A Modular Approach Dedicated to Self-Adaptive System

Mohamed Lamine Berkane, Mahmoud Boufaida, and Lionel Seinturier

Abstract—Self-adaptation is an important and desirable property for many systems, particularly for those that are deployed in dynamically changing environments. The purpose is to let the systems react and adapt autonomously to changing executing conditions without human intervention. The current lack of reusable adaptation expertise, it becomes increasingly difficult to build a system that satisfies all the requirements and constraints that might arise during its lifetime. In this paper, we propose an approach for developing policies for self-adaptive systems at multiple levels of abstraction. This approach is modular, based on the modeling of design patterns. It shows how the feedback loops and their structuring are explicitly modeled in modular way. We show the feasibility of the proposed approach with a use case based on a smart home scenario.

Index Terms—Self-adaptive systems, design patterns, modularity, reusability.

I. INTRODUCTION

Today’s software engineering implies new challenges due to the increasing complexity, size and to the unpredictable environmental changes. Adaptability is an increasingly important requirement for many systems, in particular for those that are deployed in dynamically changing environments. While the adaptation property can make systems more robust, their construction is a challenging engineering problem [1]. The important element of such systems is the Feedback Control Loop (FCL) that provides the generic mechanism for self-adaptation [1], [2]. However, the current lack of reusable adaptation expertise that can be leveraged from one adaptation system to another further makes the development difficult and costly.

In this paper, we propose an approach that enables the design of self-adaptive systems in a modular way. This approach eases the development of self-adaptive systems by using an approach based on design patterns [3], [4]. It captures the commonalities that are associated with the MAPE-K architecture [5], and associates design patterns to the modeling of MAPE-K components.

In addition, the proposed approach supports the design of the adaptation system with three levels of abstraction: the functional layer, the logical layer, and the technical one. These abstraction layers are ordered hierarchically starting with (very abstract) high layers and leading to (very concrete) low layers.

The patterns are used both in the logical layer, and the technical one. In the first layer, they provide the general models of adaptation. In the second one, they explain how the process adaptation is realized. Hence, the proposed architecture specifies the feedback loop and its structuring in a modular way. The variability [6] allows delaying design decisions in the architectural design of self-adaptive systems. Moreover, the patterns used in this work offer flexible solutions to common system development problems in each variation point.

The objectives in this paper are:
• To promote the overall view of the self-adaptive systems.
• To bridge the gap between the different levels of abstraction.
• To exploit software reuse whenever possible.

The remainder of this paper is organized as follows. Section II defines our approach for building modular self-adaptive systems. Section III refines the Monitor component of the self-adaptive systems and Section IV refines the relationship between components. In Section V, we use a case study to demonstrate the feasibility of our approach through the smart home system, and Section VI evaluates our approach. Section VII presents some related work, and finally Section VIII concludes and discusses some future work.

II. PATTERNS-BASED APPROACH FOR MODELING SELF-ADAPTIVE SYSTEMS

In this section, we present an approach that enables the design of self-adaptive systems in a modular way.

The variability in the MAPE-K architecture is expressed in each defined layer thanks to a catalog of some selected design patterns [3], [4].

The starting point is to define the feature model to design the self-adaptive systems. This feature model is a well-known approach to express variability, and can be used to build the architectural model. It contains three abstraction layers: functional layer, logical layer and technical one. The functional layer defines the common architecture of the systems with self-adaptive system components. This layer helps designer to choose the appropriate patterns in the next layer. The logical layer provides the skeleton for the overall structure of the self-adaptive systems. This layer refines the self-adaptive system components by using the design patterns defined for developing these systems [4].

The “self-adaptive” patterns impose few initial constraints upon the system being developed. Moreover, they provide general model that need to be enriched in the technical layer. With these design patterns, developers select only the adaptation mechanisms that their application will require. In this layer, our catalog contains the following patterns: Sensor Factory, Adaptation Detector, Case-based Reasoning...
and System Reconfiguration patterns [4] to define respectively the self-adaptive system components: Monitor, Analyze, Plan and Execute. The technical layer explains how the adaptation process is designed in lower level of abstraction. In this layer, we refine the design obtained from the logical level. For each pattern used in the logical level called here ‘logical patterns’, we try to find one or more GoF patterns [3] called here ‘technical patterns’. These technical patterns can specify the structure and the behavior of the whole of the logical patterns or a part of these patterns. These abstraction layers are ordered hierarchically starting with the most general (functional layer), and up to the most specific one (technical layer). During the transition from a higher layer to a more concrete one, the design of the self-adaptive systems is refined (Fig. 1(a)). Secondly, the developer builds the architectural model of self-adaptive systems by using the feature model and the selected patterns (Fig. 1(b)).

This architectural model can be used to design the self-adaptive systems. The Monitor component is further refined in the next section.

III. MONITOR COMPONENT

The main function of the Monitor component in the in the functional layer is to collect the information from the managed resource and its context that reflect changes of the monitored system.

In the logical layer, the Sensor Factory pattern [4] may be used when each observed component is distributed and provides an interface that can be probed for the required information. In this pattern, a Simple Sensor object can be used to sensor the component that needs to be monitored in the managed resource. In addition this pattern uses three main objects: Sensor Factory, Resource Manager and Registry (Fig. 1(b)).

- The first object manages the addition and the removal of sensors in the managed resource, and the Clients interact with this object in order to gain access to a sensor.
- The second object determines if an existing sensor can be shared with one or more clients, and also, it determines if the managed resource has enough resources to deploy a new sensor.
- The last object is responsible for tracking deployed sensors across the managed resource.

In the technical layer, we can find three GoF patterns from Sensor Factory pattern [3]: Observer, Composite and Command.

- The Observer pattern can be used in the Client object and the Simple Sensor one. The Client object plays the role of the observer and the Simple Sensor object plays the role of the subject. The main objective of this pattern is to enable a self-adaptive system to observe the managed resources and the environmental conditions that may lead to reconfigurations. The observer role collects the information about the monitoring system and its environment. The subject role is used to represent any component that needs to perform monitoring in system.
- The Composite pattern can be used between three objects: Abstract Sensor object, Complex Sensor object and Simple Sensor one. These three objects play the role of component, composite and leaf respectively. The

![Fig. 1. Overview of the proposed approach.](image-url)
component role shares an interface to common operations such as pushing and pulling data. The composite role allows the reporting complex data types from various leaves, and performing on board computations. The leaf role represents the basic sensor. It is able to report Boolean, integer and real data types.

- The Command pattern is used to specify the behavioral of adding a new sensor in the monitoring system. It uses four objects: Client object to play the client role, Sensor Factory object to play the invoker role, both Resource Manager object and Registry one to play the concrete command role. In this pattern, the invoker role manages the addition and the removal of sensors in the Resource Manager and Registry objects. The Clients interact with these objects in order to gain access to a sensor. The first concrete command role (defined by Resource Manager object) determines if an existing sensor can be shared with one or more clients and also, determines if the managed resource has enough resources to deploy a new sensor. The second concrete command role (defined by Registry object) is responsible for tracking deployed sensors across the managed resource.

IV. RELATIONSHIP BETWEEN COMPONENTS

In the functional layer, the relationship between components is determined by the Knowledge component. This component expresses the information base that is shared by the activities of MAPE-K paradigm.

<table>
<thead>
<tr>
<th>First Component</th>
<th>Second Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>Caller</td>
</tr>
<tr>
<td>Sensor Factory</td>
<td>Client</td>
</tr>
<tr>
<td>Adaptation Detector</td>
<td>Trigger</td>
</tr>
<tr>
<td>Case Based Reasoning</td>
<td>Decision</td>
</tr>
</tbody>
</table>

In the logical layer, the Knowledge component enables specifying the way that the design patterns of the logical layer cooperate and interoperate together. The Sensor Factory pattern uses sensors to intercept the context of the observed system. The Adaptation Detector pattern observes sensor data via its Client interface. When this pattern finds a situation where a process needs to be adapted, it creates Triggers with appropriate information to Case-based Reasoning pattern. This pattern finds appropriate decisions that are associated with the Triggers and passes them to System Reconfiguration pattern. This pattern can be used to reconfigure the components in the system. For identifying the relationships between design patterns in the logical layer, we follow the Invocation-based composition approach [7]. In this approach, two or more patterns are disjoint and they have no class in common. The roles of these patterns are only connected through one or more method calls. Table I presents an overview of the different relationships between the design patterns in the logical layer.

For identifying the relationships between design patterns in the technical layer we followed the Class-level interlacing approach [7]. In this approach, two or more patterns have one or more classes in common. The roles of these patterns are implemented by different sets of methods and attributes in these shared classes. Table II presents an overview of the some relationships between design patterns in the technical layer.

<table>
<thead>
<tr>
<th>Class</th>
<th>Patterns of Class</th>
<th>Role(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Sensor</td>
<td>Observer, Subject</td>
<td></td>
</tr>
<tr>
<td>Composite Sensor</td>
<td>Leaf</td>
<td></td>
</tr>
<tr>
<td>Client</td>
<td>Observer, Observer</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Client</td>
<td></td>
</tr>
<tr>
<td>Chain-of-responsibility</td>
<td>Handler</td>
<td></td>
</tr>
<tr>
<td>Abstract Sensor</td>
<td>Composite, Component</td>
<td></td>
</tr>
<tr>
<td>Complex Sensor</td>
<td>Composite, Composite</td>
<td></td>
</tr>
<tr>
<td>Factory</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>Sensor Registry</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>Manager</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>Trigger Pool</td>
<td>Strategy</td>
<td></td>
</tr>
<tr>
<td>Inference Engine</td>
<td>Strategy, Context</td>
<td></td>
</tr>
<tr>
<td>Fixed Rules</td>
<td>Strategy, Strategy (algorithms)</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>Strategy, Strategy (event)</td>
<td></td>
</tr>
<tr>
<td>Rule</td>
<td>Strategy, Strategy (condition)</td>
<td></td>
</tr>
<tr>
<td>Decision</td>
<td>Strategy, Strategy (action)</td>
<td></td>
</tr>
<tr>
<td>Chain-of-responsibility</td>
<td>Handler</td>
<td></td>
</tr>
</tbody>
</table>

V. CASE STUDY “SMART HOME”

In this section, a case study is used to demonstrate the feasibility of our approach. We have selected a smart home scenario. The smart home is considered as a product line for home automation. We analyze the smart home based on the three layers presented in Section II. By doing so, we aim at improving the modularity and the reusability of the software components involved in the system.

A smart home is a house equipped with a set of computing entities, such as sensors, which collect physical information (lights, window openers, thermostats, etc.) and actuators (to open the windows, to activate the alarm, etc.) that change the state of the environment [8]. The most important functional requirements for a smart home are: monitoring and changing the status of devices, and installing new devices. The considered smart home is composed of one or more floors, each floor consists of rooms, and a room holds several controlled devices. The smart home system offers higher-level functionality in which several sensors and
actuators are working together to achieve energy saving, climate control system, security system, and so on. It provides its homeowners comfort, security, energy efficiency (low operating costs) and convenience at all times. In this case, we try to study the scenario of security system.

A. Application Design

This section presents the scenario of a security system in smart home by using the proposed approach.

The functional layer defines the different components of the security scenario by using the MAPE-K architecture.

In the Monitor component we use the following sensors: doors, windows, glass break and motion detector.

The Analyze component can specify the symptoms that require the system to take specific actions. For example the glass break sensors, door sensors and motion detectors must be used to detect if persons who are not allowed to enter the home try to do so.

In the Plan component, we specify the main decisions to be taken after detecting the intrusion, like activating an alarm, and/or turning on the lights in the house.

In the Execute component, we specify the actions recommended by the plan activity. For example, in the alarm decision, we can define two kinds of actuators: the siren and the bell.

The logical layer refines the smart home system to specify the security system scenario. The rest of this section focuses on the Monitor component.

Generally, the different sensors used in smart home are distributed components that are externally available. Each sensor provides an interface that can be used to identify the context situation of the environment. In the Monitor component of this layer, we use the Sensor Factory pattern; this pattern may be used when components (Sensors) are distributed and each component (Sensor) provides an interface that can be probed for the required information. In this pattern, we can identify three parts: the sensors, the composition and the management of different sensors (registry and resource manager).

In the technical layer, we focus on two GoF patterns identified in Section III: Observer pattern and Composite pattern. In the Sensor Factory pattern of logical layer, we can identify three parts: the sensors (Observer pattern), the composition of sensors (Composite pattern), and the management of different ones: registry and resource manager (Command pattern). The different sensors used in the security scenario have the ability to be composed. They collect the context information and to communicate with each other to require the information.

B. Application Development

In this section, we present the implementation phase of the security scenario by using Observe pattern specified in technical layer. ECaesarJ is selected as the language to implement the smart home with patterns [9], [10]. This language is an extension of Java and integrates the features of its predecessor languages CaesarJ. It facilitates the better modularity and development of reusable components. In addition, it introduces distinction between required and provided interfaces of classes. The pointcuts of CaesarJ have been replaced by a more general notion of events.

C. Case of the “Observer” Pattern

The “event” notion defined by EcaesarJ language is used to specify the structure of the Observer pattern. An event has a source (or a subject) and destinations (or observers) interested in this event [8]. There are two kinds of event: explicit and implicit. The explicit events are explicitly triggered by the source and serve as notifications to the destinations that need to react to a change in the source. The implicit events consider all identifiable changes of program state. In the security scenario, we define one explicit event “activityDetected”. When the alarm system is in an armed state, then any activity within the house can be considered as intrusion like: door opened, window opened, broken glass and/or motion detectors (Fig. 2).

| 1 | abstract cclass IActivityDetection { |
| 2 | abstract event activityDetected(); |
| 3 | cclass AlarmControl requires IActivityDetection { |
| 4 | event intrusion() = activityDetected() && if(isArmed()); |
| 5 | intrusion () => |
| 6 | fireAlarm (); |
| 7 | } |

Fig. 2. Observer pattern in smart home with EcaesarJ.

VI. DISCUSSION

In this section, we evaluate our approach with some qualitative criteria from software engineering: reusability and modularity.

A. Reusability

Reusability refers to the degree to which existing applications can be reused in new applications. The MAPE-K architecture is considered as a reference for self-adaptive systems. Its value is considered as high for representing architectural models and mapping these to various applications. The self-adaptive patterns provide reusable models that can be instantiated across different domains, thus facilitating the reuse of adaptation expertise. GoF patterns also facilitate the reuse of abstract family classes as design structures makes them more readily accessible to developers of systems. In EcaesarJ, the Collaboration Interfaces minimize interdependency between components and their integration to the application. In this way components can be reused with other bindings in other context within the application or in another application. In addition to these, EcaesarJ uses the events and state machines to play the role of Observer pattern and State one.

B. Modularity

Modularity is the degree to which the components of a system may be separated and recombined. In the design, our approach supports the modularity at the different layers (of abstraction), and between the different components of self-adaptive systems. By using MAPE-K architecture, all the main functions of self-adaptive systems are defined in the fundamental building blocks; these blocks can communicate between them via the Knowledge component. Each self-adaptive pattern defines the different objects that can be used to itemize a specific situation in self-adaptive systems. Beside the GoF patterns use the notion of role to define the
modularity. The roles define the functionality of the participants in the pattern context. The different patterns assign roles to their participants, for example subject and observer for the Observer pattern. In the implementation, Ecaseard uses Collaboration Interfaces to implement the different roles of patterns. The Collaboration Interfaces minimize interdependency between components and their integration to the application.

VII. RELATED WORK

Recent works in the engineering of self-adaptive software systems have proposed various kinds of approaches: Frameworks (like Rainbow [11], StarMX [12] and AMT [13]), middleware platforms (like MADAM [14], MUSIC [15] and CAPPUCINO [16]), and model-based techniques. This section overviews selected efforts conducted by researchers to facilitate the development of self-adaptive software systems in the model-based techniques. Ramirez and Cheng [17], [18] present several patterns for developing adaptive systems. These patterns are classified into three key elements of adaptive systems: monitoring, decision-making, and reconfiguration. They facilitate the separate development of the functional logic and the adaptive logic. The authors present some patterns to specify the self-adaptive systems at high level of abstraction. However they do not provide a solution to use these patterns at low level of abstraction. Gomaa et al. [19], [20] propose some patterns to specify the dynamic behavior of software architectures (master/slave, centralized, server/client, and decentralized architectures). These patterns are helpful to the developers implementing dynamic adaptation systems. These approaches support only some kinds of software architectures (like centralized and server/client architectures). However, these patterns are only used to the specific architectures. In [21], [22], the authors propose a domain-specific modeling language for adaptive systems based on use cases called “Adapt Cases”. Adapt Cases use and refine the concept of UML use cases for the explicit and formal specification of adaptivity. However, the authors do not offer an approach to integration of the Adapt Cases into the engineering process of adaptive systems. In [23], [24], the authors propose an approach called Genie. This approach uses the model-driven engineering techniques for self-adaptive software systems. In addition, they use models for specification and generation of middleware-based software artifacts. Genie is based on architectural models to support generation and execution of adaptive systems for component-based middleware technologies.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a layer-based approach for designing self-adaptive system in a modular way. Based on design patterns our approach supports the modularity of the adaptation system in three levels: the functional layer, the logical layer and the technical one. In each layer, we have defined the different parts of self-adaptive system: monitoring, decision-making and reconfiguration. Finally, in each component, and for each layer, we use patterns to facilitate the reuse of adaptation expertise at design phase. These patterns separate the adaptation logic from the functional one. This separation of concerns facilitates the reuse of adaptation designs across multiple adaptation systems. The current implementation of self-adaptive systems is limited in languages composition. However, it still remains an open question how to compose different models and languages in a cost-effective and reliable way. We need to further investigate and experiment with different technologies to find what the feasible approaches to include multiple implementation possibilities are.

REFERENCES

Mohamed Lamine Berkane is a Ph.D student at the Computer Science Department of the University of Constantine 2, Algeria. His research interest includes design patterns, aspect-oriented programming and modularity in self-adaptive systems.

Mahmoud Boufaida is a full professor at the Computer Science Department of the University of Constantine 2, Algeria. He is the head of the LIRE laboratory. His current research activities are about advanced information systems, web databases and software engineering.

Lionel Seinturier is a full professor at University Lille 1, France. He is leader of the Inria/Lille1 Spirals research team (previously ADAM). His current research activities are about distributed systems, middleware, component based software development, service oriented computing and architecture, aspect oriented software development.