ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures

Filip Krikava, Philippe Collet, Robert France

ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures

Filip Křikava
University Lille 1 / LIFL
Inria Lille - Nord Europe, France
filip.krikava@i3s.unice.fr

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

Robert B. France
Colorado State University
Fort Collins, USA
CS Department
france@cs.colostate.edu

ABSTRACT

A common approach for engineering self-adaptive software systems is to use Feedback Control Loops (FCLs). Advances have led to more explicit and safer design of some control architectures, however, there is a need for more integrated and systematic approaches that support end-to-end integration of FCLs into software systems.

In this paper, we propose a tooled approach that enables researchers and engineers to design and integrate adaptation mechanisms into software systems through FCLs. It consists of a domain-specific modeling language that raises the level of abstraction on which FCLs are defined, making them amenable to automated analysis and implementation code synthesis. The language supports composition, distribution and reflection, thereby enabling coordination and composition of multiple distributed FCLs. Its use is facilitated by a modeling environment, ACTRESS, that provides support for modeling, verification and complete code generation. We report on its application to a concrete adaptation case study and also discuss resulting properties.

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; model-driven engineering; domain-specific modeling; domain-specific languages

1. INTRODUCTION

The growing complexity and operational costs of contemporary software systems points to an inevitable need for making them autonomously adaptable at runtime [9]. A common approach for engineering such *self-adaptive software systems* is to use *Feedback Control Loops* (FCLs) [7]. Using measurements of a system outputs (*e.g.*, response times, utilizations), a FCL adjusts the system control inputs (*e.g.*, scheduling, concurrency policies) to achieve some externally specified goals [19]. Realizing FCLs in software systems is challenging [7, 9]. It requires addressing issues related to enabling adaptation in target systems, *i.e.* providing all necessary interfaces that expose the target system state and management operations (touchpoints), designing an adaptation engine, *i.e.* a control model that drives the adaptation itself, and finally forming the architecture integrating the two together [33].

There are a number of approaches that address some of these challenges. They aim at reducing the implementation effort and provide a solid foundation for engineering of self-adaptive software systems (*cf.* surveys in Salehie and Tahvildari [33] or Villegas *et al.* [35]). However, they often target specific types of adaptation problems and require the use of certain adaptation mechanism (*e.g.* utility theory in Rainbow [16]) or are applicable to a single domain (*e.g.* mobile applications in MUSIC [30]) or technology (*e.g.* Java-based systems in StarMX [3]), thereby limiting their applicability with respect to the problem being addressed [29]. Furthermore, while there have been advances in mechanisms enabling self-adaptation and control, less effort has been put into providing a systematic approach facilitating the integration of these mechanisms from an end-to-end system perspective. Often, the integration is done manually requiring 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 propose a tooled approach that provides researchers and engineers with flexible abstractions of FCLs allowing them to more easily integrate self-adaptation mechanisms into software systems. It promotes separation of concerns whereby the development of system touchpoints, adaptation engine and the overall architecture can be decomposed and implemented by experts in the respective domains at different levels of abstraction. It is based on a technologically agnostic domain-specific modeling language called *Feedback Control Definition Language* (FCDL). It defines feedback architectures as hierarchically organized networks of adaptive elements, representing FCL processes such as monitoring, decision-making and reconfiguration. The language is statically typed, handles composition and supports element distribution via location transparency. Moreover it is reflective thereby enabling to coordinate and organize multiple control loops using different control schemes. The use of the language is facilitated by an Eclipse-based modeling environment called ACTRESS. It includes support for automated architecture consistency checking, and for code generation in which FCDL architectures are transformed into executable applications. This provides a strong mapping between the control system design and its runtime implementation.

The remainder of this paper is organized as follows. Section 2 presents a survey of related work. Section 3 introduces the adaptation scenario we use to illustrate our approach. Section 4 presents the domain-specific modeling language and is followed by an overview of the supporting tools in Section 5. Section 6 presents the evaluation and includes a discussion of the self-adaptive capabilities and properties of the approach. Finally, Section 7 concludes the paper.

SAC'14 March 24–28, 2014, Gyeongju, Korea

2. RELATED WORK

A number of approaches have been proposed to facilitate engineering of self-adaptive software systems. In this section we focus on the techniques that are the most relevant with respect to our approach and on the work that has influenced our design decisions.

Frameworks. IBM proposed what has become a widely referenced model for autonomic systems, referred to as the MAPE-K decomposition [22]. A number of MAPE-K framework-based approaches have been developed focusing on different aspects of self-adaptation in software systems. Rainbow [16] consists of a two-layer framework with an external fixed control loop for architecture-based adaptation using utility theory. While the loop is made explicit, the framework was designed for scenarios that can be solved by centralized control loop and does not support hierarchical and distributed control schemes. StarMX [3] and ASF [17] are frameworks designed for building self-managing Java-based applications using closed FCLs. They use Java management extension for target system touchpoints and a policy-rule language for adaption engine implementation. Similarly to Rainbow, they do not support runtime modification of the management logic. Other approaches focus on component adaptations, *e.g.*, K-components [10] and CASA [25]. The former introduces a component model enabling individual components self-adaption using machine learning techniques. The latter supports dynamic application adaptation by recomposing Java components.

The advantage of a framework is that it provides an architectural 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.

Middlewares. Next to frameworks, an effort has been put into extending middlewares with self-adaptation capabilities. Adaptive CORBA Template [31] focuses on CORBA applications transparently weaving adaptive behavior into object request brokers at runtime. MADAM [12] and MUSIC [30] are examples of middleware infrastructures supporting development of self-adaptive mobile applications. The former exploits architecture models to enable runtime adaptation with utility functions to compare adaptation variability. The latter provides QoS-driven adaptation including dynamic service discovery, binding, negotiation and provisioning.

These approaches aim at shielding developers from complex tasks such as resource distribution, component probing, network communication or application reconfiguration [29]. However, middleware poses highly-specific execution environments which might not be directly applicable for some systems.

Model-based Approaches. Software models have been extensively used for various parts of self-adaptive software system development. Zhang and Cheng [38] introduced an approach to create formal models of adaptive programs behavior for analysis and implementation synthesis. Their approach separates specifications of adaptive and non-adaptive behavior thereby simplifying their use.

Using models as formal specifications of self-adaptive software systems has been also proposed, *e.g.*, FORMS [37] and DYNAMICO [34]. The former supports composition of adaptation mechanisms capturing their key characteristics to allow one to compare alternative solutions. The latter is based on a three-layer architecture defining three types of FCL, each managing different parts of context dynamics (control objectives, target system adaptation and dynamic monitoring).

There is also a large body of work that concerns designing feedback control for embedded computing, for example Ptolemy II [11]. Ptolemy II 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 4.6). However, Ptolemy focus rather on simulation of the executable models and their transformations to the embedded systems.

Several approaches are exploiting the use of Model-Driven Engineering (MDE) techniques to develop particular classes of self-adaptive software. Genie [5] uses architectural models to support generation and execution of adaptive systems for component-based middlewares. The adaptive logic is specified as state machines, with each state being a system configuration and transitions being reconfiguration scripts. Diasuite [6] is a tool suite based on generative programming techniques for engineering *Sense-Compute-Control* (SCC) applications. An interesting feature of Diasuite is that the SCC architecture is enriched with a notion of interaction contract expressing the allowed interaction between its components, constraining the data and control flow. We use and extend this notion for our execution semantics (*cf.* Section 4.6).

A different model-based approach is based on the idea of using MDE 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.* [36] 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 that has some limitations for the use at runtime such as higher memory footprint and lack of thread-safe access [13].

3. ADAPTATION SCENARIO

The adaptation scenario used throughout this paper is based on the work of Abdelzaher *et al.* [1] on QoS management control of web servers by content delivery adaptation. This work notably provides (1) a control theory-based solution to a well-known and well-scoped problem, and (2) enough details for its re-engineering. For our illustration, we only consider a single server case with all requests having the same priority.

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 (*e.g.* different image quality). 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 reducing the server load when the server is under heavy load.

Figure 1 shows 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 some algebraic manipulations, the utilization from the request rate and delivered bandwidth

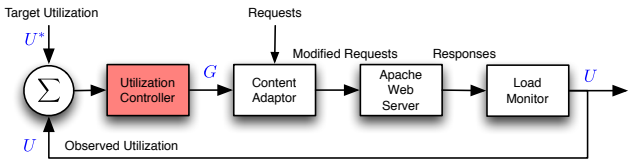

Figure 1: Block diagram of the adaptation scenario [1]

is derived as

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

where a and b are some platform constants derived by server profiling (details in Abdelzaher *et al.* [1]), $\sum r$ and $\sum w$ are respectively the number of request and the amount of bytes sent over some period of time t. The *Utilization Controller* is a *Proportional Integral* (PI) controller, which, based on the difference between the target utilization U^* (set by a system administrator) and the observed utilization U, computes an abstract parameter G representing the severity of the adaptation action. This value is used by the *Content Adaptor* to choose 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. It is computed as

$$G = G + kE = G + k(U^* - U) \quad (2)$$

where k is the controller tuning parameter that is determined *a priori* using some control analytic techniques (details in Abdelzaher *et al.* [1]). Shall $G < 0$ then $G = 0$ and similarly shall $G > M$ then $G = M$. If the server is overloaded $(U > U^*)$ the negative error will result in decrease of G which in turn changes the content tree decreasing the server utilization and vice versa.

4. MODELING FCL ARCHITECTURES

In this section, we present our approach for integrating the self-adaptive mechanisms into software systems through external FCLs.

4.1 Principles

Extracting from challenges identified in recent studies [9, 7, 33], we identify the following desirable properties for our solution:

- *Generality.* The approach should be both domain-agnostic and technology-agnostic, being applicable to a wide range of software systems and adaptation properties.
- *Visibility.* The FCLs, their processes and interactions should be made explicit at design time as well as at runtime, facilitating coordination of multiple control loops using different control schemes.
- *Tooling.* Provide tool support allowing developers to automate some recurring development tasks involving design, implementation and analysis of FCL. It should support traceability from the control design to the runtime implementation and should ensure a strong mapping between design and runtime control concepts. Together, these properties aim at increasing the overall understanding of the self-adaptive capabilities.

Furthermore, feedback control might cross boundaries of single system and thus the approach should support remote distribution of FCL. It should also follow good software engineering practices allowing modular specification, as well as composition and reuse of existing (parts of) FCLs across multiple scenarios. Finally, the approach should be efficient in terms of performance, having small execution overhead.

To meet these requirements we propose 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 on which the FCLs are described, making them amenable to automated analysis and implementation code synthesis. Indeed, it allows FCLs structure and behavior to be separated from its implementation since it is captured at a conceptual level using the problem domain concepts, rather than the implementation concepts as is the case in framework-based solutions. Since FCLs are inherently concurrent and concurrent programming is known to be difficult [24], we choose to use an actor-oriented design [20] for our model. The FCL processes are represented as message-passing actors that encapsulate their state and behavior. It allows one to implement these processes without worrying about thread safety, which greatly simplifies code [24]. The actor model is also scalable [18], supports distribution computation, and is easily applicable as there exist several high-performing actor libraries[1].

4.2 Feedback Control Definition Language

Our approach is based on a domain-specific modeling language called *Feedback Control Definition Language* (FCDL). It is grounded on an actor-oriented component meta-model representing abstractions of FCL architectures. 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 feedback control loops.

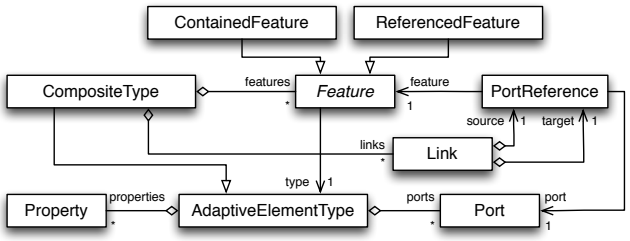

Figure 2: Excerpt of the FCDL abstract syntax

Figure 2 shown an excerpt[2] of the FCDL abstract syntax. AEs (`AdaptiveElementType`) have a well-defined interface that abstracts their internal state and behavior and restricts how they interact with their environments. It defines properties (`Property`) together with input and output ports (`Port`) that are the points of communications through which elements can exchange messages. The model supports both data-driven (push) and demand-driven (pull) communication. Once an AE receives a message, it activates and 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 active and so forth. An AE can be passive or active. The former is activated by receiving a message while the latter attaches an appropriate event listener to activate itself when an event of interest occurs. Each AE represents a process of a FCL, which may either be: *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 (`CompositeType`) from both basic adaptive elements and from other composites. A composite is also the primary unit

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

[2] The complete abstract syntax is available at the companion website `http://fikovnik.github.io/Actress/DADS14.html`

of deployment. It defines both the instances of other components (`Feature`) they contain and the connections between the instances ports (`Link`). It can also define ports which are used to promote ports of the contained features.

To enforce data type compatibility, the FCDL modeling language uses static typing. For each port and property one has to explicitly declare the data type that restricts the data values it accepts. To improve reusability, the meta-model additionally supports parametric polymorphism, making adaptive elements work uniformly on a range of data types.

4.3 Illustration

Figure 3 shows one possible FCDL implementation of the adaptation scenario. It is derived from the block diagram depicted in 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 that supports all necessary properties is presented in Section 5.1.

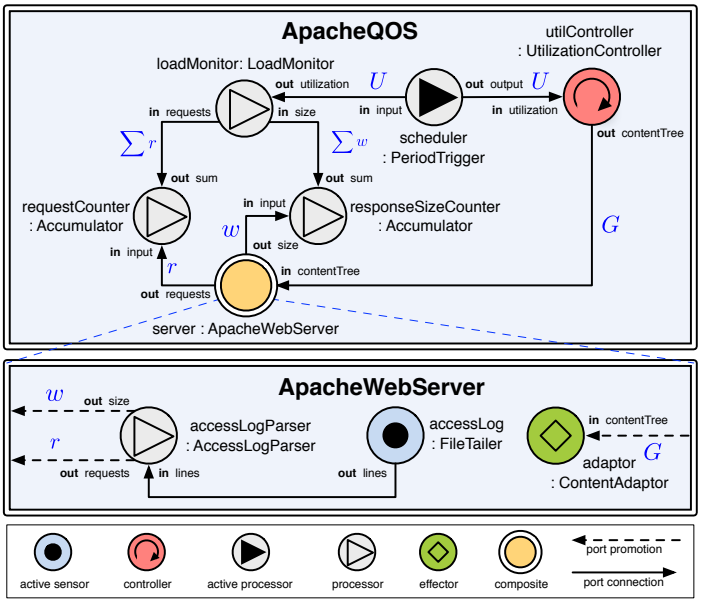

Figure 3: FCDL schema of the adaptation scenario

Decision-making. The PI controller (*Utilization Controller* from the block diagram) maps the current system utilization characteristics U into the abstract parameter G controlling which content tree should be used by the web server. In FCDL it is represented by the `UtilizationController` controller that has one push input port, `utilization`, for U and one push output port, `contentTree`, for G. Once a new utilization value is pushed to its input port, it computes G using (2) and pushes the result to the output port

Monitoring. The system utilization U depends on request rate R and bandwidth W. Both information can be obtained from Apache access log file. We create an active sensor, `FileTailer`, that activates every time a content of a file changes and sends the new lines over its push output port. It is connected to `AccessLogParser` that parses the incoming lines and computes the number of requests r and the size of the responses w, pushing the values to the corresponding `requests` and `size` ports. Consequently this increments the values of two connected counters `requestCounter` and `responseSizeCounter`, implemented as simple 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 (1) fetches the corresponding sums of requests $\sum r$ and response sizes $\sum w$ using the two pull input ports; (2) converts them to request rate R and bandwidth W; and (3) finally computes U using (1). The resulting utilization is then forwarded by the scheduler into the `UtilizationController`

Reconfiguration. Upon receiving the extent of adaptation G, the `ContentAdaptor` reconfigures 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 two composites `ApacheQOS` and `ApacheWebServer`, representing respectively the control part and the target system touchpoints.

4.4 Reflection

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 simple building blocks.

Figure 4 shows an example of an adaptive monitoring added into the adaptation scenario. Based on a periodically observed current system load using the `SystemLoad` sensor, the `PeriodController` modifies the execution timing of the `QOSControl` using the `setPeriod` effector. The `setPeriod` is a provided effector that adjusts the trigger rate t of the `PeriodicTrigger` inside the `QOSControl` composite.

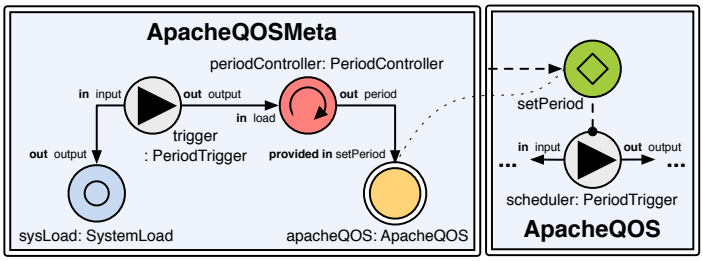

Figure 4: FCDL schema of the adaptation scenario with adaptive monitoring. *A provided sensor is visualized as an active sensor with one push output port, while provided effector is shown as a passive effector with one push input port. An additional dotted line indicates to which element do they belong.*

Technically, provided sensors and effectors are realized as AEs push output and push input ports respectively. However a crucial difference is that the messages sent from or to provided ports have a higher priority and thus will be processed before the regular messages. This is also reflected in the graphical notation (*cf.* Figure 4).

A structural adaptation, *i.e.*, changing loop composition and bindings is realized by sensors and effectors that operate on the actor model itself. These touchpoints include sensors observing adaptive

elements life-cycles (*e.g.* notifying when a new adaptive element is deployed), effectors deploying new elements, removing the existing ones or changing connections between them. By implementing the model reflection this way, we do not need any particular language support since these touchpoints are just regular AEs implemented using the underlying API. On the other hand, they have to be reimplemented for each targeted actor runtime support.

4.5 Distribution

Being based on the actor model, FCDL supports remoting using location transparency [2]. Remote elements are represented as first class entities using references. At the composite level, instead of declaring a new contained feature (`ContainedFeature`), one can declare a referenced feature (`ReferencedFeature`), formed by a reference to an existing feature in some composite, and a destination *endpoint* which is a URI of the remotely running AE. At runtime, during composite instantiation, for each referenced feature the system skips creating new AE and instead it only creates a reference that points to the given location. For example, in our adaptation scenario, we can deploy the `ApacheWebServer` composite on a different host than the `QOSControl`.

4.6 Execution Semantics

The execution semantics is based on the Ptolemy push-pull model of computation [39] coupled with an extended version of *Interaction Contracts* (IC) introduced by Cassou *et al.* [8]. The notion of IC is extended to support multiple-input, multiple-output elements, composites, optional contracts and architecture completion verification checking whether all required ports are connected [23].

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 same port-mode combination. Connecting a push output port to a pull input port indirectly implies using a queue and analogically connecting a pull output port to a push input requires to use a scheduler. In FCDL, this is intended to be explicitly modeled in the architecture in order to properly define the storage and the trigger mechanisms.

An active AE can cause its own activation from its associated event handler by sending a message to itself through an implicit self port. In FCDL, a message can only be sent by an AE. Therefore there always has to be at least one active element and for the model to be well-formed, each element has to become eventually active. An actor is eventually active if it is an active actor, or it has a pull output or push output port connected to an eventually activated element. The ordering of the activations is determined by the actor framework dispatcher.

An AE can execute different behaviors depending on what port or combination of ports caused its activation. For example, the `Accumulator` from the adaptation scenario either adds the pushed value from the `input` port or returns the accumulated sum when pulled over the `sum` port. To precise this, each non-composite AE specifies one or more basic IC that defines the element allowed interactions. It is a tuple $\langle A; R; E \rangle$ that indicates what interactions activates the AE (A), what additional data it might need to request through its pull input ports (R), and over which output ports it will push the results of its computation (E). 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 and conditionally data pushed to the `output` port. The IC for `Accumulator` is a composition of two basic interaction contracts $\langle \Uparrow (\texttt{input}); \emptyset; \emptyset \rangle \parallel \langle \Downarrow (\texttt{sum}); \emptyset; \emptyset \rangle$. Interaction contracts for composites are automatically inferred based on the IC of the contained AEs, *e.g.* `ApacheWebServer` has IC $\langle self; \emptyset; \Uparrow (\texttt{requests}, \texttt{size}) \rangle \parallel \langle \Uparrow (\texttt{contentTree}); \emptyset; \emptyset \rangle$.

The use of ICs brings following advantages. By using ICs we can assert certain architectural properties such as consistency, determinacy, and completeness. Different AE activations are clearly visible in its interface and therefore amenable to automatized analysis and verification. Furthermore, an IC denotes the type of the associate activation function. Therefore, it allows the generated code to be both *prescriptive* (guiding the developer) and *restrictive* (limiting the developer to what the architecture allows). For example, following is a Java code generated for the `PeriodicTrigger`:

```
public class PeriodicTrigger<T> extends AdaptiveElement {
  public void init();
  public void destroy();
  protected void activate(long self, Pull<T> input, Push<T> output);
  protected void onSetPeriod(Duration setPeriod);
}
```

Listing 1: Example of an AE class

The `Pull` and `Pull` interfaces denote the optional interaction for data requirements and data emission. The use of the generic parameter T is because `PeriodTrigger` is a polymorphic adaptive element capable of pulling and pushing any data type. The `init` and `destroy` methods are the AE life-cycle methods executed respectively during its initialization and termination.

5. ACTRESS

The aim of the ACTRESS modeling environment is to provide support for an integrated development of external self-adaptive software systems using FCDL. We do not focus on the control mechanisms themselves, since for this, there already exist sophisticated tools such as MATLAB [19].

In its core, ACTRESS consists of a series of model transformation and verification processes automatizing various aspects of FCDL development. This section gives a high-level overview of the main ACTRESS components. Additional details are available in a technical report [23].

5.1 Modeling Support

The ACTRESS modeling support provides a reference implementation of the FCDL meta-model and tools facilitating FCDL models authoring. The implementation is based on the EMF meta-modeling technology. The heart of the modeling support is a domain-specific language called *Extended Feedback Control Definition Language* (XFCDL) for creating FCDL models. It is a textual DSL for creating FCDL models that further supports modularization and AE implementation using a Java-like expression language. XFCDL is built using Xtext[3], a software language engineering framework that covers many aspects of a language infrastructure including sophisticated Eclipse IDE integration. 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. The following code shows an example of how to create the `PeriodicTrigger` from the running scenario[4]:

```
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    provided effector setPeriod: Duration
7    property initialPeriod: Duration = 10.seconds
8
9    act activate(selfport; input; output?)
10   act onSetPeriod(setPeriod; ;)
11 }
```

[3] http://www.eclipse.org/Xtext/

[4] The complete code is available at the companion website http://fikovnik.github.io/Actress/DAIS14.html

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. The provided effector is defined on line 6, followed by a property definition on line 7. Finally, lines 9-10 defines ICs.

Next, in order to form a FCL, we need to connect the AEs together. This is done by creating a composite in which we define all the elements of the loop and specify the data-flow by connecting their ports. For example, following is an excerpt of the `ApacheQOS` definition from Figure 4:

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

It is similar to an AE definition, but further it includes definitions of contained AEs (lines 4 and 7) port connections (line 11) and promotions (line 12). On line 4 a concrete data type is specified for the data type parameter T. Lines 5 and 8 specify values for the AEs properties including property reference.

Instead of creating a new adaptive element, it is possible to reference a remotely running one. For example, the following code creates an AE reference of the `ApacheWebServer` composite that runs at `remote-host`[5]:

```
feature server = ref ApacheWebServer @
  "akka://actress@remote-host/user/ApacheWebServer"
```

Finally, xFCDL also allows to specify the implementation of AEs (their ICs) directly using Xbase[6], a statically typed Java-like expression language that supports lambda expressions, type inference and Java interoperability. For example, the `UtilizationController` implementation using the equation (2) can be expressed as:

```
1  controller UtilizationController {
2    in push port utilization: double // U
3    out push port contentTree: double // G
4    property targetUtilization: double // U*
5
6    act activate(utilization; ; contentTree)
7
8    implementation xbase {
9      var G = M // new variable
10     // implementation of the 'act activate(utilization; ; contentTree)'
11     act activate {
12       val E = targetUtilization - utilization // computes the error
13       G = G + k * E // computes new extend of adaptation
14       if (G < 0) G = 0; if (G > M) G = M // correct bounds
15       G // returns the result
16     }
17 }
```

Next to ICs implementation, the Xbase block can contain variable declarations, life-cycle method implementations and auxiliary methods. While Xbase provides a convenient way of specifying adaptive elements implementation directly in xFCDL, it might not always be the most suitable option and a developer can use Java instead. Moreover, Xbase support for lambda expressions allows to use functions types as properties, which results in higher-order AEs definitions.

[5]The URIs are implementation dependent. Currently, ACTRESS uses Akka as the underlying actor runtime (*cf.* Section 5.2).

[6]`http://www.eclipse.org/Xtext/documentation.html\#xbaseLanguageRef_Introduction`

5.2 Code Generation and Runtime Support

Through text-to-model and model-to-model transformations the code in xFCDL is translated into FCDL. From the FCDL model, the code generator synthesizes an executable application for a concrete runtime platform. Currently, ACTRESS supports Akka[7], a scalable and lightweight framework and a runtime for actor-based applications on the *Java Virtual Machine* (JVM). Because the FCDL model is already an actor-oriented model, the source code transformation is rather straightforward as it does not need to build any other intermediate representation. Essentially, each AE type is turned into a Java class like the one shown in Listing 1. Any Xbase implementation is compiled into corresponding Java methods in the generated class. These classes are used as delegates by underlying actor classes that translate the lower level actor interactions into life-cycle and interaction contracts method calls. Using this pattern, developers never have to deal with any lower-level actor API and only use the higher-level API provided by ACTRESS. This also simplifies AE testing which can be done in isolation without any actor runtime. Additionally, the code generator outputs application launchers for top level composites providing a convenient way to execute them.

5.3 Verification Support

The verification support automates consistency checking of FCDL structural invariants including user-defined ones, as well as connectivity and data reachability properties through the means of external verification. Invariants are used in the FCDL meta-model for asserting the model well-formedness. Additionally, developers can define their own set of invariants for FCDL model instances using either OCL [27] or Xbase. Usually, they are used to identify architecture bad smells such as adaptive element overlaps (*e.g.* an effector being orchestrated by multiple controllers).

Furthermore, the use of models and MDE techniques brings the possibility of external model verification. Concretely, ACTRESS provides a FCDL transformation into Promela model in order to verify connectivity and reachability properties using linear temporal logic and the SPIN model checker [21].

6. ASSESSMENT AND DISCUSSION

In this section we discuss the application, quality attributes and limitations of both FCDL and the ACTRESS modeling environment.

Adaptation Scenario. The adaptation scenario illustrates the systematic integration of real-world control mechanisms into a real-world software system. The implementation consists of 169 xFCDL, 67 Xbase and 97 Java source lines of code (SLOC). Java was used to implement the Apache touchpoints while Xbase was used for all the other AEs. Interpreting SLOC is always problematic, however we advocate that (1) the 97 SLOC of the touchpoints code would have to be implemented in one way or another; (2) the 169 of xFCDL and 67 Xbase SLOC integrates the adaptation engine with the target system, creating an executable system; Moreover, the implementation already includes AEs that could be likely used in other adaptation scenarios since they provide some rather generic functionality (*e.g.*, `PeriodicTrigger`, `Accumulator`, `FileTailer`). Additionally two complete adaptation case studies from high throughput computing domain are available in a companion report [23].

Properties. Following is a qualitative summary of FCDL and ACTRESS support of the desirable properties identified in Section 4.1.

– *Generality.* FCDL is a domain-agnostic model language for modeling architectures of FCLs. It uses concepts from control theory and its syntax is close to the block diagram one. Unlike

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

most frameworks [29], it does not dictate any particular system architecture. Since an FCL is decomposed into a number of explicit and interconnected adaptive elements, a number of *self-** adaptation properties are likely to be expressed. Furthermore, the reflection and distribution capabilities support the organization of FCLs into complex and distributed control schemes, such as hierarchical or decentralized controls [28].

FCDL is also a technologically-agnostic model. It focuses only on the FCL architectures, hiding the details not relevant to the design. ACTRESS is based on Java technologies, however, by no means it is limited to only adapt Java systems as shown in Section 3. FCDL can also target other runtime platforms. For example, the CORONA project [26] uses FCDL for *Service Component Architecture* systems adaptation, transforming AE elements into components for the FraSCAti runtime [32].

– *Visibility.* The FCDL language is based on an actor-oriented model with known concepts such as ports and composites. The FCL processes are represented as first-class reusable entities with explicit interactions that are precisely guided by interaction contracts. Relying on the actor model, the system is highly concurrent while allowing for simple AE implementation without the need to protect mutable state. Moreover, interaction contracts make the architecture both prescriptive and restrictive, and thus guide AE implementations.

  Using FCDL developers work on a higher-level of abstraction using concepts from the self-adaptive system domain. Without a domain-specific modeling language like FCDL, developers are likely to use GPLs that do not convey domain-specific concerns and semantics [14]. 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. FCDL matches block diagrams providing an established abstraction of FCLs which is flexible, yet rigid enough for automated code synthesis.

– *Tooling.* The usage of FCDL is facilitated by the ACTRESS modeling environment. Integrated in the Eclipse IDE, it provides modeling, code generation and verification support. The modeling support uses a textual DSL, XFCDL, that enables modularization and optional AE implementations using Xbase expressions. The code generator transforms FCDL architectures into executable Java applications, providing a strong mapping between the control system design and its runtime implementation. The verifier can automatically check assumptions about modeled architectures using structural and temporal constraints. Using Xbase for implementation, the code generator emits a complete executable applications, yet with customization and configuration opportunities. During the implementation of the case studies, we observed, that the automation of the development process helps developing the solution incrementally. It allowed to start with a basic control scheme and to refine it step-by-step into a more advanced one. At the end of each step ACTRESS generates implementation code that can be tested and executed. Finally, our approach supports separation of concerns in the sense that the system architecture and control mechanisms can be defined by control engineers while the implementation of the technical/system-level processors or touchpoints can be carried out by software engineers. Thanks to the Eclipse integration both tasks can be realized within the ACTRESS modeling environment, which should simplify and promote collaboration.

Performance. We consider the overhead caused by the execution of the self-adaptive layer. A single instance of the ACTRESS runtime with no composites deployed accounts for 1.5MB[8]. The ACTRESS domain framework is based on Akka. 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[9]. The size of an adaptive element 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 adaptive element activation methods (*e.g.* a sample push/pull communication with a throughput of 5000 messages per second amounts for 5% of CPU time). 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.

Limitations. While FCDL is technologically agnostic, XFCDL is tightly coupled with Java. This currently limits the implementation of AE to Java-based languages. This might pose a problem for scenarios where the touchpoints need to interact with an API that is not accessible from Java nor JNI. 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.

Besides the FCDL uses static typing, but does not support physical units and therefore there is nothing to prevent typing errors such as `speed = time / distance`.

Xbase provides a convenient way for expressing mathematical equations, but it might be too low level for control based on concepts such as decision tables, rule-based policies or state transition diagrams. Declarative policy-rule languages can be used through their respective API, however, as in the case of StarMX or ASF, they are not directly embedded in the adaptive element definition (*i.e.* in XFCDL). Furthermore, the external adaptation relies on the fact that the target system is able to provide, or be instrumented to provide, all the required touchpoints.

7. CONCLUSIONS

In this paper, we have proposed a domain-specific modeling language, FCDL, for integrating adaptation mechanisms into software systems through external FCL. It is centered around an actor-oriented model for defining FCL architectures using a hierarchically organized networks of AEs that explicitly represent different parts of the adaptation process as first-class entities. To facilitate the development using FCDL, a modeling environment called ACTRESS has been implemented. Integrated in the Eclipse IDE, it provides a reference implementation of FCDL together with dedicated support for modeling, verification and complete code generation. The approach has been illustrated on a real-world adaptation scenario on web server QoS management control.

Current work in progress mainly concerns carrying more case studies targeting different self-adaptive properties in order to identify the strengths as well as limitations of the approach. Several improvements are planned for the future such as support for deployment in distributed environments, dealing with issues related to loop coordination, failure propagation and extending data type system with physical units. Future work also include providing a native implementation of the ACTRESS runtime and experiment with DSL embedding to allow to specify AE implementations in a variety of languages.

Acknowledgments. The work reported in this paper is partly funded by the ANR SALTY project under contract ANR-09-SEGI-012.

8. REFERENCES

[1] T. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In *International Workshop Quality of Service*, IWQoS, London, 1999.

[8] All further measurements were conducted on MacBook Pro 2.53 Ghz Intel i5, 8GB RAM, Java 1.70_17, Akka 2.2.0

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

[2] J. L. Armstrong, B. O. Dacker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real time applications. In *Software Engineering for Telecommunication Systems and Services*, 1992.

[3] 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*, 2009.

[4] O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen. The Self-Star Vision, volume 3460 of LNCS, 2005.

[5] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *International conference on Software engineering*, 2008.

[6] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel. DiaSuite: A tool suite to develop Sense/Compute/Control applications. *Science of Computer Programming*, pages 1–28, Apr. 2012.

[7] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.

[8] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. *Proceeding of the 33rd international conference on Software engineering*, 2011.

[9] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and Others. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.

[10] J. Dowling and V. Cahill. Self-managed decentralised systems using K-components and collaborative reinforcement learning. *ACM SIGSOFT workshop on Self-managed systems*, 2004.

[11] J. Eker, J. Janneck, E. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[12] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, Mar. 2006.

[13] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *MoDELS*, 2012.

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

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

[16] D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. In *International Conference on Autonomic Computing*, 2004.

[17] I. Gorton, Y. Liu, and N. Trivedi. An extensible, lightweight architecture for adaptive J2EE applications. *International Workshop on Software engineering and middleware*, 2006.

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

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

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

[22] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan. 2003.

[23] F. Křikava and P. Collet. Feedback Control Definition Language. Technical report, I3S CNRS - UMR 7271, 2013, `https://fikovnik.github.io/Actress/FCDL.pdf`

[24] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[25] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Int. Conf. on Architecture of Computing Systems*, 2005.

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

[27] Object Management Group. OMG Object Constraint Language (OCL). Technical report, 2012.

[28] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems*, 2012.

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

[30] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In *1st workshop on Mobile middleware*, 2008.

[31] S. Sadjadi and P. McKinley. ACT: an adaptive CORBA template to support unanticipated adaptation. In *Int. Conf. on Distributed Computing Systems*, 2004.

[32] L. Seinturier, P. Merle, D. Furnier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *International Conference on Service Computing*, 2009.

[33] 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.

[34] N. Villegas, G. Tamura, H. Müller, L. Duchien, and R. Casallas. DYNAMICO : A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. *Software Engineering for Self-adaptive Systems 2*, pages 265–293, 2013.

[35] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *Software Engineering for Adaptive and Self-Managing Systems*, 2011

[36] 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*, 2012.

[37] D. Weyns and S. Malek. FORMS: a formal reference model for self-adaptation. In *Proceedings of the 2010 International Conference on Autonomic Computing*, 2010.

[38] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering*, 2006.

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