A Component-Based Architecture Template for Adaptive System Design

Juan F. Inglés-Romero[1] and Cristina Vicente-Chicote[1],

[1]Dep. of Information and Communication Technologies
Technical University of Cartagena
30202 Cartagena, Spain
{juanfran.ingles, cristina.vicente}@upct.es

Abstract. In this paper we propose a component-based architecture template for adaptive system design. This template aims to provide developers with some design guideless that help them obtain efficient adaptive system implementations. Different variants of the template are proposed together with a selection criterion to help designers decide which alternative better fits (or better balance) their system reconfiguration and efficiency requirements. In order to demonstrate the feasibility and the benefits of our proposal, a case study regarding the design and implementation of a self-adaptive robotic system is presented.

Keywords: Component-Based Architecture, Design Template, Adaptive System, Runtime Software Reconfiguration, Fractal, Robotics.

1 Introduction

Nowadays, given the fast pace of technological evolutions and the diversity of computing platforms (both hardware and software), building applications that can work in such a wide range of systems is becoming more and more challenging [1]. Moreover, even on a specific platform, the execution context and available resources may significantly vary at runtime. This may require that the applications dynamically adapt their structure or behavior in order to cope with changing environments. This is particularly the case of resource-constrained systems, for which runtime adaptation could be considered a good means to achieve their best-effort performance within their available resources.

Adaptation in itself is nothing new, but it is generally implemented in an ad-hoc way, which involves trying to predict future execution conditions and embedding the adaptation decisions in the application code. This usually leads to increased complexity (business logic polluted with adaptation concerns) and poor reuse of components, due to the strong coupling among them and with the specific environment for which they have been designed.

In this paper, we propose a component-based architecture template aimed at providing designers with a more systematic (as opposed to ad-hoc) way to develop adaptive systems. The proposed template (1) advocates for a clear separation of the

business and the adaptation concerns; (2) it does not rely on any specific component model; and (3) it is fully application domain independent. Additionally, the proposed template enables a certain flexibility degree providing designers with three alternative variants, each one supporting different efficiency management mechanisms. All these features mainly seek to improve system maintainability, component reusability, and implementation efficiency.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 details the proposed component-based architecture template for adaptive system design and its variants. Section 4 describes our experience in following the design guidelines provided by the proposed template for implementing an efficient self-adaptive robotics system. Finally, Section 5 concludes and presents some future research lines.

2 Related Work

Since the late 90s, great research efforts have been made in (self-)adaptive systems. Among them, this research focuses on component-based approaches. This section reviews related works in this field, including: (1) component models supporting dynamic reconfiguration; (2) existing facilities for dynamic reconfiguration specification; and (3) existing frameworks for adaptive system development.

Among the existing component models currently supporting dynamic adaptation, it is worth noting that only some of them impose restrictions on runtime reconfigurations to assure a proper separation of the business and the adaptation concerns. Next, we review three of them, ordered by the increasingly strong constraints they impose on reconfigurations.

The SOFA 2.0 [2] component model defines a *Nested Factory* reconfiguration pattern [3] that establishes the following rule: "*Each newly created component becomes a sibling of the component who requests the factory. This latter is responsible to bind and initiate the new component*". Although this rule establishes a clear reconfiguration policy, it does not prevent that designers spread the adaptation logic through the architecture, as any of its component can have reconfiguration capabilities. In this vein, other component models limit the reconfiguration capabilities to specific components. For instance, in SaveCMM [4], *Switch* components are the only ones allowed to modify component interconnections at runtime. Switch components contain a set of connection patterns, each one defining a specific way of connecting the Switch input and output ports. Logical expressions, based on the data available at the input ports, are used to determine which connection pattern is applied. Finally, the AADL [5] component model defines a more restrictive reconfiguration mechanism based on predefined *operational modes* (i.e., fix architecture configurations) defined at design time. In this case, runtime reconfigurations imply changing from one mode to another one, i.e., replacing the overall architecture design.

Among the models that do not impose restrictions, Fractal (http://fractal.ow2.org/) defines four basic control interfaces to support architecture reconfiguration at runtime, namely, a binding-controller, a content-controller, an attribute-controller, and a

lifecycle-controller. In addition, relating to dynamic component instantiation, the Fractal component model specifies a general and a specific factory interface. The former enables the creation of all kind of component instances, while the later can only create instances of one component kind. The Fractal component model does not prescribe which components should implement these interfaces. Consequently, the designer assumes the responsibility of deciding how the adaptation logic is implemented. This commonly leads to several ad-hoc designs, which do not follow a clear pattern, making it difficult to reuse and maintain them. In the same group as Fractal, the OpenCOM [6] component model does not consider composite components, but special units called *Capsules*. Capsules can be seen as containers into which primitive components can be loaded, instantiated, and bound. Each Capsule defines a name space for its internal component instances, and offers two very primitive system-level reconfiguration facilities: *dynamic loading* (load, unload, instantiate, and destroy) and *dynamic linking*.

Concerning the existing facilities for dynamic reconfiguration specification, FScript [7] is a scripting language that enables designers to define complex reconfigurations in Fractal. FScript makes use of the FPath [7] domain-specific language, which provides it with a notation to query and navigate inside Fractal architectures. FScript is used as part of the SAFRAN [8] extension of the Fractal component model. Plastik [9], differently from FScript, relies on the ACME/Armani ADL [10], the Architecture Definition Language (ADL) used in OpenCOM to define component configurations and to specify how these configurations may be changed at runtime. Both FScript and Plastik allow designers (1) to abstract the low-level dynamic reconfiguration primitives offered by the underlying component models; (2) to deal with reconfigurations independently from the business logic, at least at design time; and (3) to asses reconfiguration reliability.

To conclude this section, we review some of the existing frameworks enabling adaptive system development. In particular, we focus on two of these frameworks that illustrate the need not only for dynamic reconfiguration, but also for contex-awareness and reasoning.

The framework presented in [1], offers a general approach to the systematic development of self-adaptive component-based applications. In this framework, adaptation is considered as a concern that needs to be treated separately from the rest of the application. To achieve this, it defines two context-awareness services and a set of adaptation polities. The former provides information about the execution context, while the latter uses this information to decide which reconfigurations need to be performed. Adaptation policies, defined as ECA (Event-Condition-Action) rules, can be classified into three groups: structural reconfigurations, parameterizations, and addition/removal of component services.

CASA [11] is a general-purpose framework that, as the previous one, promotes the separation of the adaptation and the business concerns. This separation is achieved through (1) an independent and reusable adaptation infrastructure, which enables to monitor changes in the application execution environment; and (2) a set of contract-based adaptation policies. The CASA framework supports four adaptation mechanisms, namely, dynamic re-composition of application components, dynamic weaving of aspects, dynamic change of application attributes, and dynamic change of low-level services.

Before entering into the details of our proposal, it is worth mentioning that the related works reviewed in this section (among others) have inspired many of the design guidelines included in the proposed component-based template described next.

3 Towards a Design Template for Adaptive Systems

In this section, we present the proposed component-based architecture template for adaptive system design. This template aims to provide developers with some guidance on how to design applications with dynamic reconfiguration capabilities so that they can easily translate them into efficient adaptive system implementations. Before describing the template in detail, we summarize next its main features:

- The proposed template does not assume any specific component model in order to keep it as generic as possible. This will allow developers to choose any of the existing component models to implement their designs. Although we propose a hierarchical architecture, most of its composite components are only logical groups created for sake of clarity. Thus, it is easy to obtain an equivalent non-hierarchical design if needed (e.g., if the component model selected to implement the design does not support composite component definition).
- Separation of concerns is a key design principle. Following it, the proposed template explicitly separates the adaptation logic from the business logic. The benefits of such a decision are twofold: on the one hand, it reduces the complexity and improves the maintainability of the design and, on the other hand, it promotes the reuse and sharing of adaptation mechanisms among applications.
- The proposed template relies on two basic reconfiguration facilities, namely: *dynamic interface binding* and *dynamic component instantiation* Most component models supporting dynamic reconfiguration offer similar primitives. Some of them also implement more complex reconfiguration mechanisms, although these can always be expressed as a combination of the former.
- We provide designers with three (incrementally defined) variants of the proposed template. Each variant supports different *efficiency management* mechanisms dealing with memory occupation and execution time. As we will discuss later, improving one of these factors commonly implies worsening the other. Thus, we also provide designers with a selection criterion to help them decide which template variant is the most appropriate according to their application efficiency requirements.
- In the proposed template, the two reconfiguration facilities and the efficiency management mechanisms are controlled by three separated components, enabling a fine-grained separation of concerns within the adaptation logic.
- Self-adaptive systems are context-aware adaptive systems in which internal reconfigurations are automatically triggered whenever a relevant change in the context is detected. The proposed design template enables the inclusion of an optional component with context-awareness and proactive reconfiguration capabilities, thus enabling self-adaptive system design.

Next, the general structure of the proposed design template is presented. Then, the three following subsections describe, in turn, how this template enables dynamic interface binding, dynamic component instantiation, and efficiency management.

3.1 Overview of the Proposed Design Template

As shown in Figure 1, the proposed component-based architecture template for adaptive system design comprises four main components: *Main*, *Malloc*, *Adaptive Layer* and *Adaptation*. All these components are mandatory in the proposed template, although some of them (e.g., the Adaptation component) can be implemented using alternative designs (i.e., internal templates).

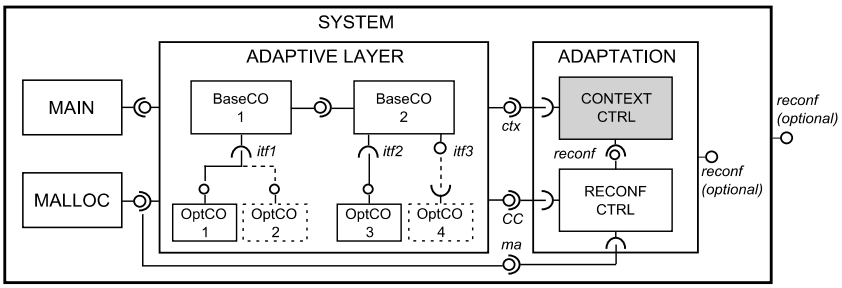

Figure 1. General overview of the proposed component-based design template. Please, note that the optional *reconf* interface, provided by the System and the Adaptation components, is only included in the template when the Context Ctrl component is not present. In this case, the Reconf Ctrl component delegates its *reconf* interface in them.

The *Main* component encapsulates the application entry point and is responsible of initializing and starting the execution by calling the services offered by the Adaptive Layer component. The *Malloc* component is responsible of the dynamic memory allocation necessary to enable runtime component instantiation.

The *Adaptive Layer* component implements the core system functionality, that is, the business logic. As its name suggests, this is the only component in the template which internal architecture can be adapted at runtime. This component may contain any number of internal components of the following types: (i) *BaseCO*, which identify mandatory components; and (ii) *OptCO*, which identify optional elements, i.e., components that might be enabled or not at runtime. Please note that the template does not restrict the nature of these components (i.e., each BaseCO and OptCO can be either a primitive or a composite components) as, from a dynamic reconfiguration perspective, they will be always treated as black box components (i.e, the reconfiguration will never affect their internal structure).

The *Adaptation* component implements the system adaptation logic which, as previously discussed, appears conveniently decoupled from the business logic. The Adaptation component contains an optional *Context Control* component (only included in self-adaptive systems) and a mandatory *Reconfiguration Control* component (labeled in Figure 1 as “Context Ctrl” and “Reconf Ctrl”, respectively).

The *Context Control* component implements context-awareness and proactive reconfiguration capabilities. Its three main functions are: (1) to acquire and structure the application context, i.e., to obtain the relevant context information (raw data) and to derive the value of significant variables from it; (2) to reason on these variables in order to decide whether a reconfiguration is needed or not; and, (3) when a reconfiguration is triggered, to calculate the best possible architecture according to the current context. Note that the Adaptive Layer component provides a *context* interface aimed to make contextual information (raw data) available to the Context Control component. Although the proposed template does not prescribe how the Adaptive Layer internally implements this interface, an elegant solution could be to adopt a Façade Pattern [12]. In this vein, the Adaptive Layer component could delegate the implementation of its *context* interface in an internal façade component responsible of gathering (namely from other components also in the Adaptive Layer) all the relevant contextual information.

As previously stated, the optional Context Control component provides the system with self-adaptation capabilities. When this component is not included in the design we assume that its functions are performed by an external actor (either a user or another system). In this case, the proposed template varies as follows: (1) the *context* interface, provided by the Adaptive Layer component, is removed or simply left unbound; and (2) both the System (root element) and the Adaptation components activate their optional *reconf* interfaces, enabling the Reconfiguration Control component to delegate its provided interface to them.

Regarding the *Reconfiguration Control* component, the proposed template considers three possible design variants, each one providing developers with different dynamic reconfiguration capabilities, i.e., (a) *dynamic interface binding*; (b) *dynamic component instantiation*; and (c) *efficiency management*. Figure 2 outlines the internal structure of the Reconfiguration Control component for each of these variants and the three following subsections describe them in detail.

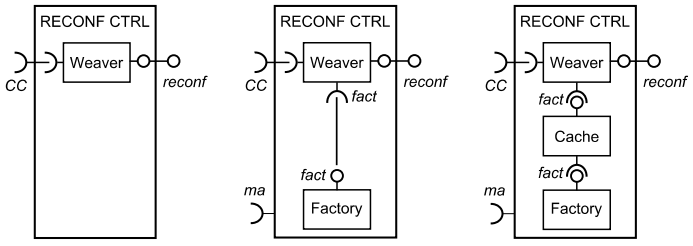

Figure 2. Design variants for the *Reconfiguration Control* component.

3.2 Enabling Dynamic Interface Binding

Among the three design variants supported by the proposed template for the Reconfiguration Control component, Figure 2a outlines the simplest. This variant supports dynamic interface binding as the only reconfiguration mechanism. The *Weaver* component responds to the reconfiguration commands it receives through its

reconf interface (either from a Context Control component or from an external actor) and changes the internal architecture of the Adaptive Layer component accordingly. To do this, the Weaver sends the appropriate bind/unbind commands to the Adaptive Layer through its *CC* interface. As a result, some of its internal optional components (*OptCO*) are bound/unbound according to the prescribed reconfiguration. It is worth noting that, as this design variant does not support dynamic component instantiation, all the *OptCO* components in the Adaptive Layer must be created at compilation-time and kept in memory throughout the execution, regardless of they are in use or not.

It is also worth noting that the template does not prescribe how to implement the Weaver interfaces, that is, the decision of what kind of reconfiguration commands can the Weaver deal with and how it performs the corresponding binding reconfigurations on the Adaptive Layer, relies exclusively on the adaptive system developer. For instance, the Weaver may accept primitive reconfiguration commands (e.g., to bind/unbind a single component) or more complex ones (e.g., to set a certain configuration or mode that may require several bind/unbind operations at a time).

3.3 Enabling Dynamic Component Instantiation

As previously discussed, the former template variant enables the dynamic binding of optional component interfaces, although all these components need to have been instantiated at compilation-time. The second template variant, depicted in Figure 2b and described in this subsection, enables a more efficient memory strategy based on the inclusion of a *Factory* component in the Reconfiguration Control.

The Factory component enables the dynamic creation of component instances. Similar to the classical Factory Pattern [12], the Factory component provides an interface with a generic service that creates and returns new component instances of the requested type. To support this action, the Factory requires the dynamic memory allocation operations provided by the Malloc component (see Figure 1).

This template variant assumes that when the Weaver receives a reconfiguration command it will proceed as follows. Firstly, if some of the optional components in the Adaptive Layer need to be removed, the Weaver will disconnect them and remove all the optional components left unbound. Similarly, if new optional components need to be added, the Weaver will first ask the Factory to create the corresponding instances, and then it will add them and appropriately bind them in the Adaptive Layer.

Here again, as in the previous variant, the template does not prescribe how to implement the Factory component, leaving this decision to the developer. For instance, the Factory can be implemented as a primitive component that stores all component definitions, so that it can create instances of all kind of components. Similarly, it can also be implemented as a composite component that contains internal factories for creating specific component (or groups of component) instances.

3.4 Enabling Efficiency Management

Efficiency concerns need to be carefully taken into account, in particular, when working with resource-constrained systems (e.g., robots, embedded systems, etc.).

When this is the case, it commonly becomes necessary to take and balance different efficiency metrics.

The two template variants, previously described, have each one their own pros and cons in terms of efficiency. For instance, if we consider the CPU processing time, the first scheme (supporting only dynamic interface binding) is significantly more efficient than the second one, as it saves the time of creating and removing component instances at runtime. However, if we consider memory occupation, the balance tips towards the second variant (supporting also dynamic instance creation) as the memory will contain, at each moment, only the strictly required components.

In order to provide designers with some additional efficiency management mechanism, we propose a third template variant that, built on the previous one, includes a *Cache* component between the Weaver and the Factory components. As shown in Figure 2c, the Cache provides (to the Weaver) and requires (from the Factory) the same *fact* interface, serving as a transparent layer between the two other components.

The Cache stores a set of component instances selected according to a predefined policy (e.g., those more recently or frequently used), making them readily available on demand for the Weaver. Depending on the number of instances allowed in the Cache, the system will be more or less efficient in terms of memory usage and execution time. Generally, the bigger the Cache the higher the memory occupation and the faster the reconfigurations. In fact, a degenerate use of this template variant can serve to simulate the other two. For instance, the first one can be simulated with a Cache big enough to store all the possible component instances. Similarly, the second variant can be simulated with a Cache small enough to prevent the storage of any component instance, implying that the Factory will have to create all the required instances at each reconfiguration step.

As in the two previous variants, the template does not prescribe how to implement the Cache, leaving this decision to the developer, who will need to specify, at least, its size and instance selection policy. Besides, the developer could also decide implementing further additional features, e.g., a multi-layered Cache.

4 Tool Support for the Proposed Template

In the previous section, we have proposed a component-based architecture template for adaptive systems design, where efficiency was regarded as a main concern. In order to probe the feasibility and the benefits of applying the proposed template, we have selected Cecilia (http://fractal.ow2.org/cecilia-site/current/), the C reference implementation of the Fractal component model, to implement a self-adaptive robotics system. The reasons that led us to select Fractal and, in particular, Cecila for implementing our case study were the following:

- Fractal is a well known hierarchical and reflective component model intended to implement, deploy and manage a wide range of software systems including operating systems and middleware.
- Fractal does not impose any design constraint and, thus, it allows us to apply the proposed design template without any restriction.

- Fractal offers some generic dynamic reconfiguration capabilities by means of standard control interfaces that enable the manipulation of components, their interfaces, subcomponents, client bindings, and attributes.
- The efficient use of the available resources is crucial in resource-constrained systems (e.g., robotic or embedded systems). Many of these systems only support C applications as they require minimal run-time support. Besides, C provides designers with low-level machine control.

Component-based Fractal architectures are described using a XML-based Architecture Description Language (ADL). This ADL allows designers to specify software architectures in terms of primitive components, attributes, interfaces, component bindings, and composite components, which may contain any number of internal components (either primitive or composite) and bindings among them.

There exist two reference implementations of the Fractal component model, namely, Cecilia for C, and Julia (http://fractal.ow2.org/julia) for Java. Both of them provide developers with a tool-chain that takes ADL files as an input and automatically generates (1) the necessary glue code between components; and (2) the implementation of the Fractal interfaces needed to support features such as runtime reconfiguration.

The F4E (Fractal for Eclipse) plug-in provides designers with a graphical environment for editing ADL specifications. Furthermore, F4E use a Model-Driven Engineering [13] (MDE) approach that enables designers (1) to formally validate their graphical specifications (models) against a Fractal meta-model, and (2) to automatically generate a Java implementation using the Julia API.

As previously described in Section 3, the proposed template requires defining both mandatory components (e.g., Main, Malloc or Weaver) and optional components (e.g., Context Control). However, the Fractal ADL only considers the former. The inclusion of variability descriptions remains an open issue in most component-based ADLs. Some works have already addressed this problem for component models different from Fractal [9, 14]. In order to enable the inclusion of optional components as part of Fractal ADL specifications, we considered two alternatives: (1) creating a new Fractal ADL editor from scratch that explicitly supported component variability; and (2) reusing the F4E graphical editor and including ad-hoc variability annotations to designate optional components. We opted for the second solution, as explained next in Section 4.1.

4.1 Description of the Case Study Implementation Process

Figure 3 illustrates the process we followed to implement our case study according to the design guidelines provided by the proposed template. Before detailing the three main steps in this process, it is worth highlighting that this solution does not intend to compete in any sense neither with F4E nor with the Cecilia tool-chain. Conversely, we have build upon them, trying to reuse their features in as much as possible. Besides, it is also important to remark that the main objective of developing the proposed case study and, in general of this work, was to demonstrate the feasibility and benefits of the design template, rather than to develop new tools or to adapt existing ones.

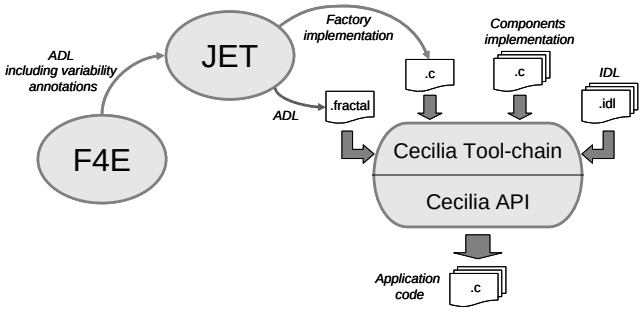

Figure 3. Overview of the process we followed to implement the case study according to the design guidelines provided by the proposed template.

Architecture Design Using the F4E Graphical Model Editor

According to the design guidelines described in Section 3, developers should focus on the description of the business logic, which, in the proposed template, is contained in the Adaptive Layer component. In this regard, the graphical ADL model editor, provided by F4E, can be used to depict the internal architecture of the Adaptive Layer component. As previously mentioned, the Fractal ADL only considers mandatory components (*BaseCO* in the proposed template). In order to provide developers with some mechanism that allows them to define optional components in their designs, we have used the F4E feature that enables the inclusion of *comments* in the Fractal components. In our case, these comments may take two possible values: *OPTIONAL* and *INITIAL*, both associated to optional components (*OptCO* in the proposed template). The only difference between these annotations is that those components marked as INITIAL will be included in the initial architecture configuration, while those marked ad OPTIONAL will be removed. Those components containing comments different from the two we have considered, or containing no comments will be considered mandatory and included in the initial architecture configuration.

Model-to-Text Transformations

At this point, we have an annotated ADL model (conforming the Fractal ADL meta-model) that represents the internal structure of the Adaptive Layer component. In order to translate this model into a Cecilia implementation, following the design guidelines prescribed by the proposed template, a Model-to-Text (M2T) transformation has been implemented using JET (Java Emitted Templates [15]). This transformation generates (1) an ADL file containing the whole architecture description, i.e., the Main, Malloc and Adaptation[1] components , and a filtered version of the Adaptive Layer component, previously designed, from which all the components marked as OPTIONAL have been removed; (2) the code of the Factory component that will allow the dynamic creation of instances for all the components marked as OPTIONAL and INITIAL in the Adaptive Layer component.

[1] Note that the M2T transformation, implemented as part of this research, assumes the third variant of the proposed template for the Adaptation component.

The M2T transformation implemented as part of this research, overcomes two of the main limitations of the current Cecilia tool-chain. On the one hand, although the Fractal specification includes a factory interface for dynamic instance creation, Cecilia does not implement that interface. Given that the ADL file created by the designer contains the definitions of all the optional components in the architecture (i.e., components that need to be dynamically created by the Factory component) it is possible to automatically generate the implementation of the Fractal factory interface from it. On the other hand, according to the latest version of the Cecilia tool-chain (v.2.1), those components that appear disconnected in the input ADL file are automatically removed from the design. This implies that the Cecilia tool-chain will not generate the definitions for the unbound optional components. Thus, our M2T transformation overcomes this limitation creating all the optional component definitions as part of the Factory component.

Using the Cecilia Tool-Chain to Obtain a C Implementation

The Cecilia tool-chain allows developers to generate the C code of their application. It takes as inputs (1) the ADL files (**.fractal*); (2) the interface description files (**.idl*), that are manually defined as F4E does not provide a Fractal IDL editor, (3) the files containing the implementation of the system functionality (*.c), that is, the implementation of the services included in each provided interface. Note that this tool-chain relies on the Cecilia API (see Figure 3), which should provide the C implementation of the Fractal control interfaces involved in the dynamic reconfiguration. For instance, for the design template proposed in Section 3, the Cecilia API should implement the *BindingController* interface (to bind and unbind optional components) and the *ContentController* interface (to add and remove optional component instances into the Adaptive Layer component). However, as the current version of the Cecilia API does not implement the ContentControler interface, we had to manually implement it and add it to the API. As a result of executing the Cecilia tool-chain with the files generated by the M2T transformation in the previous step, we obtained the ANSI C application code, ready to be compiled for a target platform and executed on it.

4.2 Case Study

This section practically illustrates the result of applying the process, previously described, to implement a self-adaptive robotics system. The setting in which we tested our case study was a room without obstacles, where the robot was initially placed at an arbitrary position. The robot perceives its environment thanks to the information provided by its light and noise detectors, which measure the level of light and noise in the room, respectively. Depending on the information provided by these sensors, the robot changes its signaling policy, which can be (i) Acoustic (i.e., playing rhythmic beeps); (2) Light (i.e., turning on its led lights), or (3) None. Additionally, the robot can adopt two different movement strategies, namely a Linear Motion or a Circular Motion Strategy. Both the robot signaling policy and motion strategy can change at runtime depending on certain context changes detailed latter.

In order to implement this case study we have selected a low-cost mobile robotic platform known as e-puck (http://www.e-puck.org). E-pucks are compact mobile robots with a large range of sensors and actuators, which make them appropriate for testing the proposed self-adaptation approach. In spite of that, the computational capabilities of the e-pucks are quite limited as they rely on a dsPIC microcontroller.

As a first step, we designed the internal architecture of the Adpative Layer component for our case study (see Figure 4) using F4E and the variability annotation mechanism described in the previous section. This architecture consists of one BaseCO component (Control), and six OptCO components (namely, Light Detector Noise Detector, Acoustic Signaling, Light Signaling, Linear Motion Strategy, and Circular Motion Strategy). Among the latter, only Circular Motion Strategy and Acoustic Signaling are marked as OPTIONAL, while the other four components are marked as INITIAL. It is worth noting that some of the required interfaces have been marked as optional implying that they do not need to be always bound to another component. For instance, the Control.*LightDet* interface sometimes will be bound to the LightDetector.*LightDet* and sometimes (when, a reconfiguration decides to remove the Light Detector component) it will be left unbound. Conversely, other required interfaces have been marked as mandatory implying that they cannot be left unbound. For instance, the Control.*MotionStr* is mandatory implying that it needs to be bound either to the LinearMotStr.*MotionStr* or to the CircularMotStr.*MotionStr*.

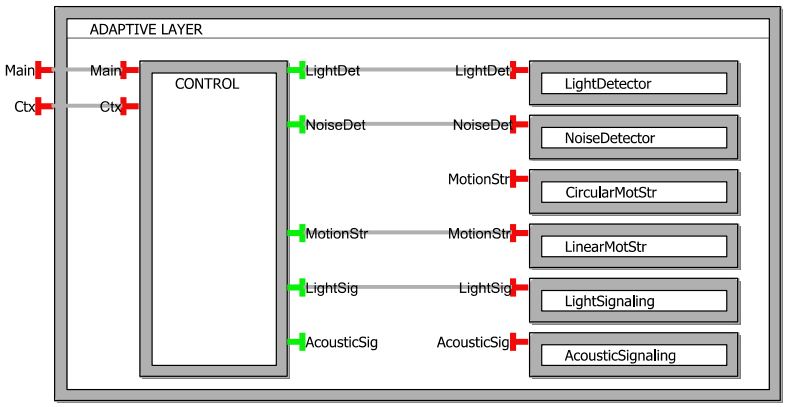

Figure 4. Initial design developed using the graphical model editor provided by F4E.

Figure 5 show the ADL specification obtained by the M2T transformation when executed on the initial ADL file (shown in Figure 4). Please note that the two components marked as OPTIONAL in the initial ADL file have been removed.

In addition, the M2T transformation also generates the factory.c file, containing the implementation of the Factory component (see Figure 3). This Factory is ready to dynamically create (on the Weaver demand) instances for the six optional components. The code associated to all the other components was manually implemented, including the robotic domain-specific components (e.g., Control) and the adaptation logic (i.e., all the internal components included in Adaptation, except the Factory). The information coded for all the components dealing with the adaptation logic is outlined in Table 1.

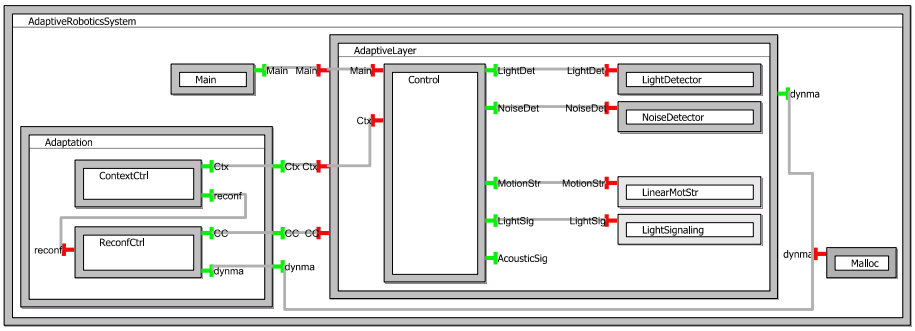

Figure 5. Initial design developed using the graphical model editor provided by F4E.

Table 1. Information coded for the Adaptation internal components.

CONTEXT CONTROL	**ADAPTATION RULES** **If** *LightDetection* **&** **!***NoiseDetection* **then** *Mode 1* **If** **!***LightDetection* **&** *NoiseDetection* **then** *Mode 2* **If** *LightDetection* **&** *NoiseDetection* **then** *Mode 3* **If** **!***LightDetection* **&** **!***NoiseDetection* **then** *Mode 4*
WEAVER	**RECONFIGURATION MODES** **Mode 1:** *AcousticSignaling* **&** *LinearMotStr* **Mode 2:** *LightSignaling* **&** *LinearMotStr* **Mode 3:** *CircularMotStr* **Mode 4:** *LightSignaling* **&** *AcousticSignaling* **&** *CircularMotStr*
CACHE	**CACHE POLICY** **Retention Policy:** The most frequently created remains in the cache. **Size Policy**: Variable size depending on memory occupation (from 0 to 3 components)
FACTORY (automatically generated)	**SUPPORTED COMPONENT TYPES** *LinearMotStr* *CircularMotStr* *LightSignaling* *AcousticSignalig*

5 Conclusions and Future Research

In this paper, we have presented a component-based architecture template for adaptive system design. This template aims to provide developers with some design guideless that help them obtain efficient adaptive system implementations. It is worth highlighting that the proposed template promotes a clear separation between the adaptation and the business logic, and that it provides explicit mechanisms to separately deal with dynamic interface binding, dynamic component instantiation, and efficiency management. Each of these mechanisms is supported by one of the three template variants described in this paper. We have also shown the feasibility and benefits of using the proposed template by developing a self-adaptive robotics system.

For the future, we plan to formalize the proposed template in order to obtain a component-based design pattern (similar to existing object-oriented design patterns) that prescribes how to systematically develop (self-) adaptive systems. Additionally,

we intend to develop a model-driven tool-chain that, based on this pattern, enables designers to model component-based adaptive systems and generate the corresponding implementations for some of the existing component-based platforms.

Acknowledgements

This work has been partially funded by the EXPLORE (MICINN, TIN2009-08572) and the MISSION (Fundación Séneca-CARM, 15374/PI/10) projects. Juan F. Inglés-Romero thanks Fundación Séneca-CARM for a research grant (Exp. 15561/FPI/10).

References

1. David, P., Ledoux, T.: Towards a Framework for Self-adaptive Component-Based Applications. In: DAIS'03. LNCS, vol. 2893, pp. 1–14 (2003)
2. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In: 4th International Conference on Software Engineering Research, Management and Applications, pp. 40–48. Seattle, Washington (2006)
3. Hnetynka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models. In: Gordon, I., et al. (eds.) CBSE 2006. LNCS, vol.4063, pp. 352–359. Springer-Verlag, Berlin Hidelberg (2006)
4. Akerholm, M., et al.: The SaveCCM Language Referente Manual. Technical report, Malardalen University (2007)
5. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Carnegie Mellon University (2006)
6. Coulson, G., et al.: A generic component model for building systems software. ACM Trans. Comput. Syst. 26, 1, Article 1 (2008)
7. David, P., Ledoux, T., Léger, M., Coupaye, T.: FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. Annales des Télécommunications, 45–63 (2009)
8. David, P.C., Ledoux T.: An aspect-oriented approach for developing self-adaptive Fractal components. In: 5th international symposium on software composition (SC'06), (2006)
9. Joolia, A., et al.: Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In: WICSA, pp. 131–140 (2005)
10. Garlan, D., Monroe, R., Wile, D.: ACME: Architectural Description of Component-based Systems. In: Foundations of Component-based Systems, Leavens, G. T., and Sitaraman, M. (eds), Cambridge University Press, pp. 47–68 (2000).
11. Mukhija, A., Glinz, M.: The CASA Approach to Automatic Applications. In: 5th IEEE Workshop on Applications and Services in Wireless Networks, pp. 173–189 (2005).
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented. Addison Wesley Professional (1994)
13. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, (2006)
14. Roshandel R., Hoek A., Mikic-Rakic M., Medvidovic N.: Mae: a System Model and Environment for Managing Architectural Evolution. ACM Trans Softw EngMethodol, 13(2), pp.240–276 (2004)
15. The Java Emitter Templates (JET), http://www.eclipse.org/modeling/m2t/?project=jet