Contracts-based Control Integration into Software Systems

Filip Křikava[1], Philippe Collet[3], Romain Rouvoy[2], and Lionel Seinturier[2]

[1] Czech Technical University
filip.krikava@fit.cvut.cz
[2] University of Lille / Inria
{romain.rouvoy,lionel.seinturier}@inria.fr
[3] Université Nice Sophia Antipolis
philippe.collet@unice.fr

Abstract. Among the different techniques that are used to design self-adaptive software systems, control theory allows one to design an adaptation policy whose properties, such as stability and accuracy, can be formally guaranteed under certain assumptions. However, in the case of software systems, the integration of these controllers to build complete feedback control loops remains manual. More importantly, it requires an extensive handcrafting of non-trivial implementation code. This may lead to inconsistencies and instabilities as no systematic and automated assurance can be obtained on the fact that the initial assumptions for the designed controller still hold in the resulting system.
In this chapter, we rely on the principles of design-by-contract to ensure the correction and robustness of a self-adaptive software system built using feedback control loops. Our solution raises the level of abstraction upon which the loops are specified by allowing one to define and automatically verify system-level properties organized in contracts. They cover behavioral, structural and temporal architectural constraints as well as explicit interaction. These contracts are complemented by a first-class support for systematic fault handling. As a result, assumptions about the system operation conditions become more explicit and verifiable in a systematic way.

1 Introduction

Self-Adaptive Software Systems (SASS) are characterized by the ability to continuously operate under varying runtime conditions [11]. They autonomously adapt themselves based on the perception of both their own state and the state of their environment. The heart of this adaptation capability can be based on the notion of a *Feedback Control Loop* (FCL) that regulates the characteristics of the system to achieve its goals despite changes that may occur during operation [20]. In particular, it uses measurements of system outputs (*e.g.* response time) to automatically adjusts system control inputs (*e.g.* level of concurrency) based on a given control strategy.

There are different techniques that can be used to design SASS. Among them, control theory offers a promising solution by providing a well-established mathematical foundations for designing controllers. It has been extensively used in other engineering disciplines for controlling behavior of industrial plants. It allows one to develop a control strategy whose properties, such as stability and accuracy, can be formally guaranteed under certain *assumptions on the operating conditions* [17] such as input data bounds or timing properties.

However, there are important difficulties in systematically applying control techniques into software systems [17]. First, it is quite hard for non-experts to develop mathematical models of software behavior that are actually usable for a control design. Next, the design, implementation and integration of the controller into a complete self-adaptive software system are activities known to be both error-prone and difficult [21]. Despite support provided by tools, such as MATLAB, SIMULINK, or SYSWEAVER [32], that provide code generation capabilities, the controller integration still requires an extensive handcrafting of non-trivial code, in particular in the case of distributed systems. Consequently, a FCL is often tangled with the source of the target application [1,2,27] or composed of several *ad hoc* scripts [17,5]. As a result, this may lead to inconsistencies and instabilities in the resulting SASS. This is essentially due to the fact that the initial assumptions on the operation conditions for the designed controller are usually not explicitly specified and that no systematic verification is conducted to ensure they still hold at runtime.

In our previous work [25], we have proposed a domain-specific modeling language, called *Feedback Control Definition Language* (FCDL), that is addressing some of the integration challenges related to the visibility of FCLs. It provides system-level abstractions for integrating external control mechanisms into existing software systems, notably through an underlying actor-oriented component model. In this chapter, we go beyond this contribution by focusing on the reliability aspect of the integration activity. In particular, we present an extension of the FCDL language to support a *design-by-contract* methodology [29]. Design-by-contract is a pragmatic and a lightweight method for embedding elements of formal specification into software elements (*e.g.*, objects, components, services). The approach strongly focuses on the correctness of the contracted system and thus contributes to improve the overall system reliability assurance [26]. Concretely, we provide a first-class language support for defining:

- *behavioral contracts* that assert component behavior through state invariants and pre and postconditions,
- *interaction contracts* that express allowed component interactions, and
- *structural and temporal invariants* that define architecture constraints as well as design and execution time interaction invariants.

Next to contracts, we also equip FCDL with first-class support for systematic fault handling. These constructs can be further used to respond to runtime deviations from expected quality of service (*e.g.* response time violation in the sense of a timeout).

Including the support directly into the FCDL language offers several advantages. With reasonable development effort, contracts make the system specification more rigorous and therefore improve its verification capabilities. Static structural and interaction invariants can be checked at design time, allowing developers to verify system consistency and spot design flaws early. The fault handling strategies enable developers to define coherent responses to runtime contract violations, contributing to the construction and maintenance of more fault tolerant systems.

Applied to the engineering of self-adaptive software systems, contracts make the assumptions about the system operation conditions more explicit, and verifiable in a systematic way to handle system failures. Separation of concerns is also promoted as the controller design and integration can be decomposed and implemented by experts in their respective domains and at different levels of abstraction, still under an explicit specification.

The chapter is structured as follows. Section 2 introduces the adaptation scenario we use to illustrate our approach. Section 3 gives an overview of the FCDL language. Section 4 presents the different contract extensions for FCDL while the description of the failure handling support is provided in Section 5. The trade-offs of using contracts are discussed in Section 6. Section 7 surveys related work and finally, Section 8 concludes the chapter.

2 Adaptation Scenario

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

The aim of the adaptation is to maintain a web server load at a given 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

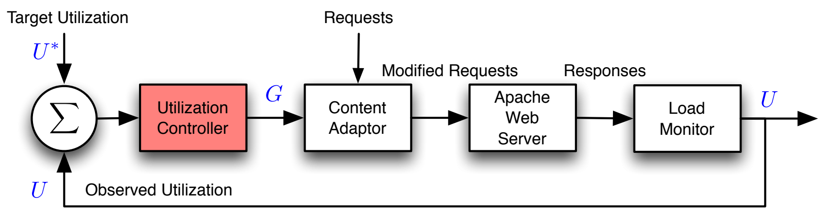

Fig. 1. Block diagram of the adaptation scenario [1]

overhead and a data-size dependent overhead, using some 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 some platform constants derived by server profiling (details in Abdelzaher *et al.* [1]), $\sum r$ and $\sum w$ are the number of requests and the amount of bytes sent over some period of time t, respectively. The *Utilization Controller* is an *integral* (I) 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:

$$G = G + kE = G + k(U^* - U) \tag{2}$$

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.

Next to the integral control, Abdelzaher *et al.* [1,2] also propose a more sophisticated *proportional integral* (*PI*) controller. For the simplification, in this chapter we only consider the former one, since from the software architecture perspective the only difference between them is the type of AE that is instantiated. The FCDL primary focus is facilitating controller integration not its design, since for this, there already exist sophisticated tools such as MATLAB [20].

This adaptation scenario is essentially a simplified version of the *znn.com case study* [12] in which only the server content fidelity is considered. While in this chapter we are mostly focus on the control theory based solution, there exist other approaches as well. In the related work section, we give an example how FCDL can be used for their integration.

3 Feedback Control Definition Language in a Nutshell

In this section we present the essentials of the FCDL language and illustrate how it can be used to integrate the adaptation scenario from the previous section. We deliberately skip some technical details about the language and instead provide pointers where details can be found. The complete FCDL syntax and semantics can be found in Křikava's PhD thesis [24] as well as on the FCDL web site[4].

3.1 Modeling Feedback Control Loops

FCDL is a *Domain-Specific Modeling Language* (DSML) tailored for defining feedback control architectures of external self-adaptive software systems—*i.e.*, systems where the adaptation engine is isolated from the target (adaptable) system and interact through identified touchpoints. It is based on an actor-oriented model [22] where the components (actors) are called *Adaptive Elements* (AE). They represent the different FCL processes, such as monitoring, decision making and reconfiguration and feedback control loops are formed by connecting them together.

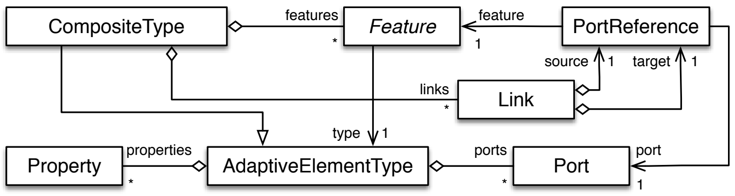

Fig. 2. Excerpt of the FCDL abstract syntax (type meta-model)

Figure 2 shows an excerpt of the FCDL abstract syntax. An AE (`AdaptiveElementType`) has a well-defined interface that abstracts its internal state and behavior. It consists of properties (`Property`), ports (`Port`) and operations[5]. Properties define the AE initial configuration as well as its runtime state variables. The input and output ports are the points of communications through which elements can exchange messages. An operation is a function *input* $\times$ *state* $\rightarrow$ *output* executed upon receiving a message. The concrete function code can be expressed for example in Java.

The execution semantics is based on the Ptolemy [16] push-pull model of computation [45]. 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 on and so forth. An AE can be *passive* or *active*. The former is activated by

[4] `http://fikovnik.github.io/Actress`
[5] Not shown in the excerpt, details are in Chapter 3 of Křikava's PhD thesis [24].

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 (*e.g.* log files, hardware sensors),
- *an effector* carrying out changes on the target system using provided management operations (*e.g.* configuration file modification, system calls),
- *a processor* processing and analyzing incoming data both in the monitoring and reconfiguration parts (*e.g.* data filters, converters), or
- *a controller*, a special case of a passive processor that is directly responsible for the decision making (*e.g.* PI controller, rule engine, utility controller).

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.

FCDL also supports constructing *composite* components (`CompositeType`) which is also the primary unit of deployment. It defines both the instances of other components (`Feature`) they contain and the connections between the instances ports (`Link`).

Additionally, the language supports distribution and reflection[6], thereby enabling coordination and composition of multiple distributed FCLs [24, Sections 3.3.5 and 3.3.6].

3.2 Illustration

Figure 3 shows how the control mechanism elaborated in the previous section can be integrated in the target system (Apache web server) in FCDL. The figure uses an informal FCDL graphical notation, an informal visual representation of FCDL models. A formal textual syntax is presented in Section 3.3 with an example corresponding to this illustration in Listing 1.

The feedback control loop consists of three composites. The `ApacheWebServer` that collects elements related to the Apache web server, `LoadMonitor` that is responsible for computing the server utilization U and `ApacheQOS` that embodies them and assembles the FCL. It works as follows:

- *Decision making.* The controller 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 `utilController` AE that defines one push input port, `input`, for U and one push output port, `output`, 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. The `IController`

[6] Conceptually, each AE can be seen as a target system itself, and as such it can provide sensors and effectors enabling the AE reflection — hence the name *adaptive element*.

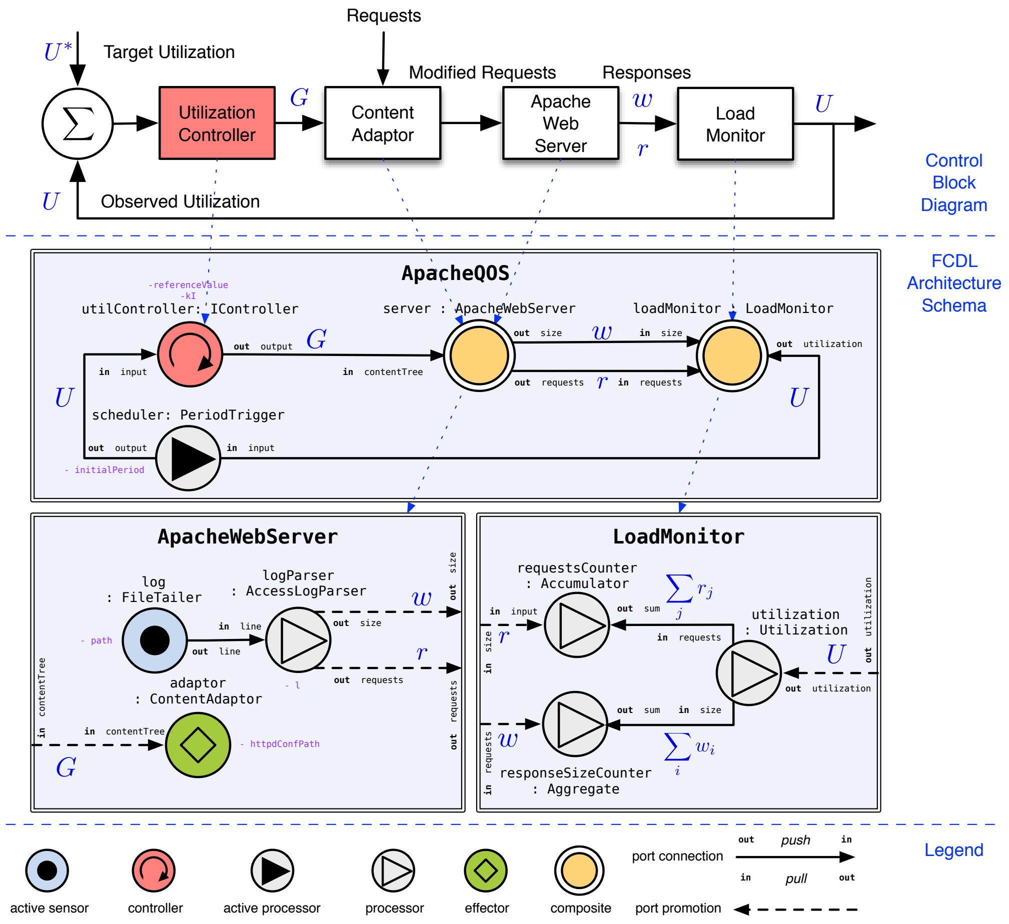

Fig. 3. FCDL schema of the adaptation scenario.

also includes properties for the controller set point `referenceValue` and the K_I controller integral gain `kI`.

– *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 `logParser` 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 requests and bytes sent over a certain time period t. One way of doing this is by adding a `scheduler`, an active processor that every t milliseconds pulls data from its pull input port (the pull is indicated by the arrow going from the `scheduler` input port to the `utilization` output 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 `utilization` processor that (i) fetches the corresponding sums of requests $\sum r$ and response sizes $\sum w$ using the two pull input ports; (ii) converts them to request rate R and bandwidth W; and (iii) finally computes U using (1). The resulting utilization is then forwarded by the scheduler into the `utilController`.

– *Reconfiguration.* Upon receiving the extent of adaptation G, the `Content-Adaptor` reconfigures the web server URL rewrite rules so that the newly computed content tree is used to serve the upcoming requests.

3.3 Modeling Support

The idea behind a DSML, such as FCDL, is to raise the level of abstraction, which should lower accidental complexities and in turn increase productivity. For this to be true, however, DSMLs have to be associated with software development tools that automate tasks such as model construction and code generation [42].

The model creation is facilitated by a textual *domain-specific language* (DSL). It is built using the Xtext[7] software language engineering framework. The Listing 1 shows a code excerpt of the adaptation scenario (cf. Figure 3).

FCDL model is used as an input to a model transformation that synthesizes an executable system based on Akka[8], an actor-based framework and runtime for the Java virtual machine. Because the FCDL model is already an actor-oriented model, the source code transformation is rather straightforward. Essentially, each AE type is turned into a Java class with generated skeleton methods left for developers to complete with the logic (*e.g.* parsing access log records in the `ApacheLogParser`). These classes are used as delegates translating the lower level actor interactions (*e.g.* messages initializing actors) into corresponding method calls (*e.g.* initialization or activation method). Using this pattern, developers never have to deal with any lower-level actor API making it portable across different actor frameworks. This also simplifies AE testing, which can be done in isolation without any actor runtime.

4 Adaptive Element Contracts

We now present a set of extensions to FCDL for adaptive element contracts specification. The aim is to make the assumptions about the operation conditions

[7] `http://eclipse.org/Xtext`
[8] `http://akka.io`

```
active processor PeriodicTrigger<T> { // polymorphic processor with T parameter
 push in port output: T
 pull in port input: T
 self port selfport: Long // active AE contains a self port for self-activation
 property initialPeriod: Long
 // ...
}

controller IController {
 push in port input: Double
 push out port output: Double
 // ...
}

composite ApacheQOS {
 // instantate a contained AE with a concrete data type for the T parameter
 feature scheduler = new PeriodicTrigger<Double> {
  initialPeriod = 5.seconds // set properties
 }
 feature utilController = new IController { /* ... */ }
 // port connection
 connect scheduler.output to utilController.input
 // ...
}
```

Listing 1. Adaptation scenatio FCDL code excerpt

explicit. In FCDL, this can be realized at the architecture level by specifying invariants that constrain adaptive element structure, interactions, and behavior.

We start by introducing *Interaction Contracts* (IC) as they define the semantics of an adaptive element (AE) execution. Next, we extend ICs with *Behavioral Contracts* (BC) that assert AE behavior. Finally, we complete the section with adaptive element structural and temporal invariants.

In this section, we only focus on the contract definition—*i.e.*, on the definition of the *expected* behavior. The next section will discuss when contract violation and will detail the mechanisms to handle *exceptional cases*.

4.1 Interaction Contracts

Motivation Let us consider a more sophisticated version[9] of the `Accumulator` from our adaptation scenario (*cf.* Figure 4). It works as follows: When it receives data on its `input` port, it pushes to its `output` port the received value plus the sum of all the input values it has received since the last time the `reset` port was triggered. Similarly, when pulled on the `sum` port, it returns the sum of all the input values since the last reset. Finally, when any data are pushed to its `reset` port, it sets the accumulated value back to 0.

The above description makes the element interactions rather intuitive. However, the fact that every time an input is received data will be pushed over the output port is not explicitly stated in the architecture. It is but only mentioned in its documentation. Such architecture underspecification leads to several shortcomings:

[9] Inspired by the Ptolemy 2 Accumulator actor cf. `http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.1/ptII/doc/codeDoc/ptolemy/actor/lib/Accumulator.html`

```
processor Accumulator {
   in push port input: Long
   in push port reset: Boolean
   out pull port sum: Long
   out push port output: Long
}
```

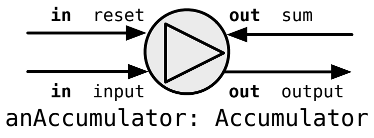

Fig. 4. Improved `Accumulator` processor

1. There is no systematic way to verify that an implementation matches its documentation.
2. In more complex cases or with less rigid documentation, it may result in different interpretations and thus incompatible implementations.
3. The implementation has to manually follow the data flow to check which AE *functionality* should be executed, which is tedious in cases when multiple inputs have to be synchronized.
4. The element interactions are an integral part of its implementation, strongly reducing the possibility of formal analysis.

To address this architecture underspecification issue, we enrich the adaptive element definition with *interaction contracts* that express allowed interactions, making them explicit.

In the following we give an overview of the interaction contracts that have been implemented in the FCDL language based on the work of Cassou *et al.* [9] (cf. Section 7). The complete formal specification of interaction contracts is available in Křikava's PhD thesis [24, Section 4.2 from page 74].

Contract Specification The objective of an interaction contract is to describe the interactions allowed by an adaptive element. More formally, we define *a basic interaction contract* as a tuple $\langle A; R; E\rangle$ where

- A is the activation condition that indicates what interactions activate the AE—*i.e.*, a set of push ($\Uparrow$) input ports or a pull ($\Downarrow$) output port;
- R is the data requirements denoting what additional data might be needed during its execution—*i.e.*, a set of pull input ports — and
- E is the data emission—*i.e.*, a set of push output ports through which the results of computation will be distributed.

For example, the interaction contract associated with `PeriodicTrigger` is $\langle self; \Downarrow(\texttt{input}); \Uparrow(\texttt{output}?)\rangle$ denoting an interaction caused by *self* activation (as it is an active processor) where `input` port might be pulled and conditionally (?) data pushed to the `output` port. In FCDL this corresponds to the following definition:

```
active processor PeriodicTrigger<T> {
   push out port output: T
   pull in port input: T
   port selfport: Long

   act activate(selfport; input; output?) // ⟨self;⇓(input);⇑(output?)⟩
}
```

For adaptive elements with multiple operations, an interaction contract is a composition ($\|$) of all the individual ICs. For example, the IC of the `Accumulator` described above is a composition of three basic ICs $\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output})\rangle \parallel \langle \Downarrow (\texttt{sum}); \emptyset; \emptyset\rangle \parallel \langle \Uparrow (\texttt{reset}); \emptyset; \emptyset\rangle$ which in FCDL corresponds to:

```
processor Accumulator {
   in push port input: Long
   in push port reset: Object // any data pushed will reset the counter
   out pull port sum: Long
   out push port output: Long

   act onInput(input; ; output) // ⟨⇑ (input); ∅; ⇑ (output)⟩
   act onSum(sum; ;) // ⟨⇓ (sum); ∅; ∅⟩
   act onReset(reset; ;) // ⟨⇑ (reset); ∅; ∅⟩

   // ...
}
```

In the case of composites, an IC is automatically inferred [24, Section 4.2.4 on page 84] based on the ICs of the contained AEs. For example, the `ApacheWeb-Server` IC is specified as $\langle self; \emptyset; \Uparrow (\texttt{requests}, \texttt{size})\rangle \parallel \langle \Uparrow (\texttt{contentTree}); \emptyset; \emptyset\rangle$.

Denotation A denotation of an IC $\langle A; R; E\rangle$ is a function $A \times R \to E$. Each of the contracts can then be used to synthesize an AE activation method. For instance, the following listing shows the generated Java interface that corresponds to the `Accumulator`:

```
public interface Accumulator {
   Long onInput(Long input); // ⟨⇑ (input); ∅; ⇑ (output)⟩
   Long onSum(); // ⟨⇓ (sum); ∅; ∅⟩
   void onReset(Object reset); // ⟨⇑ (reset); ∅; ∅⟩
}
```

Since the information about AE activation are explicitly stated, the low-level details about input data synchronization can be automatically generated. Developers therefore only need to focus on the actual functionality. The synthesized methods further help the implementation in the way that they are both *prescriptive*—*i.e.* guiding the developer — and *restrictive*—*i.e.* limiting the developer to what the architecture allows — in contrast to having only one monolithic activation method.

Input Synchronization and Disjunction Interaction contracts support complex activation patterns, such as input synchronization and input disjunction. Let us consider a passive adaptive element $\mathcal{A}$ with two input push ports $(\texttt{in}_1, \texttt{in}_2)$ and one push output port $(\texttt{out})$. An interaction contract $\langle \Uparrow (\texttt{in}_1, \texttt{in}_2); \emptyset; \Uparrow (\texttt{out})\rangle$ synchronizes the input ports and will only activate $\mathcal{A}$ if data have been pushed to both input ports. This corresponds to a Java method:

```
   <T> T onIn1In2(T in1, T in2); // ⟨⇑ (in₁, in₂); ∅; ⇑ (out)⟩
```

On the other hand a input disjunction ($\vee$) used in the activation condition of an IC such as $\langle \Uparrow (\texttt{in}_1 \vee \texttt{in}_2); \emptyset; \Uparrow (\texttt{out})\rangle$ will activate $\mathcal{A}$ any time any of the input ports receives data. In Java:

```
   <T> T onIn1In2(T in); // ⟨⇑ (in₁ ∨ in₂); ∅; ⇑ (out)⟩
```

Finally, an IC $\langle \Uparrow (\mathtt{in}_1); \emptyset; \Uparrow (\mathtt{out})\rangle \parallel \langle \Uparrow (\mathtt{in}_2); \emptyset; \Uparrow (\mathtt{out})\rangle$ will also activate $\mathcal{A}$ any time there has been a data pushed on any of its input ports. However, the crucial difference here is that, in this latter case, there is a different behavior associated with each of the basic contracts corresponding to two different activation methods:

```
<T> T onIn1(T in1); // ⟨⇑ (in₁); ∅; ⇑ (out)⟩
<T> T onIn2(T in2); // ⟨⇑ (in₂); ∅; ⇑ (out)⟩
```

Architecture Properties Interaction contracts make the component interactions explicit. The resulting architecture is therefore amenable to automated analysis and the following properties can be statically checked at design time [24, Sections 4.2.5-4.2.7].

- *Consistency.* ICs not only define the elements interactions but also imply certain interactions requirements for the other elements in order for the assembly to be consistent. For example, for the `scheduler` (cf. Figure 3) to be able to pull data from its input port, it must define a contract with the following data requirement: $R = \Downarrow (\mathtt{input})$. This implies that the connected element, `loadMonitor`, must have in one of its ICs and activation condition $A = \Downarrow (\mathtt{output})$ since `output` port is connected to the `PeriodicTrigger` `input` port.
- *Determinacy.* An interaction contract can be composed of one or more basic interaction contracts defining multiple activation conditions for an AE. It is important to make sure that these activation conditions do not interfere with each other. An interference occurs for example if two or more interaction contracts share the same push input port in their activation condition. In such a case, the activation is not *deterministic* since it is not possible to identify which contract should be executed when the input data arrives.
- *Completeness.* An adaptive element might define a multitude of interaction contracts from which some are required and some are optional. For example, in the case of the `Accumulator`, the interaction contract linked to the reset functionality is optional while the other two are required. It is therefore not possible to use the element without connecting its `input` and `sum` ports while the `reset` might be left out.

4.2 Behavioral Contracts

Motivation The interaction contracts precisely specify which adaptive element behavior is triggered by what interaction. However, they say very little about the behavior itself. This might be problematic from at least two reasons. First, from the correctness point of view, less constrained implementation may lead to a higher chance of containing bugs than an implementation satisfying well-understood specifications [29]. Second, from the robustness perspective, Cámara *et al.* [8] showed that the unconstrained inputs and outputs may lead to salient failures, which are both hard to detect and may cause unexpected or undesired behavior. In the case of self-adaptive software systems, this is particularly worrying as the controller may steer the system into a wrong state without any obvious reason.

A systematic approach to address both concerns is to use behavioral contracts—*i.e.* pre and postconditions and state invariants. They have been successfully used in other programming languages, such as Eiffel[10], OCL [33] or Scala [34]. In our case, these contracts will allow developers to augment the type specification and express AE properties and behavioral requirements.

State Invariants A state invariant asserts the values of AE properties since that is the only component of the adaptive element definition that is directly modifiable by users (during AE instantiation in a composite). It is specified as a boolean expression using a `state inv` construct.

Let us consider the `AccessLogParser` from the adaptation scenario (cf. Figure 3), which is responsible for parsing the Apache access log file. To accommodate for different log formats, we define a `logFormat` property:

```
property logFormat: String
```

However, we need to make sure that it is always set and that the value is a valid regular expression. In FCDL, these concerns are expressed using state invariants. First, we define an invariant that makes sure the property is set to a non-empty string:

```
state inv NonEmptyLogFormat = self.logFormat != null && self.logFormat.length > 0
```

Second, we ensure a valid regular expression:

```
 1 state inv ValidLogFormat if NonEmptyLogFormat = new StateInvariant {
 2    override check() {
 3       try {
 4          java.util.regex.Pattern::compile(logFormat)
 5          pass() // an invariant is satisfied
 6       } catch (java.util.regex.PattermSyntaxException e) {
 7          fail("Invalid pattern: "+e.message) // invariant is violated
 8       }
 9    }
10 }
```

The second invariant shows an alternative implementation that uses an anonymous class implementing a designated interface. It also shows some additional features that are supported by the FCDL invariants. On line 1 we define a dependency between invariants—*i.e.*, the check will only be evaluated if `NonEmptyLogFormat` has not been violated (that is why we can skip a nullity check for `logFormat`). On line 7 we further provide a user-friendly error message with more details about what went wrong.

Assertion Language The code listing above reports examples of typical assertions used in BCs. Assertions are boolean expressions that come from the Xbase language [15]. Xbase is a Java-like expression language especially designed to be embedded in DSLs that are created using the Xtext language engineering workbench. It is interoperable with Java and the expressions can instantiate Java

[10] First appeared in the Eiffel language under the name *Design-by-contract* [30].

classes, implement Java interfaces and call Java methods. The language is statically typed and it uses type inference to provide type safety without unnecessary syntactic clutter. It includes support for first-order logic collection operations, which makes assertions, such as $\forall x \in T.p(x)$ or $\exists x \in T.p(x)$, convenient to define using expressions like `T.forall[x | p(x)]` and `T.exists[x | p(x)]`.

Many of the assertions are usually simple expressions for which Xbase provides a suitable option. However, it might not always be the case and complex assertions can be equally implemented in Java. A developer can either instantiate an existing class that conforms to the right interface or omit the expression in which case a skeleton Java class will be generated instead. This gives a possibility to reuse state invariants across adaptive elements.

Next to the BCs, Xbase can also be used to directly implement AE operations in FCDL.

Pre/Postconditions Pre and postconditions are related to the executions of AEs operations. They are specified as boolean expressions using the **`require`** and **`ensure`** keywords.

Let us consider again the `AccessLogParser`. To prevent potential salient failures, we should check that the received input line matches the specified access log format. In FCDL, we can express it using the following precondition:

```
act activate(line; ; requests, size) { // ⟨⇑(line); ∅; ⇑(requests, size)⟩
   require LineMatchingPattern if LineNotEmpty = new PreCondition {
      val Pattern = java.util.regex.Pattern::compile(self.logFormat)
      override check() {
         val m = Pattern.matcher(line)
         assertTrue("The input line must match the log format", m.matches())
      }
   }
}
```

The structure is similar to the structural invariant definition. The main difference is that, in its scope, it can additionally access input data (`line`), which is injected automatically into the class definition. Like Java anonymous classes, it also allows one to declare variables and constants (`Pattern`). We can therefore express more complex invariants—*e.g.* to ensure that the log entries are successive:

```
require SuccessiveTimestamp if ValidTimestamp = new PreCondition {
   val Pattern = java.util.regex.Pattern::compile(self.logFormat)
   val TimestampFormat = new SimpleDateFormat("dd/MMM/YYYY:HH:mm:ss Z")
   var lastTime = new Date(0)

   override check() {
      val timestamp = Pattern.matcher(line).group("time")
      val time = TimestampFormat.parse(timestamp)

      if (time.after(lastTime)) {
         lastTime = time
         pass()
      } else {
         fail("Invalid record: "+time+" appears before the last record")
      }
   }
}
```

It is important to note that it makes perfect sense to express this as a contract instead of having the same check within the operation method of the adaptive element. The reason is that non-successive timestamps present a deviation from the expected behavior, concretely a bug in the connected component that supplies the invalid log entries (`log` in this case), and not a special case that should be handled in the operation body. Following this approach we clearly express the assumptions on the adaptive element inputs.

4.3 Interaction Invariants

Motivation So far, we have presented contracts that consider adaptive elements in isolation. While it is important to express assumptions on the interactions and behavioral properties of individual components, these contracts do not provide any assumptions about the complete loop architecture—*i.e.* about the feedback control they implement. These properties mostly include *liveness* (*e.g.* data collected by a given sensor always trigger a particular reconfiguration action) and *safety* (*e.g.* data from a given sensor never lead to a certain reconfiguration). They are usually expressed in temporal logic [38]. In FCDL, we use *Linear Temporal Logic* (LTL) to characterize these properties in the sense of interaction invariants.

Static Interaction Invariants An interaction contract constrains the interactions allowed at the level of a single adaptive element. An interaction invariant does the same but at the composite level. It expresses requirements and restrictions about the data flow within an assembly of components.

For example, in the adaptation scenario we would like to ensure that whenever someone accesses the web server a controller will be activated. Using the LTL, this invariant can be expressed as:

$$\Box\,(\,\texttt{log\_activate} \rightarrow (\Diamond\,\texttt{utilController\_activate}))$$

The predicate variables **`log_activate`** and **`utilController_activate`** relate to the `log` and `utilController` elements and are **true** when they have been activated. The LTL formula means that: "*always* ($\Box$) when the `log activate` interaction contract is executed, then the `utilController activate` interaction contract will *eventually* ($\Diamond$) be activated as well."

In FCDL this interaction invariant is defined in the `ApacheQOS` composite using the following code (both the `FileTailer` and `IController` define one interaction contract called `activate`):

```
composite ApacheWebServer {
   feature log = new FileTailer (/* ... */)
   // ...
}
composite ApacheQOS {
   feature server = new ApacheWebServer (/* ... */)
   feature utilController = new IController (/* ... */)
   // ...

   temp inv loopLiveness =
     // LTL formula
     [] (activate@server.log -> <> activate@utilController)
}
```

These properties can be easily verified by an appropriate model checker, such as SPIN [23]. Because the verification is done statically at design time, we refer to these interaction contracts as *static*. SPIN takes a model of a system described in the Promela modeling language. It will try to find a counter example in which the negated LTL formula holds providing the corresponding stack trace. The Promela model can be generated from our architecture model by mapping the element interaction contracts and message flow into the corresponding Promela concepts. For example, the `utilController : IController` interaction contract $\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \rangle$ is translated into Promela code shown in Listing 2.

```
// act activate(input; ; output)
#define utilController_activate (util_controller@act_activate)

// ports
chan utilController_port_input = [1] of { mtype }; // push in port input: Double
chan utilController_port_output = [1] of { mtype }; // push out port output: Double

active proctype utilController() {
  byte act = 0;

  end: // infinite process

  waiting: // waiting for activation
    if
    :: utilController_port_input ? PUSH -> act = 1; // act activate(input; ; output)
    fi;

  executing: // element activations
    if
    :: act == 1 -> // act activate(input; ; output)
       act_activate:
         utilController_port_output ! PUSH;
    fi;

  act = 0;
  goto waiting;
}
```

Listing 2. Example of the generated Promela code for `IController`

Since the interaction contracts are also used to synthesize the AE operation methods, these properties are also preserved at the implementation level.

Timed Interaction Invariants The static interaction invariants above allowed us to define an invariant ensuring that, whenever someone accesses the web server, a controller will be activated. The natural extension is to add a time constraint—*i.e.*, to ensure that not only something will happen, but also to put a deadline until when it has to happen. We refer to these types of constraints as *timed interaction invariants*. They extend the static interaction invariants with quantitative time.

The following formula adds a 6 seconds (5 seconds is the initial scheduling period −cf. Listing 1) deadline for the controller activation from the new access log record:

$$\Box\,(\,\texttt{log\_activate} \rightarrow (\Diamond_{\texttt{in}<6seconds}\ \texttt{utilController\_activate}))$$

In FCDL it is expressed as:

```
temp inv loopLivenessWithDeadline =
   [] (activate@server.log -> <>(in < 6.seconds) activate@server.adaptor)
```

Similarly, we can track a progress of an individual component. For example, we might want to ensure that the controller is triggered regardless the web server activity every at least 6 seconds:

```
temp inv controllerLiveness =
   [] (<>(in < 6.seconds) activate@utilController)
```

Unlike the static interaction invariants, the timed invariants are only verified at runtime. Our realization is based on the approach developed by Stolz and Bodden [43] for temporal assertions of Java-based programs.

4.4 Structural Invariants

Motivation Interaction contracts make possible to check architecture consistency. However, because of their generality, they do not help to spot domain-specific problems and FCL architecture issues. For example, an effector that is manipulated by multiple controllers may lead to undesired interference [19].

Essentially, these problems are related to the model structure, concretely the structural configuration of the adaptive elements forming the FCLs. A general mechanism to constraint model structure is provided in languages like OCL that defines model invariants as boolean assertions over model elements structure [33]. In a similar way, we extend the FCDL language with structural invariants that operate on the FCDL meta-model and make possible to statically assert adaptive element structures at design time.

Invariant Specification A structural invariant operates at the level of the FCDL instance meta-model (cf. Figure 5), which corresponds to the type meta-model shown in Figure 2. For example, instead of using an instance of `PeriodicTrigger`, which is an instance of `AdaptiveElementType`, we work with the instance of `scheduler`, which is an instance of `AdaptiveElementInstance` (that has a reference called `type` pointing to the `AdaptiveElementType`, concretely to `PeriodicTrigger`).

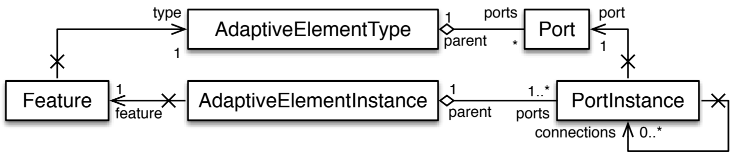

Fig. 5. Excerpt of the FCDL instance meta-model

For example, to ensure that the `requests` and `size` ports of the `Utilization` processor are connected to two different sources (in order to prevent mistakenly connecting them to the same `Accumulator` for instance), we use the following code that traverses the FCDL instance model (**`self`** is an instance of the currently checked `AdaptiveElementInstance`):

```
struct inv DifferentInputSources =
 self.ports
    .filter[p | p.name = 'size' || p.name = 'requests'] // select ports
    .map[p | p.connections] // select their connects
    .map[p | p.parent] // select owning instances
    .toSet
    .size == 2 // there must be two different ones
```

Similarly to the other contracts introduced in this section, structural invariants also support dependencies (to other structural constructs) and can be defined using Xbase anonymous classes or Java implementations.

5 Failure Handling

The contracts introduced in the previous section aim at explicitly specifying architecture-level and behavioral-level assumptions about feedback control loops—*i.e.* they define what is *expected*. This section focuses on the opposite *exceptional* cases—*i.e.* on what happens when these contracts are violated. It describes the mechanisms provided in FCDL that can be used to detect exceptions and to coherently handle them.

5.1 Failures and Exceptions

When associated to contracts, Meyer [30] defines an *exception* as a runtime event that may cause a *failure*—*i.e.* the termination of a routine call that does not satisfy the routine contract. What is important to note on this definition is that every failure is the result of an exception, whereas not every exception leads to a failure. An exception occurs when an operation cannot achieve its intention. In FCDL there are two sources of exceptions: a *contract violation* of either an invariant (state or timed temporal) or a pre/postcondition, or a *runtime exception* that has been thrown by an adaptive element operation method[11]. Possible causes of runtime exceptions fall into one of the three categories: *systematic* caused by programming errors, *accidental* caused by corrupt internal state, or *transient* caused by failures of some external resource used during computation including exceptions caused by connected elements (*e.g.* timeouts).

As previously stated, an exception does not necessary need to lead to a failure and it is possible to detect and recover from an exceptional state. In FCDL, actors are organized in composites. This introduces a hierarchical structure (cf. Figure 6) whereby a composite supervises its subordinates (nested adaptive elements). This therefore implies a dependency relation between the actors. Besides that, the composite is responsible for routing messages from promoted ports to the connected adaptive elements and vice-versa, it must also respond to their failures. A subordinate failure then becomes an exception in a supervisor.

[11] By the term *runtime exception*, we mean all exceptions that are possibly thrown at runtime, which in the case of Xbase and Java include both checked and unchecked exceptions

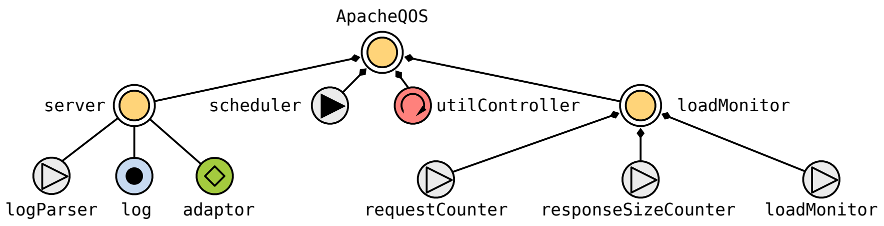

Fig. 6. Adaptive Elements hierarchy

Depending on the nature of the exception, the supervisor can choose one of the following action[12]: *resume* and thus keeping the state of the error adaptive element as it is, *restart* adaptive element(s) with a potentially different configuration or implementation, or finally *escalate*. An escalation converts the exception into a failure, which in turn, either becomes an exception in a higher-level composite or fail the complete system when there are no more layers.

5.2 Exception Handling

The goal of exception handling is to make coherent responses to exceptional cases. These are exceptions that are not handled within AE operation methods themselves, since otherwise they would be expected regardless any special treatment they require. Exceptions are thus handled at the composite level.

The following listing shows the exception handling constructs:

```
composite C {
  // ...

  catch {
    case [<variable_name>:] <exception_sel> @ <feature_sel> = ...
    // ...
  }
}
```

`<exception_sel>@<feature_sel>` is an expression that specifies which exceptional case(s) are to be handled—*i.e.* which exception(s) from which adaptive element(s). It has two parts: a set of exception types (or an asterisk matching them all), and a set of considered features in which the exception can occur (or an asterisk matching all features declared within the composite). For example `case ValidLogFormatViolation @ logParser` handles the case of `ValidLogFormat` contract violation that occurs in `logParser` instance of `AccessLogParser` adaptive element.

Before we show how the above constructs can be used to handle an exception, we need to discuss where are the exceptions defined. As we have hinted, there is a tight relation between a contract violation and an exception. Concretely, all state and timed interaction invariants as well as both pre and postconditions

[12] Making a parallel with the Meyer's *Disciplined Exception Handling principle* [30], the resume and restart actions corresponds to the *retrying* response and the escalation falls into *failure*.

are turned into exceptions (whose name matches the contract name with the "Violation" suffix[13]). Turning contract violations into exceptions allows us to respond to all abnormal cases in a consistent way. As mentioned above, the two possible sources of exception are contract violations, which we have just turned into exceptions, and runtime exceptions, which are all the exceptions possibly thrown by the operation methods.

To illustrate the ramification of unified exceptions, let us consider the `AccessLogParser` again. Its IC is **act** `activate(line; ; requests, size)` which, together with the two preconditions defined in the previous section, translates into the following Java interface:

```
public interface AccessLogParser {
   Tuple2<Int,Long> activate(String line)
      throws LineMatchingPatternViolation, SuccessiveTimestampViolation;
}
```

Now, if we had modified the interaction contract to:

```
  act activate(line; ; requests, size) throws NumberFormatException
```

the `NumberFormatException` would have been added to the method throws clause, allowing us to handle[14] it in the supervisor using:

```
  case NumberFormatException @ logParser
```

By making both exception sources explicit, we can statically check at design time whether all cases are covered.

5.3 Supervision Strategies

Equipped with the exception handling constructs, coherent strategies can now be defined to respond to exceptional cases. Essentially, a strategy is a function that, given an exception context—*i.e.* the exception itself and the reference to the failed adaptive element instance — executes one or more actions. Similarly to contract definition, it can be a Xbase expression including an anonymous class or an existing class instance, or can be left empty so that an empty handler class is generated. The implementation should result into one of the three possible actions. The simplest case is to resume the operation leaving the state of the failed adaptive element untouched. This corresponds to an empty implementation. The second option is to restart one or more adaptive elements depending on how directly the elements affect one another. One can either create an *one-for-one* strategy in which only the failed element is restarted or an *all-for-one* where all inextricably linked elements are restarted. By restarting an element, not only its internal state is cleared, but it also gives an opportunity to select a different configuration—*i.e.* combination of implementation property settings. Finally, by re-throwing the exception the issue is escalated to the higher level of the hierarchy.

[13] There are two reasons for the suffix: (i) it makes a clear distinction between contract logic and contract exception, and (i) it makes more sense English wise.

[14] It is unlikely that such an exception would occur. However since the size of the response has to be converted from a `String` into a `Long` declaring this runtime exception explicitly contributes to the AE robustness.

Often a combination of actions can be used to define an exception handling strategy. In the following example, we handle a timeout exception that might occur in the `PeriodicTrigger` when pulling data from a sensor (`aSensor`) through the input pull port. This shows an example of a quality-of-service violation in which the contracted part, the sensor, does not meet its response time:

```
catch {
  case e: InputPortTimeoutViolation@scheduler = new ExceptionHandler {
    var failures = 0
    override handle() {
      failures += 1
      switch failures {
        case failures < 5 : {} // no action = resume action
        case failures > 5 && failures < 10 :
          switch aSensor { // restart action
            case SensorImpl1 : restart(aSensor, new SensorImpl2)
            case SensorImpl2 : restart(aSensor, new SensorImpl1)
          }
        default : throw e // escalation
      }
    }
  }
}
```

The realized strategy[15] alternates between two different sensor implementations up to a point where it gives up and escalates the problem.

6 Assessment

FCDL models an architecture of a feedback control system. In general architecture models are used for two main engineering concerns: for statical analysis and for mapping the architecture into an implementation [35]. A key element in both cases concerns the amount of details required about components and their interactions in the architectural description which determines the degree of available implementation guidance and verification support. The use of contracts in FCDL increases the amount of details with explicit assumptions about the adaptive element interactions, behavior and structure. This in turn contributes to a better programming and verification support. However, as there is no silver bullet [7], contracts come with their own costs and drawbacks. In this section we assess their impact and trade-offs.

6.1 Modeling with Contracts

For the adaptation scenario from Section 2, we have implemented in total 45 contracts (7 interaction contracts, 13 state invariants, 8 preconditions, 10 post-conditions, 1 structural invariant, 1 static and 1 timed interaction invariant). The summary of all these contracts is presented in Table 1. While the control presented in the chapter is of a rather modest size, it represents a real-world adaptation scenario and gives us an interesting insights into modeling feedback control loops with contract support. It allows us to do a preliminary assessment

[15] The implementation is rather naive, as the purpose is to demonstrate the language features.

on the impact of the contracts on assumptions *visibility*, implementation *effort*, *performance*, and the overall system *reliability*.

Adaptive Element	**Contract Type**	**Definition**
`FileTailer`	Interaction	$\langle self; \emptyset; \Uparrow (\texttt{line})\rangle$
	State	`path` is non empty
	State	`path` exists
	State	`path` is a file
	State	`path` is readable
	Post	`line` is not empty
`AccessLogParser`	Interaction	$\langle \Uparrow (\texttt{line}); \emptyset; \Uparrow (\texttt{requests}, \texttt{size})\rangle$
	State	`logFormat` is not empty
	State	`logFormat` is a valid pattern
	State	`logFormat` includes a named group for timestamp
	State	`logFormat` includes a named group for response size
	Pre	`line` matches the `logFormat` pattern
	Pre	`line` contains a valid timestamp
	Pre	successive log timestamps
	Runtime	reported response size is a valid integer
	Post	$\texttt{size} > 0$
	Post	$\texttt{requests} = 1$
`Accumulator`	Interaction	$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output})\rangle \parallel \langle \Downarrow (\texttt{sum}); \emptyset; \emptyset\rangle \parallel \langle \Uparrow (\texttt{reset}); \emptyset; \emptyset\rangle$
	Pre	$\texttt{input} <= \texttt{Long\_MAX\_VALUE} - \texttt{value}$
	Pre	$\texttt{input} >= \texttt{Long\_MIN\_VALUE} - \texttt{value}$
	Post	$\texttt{value} == \texttt{old.value} + \texttt{input}$
	Post	$\texttt{output} == \texttt{value}$
	Post	$\texttt{value} == 0$
	Post	$\texttt{sum} == \texttt{value}$
`LoadMonitor`	Interaction	$\langle \Downarrow (\texttt{utilization}); \Downarrow (\texttt{requests}, \texttt{size}); \emptyset\rangle$
	State	`a` is set in range cf. (1)
	State	`b` is set in range cf. (1)
	Pre	$\texttt{requests} >= 0$
	Pre	$\texttt{size} >= 0$
	Post	`utilization` is computed using (1)
	Structural	`requests` and `size` are connected to different targets
	Runtime	`requests` pull input port timeout
	Runtime	`size` pulls the input port timeout
`PeriodicTrigger`	Interaction	$\langle self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output?})\rangle$
	State	$\texttt{initialPeriod} > 0$
	Post	$\texttt{output} == \texttt{input}$
	Runtime	`input` pulls the input port timeout
`IController`	Interaction	$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output})\rangle$
	State	$\texttt{referenceValue} > 0$
	State	$\texttt{kI} > 0$
	Post	`output` is computed using (2)
`ContentAdaptor`	Interaction	$\langle \Uparrow (\texttt{contentTree}); \emptyset; \emptyset\rangle$
	Pre	$0 <= \texttt{contentTree} <= M$
`ApacheQOS`	Temp Int. Inv.	$\Box\, (\Diamond\, (\texttt{in} <= 32s)\, \texttt{utilController\_activate})$
	Static Int. Inv.	$\Box\, (\texttt{server.log\_activate} \rightarrow \Diamond \texttt{server.adaptor\_activate})$

Table 1. Interactions contracts for the adaptation scenario (cf. Section 2).

Visibility One of the design goal of FCDL is visibility—*i.e.* FCLs, their processes and interactions should be made explicit at design time as well as at runtime. The contracts contribute to this goal by making visible not only the interaction but also the assumptions about their behavior at the level of a single element as well as at the level of the assembly.

The explicit specification also helps the separation of concerns between control engineers and software developers. Control engineers can use FCDL to define the overall FCL architecture specifying required assumptions on the loop components whose implementation can be then carried out by developers. Contracts therefore helps to mediate communication and improve adaptive element documentation.

Implementation Effort The FCDL contracts implement what has been acknowledged to be a reasonable trade-off between a full extend of formal specifications and acceptable effort to developers [31]. Without contracts, the only option would be to follow defensive programming to check and protect incorrect input and invalid state using control flow constructs. Contracts on the other hand make these checks explicit and separate them from the operational method code. Furthermore, they allow developers to systematically handle exceptional cases.

Implementing all the contracts from Table 1 is associated with some development effort. In the case of behavioral contracts, it was slightly higher than having similar checks directly intertwine in the AE operational methods. Also, as the number of contracts increases the complexity of handling them rises as well. On the other hand, making the exceptions explicit allows one to statically check whether all the cases are covered which would have to be otherwise done manually.

Performance Impact As with any runtime verification, there is a certain overhead in instrumenting software systems. In the case of FCDL contracts, this includes the penalty of evaluating the assertions, as well as the cost of the hooks that trigger them. In the case of the behavioral contracts, the impact is small since the actual check is synthesized into corresponding AE operation methods. In the case of timed interaction invariant, the instrumentation comes at a cost of an extra actor per invariant and extra notification messages. This overhead is linear to the number of formulae and to the number of predicates they contain.

In the current Akka 2.0 based prototype, 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[16]. A sample push/pull communication with a throughput of 5000 messages per second amounts for 5% of CPU time. Therefore, the performance impact of the evaluation itself largely depends on the complexity of the assertion itself. The assertion would however need to be evaluated anyway, regardless if contracts are used or not—*i.e.* without contracts it would be an if−condition in the code.

Like in other languages, the runtime verification of contracts can be turned off leaving the operational methods unaffected (with all the consequences of unprotected code).

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

Reliability Contracts impact both correctness and robustness properties [29]. Correctness is linked to contract efficiency—*i.e.* the ability of a contract to detect a failure. A contract violation implies the execution of an erroneous system. By injecting errors into the system and recording contract violations, it is possible to estimate their impact on software vigilance and diagnosability [26]. This approach is part of our plans for further work.

In addition to record state semantics, contracts also aid with bug assessment [30]. A violation of a precondition indicates a problem in the client while postcondition violation manifests a service failure.

In a recent study, Cámara *et al.* [8] made an experimental evaluation of the robustness of the Rainbow framework for architecture-based self-adaptation [18]. They defined a set of mutation rules, which were systematically applied to controller input in order to explore the limit conditions that according to the study are the typical sources of robustness issues. Essentially, the mutation rules feed the controller with invalid inputs, such as nulls, empty strings, wrong timestamps, or by overflowing or underflowing the input domain bounds. Their experiment uncovered robustness issues in about half of the tests they conducted, majority of which resulted in salient failures and a few even crash the controller. These results confirm the importance of having explicit assumption on inputs which in FCDL is supported by the behavioral contracts.

6.2 Limitations

While the use of contracts brings many advantages, they also come with some shortcomings. Contracts benefits are directly linked to their efficiency—*i.e.* what the contracts say and, often more importantly, what they do not, as they might give a false sense of correctness.

A low efficient contract only contributes to performance penalty instead of system reliability. In the case of poorly implemented contracts, such penalty might seriously affect the overall performance of the whole system. Furthermore, this can lead to an even worse situation when a contract is wrong. There are three types of incorrect contracts: (i) an inaccurate or false assertion that executes faulty system, (ii) a bug in the assertion throwing a runtime exception, and (iii) an impure function that violates the descriptive nature of contract assertions and mutates the component state upon evaluation. The last one in particular might cause a serious harm and is usually difficult to detect. Currently, in the assertion language, there is no support to detect an impure functions.

6.3 Discussion

The objective of FCDL is to provide engineers and researchers with a flexible abstraction that allows them to easily experiment with self-adaptation without the need to deal with low-level system implementation details. Including contracts into FCDL contributes to this goal by making this abstraction more rigid, yet without hindering its use. Concretely, they allow developers to precisely specify the assumptions about the component operational conditions, interactions and behavior in a systematic way with a simple, yet expressive programming language. One of the main benefits of the contracts is that it guides developers to make a clear distinction between what is an expected, what is a special, and what is an exceptional case and about how to handle them. Therefore, even though, there is an initial higher development effort, which could make some software engineers reluctant, according to our experience, as the system grows, the advantages of clear separation outweighs it. Moreover, the generative approach ensures that statically verifiable contracts—*i.e.* interaction contract, static interaction and structural invariants remain preserved at the implementation level. While the adoption of contracts fosters the reuse of adaptive elements in FCLs, we also contribute to improve the separation of concerns by isolating contract verification from failure handling. This approach supports the definition of custom repair strategies depending on the context of deployment of the software system.

7 Related Work

Our work is related to interaction specification, component contracts and self-adaptive software systems engineering.

7.1 Interaction Specification

To address the architecture underspecification, Cassou *et al.* [9] propose to enrich SCC architecture descriptions by annotating components with *interaction contracts* that precise their interactions. We extend this notion and make it applicable to FCDL. Concretely, our extension to the original proposal includes support for: (i) components with multiple output ports, (ii) multiports, (iii) composites including IC inference algorithm, (iv) optional interaction contracts, (v) interaction contract completion verification.

There are other formalisms that are commonly used to specify interactions between components or processes in distributed systems [3,28] or in hierarchical component-based architectures [37]. However, since they offer full description of an interaction sequences, they require more expertise to use them properly. It is also much harder to enforce a complete automata behavior by the generated code while the interaction contracts remain preserved at the implementation level [36]. Finally, as interaction contracts capture the specific properties of data and demand driven communication, they were easier to tailor to FCDL than more general automata-based models.

7.2 Component Contracts

Beugnard *et al.* [6] describe four levels of contracts in component-based systems: syntactic, behavior (as defined by Meyer [29]), synchronous, and quality-of-service (QoS). The contracts we introduced into FCDL partially follows this hierarchy. The syntactic contracts include the adaptive element interface specification together with interaction contracts. The pre and postconditions with state invariants behave similarly as the contracts defined by Meyer with the small extension of contract dependency and custom error messages. Since in actor-oriented programming model there is no need to protect mutable state, we do not need to support synchronous contracts *per se*. The data flow synchronization is already covered by interaction contracts and we further include interaction invariants to enforce liveness properties through LTL formulae. Finally, the QoS contracts are partially supported through the runtime exceptions and their consequent supervision strategy. However, explicit QoS negotiation and component rewiring at runtime is not supported and restart reconfiguration is used instead. Negotiation strategies inspired by agent-based systems have been proposed for hierarchical component-based systems [10]. They could be envisaged in our context, but the relationship with the control models would need a thorough study.

The combination of several kinds of contracts have also been proposed for hierarchical components, with the coupling of executable assertions and temporal logic [13] and with some composition properties enabling the creation of a composite contract [14]. Contrary to these approaches, our work is tailored to the feedback loop architecture, ensuring more properties on the data flow synchronization, framing the implementation while being more technology agnostic.

In LeTraon *et al.* [26] the authors propose a formal way to evaluate the impact of contracts on system vigilance and ability to detect bugs. Since the work considers Meyer's behavioral contract, it shall be usable for FCDL and thus we plan to incorporate it into the FCDL tool support.

7.3 Self-Adaptive Software Systems Engineering

There is a number of frameworks, middlewares and model-driven engineering approaches to self-adaptive systems engineering. They aim to reduce the design and implementation effort and provide a solid foundation for engineering of self-adaptive software systems (cf. surveys in Salehie and Tahvildari [41] or Villegas *et al.* [44]). However, they often target specific types of adaptation problems and require the use of certain adaptation mechanism (*e.g.* utility theory in Rainbow [18]) or are applicable to a single domain (*e.g.* mobile applications in MUSIC [40]) or technology (*e.g.* Java-based systems in StarMX [4]), thereby limiting their applicability with respect to the problem being addressed [39]. Furthermore, they do not particularly focus on control theoretical controllers.

FCDL on the other hand focuses primarily on the integration aspect of SASS engineering. It does not promote any particular control approach and instead provides an abstraction of feedback control loops in which various scenarios can be modeled with diverse control mechanisms. Having based the implementation on Akka and Xbase currently limits the adaptive element implementation to JVM 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. However, thanks to the model-driven engineering approach, it is possible to target different actor runtimes and use Xbase to synthesize code to languages other than Java[17]. The increasing popularity of the actor model gives us a variety of different frameworks available in various programming languages[18]. This should allow for deploying the proposed solution on top of a wide range of systems.

While across this paper we have mostly followed a control theoretical approach, it should be equally possible to use Rainbow [18] instead of the integral controller. Rainbow could then take advantage from all the contracts introduced in this chapter, which should in turn improve its robustness in the experiments such as the one mentioned in the previous section.

There is also a large body of work that focuses on design and simulation of feedback control primarily for embedded systems, for example Ptolemy II [16]. Ptolemy II is an extensive framework for simulation of concurrent actor-oriented systems with the ability to combine heterogeneous models of computation. We follow a similar actor-oriented approach and our execution semantics is comparable to Ptolemy *Push-Pull* model of computation. Similarly to Rainbow, it should be possible to create a Ptolemy-based controller that can be used in FCDL. The same applies to other tools that are often used by control theorists such as Matlab / Simulink[19] or Modelica[20].

8 Conclusion

In this chapter we have presented a design-by-contract extension into *Feedback Control Definition Language*, a domain-specific modeling language for integrating control mechanisms into software systems through external feedback control loops. The aim is to allow researches and engineer to explicitly specify their assumptions about the operational condition of the different components that form feedback control loops.

The presented contracts support this by embedding elements of formal specification into the feedback control loop element definitions. Concretely, we have defined first-class language support to specify (i) behavioral contracts to assert component behavior through state invariants and pre and postconditions, (ii) interaction contracts to express allowed component interactions, and (iii) structural and temporal invariants to define architecture constraints as well as design and execution time interaction invariants. The temporal invariants are specified using linear temporal logic while the assertion of the other contracts use a Java-like expression language. Next to these contracts, we have also defined a first-class language support for systematic fault handling.

All the different type of contracts have been illustrated on a real-world adaptation scenario of web server QoS management control.

As for future work there are several opportunities for further extending the contracts specifications and validation of our approach: (i) explore the extend

[17] https://wiki.eclipse.org/Xbase
[18] http://en.wikipedia.org/wiki/Actor_model
[19] http://www.mathworks.com/products/simulink/
[20] https://openmodelica.org/

to which contracts could express causalities to ensure that a particular action always leads to a certain system state change; (ii) enable runtime modifications of invariants time thresholds; (iii) provide more insights into contracts evaluation by combining the mutation rules from Cámara *et al.* [8] and the quantifying approach proposed by Le Traon *et al.* [26], shall allow us to have an automate way to estimate the levels of vigilance and diagnosability of given set of contracts, and finally (iv) develop a tool support to better facilitate the use of the presented FCDL contracts.

Acknowledgments This work is partially supported by the Datalyse project `www.datalyse.fr` and was previously supported by the ANR SALTY project under contract ANR-09-SEGI-012.

References

1. Abdelzaher, T., Bhatti, N.: Web server QoS management by adaptive content delivery. In: 7th International Workshop on Quality of Service. (1999)
2. Abdelzaher, T., Shin, K., Bhatti, N.: Performance guarantees for Web server end-systems: a control-theoretical approach. IEEE Transactions on Parallel and Distributed Systems 13(1) (2002)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ACM SIGSOFT Software Engineering Notes. vol. 26 (2001)
4. Asadollahi, R., Salehie, M., Tahvildari, L.: StarMX: A framework for developing self-managing Java-based systems. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. (2009)
5. Berekmeri, M., Serrano, D.: A Control Approach for Performance of Big Data Systems. In: Proceeding of the 2014 IFAC World Congress. (2014)
6. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. Computer 32(7) (1999)
7. Brooks, F.P.: No Silver Bullet Essence and Accidents of Software Engineering. Computer 20(4) (1987)
8. Cámara, J., de Lemos, R., Laranjeiro, N., Ventura, R., Vieira, M.: Robustness Evaluation of the Rainbow Framework for Self-Adaptation. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. (2014)
9. Cassou, D., Balland, E., Consel, C., Lawall, J.: Leveraging software architectures to guide and verify the development of sense/compute/control applications. In: 33rd International Conference on Software Engineering. (2011)
10. Chang, H., Collet, P.: Fine-grained Contract Negotiation for Hierarchical Software Components. In: 31th EUROMICRO-SEAA Conference - CBSE Track. (2005)
11. Cheng, B.H.C. *et al.*: Software Engineering for Self-Adaptive Systems: A Research Roadmap. Software Engineering for Self-Adaptive Systems, LLNC 5525 (2008)
12. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system. In: 4th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. (2009)
13. Collet, P., Ozanne, A., Rivierre, N.: Enforcing Different Contracts in Hierarchical Component-Based Systems. In: 5th International Symposium on Software Composition. (2006)

14. Collet, P., Malenfant, J., Ozanne, A., Rivierre, N.: Composite Contract Enforcement in Hierarchical Component Systems. In: 6th International Symposium on Software Composition. (2007)
15. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering (2012)
16. Eker, J., Janneck, J., Lee, E., Ludvig, J., Neuendorffer, S., Sachs, S.: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE 91(1) (2003)
17. Filieri, A., Hoffmann, H., Maggio, M.: Automated Design of Self-Adaptive Software with Control-Theoretical Formal Guarantees. In: Proc. 36th International Conference on Software Engineering. (2014)
18. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. IEEE Computer 37(10) (2004)
19. Hebig, R., Giese, H., Becker, B.: Making control loops explicit when architecting self-adaptive systems. In: Proceeding of the second international workshop on Self-organizing architectures. (2010)
20. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback control of computing systems. Wiley Online Library (2004)
21. Hellerstein, J.L.: Engineering Autonomic Systems. In: Proceedings of the 6th International Conference on Autonomic Computing. (2009)
22. Hewitt, C.: Viewing control structures as patterns of passing messages. Artificial Intelligence 8(3) (1977)
23. Holzmann, G.J.: Spin Model Checker. Addison-Wesley Professional, 1. edition edn. (2003)
24. Křikava, F.: Domain-Specific Modeling Language for Self-Adaptive Software System Architectures. Ph.D. thesis, University of Nice Sophia-Antipolis (2013)
25. Křikava, F., Collet, P., France, R.B.: ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (2014)
26. Le Traon, Y., Baudry, B., Jézéquel, J.M.: Design by contract to improve software vigilance. IEEE Transactions on Software Engineering 32(8) (2006)
27. Lu, Y., Abdelzaher, T., Lu, C., Tao, G.: An adaptive control framework for QoS guarantees and its application to differentiated caching. In: 10th International Workshop on Quality of Service. (2002)
28. Lynch, N.A., Tuttle, M.R.: Hierarchical Correctness Proofs for Distributed Algorithms. In: PODC'87: Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing (1987)
29. Meyer, B.: Applying 'design by contract'. Computer 25, (1992)
30. Meyer, B.: Object-oriented Software Construction (1997)
31. Meyer, B.: Toward More Expressive Contracts. Journal on Object Oriented Programming 13(4) (2000)
32. Niz, D.D., Bhatia, G., Rajkumar, R.: Model-Based Development of Embedded Systems: The SysWeaver Approach. In: 12th IEEE Real-Time and Embedded Technology and Applications Symposium. (2006)
33. Object Management Group: OMG Object Constraint Language (OCL). Tech. rep.
34. Odersky, M.: Contracts for scala. Runtime Verification 6418, (2010)
35. Oreizy, P., Rosenblum, D.S., Taylor, R.N.: On the Role of Connectors in Modeling and Implementing Software Architectures. Tech. rep., Department of Information and Computer Science, University of California (1998)

36. Parizek P., Plasil, F., Kofron, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In: 30th Annual IEEE / NASA Software Engineering Workshop (SEW-30) (2006)
37. Plasil, F., Visnovsky, S.Behavior protocols for software components. IEEE Transactions on Software Engineering 28(11) (2002)
38. Pnueli, A.: The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science (1977)
39. Ramirez, A.J., Cheng, B.H.C.: Design patterns for developing dynamically adaptive systems. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. (2010)
40. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In: MobMid '08: Proceedings of the 1st workshop on Mobile middleware. (2008)
41. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4(2) (2009)
42. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. Software, IEEE 20(5), (2003)
43. Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. Electronic Notes in Theoretical Computer Science 144, (2006)
44. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. (2011)
45. Zhao, Y.: A Model of Computation with Push and Pull Processing. Tech. rep., Technical Memorandum UCB/ERL M03/51, University of California, Berkeley (2003)