attributes together with a detailed discussion are available in Krikava's thesis [25].

Self-Adaptation Criteria. The following are the selected self-adaptive criteria based on Asadollahi *et al.* [6]. Table 1 summarizes the corresponding capacities of FCDL and ACTRESS.

- *Degree of autonomy.* The capability of a framework in automating the management process.
- *Control scope.* The granularity of control scope.
- *Self-* properties support.* The ability of the framework to address self-* properties [23].
- *Management logic expression.* The mechanisms to define the self-managing requirements (*e.g.* descriptive or programmatic).
- *Runtime modification of policies* The capability of the framework in allowing runtime modification of the management logic.

Criteria	FCDL/ACTRESS Capacity
Degree of autonomy	Both open and closed loops.
Control scope	Multiple application instances.
Self-* properties	All self-*. While the FCL from section 4 all relate to self-optimization, the general FCL decomposition should be applicable to to other self-* properties as well.
Management logic	Programmatic using Xbase or Java.
Runtime modification	Supported through AE reflection.

Table 1: FCDL/ACTRESS characteristics.

Quality Attributes. The following is the analysis of FCDL and ACTRESS quality attributes as selected by Asadollahi *et al.* [6].

- *Flexibility.* FCDL provides an abstraction close to control theory block diagrams. It can represent both closed and open control loops and AE reflection allows for designing complex control schemes. Unlike most frameworks, FCDL does not dictate any system architecture nor any specific technology. Furthermore, it promotes separation of concerns in the sense that the FCL architecture and control mechanisms may be defined by control engineers while the technical/system-level processors and touchpoints may be implemented by software engineers.

  In this paper we focused on control theoretical approach, however, control theory might not always provide the most suitable approach. While the FCDL is particularly suited for developing classical feedback control, in the end it provides a general FCL abstraction and other kind of control can be realized including event-condition-actions or transition diagrams.

  Next to the *Znn.com* case study, FCDL was also used to build overload control adaptation scenarios in the domain of high-throughput computing. Details with experimental evaluation is presented in Krikava [25, §8.1].

- *Scalability.* The FCDL support for composition, polymorphic data types and ICs allowed us to incrementally refine the needed FCLs throughout the case study. These techniques are likely to allow for building larger models. ACTRESS further includes support for automatically checking model structural and temporal consistency and the Akka based runtime has low footprint (see below).

- *Usability.* Assessing software usability is difficult since it also depends on the preferences and background of its users, which is subjective by definition. Still, our approach relies on known concepts, as FCDL is using notions from control theory and component-based software engineering, XFCDL follows known concepts from Java, and the ACTRESS modeling environment is integrated in the Eclipse IDE, which might simplify adoption for the users already familiar with it. Furthermore, relying on the actor-model simplifies AE implementation without the need to protect mutable state [18].

- *Reusability.* There are two features that contributes to AE reusability: the FCDL support for data type polymorphism and the Xbase support for lambda expressions that allows to use functions types as properties. This results in higher-order polymorphic AEs definitions. Table 2 compares the AEs created and

reused[11] in the FCLs from Section 4.

FCL	New AEs	Reused AEs
Local content delivery	11	0
Distributed content delivery	12	9
Distributed resource mgmt.	3	5
Adaptive control	5	11

Table 2: Adaptive element reusability

– *Extensibility.* FCDL and XFCDL are both defined using their respective EMF meta-models. Therefore, extending their core functionality is only possible by modifying the ACTRESS source code. On the other hand, thanks to MDE, it is possible to use the FCDL models and target different systems, providing new code generators, verification techniques and the like. This is already the case as the CORONA project [32] uses FCDL for adapting service component architecture systems, transforming AE elements into components for the FraSCAti runtime [35].

– *Performance.* We consider the overhead caused by the execution of the self-adaptive layer. It can be decomposed into a fixed component related to the ACTRESS runtime platform and a variable component that depends on the complexity of the adaptation mechanism, *i.e.* the amount of memory and CPU time used by the different AEs. A single instance of the ACTRESS runtime with no composite deployed accounts for 1.5MB[12]. In Akka 2.0 version, the memory overhead is about 400 bytes per actor instance (2.7 million actors per GB of heap) with a possible throughput of 50 million messages per sec on a single machine[13].

  The size of an AE is mostly affected by the amount of state it keeps. The same applies for the execution time whose majority is spent in running the user-code of AE activation methods (*e.g.* a sample push/pull communication between two AEs with a throughput of 5000 messages per second amounts for 5% of CPU time). However, the main potential performance issues is in the indirect load caused by the sensors and effectors, which might become significant and as such it must be taken into account while designing any self-adaptive software system.

– *Testability.* FCDL models are amenable to automated consistency checking. For testing AEs implementation, there is no particular support needed, since they can be simply tested in isolation, without the ACTRESS runtime, using an unit testing framework.

5.3 Limitations

While FCDL is technologically agnostic, the ACTRESS is tightly coupled with Java. This currently limits the implementation of AEs to Java-based languages. It might pose a problem for scenarios where the touchpoints need to interact with an API that is not accessible from Java, JNI or from the network (*e.g.*, HTTP). With the MDE approach, however, it is possible to target different runtime platforms that are themselves based on the actor model. The increasing popularity of the actor model gives us a variety of different frameworks available in various programming languages.

Xbase provides a convenient way for expressing mathematical equations, but it might be too low level for expressing control based on concepts such as decision tables or state transition diagrams. These techniques can be still used used, but only through their respective Java APIs. Similarly to the other approaches that use external adaptation, FCDL relies on the fact that the target system is able to provide, or be instrumented to provide, all the required touchpoints.

It is important to note here that the abstraction we have chosen is not the only one and it is possible to have higher-level models such as goal-oriented models. FCDL matches block diagrams providing an established abstraction of FCLs which is flexible, yet rigid enough for automated code synthesis.

Finally, the control models proposed in Sections 4.2 and 4.3, *i.e.* (3) and (4) are rather illustrative. The aim is to demonstrate the capabilities of FCDL allowing one to progressively refine adaptation mechanisms and not to develop state-of-the-art load balance scheduler and resource manager.

6. RELATED WORK

IBM proposed what has become a widely referenced model for autonomic systems, referred to as the MAPE-K decomposition [23]. Since that, a number of MAPE-K framework-based approaches have been developed by focusing on different aspects of self-adaptation in software systems.

The Rainbow framework [17] provides an architecture-based approach for developing external self-adaptive software systems, using utility theory for selecting an optimal adaptation strategy in the given QoS context. It also introduced the *Znn.com* case study, implementing several strategies to address the adaptation challenges using content delivery and server pool size adaptation [10]. Unlike the control theory based controllers that are driven by mathematical relation between the control inputs and system outputs, Rainbow is using fixed threshold-based event-condition actions. As a result the controlled system is more likely to experience instability due to oscillations. For example, the system might continuously enlist or discharge servers since the adaptation condition is expressed as `c.experRespTime > M.MAX_RESPTIME` [11, p. 187]. Using the utility theory brings the possibility to link adaptation strategies with business objectives. On the other hand, it requires to precisely tune the utilities [11, p. 188] of used QoS dimensions, tactics impact vectors and weighted utility preferences which might be rather laborious to do particularly when the number of tactics or QoS dimension increases. Furthermore, these values together with some of the thresholds used in the adaptation conditions have to be at least partially re-evaluated every time the workload or system configuration changes. In contrast, the control theoretic approach enables systematic controller design with established analytical methods for tuning and evaluation providing certain guarantees [3, 20, 30]. Moreover, adaptive control (*cf.* Section 4.5) can automate this process [39]. The use of architecture models also implies a fixed number of clients and server, unlike our approach in which we do not pose any restriction as to how many servers or clients are in use. Finally, the framework was designed for scenarios that can be solved by centralized control loop and does not support hierarchical control schemes nor runtime modifications of adaptation strategies or thresholds. *The DYNAMICO Reference Model* [36] provides guidelines for designing self-adaptive software architectures with amenable adaptation goals. It consists of fixed three-layers architecture defining three FCLs, each managing different parts of context dynamics (control objectives, target system adaptation and dynamic monitoring). Similarly to Rainbow, the adaption conditions are also based on boolean expressions involving fixed thresholds. *The Zanshin Framework* [5] provides a requirements-based approach for developing external self-adaptive software systems using goal-oriented requirements engineering models. The advantage of such models is that they provide a higher level abstraction in comparison to FCDL. The adaptation

[11] AEs reused within a composite are not counted,

[12] All further measurements were conducted on MacBook Pro 2.53 Ghz, 8GB RAM, Java 1.70_17 64bit, Akka 2.2.0

[13] `http://bit.ly/1gHM975`

strategies have similar format to Rainbow, however there is no support for utility-based strategy selection and simply the first applicable one is selected. *Adapt Cases* [29] extends UML Use Case diagrams allowing to explicitly model self-adaptivity in systems using UML for its specification. The adaptation strategies are implemented in OCL using similar event-condition-action adaptation mechanism as the frameworks above. *The StarMX framework* [6] is designed for building self-managing Java-based applications using closed FCLs. It uses Java management extension for target system touchpoints and a policy-rule language for adaption engine implementation. StarMX is by the choice of technology primarily focused on adapting Java-based systems. It does not support runtime modification of the management logic nor complex control schemes.

In general, the advantage of the above framework based solutions is that they provides an architecture basis of an application, defining its structure and control, and therefore it can simplify its development [15]. On the other hand, frameworks operate within boundaries of some programming language and therefore they are limited in the level of abstraction they can provide. The possibility of a formal reasoning and verification is also limited since the structure and behavior is an integral part of the implementation. Furthermore, they always impose the use of a certain technological stack. Finally, some creative freedom is lost since many design decisions have been made by the framework designers [15].

Several approaches are exploiting the use of model-driven engineering techniques to develop self-adaptive software, in particular by using these techniques at runtime. The *model@run.time* represents an abstraction of a running system or its part and can be used to support dynamic adaptation of structure, behavior or goals of the underlying software systems [14]. For example, Vogel *et al.* [38] promotes the use of runtime executable megamodels. They present a modeling language for adaptation logic modeling together with a runtime interpreter that executes the megamodels. This is similar to what we develop in our approach, as they can also represent loop coordination and hierarchically organize them into layers. However, this solution is only a high-level overview of how the actual adaptations look like. They rely on an implicit synchronization between the megamodels and running system. Finally, their metamodel is based on EMF which has been primarily designed for the use at design time and as such it has some limitations for the use at runtime (*e.g.* high memory footprint, lack of thread-safe access).

There is also a large body of work that concerns designing feedback control for embedded computing, for example *Ptolemy II* [13]. It is an extensive framework for simulation of concurrent actor-oriented systems with the major emphasis on the ability to combine heterogeneous models of computation. We follow a similar actor-oriented approach and our execution semantics is comparable with Ptolemy push-pull model of computation (*cf.* Section 3.2). However, Ptolemy focus rather on simulation of the executable models and their transformations to the embedded systems.

7. CONCLUSIONS

While control theory provides solid foundations for designing self-adaptive systems, its mapping into implementation artifacts often results in the development of dedicated assets (*e.g.* code, models) which inevitably prevents their reuse and adoption at a larger scale. To overcome this limitation, we use FCDL, a domain-specific modeling language for integrating adaptation mechanisms into software systems. We demonstrated its use on the *Znn.com* case study, with an implementation of local and distribute content deliver adaptation, distributed resource management and adaptive control. FCDL is a domain-specific and technologically-agnostic architecture model that provides an actor-based programming model. It is coupled with runtime support for executing the designed FCLs as well as with a dedicated support for modeling, verification and implementation code synthesis.

Feedback controllers provide stability guarantees as long as the conditions observed during the system identification hold. As a matter of future work, we intend to investigate the definition of self-adaptive feedback controllers by studying the principles of defensive programming in order to better control the execution of feedback control loops.

Final Remarks. The *Znn.com* case study as provided by the SEAMS community[14] serves as a good starting point for benchmarking self-adaptive engineering approaches. However, we found two main issues with it: (1) its relevance to current industrial practice and problems should be reviewed since the distributed computation has changed dramatically from the case study inception in 2008; (2) its availability should be improved so it is easier to reproduce it with standardized benchmarks (*e.g.*. latency and throughput). More specifically, we observed that the various implementations of the *Znn.com* case study tend to assess a different set of properties, which does not help to quantitatively compare the strengths and weaknesses of each approach. Furthermore, evaluation frameworks (*e.g.* Villegas *et al.* [37]) do not highlight the core differences among autonomic frameworks, thus making it hard to understand their key orientations.

We therefore believe that the evaluation of self-adaptive systems require the definition of a comprehensive evaluation platform that does not limit itself to the identification of specific properties. Such a platform should be open to a wide variety of realistic application scenarios, which clearly identify the adaptation challenges (*e.g.* using the requirements identified by Cheng *et al.* [12]) and the quantitative and qualitative metrics to be used as part of the evaluation. This platform should also provide a packaging format providing a simple to use reproducible environment to deploy, experiment, and evaluate self-adaptive systems. From this platform, authors should be encouraged to share their assets and results they obtained in order to acknowledge their contributions and provide concrete datasets to be used by the community.

8. REFERENCES

[1] T. Abdelzaher. Modeling and performance control of Internet servers. In *39th IEEE Conference on Decision and Control*, volume 3, IEEE, 2000.

[2] T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *7th International Workshop on Quality of Service*, IWQoS, London, 1999. IEEE.

[3] T. Abdelzaher, K. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[4] T. Abdelzaher and J. Stankovic. Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, Sept. 2006.

[5] K. Angelopoulos, V. E. Silva Souza, and J. Pimentel. Requirements and architectural approaches to adaptive software systems: A comparative study. In *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, IEEE, May 2013.

[6] R. Asadollahi, M. Salehie, and L. Tahvildari. StarMX: A framework for developing self-managing Java-based systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Ieee, May 2009.

[7] K. J. Astöm and B. Wittenmark. Adaptive Control, 1995.

[14] http://seams.self-adapt.org/wiki/ModelProblem:ZnnExemplar

[8] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, 2009.

[9] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *33rd International Conference on Software Engineering*, ICSE, 2011.

[10] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, J. Whittle, R. Lemos, and Others. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, 2009.

[11] S.-w. Cheng. *Rainbow: Cost-Effective Software*. PhD thesis, Carnegie Mellon University, 2008.

[12] S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In *4th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, IEEE, May 2009.

[13] J. Eker, J. Janneck, E. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity - the Ptolemy approach. *IEEE*, 2003.

[14] R. B. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering*, FOSE, 2007.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[16] N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *EEE/IFIP Network Operations and Management Symposium.*, pages 219–234. IEEE, 2002.

[17] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[18] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, Feb. 2009.

[19] T. He, J. a. Stankovic, M. Marley, C. Lu, Y. Lu, T. Abdelzaher, S. Son, and G. Tao. Feedback control-based dynamic resource management in distributed real-time systems. *Journal of Systems and Software*, 2007.

[20] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback control of computing systems*. Wiley Online Library, 2004.

[21] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[22] G. J. Holzmann. *Spin Model Checker*. Addison-Wesley Professional, 1. edition edition, 2003.

[23] IBM. An Architectural Blueprint for Autonomic Computing, 4. edition. Technical report, IBM, 2006.

[24] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE, 2008.

[25] F. Krikava. *Domain-Specific Modeling Language for Self-Adaptive Software System Architectures*. PhD thesis, University of Nice Sophia-Antipolis, 2013.

[26] F. Krikava, P. Collet, and R. France. ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures. In *Symposium on Applied Computing (SAC), track on Dependable and Adaptive Distributed Systems (DADS)*, 2014.

[27] E. A. Lee. The Problem with Threads. *Computer*, 2006.

[28] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching. In *10th International Workshop on Quality of Service*, IWQoS, IEEE, 2002.

[29] M. Luckey, B. Nagel, C. Gerth, and G. Engels. Adapt cases. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, page 30, New York, New York, USA, 2011. ACM Press.

[30] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Decision making in autonomic computing systems. In *8th ACM international conference on Autonomic computing*, ICAC, 2011.

[31] H. Müller, M. Pezzè, and M. Shaw. Visibility of control in adaptive systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ULSSIS, 2008.

[32] R. Nzekwa. *Building Manageable Autonomic Control Loops for Large Scale Systems*. PhD thesis, Université des Sciences et Technologie de Lille - Lille I, 2013.

[33] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2010.

[34] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, 2009.

[35] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 42:559–583, 2012.

[36] G. Tamura, N. M. Villegas, H. A. Muller, L. Duchien, and L. Seinturier. Improving context-awareness in self-adaptation using the DYNAMICO reference model. In *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2013.

[37] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, volume 1 of *SEAM*, page 80, New York, New York, USA, 2011. ACM Press.

[38] T. Vogel and H. Giese. A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, number 3 in SEAMS, IEEE, June 2012.

[39] K. Wu, D. J. Lilja, and H. Bai. The Applicability of Adaptive Control Theory to QoS Design: Limitations and Solutions. In *19th International Parallel and Distributed Processing Symposium*, IPDPS, pages 272b–272b. IEEE, 2005.

[40] Y. Zhao. A Model of Computation with Push and Pull Processing. Technical report, Technical Memorandum UCB/ERL M03/51, University of California, Berkeley, 2003.