Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2014, Article ID 536362, 21 pages
http://dx.doi.org/10.1155/2014/536362

Research Article

Design Patterns for Self-Adaptive RTE Systems Specification

Mouna Ben Said,[1] Yessine Hadj Kacem,[1] Mickaël Kerboeuf,[2] Nader Ben Amor,[1] and Mohamed Abid[1]

[1] *University of Sfax, ENIS, CES Laboratory, Soukra km 3,5, BP 1173, 3000 Sfax, Tunisia*
[2] *University of Brest, Lab-STICC, MOCS Team, France*

Correspondence should be addressed to Mouna Ben Said; mouna.ben-said@ceslab.org

Received 30 November 2013; Revised 2 April 2014; Accepted 28 April 2014; Published 14 July 2014

Academic Editor: Markus Happe

The development of self-adaptive real-time embedded (RTE) systems is an increasingly hard task due to the growing complexity of both hardware and software and the high variability of the execution environment. Different approaches, platforms, and middleware have been proposed in the field, from low to high abstraction level. However, there is still a lack of generic and reusable designs for self-adaptive RTE systems that fit different system domains, lighten designers' task, and decrease development cost. In this paper, we propose five design patterns for self-adaptive RTE systems modeling resulting from the generalization of relevant existing adaptation-related works. Combined together, the patterns form the design of an adaptation loop composed of five adaptation modules. The proposed solution offers a modular, reusable, and flexible specification of these modules and enables the separation of concerns. It also permits dealing with concurrency, real-time features, and adaptation cost relative to the adaptation activities. To validate our solution, we applied it to a complex case study, a cross-layer self-adaptive object tracking system, to show patterns utilization and prove the solution benefits.

1. Introduction

Compared to desktop systems, embedded systems, and particularly RTE systems, are more complex and difficult to develop. They are subject to a multitude of constraints, such as resource limitations and time, and execute in a continuously and highly variable environment. The addition of self-adaptivity to such systems further hardens and delays their development especially with the current lack of reusable designs and development tools for self-adaptive RTE systems [1–3]. Lightening the task of self-adaptive system designers and reducing the development cost and time to market represent a major challenge in the field. Numerous research works aiming at decreasing the complexity of self-adaptive systems development have been proposed in the literature. They tried to exploit software engineering approaches, such as the model driven engineering (MDE) paradigm, to provide rapid and simplified designs [4]. They tackled different system layers with different adaptation granularities and goals and different environmental constraints [3, 5]. However, there is still a lack of generic and reusable designs that are target independent, fit different system domains, and ease the design of these complex systems.

Design pattern development is a relatively old discipline that has been proven to be beneficial for getting fast and reusable designs [6]. Unfortunately, patterns that are devoted to adaptive systems are still not well tackled in the literature. Existing solutions are mostly dedicated to desktop and distributed systems, thus ignoring important constraints that characterize RTE systems (e.g., resource limitations). They also tackle only the software part of the system, thus disregarding the hardware side which is as essential as the software side in embedded systems design [7]. Those particularly intended for RTE systems domain are, to the best of our knowledge, almost absent.

In this paper, we aim at proposing a pattern-based specification of a modular self-adaptive system based on the external adaptation approach [3] which separates the adaptable system from the adaptation logic. We append to the adaptable system an adaptation loop formed by five modules: four modules forming the so-called MAPE loop [8] (*Monitor*, *Analyze*, *Plan* or *Decide,* and *Execute* or *Act*) and an additional fifth one, named *Assess*, that we propose to include in the loop. To promote reusability and flexibility of the design, the adaptation modules are represented as design patterns. We survey relevant adaptation-related works and generalize them to extract a generic adaptation terminology that we use to construct our design patterns. The proposed patterns handle concurrency, real-time features, and adaptation cost relative to the adaptation activities. Our patterns are then used in the specification of a loop-based self-adaptive RTE system by applying them to a case study representing an adaptive object tracking system. This work is to be integrated in a model-based approach for design pattern recognition to guide RTE system designers to build self-adaptive RTE systems models.

This work involves different contributions. The first one lies in the development of five patterns for the MAPE loop adaptation modules which permits the promotion of their reuse, separately, in other contexts. It also promotes reusability and modularity of the design. The second contribution consists in the utilization of the proposed patterns for the specification of a modular self-adaptive RTE system whose structure is based on the MAPE loop and the external adaptation approach. The use of patterns enables designers to model their specific adaptive systems and gives them the freedom to insert additional options that have not been specified in the design. Moreover, a complete self-adaptive system is composed of the five modules of the adaptation loop. However, being scalable and extensible, our solution enables the designers to specify their own self-adaptive systems using the modules they need. They have the liberty to remove modules from the loop or add specific information to it. We prove this benefit through case studies illustrating different use of the patterns. In addition to these contributions, an extended version of the adaptation loop has been proposed by adding the *Assessor* module to the MAPE loop in order to enable the evaluation of its behavior. Another novelty of this work is that patterns are described using the unified modeling language (UML) models that are annotated using the MARTE (modeling and analysis of real-time and embedded systems) [9] profile. The MARTE profile permits the jointly modeling of the hardware and software parts of embedded systems using a rich terminology for the specification and analysis of RTE systems. The above benefits have been illustrated through the application of our patterns to different examples of self-adaptive RTE systems.

The remainder of this paper is organized as follows. Section 2 surveys relevant related works in the adaptive embedded systems field. Section 3 introduces the RTE systems domain through the description of the external adaptation approach and the presentation of an architectural view of the extended MAPE loop. The description of the five proposed patterns is then given in Section 4. In Section 5, we illustrate, through two case studies, the patterns utilization for the specification of different self-adaptive RTE systems with different patterns combination. Finally, Section 6 concludes the paper and mentions future work.

2. Related Work

A self-adaptive system is a system that is able to change its structure or behavior at run-time in response to the execution context variations and according to adaptation engine decisions [10]. The design of adaptive embedded systems presents many challenges due to the complexity of the problem it handles. A common basic challenge is optimizing system nonfunctional properties (e.g., maximizing output quality) while meeting internal and external constraints (e.g., real-time constraint). For example, a high quality of service may require a high utilization of system resources, such as CPU cycles and memory space, and implies high energy consumption.

The self-adaptive systems domain is relatively old and thus very rich. In the present review, we limit our study to research works particularly tackling self-adaptive embedded systems, which are relevant to the definition of our patterns.

Self-adaptation in embedded systems has been well tackled in the literature. Several approaches are based on low-level development process integrating adaptation techniques in the classic system on chip codesign flow [11–16]. Later, the development of adaptive systems at a low level has become a tedious task due to the growing complexity of modern systems and dedicated applications. Designers have then resorted to high abstraction level approaches [17–21], typically based on the model driven engineering (MDE) paradigm [22] with the UML/MARTE profile which is the most upcoming standard for embedded systems model driven development. This methodology has been proven to be well appropriate to embedded systems design [23]. It eases the modeling of self-adaptive systems by avoiding dealing with technical details thus promoting reusability. Details about the previously cited works may be found in [5].

In addition to the above approaches, several projects [24, 25] have been realized in the literature to help guide and ease self-adaptive systems development. Researchers have equally developed middleware [26–28], frameworks [29, 30], languages [31], and tools [32] for this aim. CARISMA (context-aware reflective middleware system for mobile applications) [27] and MADAM (mobility and adaptation enabling middleware) [26] offer adaptation middleware to facilitate the development of mobile applications. CARISMA exploits the principle of reflection to enable mobile application to dynamically adapt to context changes. It also offers a conflict resolution approach that treats conflicts that may be incurred by the reflective behavior. The MADAM project proposes a component-based design of both adaptation middleware and mobile applications. Each component has a set of implementations offering the same functional properties but different nonfunctional ones. The adaptation middleware decides on the implementation that best meets user needs. It is composed of three core components: a context manager which monitors the change in user requirements and context elements, an adaptation manager which takes the appropriate adaptation

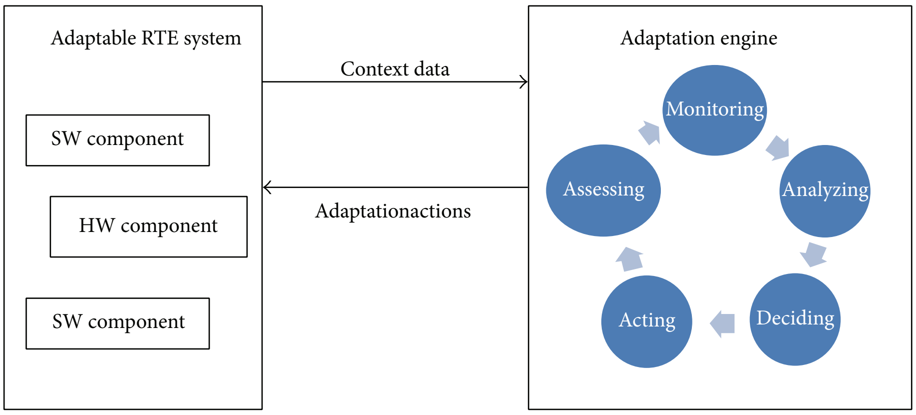

Figure 1: External adaptation approach: adaptable system + adaptation loop.

decisions, and a configurator which reconfigures the mobile application according to the adaptation decisions.

The author of [29] proposed a model-based framework that helps automate the development of self-adaptive embedded systems. He uses formal methods to specify system features which are the embedded system, events triggering reconfiguration, and reconfiguration requirements. The author proposes a process formalization that permits extending of the original system model to a self-adaptive one based on its formal specification. The resulting model can be the input of existing model-based simulator or code generator.

All the previously described approaches are beneficial since they facilitate and fasten the development of adaptive systems. However, they present some weaknesses. They are generally domain-specific which limits their applicability for diverse systems. They are also not sufficiently generic since they tackle specific adaptation problems, which consequently compromises their reusability as well as their ability to adapt to new system requirements and constraints. Additionally, most of them only focus on the software side adaptation while ignoring the hardware side which is essential in the embedded systems design.

The development of design patterns is a promising alternative approach to deal with the above problems. A design pattern gives a higher abstraction view of a commonly recurring problem, thus promoting the reusability and extensibility of the design. Works dealing with pattern-based adaptation are not numerous. Some were interested in defining the internal functioning of adaptation modules [6, 7, 33] while others rather focused on the structure and organization of adaptation functions. Concerning patterns dedicated to the architecture of adaptive systems, we cite Weyns et al. [34] who proposed patterns to decentralize multiple adaptation loops in large and complex self-adaptive systems. In [35] authors proposed a dynamic self-adaptation pattern for distributed transaction management in service-oriented applications (SOA). SOA coordination patterns are used to deal with the coordination of distributed transactions. In [36] a taxonomy was proposed for self-adaptation patterns at both component and ensemble levels. At the component level, authors describe the basic components that may compose self-adaptation patterns. At the ensemble level, mechanisms by which components can be composed into ensembles are presented.

As for patterns dealing with the internals of the adaptation functions, Gamma and colleagues [6] proposed design patterns to specify the behavior to dynamically reconfigure four types of software architectures; master/slave, centralized, server/client, and decentralized architectures. Schmidt et al. [33] proposed a set of patterns that can be used for the development of adaptive middleware. For instance, the virtual component pattern [37] permits the adapting of a distributed application to embedded systems' memory constraints. The component configurator pattern enables an application to change its components' implementations at run-time. In [7] authors proposed a set of patterns aiming at adapting distributed networked systems in order to satisfy requirements and constraints that arise at execution time. Patterns are classified, according to their purpose, into three principle categories: monitoring, decision-making, and reconfiguration activities. For example, Sensor Factory pattern is a monitoring pattern dedicated to component-based distributed infrastructure and intends to automatically deploy sensors across a network and probe the distributed components. These patterns are useful for the development of adaptive systems in different domains. However, they do not fit the real-time and embedded systems domain since they do not deal with RTE systems constraints. Also, they are limited to only addressing the software part of the system and are most appropriate for distributed systems.

3. Self-Adaptive RTE Systems

3.1. External Adaptation Approach. Two types of adaptation approaches have been identified in [3] to define how adaptivity is incorporated into a self-adaptive system: external and internal approaches. In internal approach, the adaptation logic is mixed with the system functional logic. In this case, the adaptation engine is system dependent and thus difficult to maintain, evolve, and reuse. However, in the case of external approach, illustrated by Figure 1, the self-adaptive system is defined by an adaptable system and an external adaptation loop. This approach permits the separation of concerns thus offering a reusable and customizable adaptation engine. Since most existing works are based on an external approach, in order to promote scalability, maintainability, and reusability, we adopt the external approach when developing our patterns.

3.2. The MAPE Adaptation Loop. A common structure of the adaptation mechanism owned by a self-adaptive system has been defined in the literature through the MAPE loop [3, 8, 38, 39]. It is composed of sensors, effectors, and four adaptation modules: *Monitor*, *Analyzer*, *DecisionMaker*, and *Actor*. In this work, we propose an additional module named Assessor that serves to adapt the adaptation loop in order to guarantee the required performances of the self-adaptive system. The above adaptation modules are briefly described hereafter as follows.

(i) Sensor collects data about the status of the system and its environment.

(ii) *Monitor* processes the collected data to decide about relevant changes and then trigger change events.

(iii) *Analyzer* examines change events that occur in the system to detect if an adaptation is required. It can also identify the source of the change. Monitoring and analyzing modules stand for all forms of observation and evaluation of systems' execution such as performance monitoring, safety inspection, and constraint verification [10].

(iv) *DecisionMaker* generates an adaptation decision which specifies what elements to change and how to change them in order to best meet system requirements. Two common approaches are used in the literature to construct *DecisionMakers*: a rule-based approach and an intelligent approach. The second approach does not fit the real-time and embedded systems domain because of its requirements in terms of computing time. Adaptationactions can be classified into two categories [40]: parameter adaptation/tuning and compositional adaptation mechanisms. The former modifies application parameters that determine the behavior. The latter exchanges algorithmic or structural system components with others to improve the system outcome.

(v) *Actor* applies the decision to the system. It maps actions to effectors' interfaces.

(vi) Effector is related to an adaptable system element and is responsible for applying adaptationactions to it.

(vii) Assessor evaluates the adaptation cost-effectiveness and performs, if needed, adjustments of the adaptation controller in order to meet the required performances.

Figure 2 illustrates the global architectural view of the loop-based self-adaptive system. It specifies an abstract view of the system with low coupled modules. The adaptation logic is presented by five modules and each refers to an element of

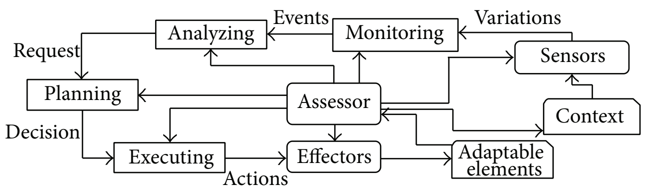

Figure 2: Global architectural view of the adaptive system.

the adaptation loop. The adaptable system is composed of a set of adaptable elements which are modified by effectors. The contribution of this paper consists in developing a modular architectural template for self-adaptive RTE systems based on the extended MAPE loop. The modules forming the proposed template are presented as design patterns in order to permit their reuse, separately, in other contexts.

4. The Proposed Patterns Description

In this section, we present the description of five patterns: RTE *Monitor*, RTE *Analyzer*, RTE *DecisionMaker*, RTE *Actor*, and RTE Assessor. The description follows the pattern template in [41]. In this paper, we give details of six fundamental fields which are the pattern name, problem, intent, context, motivation, and solution. We use the UML standard diagrams annotated with the UML/MARTE profile stereotypes to present structural and behavioral views of patterns' solutions. These solutions are the result of the abstraction and generalization of many relevant existing adaptation-related works [12–16, 24–26, 29, 42].

4.1. RTE Monitor Pattern

Name. RTE *Monitor*.

Problem. The problem treated by this pattern is the detection of an irregular status of an RTE system which results from relevant variations of internal and external context elements.

Intent. The RTE *Monitor* pattern permits the continuous control of the status of one or more RTE system properties in order to detect relevant changes and trigger events. It takes into consideration the system stability issue by minimizing events trigger through the selection of only important context variations. It also handles concurrency and real-time features relative to the control operations.

Context. This pattern is used in the first step of development of a self-adaptive RTE system. The designer has to define the system to adapt and his adaptation requirements by answering the "what to *monitor*?" question.

Motivation. The starting point that triggers adaptation mechanism in an RTE system is the context variation detection. Therefore, to be self-adaptive, an RTE system first needs to integrate a monitoring module that permits continuously controlling and updating of the status of its execution context. Additionally, the execution context of an RTE system is very

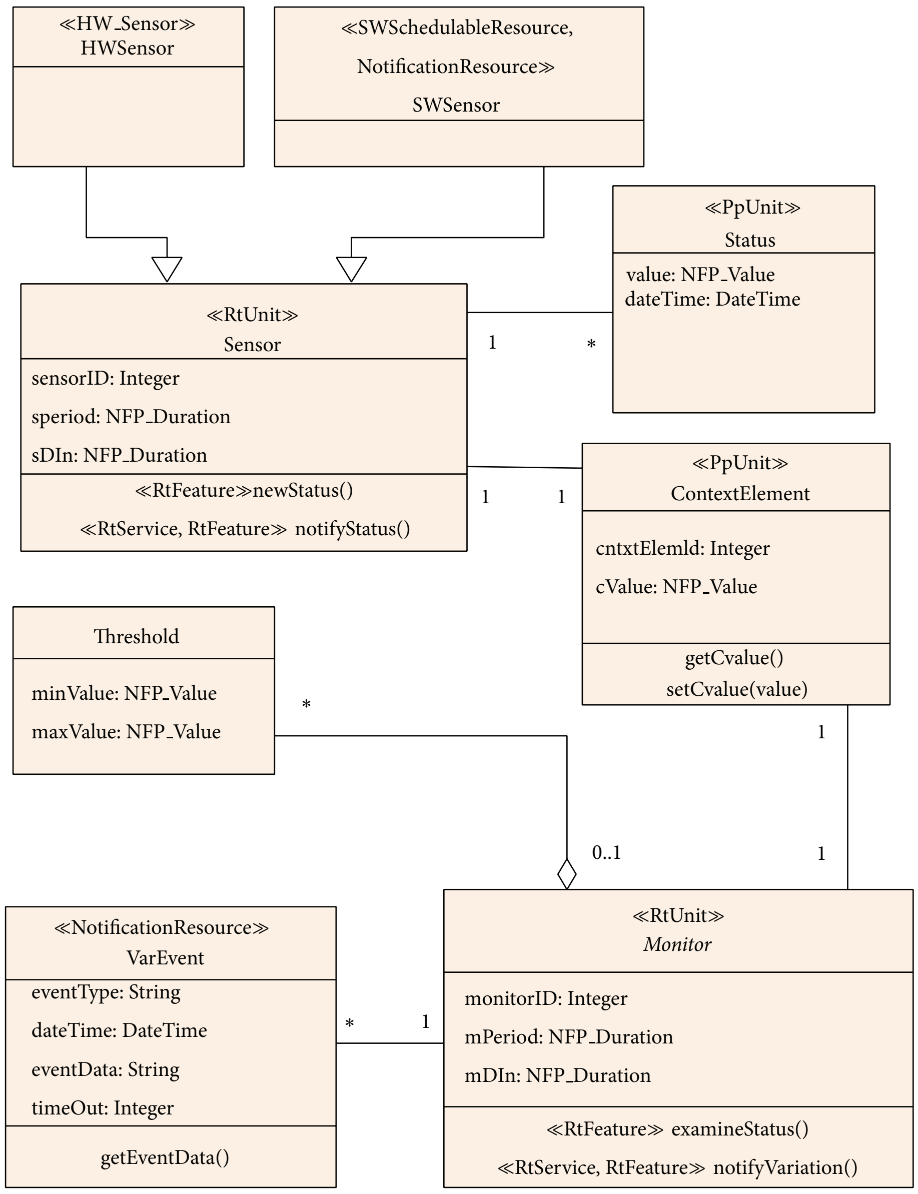

Figure 3: Structural view of the RTE *Monitor* pattern.

fluctuant so that context variation detection risks are very frequent. Therefore, in order to have a stable adaptive system with the minimum of reconfigurations, a monitoring step is required to restrict the number of treated changes by approving only relevant ones.

Solution. This pattern represents a monitoring module that permits the observation of status of RTE system context properties.

Structural View. Figure 3 shows the class diagram that explains the structural view of the pattern.

Participants

(i) *ContextElement* represents an internal or external property of the system which is observed by the *Monitor*, such as CPU load, battery life, and network bandwidth. It is a passive unit that carries information about the status of a system property and is concurrently accessed by the Sensor and *Monitor*. It is thus stereotyped as "PpUnit." It specifies its concurrency policy through the *concPolicy* attribute ("sequential", "guarded", or "concurrent").

(ii) *Sensors* are responsible for data collection about the status of *Context Elements*. A Sensor is associated with each *ContextElement* and provides measures of its status. We classify sensors into two categories: a hardware sensor, stereotyped as "HW_Sensor" represents a hardware device providing a measure of a physical quantity and converting it into a signal. A software sensor is defined by a software task running

concurrently on the system to measure a system property, such as CPU usage, and notify a *Monitor*. It is thus stereotyped as "SwSchedulableResource" and "NotificationResource." A notification resource is a software synchronization resource used to notify events. To keep events history, the notified occurrences can be memorized in a buffer by setting the *policy* attribute value to "memorized".
Since we are in the context of real-time and embedded systems, two basic issues have to be taken into consideration: the concurrency and the real-time features. In order to handle these issues, we annotate the active classes by the HLAM "RtUnit" stereotype [9] indicating that it is a real-time unit. An RtUnit is an autonomous execution resource that may own one or more schedulable resources and one or several behaviors. It is also capable of handling different incoming messages at the same time without worrying about concurrency issues thanks to its own concurrency and behavior controller. It owns a message queue permitting the saving of messages it receives. Messages can represent operation calls, signal occurrences, or data receptions. A message can be used to trigger the execution of a behavior owned by the real-time unit. A sensor is an active class that we annotate "RtUnit."

(iii) *Status* stores the measures realized by sensors in order to keep track of context information history which is important to determine the trends of context elements variations [26] and consequently to improve predictions. Status indicates for each measure the date and time and the value. The latter is typed *NFPValue* which has different attributes that permit precisely the specifying of NFP values, such as statistical qualifier, precision, and source.

(iv) The *Monitor* is an "RtUnit" that is associated with each *ContextElement*. It examines sensing data using minimum and maximum values stored in a *Threshold* to decide if a significant variation has occurred or a certain threshold has been exceeded. Threshold may represent either interval limits indicating a regular status or allowed variation margins to be used to decide about the variation relevance. If a variation is relevant, the *Monitor* generates a variation event, stereotyped as "NotificationResource."

For the sake of system stability when self-adapting, it is recommended to define an adaptation period in order to manage the adaptation mechanism occurrence. Commonly, this period is equal to a defined number *Ne* of application iterations [12]. At every period, the monitoring module starts a control session and then the adaptation cycle is executed. Therefore, the operations executed by the sensing and monitoring modules need to specify their occurrence kind (such as periodic, aperiodic, and sporadic). Moreover, in order to respect the real-time constraint, these operations have deadlines that they are asked to meet. In order to model these real-time properties, we annotate the sensing and monitoring methods with the "RtFeature" stereotype which has the occKind, relDl, and absDl (for occurrence kind, relative deadline, and absolute deadline, resp.) attributes. In order to specify additional attributes for real-time constraints of these operations, we use the "RtService" stereotype. It permits managing of the execution priority of a real-time service by the specification of the execution kind (*exeKind* attribute) which can be either "deferred", "remoteImmediate", or "localImmediate".

Behavioral View. Figure 4 shows the UML sequence diagram presenting the execution scenario of the RTE *Monitor* pattern by showing the communication between the different objects forming it. The monitoring module starts by *Sensor* which periodically delivers a new measure of the status of the supervised *Context Element* and then it notifies the *Monitor*. The *NotifyStatus()* method execution kind is *localImmediate* in order to be immediately executed by the *Sensor*. The *Monitor* receives the new measure and updates the current status value of the *Context Element*. Then it examines the new status to decide about the relevance of the change. It can use thresholds to verify whether the measure is in the interval delimited by minimum and maximum values. The negative case indicates an irregular state causing the *Monitor* to generate a variation event and send it to the *Adaptation Controller* of the system through its *notifyVariation()* method in order to be processed and decided upon. This method occurrence is aperiodic since its execution depends on the verification result of the *examineStatus()* method. However, when executed, it has the highest priority, thus having its execution kind set to *localImmediate*.

4.2. RTE Analyzer Pattern

Name. RTE *Analyzer*.

Problem. Having the status of an RTE system context, the RTE *Analyzer* pattern responds to the question of "Does an adaptation need to be applied?"

Intent. This pattern permits the verification of constraints meeting of an RTE system and then asks for adaptation if needed. It contributes to providing a stable adaptive system by minimizing adaptation requests. It handles concurrency and real-time features relative to the control operations.

Context. This pattern is used when designing a self-adaptive RTE system, specifically when information about changes in the system context is available and system constraints are defined.

Motivation. A change in the execution context does not necessarily affect the functioning of the system, that is, violate system constraints, thus not requiring an adaptation. Therefore, a verification step is needed in order to avoid useless adaptations.

Solution

Structural View. Figure 5 shows the class diagram relative to the structural view of the RTE *Analyzer* pattern.

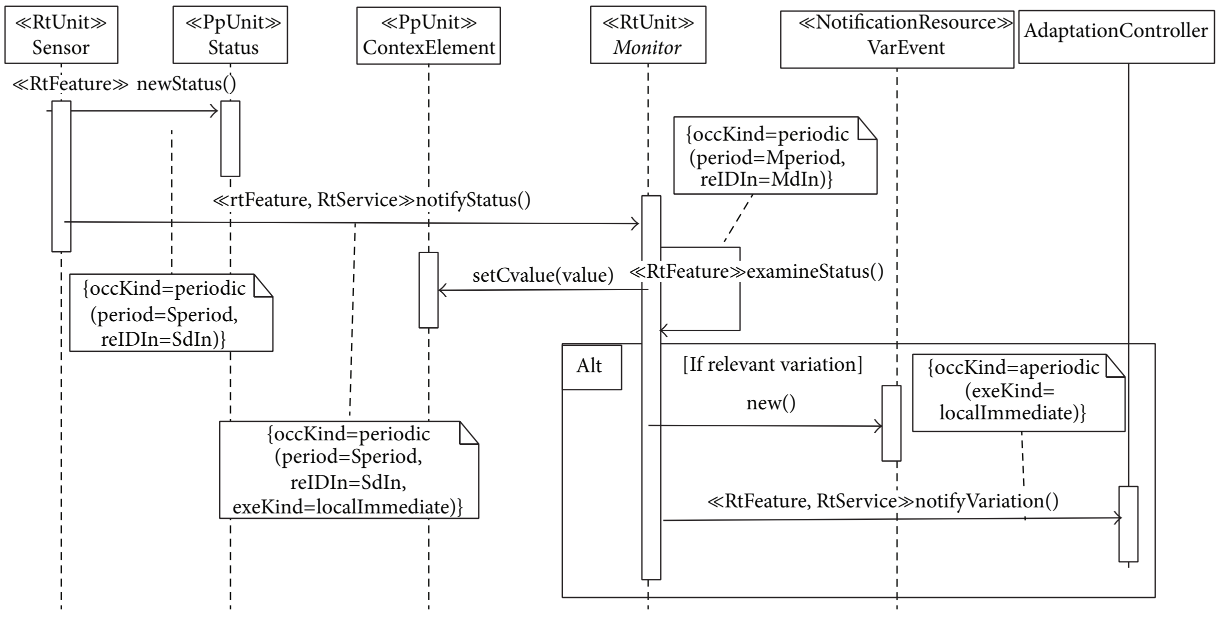
≪RtUnit≫ Sensor
≪PpUnit≫ Status
≪PpUnit≫ ContexElement
≪RtUnit≫ *Monitor*
≪NotificationResource≫ VarEvent
AdaptationController
≪RtFeature≫ newStatus()
{occKind=periodic (period=Mperiod, reIDIn=MdIn)}
≪rtFeature, RtService≫notifyStatus()
setCvalue(value)
≪RtFeature≫examineStatus()
{occKind=periodic (period=Speriod, reIDIn=SdIn)}
Alt
[If relevant variation]
new()
{occKind=aperiodic (exeKind= localImmediate)}
{occKind=periodic (period=Speriod, reIDIn=SdIn, exeKind=localImmediate)}
≪RtFeature, RtService≫notifyVariation()

Figure 4: Behavioral view of the RTE *Monitor* pattern.

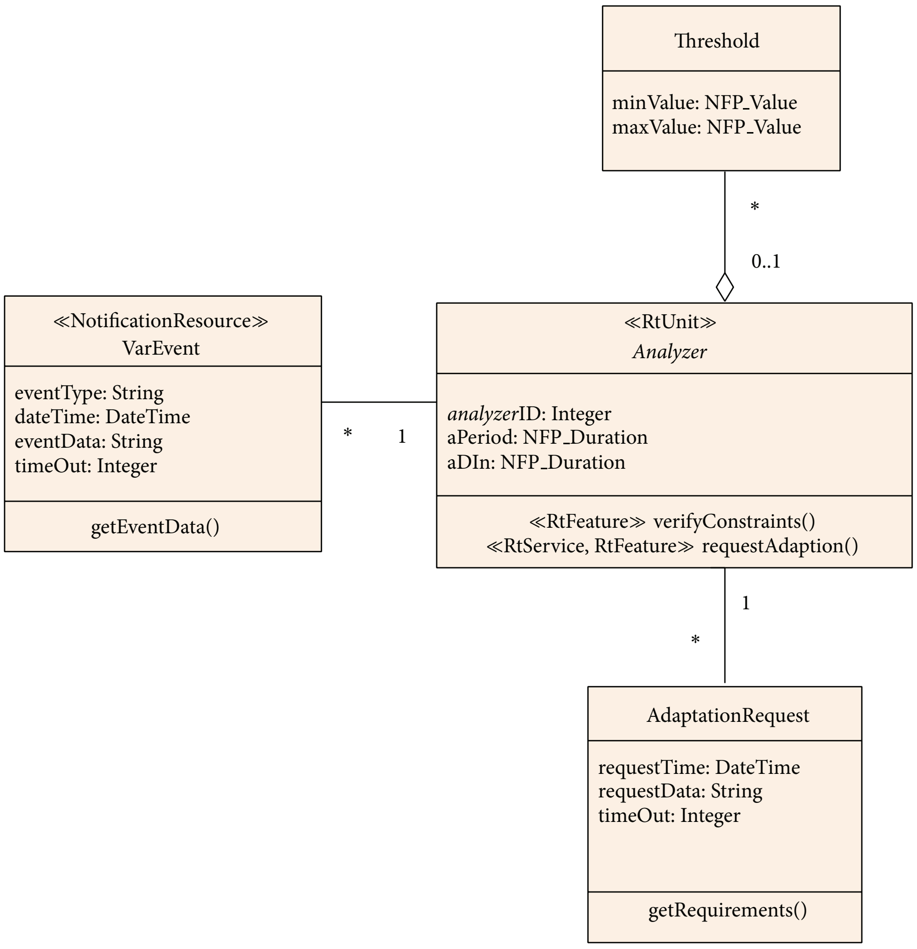

Figure 5: Structural view of the RTE *Analyzer* pattern.

Participants

(i) The *Analyzer* is responsible for the verification of the system constraints meeting. It processes a *Variation Event* that occurs to the system to decide whether an adaptationaction is required or not. It is thus an active class stereotyped as "RtUnit." It has an analysis method, *verifyConstraints()*, which generally executes a constraint miss test. The miss test may require *thresholds*. The *requireAdaptation()* method generates an *Adaptation Request*.

(ii) An *AdaptationRequest* carries request data indicating the analysis results such as the source of constraint violation. It has a timeout to be considered when treated.

Behavioral View. The behavior of the RTE *Analyzer* pattern is depicted by the UML sequence diagram in Figure 6. Having variation events received in its message queue, the *Analyzer* treats them in a loop. It asks for event data if the event is still valid; that is, its timeout is not achieved. Then it uses the collected data to verify the system constraints meeting through the *verifyConstraints()* method. Since events occurrence is aperiodic, this method's occurrence kind is aperiodic too. If constraints are not met, the *analyzer* asks for adaptation by sending an *AdaptationRequest* to the *Adaptation Controller* of the system. For more clarity, we can cite an example of real scenario: when a task entry event occurs in the system, the *Analyzer* performs a schedulability test to verify the real-time constraints meeting. If tasks' deadlines are not met, it asks for adaptation by generating an adaptation request carrying new context data.

4.3. Decision-Making Pattern. Different classifications of decision strategies have been proposed in the literature. In [3, 40, 42], authors proposed a classification in two types according to the level of granularity and complexity of the change: parametric and structural strategies. The parametric strategy modifies parameters of system components which have effect on the system behavior. It is a low-cost fine-grain adaptation applied locally on system elements. However, the structural strategy modifies the system structure such as components allocation change and their activation/inactivation. It is a high-cost coarse-grain adaptation that involves the entire system. A self-adaptive system may have a parametric or structural adaptation strategy or a combination of these. This latter case is called hierarchical adaptation that has been tackled by a number of research works like [12, 14] and has been proven to be effective.

Name. RTE *DecisionMaker*.

Problem. The problem treated by this pattern is to decide what artifact in an RTE system to adapt and how to adapt it to meet a set of requirements and constraints.

Intent. When an adaptation decision is required, the RTE *DecisionMaker* pattern decides what system elements to change and how to meet requirements and constraints. This pattern defines the adaptation strategy to apply. It can be based on parameters tuning of system's changeable elements, the modification of system's structure, or a hierarchical adaptation coordinating both strategies.

Context. This pattern is used when an RTE system exhibits new constraints or requirements due to change in its execution context. It is used in conjunction with the RTE *Analyzer* pattern.

Motivation. When modeling a self-adaptive RTE system, designers need to specify the adaptation strategy to use to calculate the adaptation decision. The RTE *DecisionMaker* pattern permits modeling of three types of adaptation strategies.

Solution

Structural View. We designed the RTE *DecisionMaker* pattern at a high abstraction level so that it is simple enough and generic to permit the design of a hierarchical adaptation decision-making by considering two different adaptation strategies at once: the parametric fine-grain and the structural coarse-grain strategies.

Participants. The structure of this pattern, depicted in Figure 7 , is composed of two basic classes: a *DecisionMaker* (DM) and a *ConfigurationManager.*

(i) *DecisionMaker* is the principal class of the pattern. It is responsible for generating the adaptation decision that best meets the requirements of an *adaptation request*. It initiates a hierarchical decision-making by cooperating with fine- and coarse-grain configurationmanagers. It asks for partial reconfiguration decisions. Then it coordinates between them through its *coordinateDecision()* method which generates the final decision that is encapsulated in an *AdaptationPlan* and sent through the *notifyPlan()* method to the *Adaptation Controller* of the system. The DM is therefore an active class stereotyped as "RtUnit." An illustrative example of the hierarchical adaptation decision-making is the GRACE platform proposed in [14]. Authors proposed a hierarchical adaptation approach performing expensive global adaptations occasionally at large system changes (e.g., application entry or exit) and low-cost limited-scope per-application adaptations frequently at the start of every frame. Since we are in the context of RTE systems, whose behavior needs to be predictable, the overhead of adaptation activities has to be taken into account while designing such systems. Adaptation cost refers to the worst case execution time (WCET) and required resources of adaptation modules. In our case, the adaptation cost is equal to the cost of deciding and acting modules which necessitate time to find the best adaptation solution and then apply the changes on the system. We suppose that the cost of the other loop modules is negligible. In order to capture the adaptation cost, we use

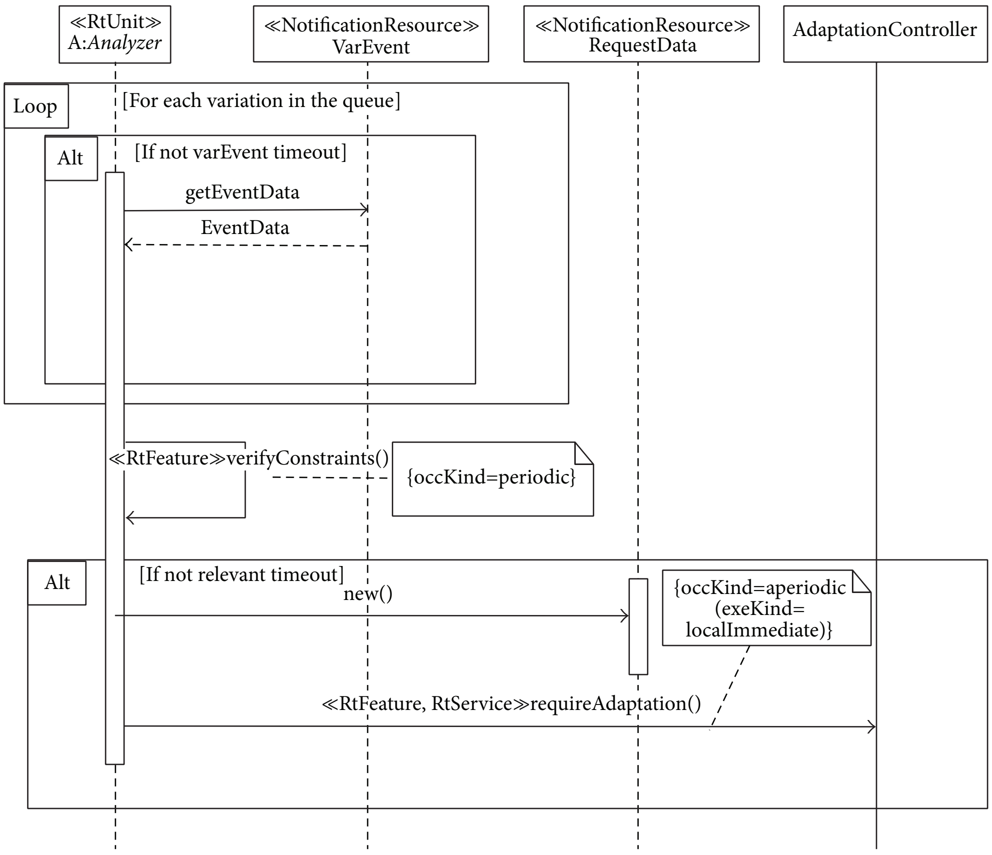

Figure 6: Behavioral view of the RTE *Analyzer* pattern.

the "ResourceUsage" stereotype which offers a set of nonfunctional properties representing consumed amounts of resources, such as the WCET taken from a computing resource, the used memory, and the consumed energy.

(ii) *ConfigurationManager* is the generalization of *adaptableSystem* and *ChangeableElement*. It is an "RtUnit" responsible for the management of *adaptableElements* configurations. It delivers, when required, the next mode that best responds to system requirements using its *getNextMode()* method. It is stereotyped as "ResourceUsage" to capture its adaptation cost.

(iii) *AdaptableSystem* represents the system to adapt as a whole. Its behavior is specified using a UML state machine which manages global system reconfiguration through structural modifications.

(iv) *ChangeableElement* represents an element of the adaptable system that is amenable to change. Similarly to the *AdaptableSystem*, it owns a state machine managing fine-grain reconfiguration of a *ChangeableElement* which is based on simple parameters modification.

Behavioral View. The behavior of the *DecisionMaker* pattern is modeled using a sequence diagram illustrated in Figure 8 accompanied with state machines of the adaptable system and changeable system elements. In fact, the configuration selection is modeled by state machines composed of a set of modes and transitions between them. The triggering of an event ensures the transition from one mode to another.

To model global adaptation decision, we use MARTE capabilities for reconfigurable systems modeling defined by the modal behavior model of the CommonBehavior package [9]. A state machine, stereotyped as "ModeBehavior," is used to model the dynamics of the adaptable system configurations. It is composed of a set of mutually exclusive modes, stereotyped as "Mode," each characterized by a configuration, and transitions between modes, stereotyped as "ModeTransition." As for the design of local adaptation decision, we use the MARTE extension proposed in [43] for the design of software fine-grain adaptation in RTE systems. The behavior of a changeable element is controlled by a state machine stereotyped as "SwAdaptor" which is composed of a set of configurations stereotyped as "ElementaryMode" and switching transitions stereotyped as "ElementaryModeTransition." An elementary mode represents a quality level of the changeable element. It is characterized by a combination of configuration parameters and its implied output quality and resources usage. These characteristics are to be compared to change requirements and constraints in order to select the best next mode. Examples of fine- and coarse-grain state machines are found in [9, 43], respectively.

When the DM receives a valid adaptation request, it processes the data that it captures to determine requirements and constraints to take into consideration. Then it decides

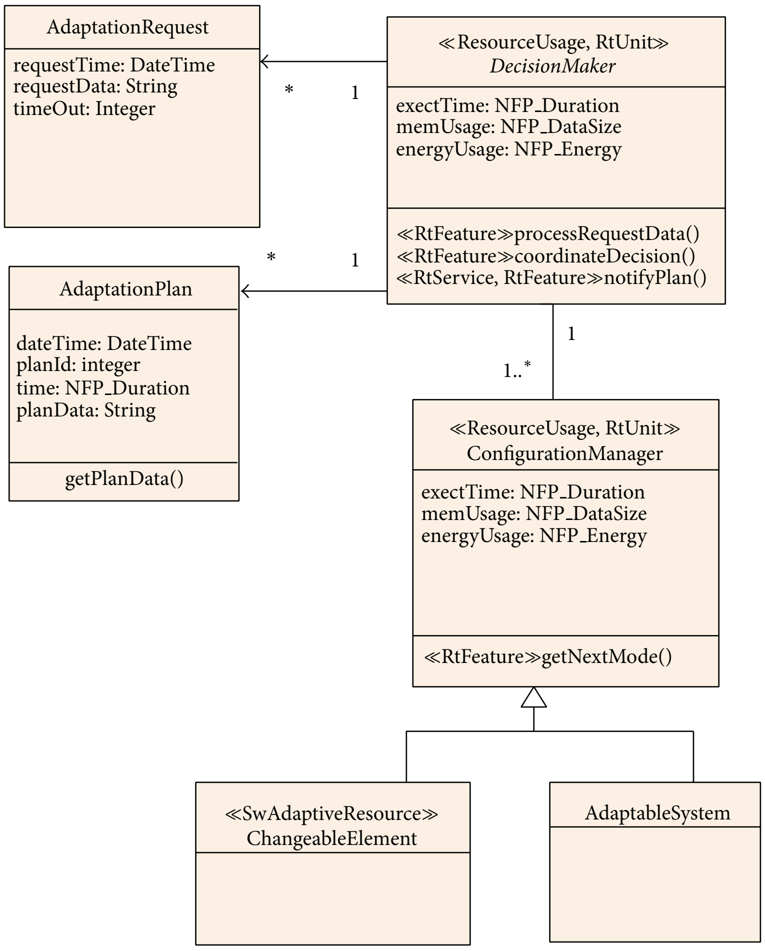

Figure 7: Structural view of the RTE *DecisionMaker* pattern.

whether a local, structural, or both decision strategies are necessary and which elements of the system to change. If local adaptation of a *ChangeableElement* is required, the DM triggers mode switch of this element by invoking the *getNextMode()* method, giving then requirements and constraints to respect. The "ElementaryTransition" that best meets constraints is then activated and the destination "ElementaryMode" is selected and returned back to the DM. The same scenario is applied to the adaptable system which returns back the next "Mode" of the whole system. Having received destination modes decisions, the DM coordinates between them, if needed, generates the final adaptation plan, and notifies the system *AdaptationController*.

4.4. Acting Pattern

Name. RTE *Actor*.

Problem. Having an adaptation decision for an RTE system, the RTE *Actor* pattern deals with the definition of who and how to apply this decision.

Intent. This pattern permits refining of a final adaptation plan into a set of adaptationactions, each to be applied to a changeable element of an adaptable RTE system. It defines the effector responsible for the application of an adaptationaction.

Context. The RTE *Actor* pattern is used when an adaptation plan needs to be applied to a RTE system.

Solution

Structural View. Figure 9 represents the structure of the RTE *Actor* pattern.

Participants

(i) The *Actor* is an active class that is responsible for analyzing the received *AdaptationPlan* in order to refine it into a set of one or more *AdaptationActions*. It also assigns each action to its specific *Effector* through its *assignEffector()* method.

(ii) An *AdaptationAction* is an atomic activity permitting the reconfiguration of a system *ChangeableElement*. It is the generalization of two types of actions: *ParametricAction* and *StructuralAction*. The former consists in parameters tuning of a system element. The latter represents the set of possible structural modifications

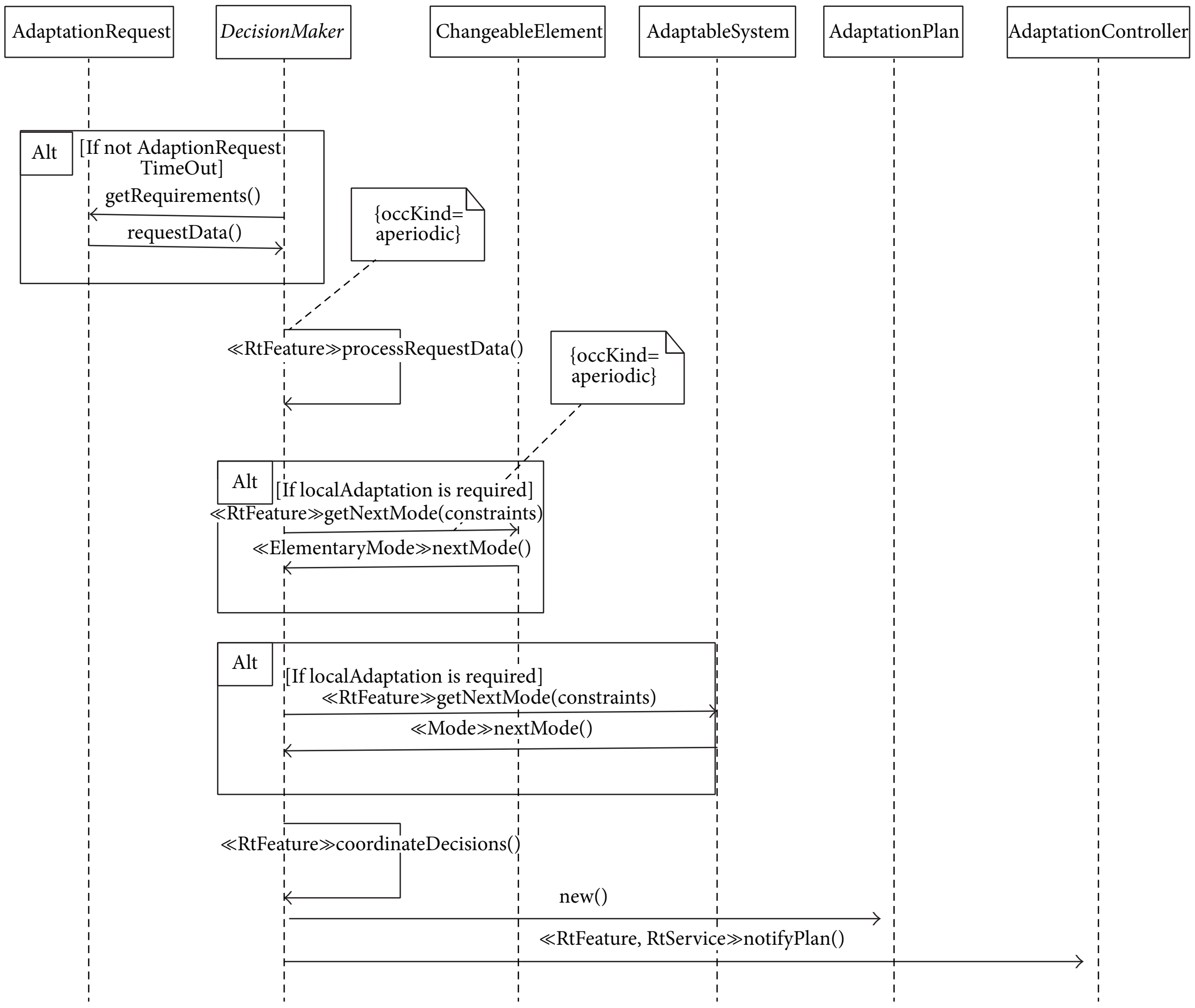
AdaptationRequest
DecisionMaker
ChangeableElement
AdaptableSystem
AdaptationPlan
AdaptationController
Alt
[If not AdaptionRequest TimeOut]
getRequirements()
requestData()
{occKind= aperiodic}
≪RtFeature≫processRequestData()
{occKind= aperiodic}
Alt
[If localAdaptation is required]
≪RtFeature≫getNextMode(constraints)
≪ElementaryMode≫nextMode()
Alt
[If localAdaptation is required]
≪RtFeature≫getNextMode(constraints)
≪Mode≫nextMode()
≪RtFeature≫coordinateDecisions()
new()
≪RtFeature, RtService≫notifyPlan()

Figure 8: Behavioral view of the RTE *DecisionMaker* pattern.

that can be applied on a system component, such as the addition/removal and activation/disactivation of a component.

(iii) The *Effector* is an interface between the adaptation decision and the adaptable system elements. It is a real-time unit responsible for applying an adaptationaction on its associated *ChangeableElement*. It is therefore target specific. An *Effector* can be either a *HWEffector* stereotyped as “HWI/O” or a *SWEffector* implemented as a software task. The application of an *AdaptationAction* is the main adaptation operation having an effect on the adaptation costs. Therefore, each *Effector* is stereotyped as “ResourceUsage” to specify its resource consumption amounts when applying an action in order to be taken into account in the adaptation cost evaluation.

Behavioral View. Figure 10 represents the behavior of the RTE *Actor* pattern.

The *actor* performs the refinement of an *AdaptationPlan*, if it is still valid, into a set of atomic *AdaptationActions*. Then, it assigns each adaptationaction to the appropriate *Effector* which is responsible for its application on the associated *changeableElement* via its *applyAction()* method.

4.5. Assessing Pattern

Name. RTE Assessor.

Problem. The RTE *Assessor* pattern deals with the issue of adaptation cost-effectiveness. It treats the question of “How well do we adapt?”

Intent. This pattern is used for evaluating and adapting a self-adaptation loop controlling an RTE system in order to enhance the self-adaptation capabilities to better deal with context variations. It performs statistics and estimations and applies parameters tuning adaptation.

Context. This pattern represents the final module in the RTE systems adaptation loop. It is therefore used when an

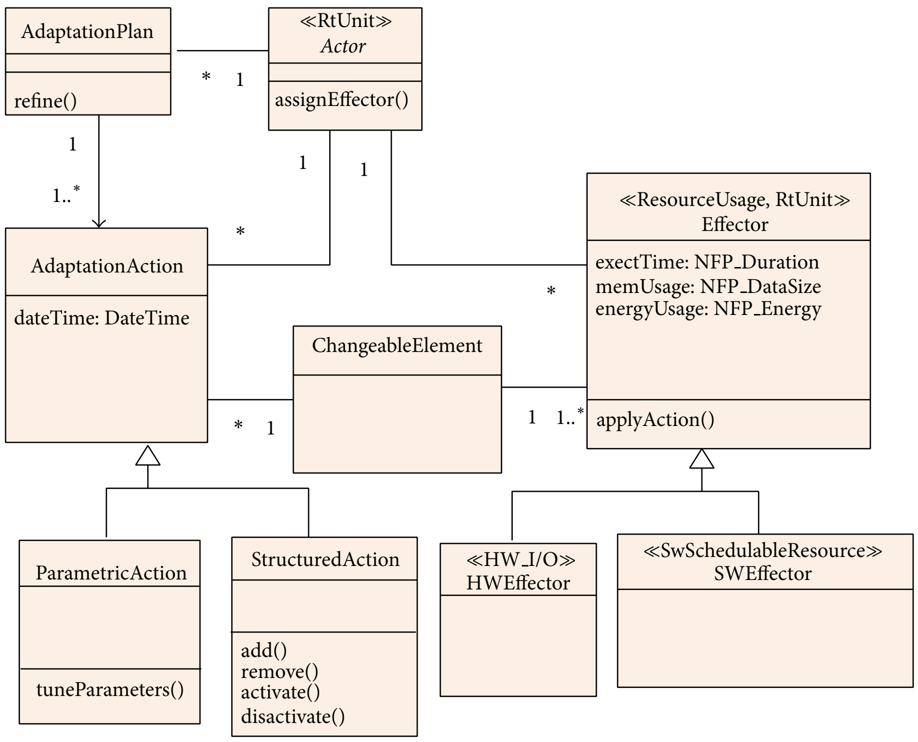

FIGURE 9: Structural view of the RTE *Actor* pattern.

adaptation loop design exists and is attached to an adaptable RTE system, in order to control and adapt the loop.

Motivation. The execution context of RTE systems is continuously variable. New environmental changes may arise during the system execution. Therefore, there is a need for a control module that evaluates the cost-effectiveness of the adaptation decisions, detects their inefficiency, and regulates the adaptation parameters to meet the required performances and better handle new context variations.

Solution

Structural View. We propose a simple and generic structure of the assessing pattern depicted in Figure 11.

Participants

(i) *AdaptationHistory* is a repository of decisions historically generated by the different adaptation modules. It can be the history of monitored status, generated variation events, adaptation requests, or adaptation decisions.

(ii) *ControlParameter* represents a parameter of an adaptation module which can be tuned in order to adapt the adaptation loop to new context variations. It can be, for example, the monitoring period or the analyzing thresholds.

(iii) The *Assessor* performs statistics and estimations, evaluates the performance of the adaptation loop, and adjusts parameters of the latter to improve the adaptation cost-effectiveness. It uses a repository of *AdaptationHistory* to make statistics and evaluates adaptation performance by calculating *Metrics*. It is also responsible for generating an adaptation decision through its *assess()* method and applying it on *ControlParameters* through the *adjust()* method.

Behavioral View. The behavior of the RTE *Assessor* pattern is represented by Figure 12. The *Assessor* evaluates performance *Metrics*, such as adaptation cost, and *analyzes* a repository of *AdaptationHistory* to make statistics. Based on these calculations, it generates via the *assess()* method an assessment *Report* according to which it adapts the adaptation engine by acting on some *Control Parameters* through the *adjust()* method. For example, having noticed a high adaptation overhead, the *Assessor* augments the monitoring period or raises the analyzing thresholds in order to guarantee system stability. The *Assessor* is an RtUnit which may run its job either periodically, at every predefined period, or aperiodically after every adaptation loop execution. We give the designer the liberty to choose the occurrence kind of the assessment module.

Patterns Combination. The RTE *Assessor* pattern represents an adaptation engine applied to an adaptable adaptation loop. Therefore, it performs the adaptation operations of the previously studied adaptation modules. After preparing analysis data (statistics and metrics values), the *Assessor* generates, via its *assess()* method, a *Report* which represents the adaptation decision. The *assess()* method may implement an adaptation strategy to decide which control parameters to adapt and how. This issue is handled by a *DecisionMaker*

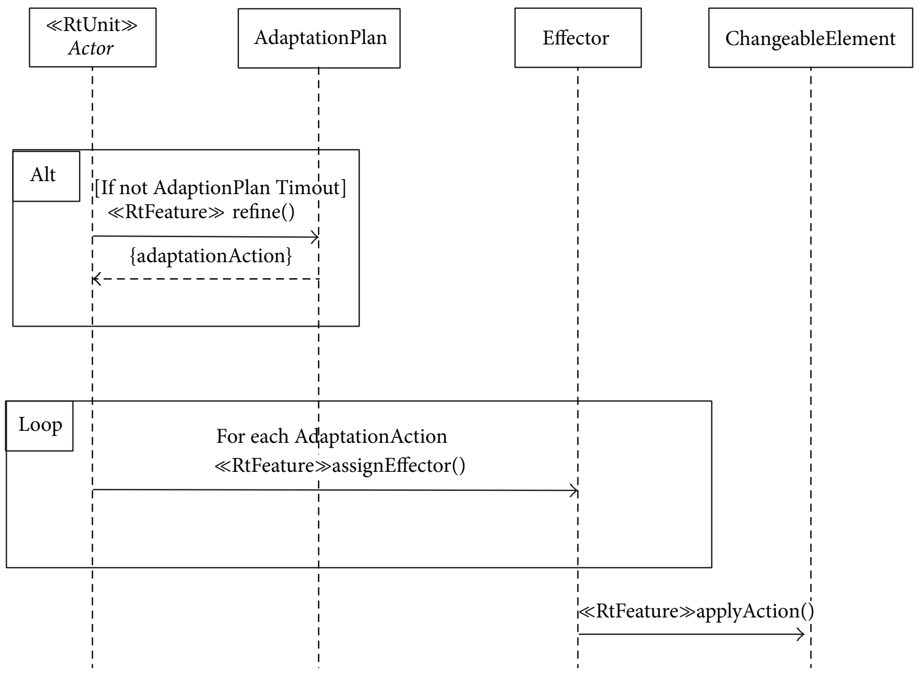

Figure 10: Behavioral view of the RTE *Actor* pattern.

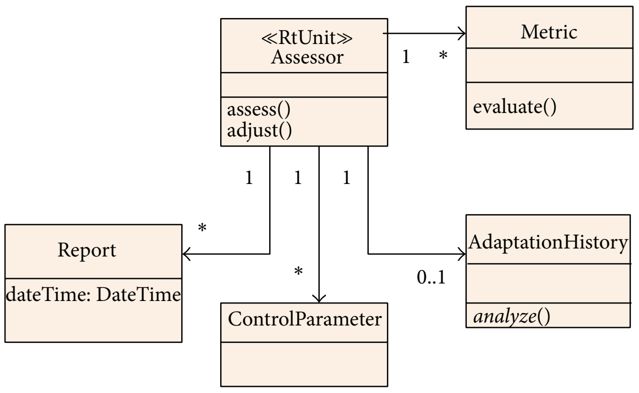

Figure 11: Structural view of the RTE Assessor pattern.

module. It is therefore convenient for designers to replace this method by the RTE *DecisionMaker* pattern. In this case, only one adaptation strategy type is used: the parametric strategy, which is applied to *ControlParameters*, which represent the changeable elements of the adaptable loop. Furthermore, the *adjust()* method, which performs the application of the *adaptationPlan* on the *ControlParameters*, can also be replaced by the RTE *Actor* pattern which permits decomposing of the adaptation decision into *parametricActions,* with each being applicable to a *ControlParameter*. The possibility of combining the RTE *DecisionMaker* pattern and the RTE *Actor* pattern with the RTE *Assessor* pattern proves the reusability and modularity of the proposed designs.

5. Patterns Utilization

This section includes three objectives: (i) illustrating the utilization of the proposed patterns through case studies of adaptive RTE systems, (ii) presenting the specification of a loop-based self-adaptive system using the five patterns, and (iii) demonstrating different and independent use of the patterns through a second case study.

5.1. Case Study 1 (Object Tracking Application). In this section, we present a case study where we apply our patterns for the design of a dynamically adaptive object tracking application proposed in [12]. We chose this case study because it is one of the rare existing works that has dealt with almost all modules of the adaptation loop, carrying adaptation at different system layers and for both hardware and software parts of an RTE system and dealing with the system stability issue. However, this work has been developed at a low level, thus lacking a high abstraction level modeling step, which is the case of most state-of-the-art works in the field. Consequently, to apply our patterns, we start by providing a table of correspondence between adaptation concepts offered by the patterns and those considered in the application example in order to evaluate the generic aspect of the patterns' solutions. Then, we present a pattern-based structural model for the case study to show how to append the proposed patterns to the framework based self-adaptive system structure.

5.1.1. Case Study Overview. The authors of [12] present a self-adaptive object tracking application, which was implemented on an FPGA-based camera. The application is composed of 10 tasks which can be implemented in HW or in SW. An

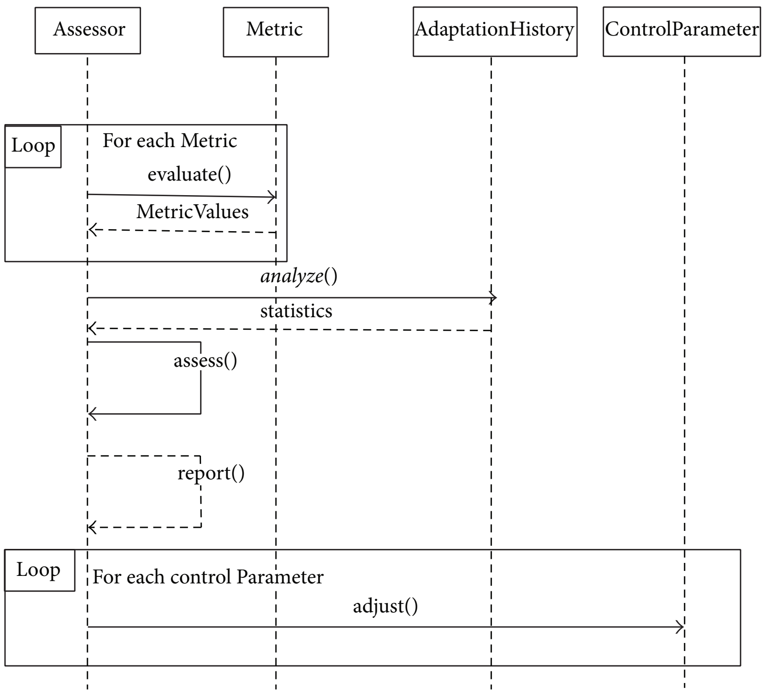

Figure 12: Behavioral view of the RTE Assessor pattern.

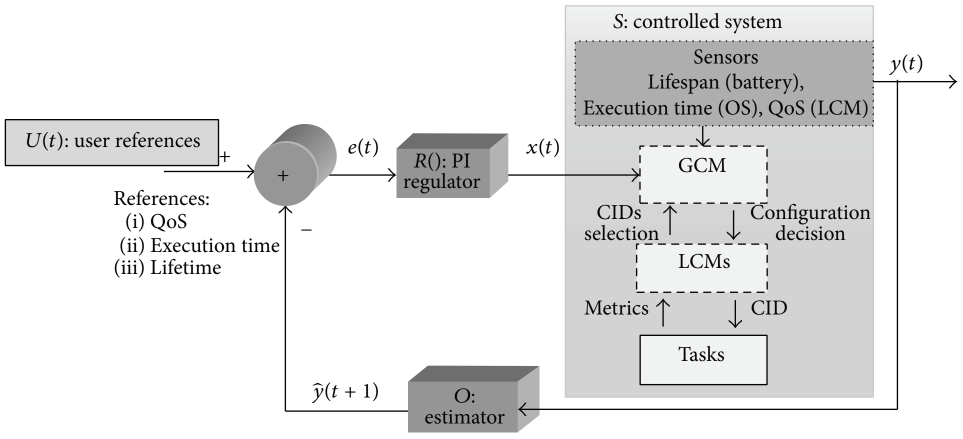

Figure 13: Global structure of the closed-loop self-adaptive system proposed in [12].

electric toy train tracking scenario is proposed to illustrate the system self-adaptivity. The scenario contains various events provoking configuration decisions. The goal is to design an embedded system able to respect a constraint while optimizing secondary magnitudes. In this case study, the regulated magnitude is the QoS indicating the tracking accuracy and the optimized ones are power and execution time.

This self-adaptive system is developed based on a cross-layer adaptation approach for self-adaptive RTE systems development. The authors propose a hierarchy of local and global configurationmanagers (LCM/GCM), as depicted in Figure 13, to separately deal with application-specific and application-independent reconfigurations, respectively. The LCM is responsible for local application algorithmic reconfigurations. An LCM is defined for each application. The GCM is responsible for architectural reconfigurations which consist in tasks migration from software to hardware on a multiprocessor heterogeneous architecture. Only one GCM is defined for the whole system.

5.1.2. Patterns/Application Features Matching. The adaptation features matching is summarized in Table 1. The train tracking application considers three context elements, the QoS metric indicating the tracking error, the execution time, and the battery life. The first regulated magnitude is the application QoS. Three sensors, each corresponding to a

Table 1: Adaptation concepts correspondence between the proposed patterns and the object tracking application.

Patterns feature	Case study instance

Context elements	Three magnitudes:
	(i) application QoS (the tracking error);
	(ii) execution time;
	(iii) power consumption

Sensors	Four observers:
	(i) a task (T10);
	(ii) SW timer of RTOS;
	(iii) battery gauge component;
	(iv) observer estimator

Adaptation period	Configuration period $Ne = 1$; that is, a new configuration is evaluated after each application iteration
Monitor + *Analyzer*	Local configurationmanager (LCM)
Monitoring Thresholds, irregular status	QoS reference (the tracking maximum error) is set to 10% and reduced to 2% within the critical area to guarantee good reactivity. Task T10 provides the LCM with the application QoS metric (error between prediction and object position): (i) a value close to 0 but lower than the reference (10%) means a very high tracking quality $\rightarrow$ it can be relaxed by reducing the application speed; (ii) a value higher than the reference $\rightarrow$ the application rate must be increased with a faster configuration.
Adaptation request	User requirements (e.g., QoS, power, and performance references) and magnitudes' values
Decisionmaker	(i) LCM for application specific algorithmic configuration decisions (ii) Global ConfigurationManager (GCM) for global architectural configuration decisions
Decision strategy	(i) LCM: algorithmic configuration selection using LCM rules; lists of configurations and transition rules based on tasks metrics. (ii) GCM: selection of the closest solution below the user reference (QoS reference). A Borda vote is used in the case of multiple solutions.
Adaptation Plan	Architectural configuration CID
Actor (refinement)	A mask-based CID analysis
AdaptationAction	Four cases result from the CID analysis: (i) nothing to do; (ii) requests to HAL to get new I/O information (physical addresses); (iii) a reconfiguration message sent by a HW task to its legal representative (LR) to activate its SW implementation; (iv) the LR directly intercepts the new CID and activates the SW task
Effector	(i) HAL (ii) HW task LR
Changeable elements	Tasks
Assessor	System stability and avoidance of reconfiguration
Performance Metrics	(i) System stability: use of proportional integrator (ii) Requests to HAL to get new I/O information (PI) regulator (coefficients: $kp = 0,25; ki = 0,25$) and a least mean square (LMS) observer (coefficient: $kL = 2^{21}$) (ii) Reconfiguration avoidance: use of a minimum delay Tk required to accept the reconfiguration overhead compared to expected benefits
Control Parameter	Configuration space restriction

magnitude, and a fourth one providing estimations of the context behavior are used. A configuration period Ne is defined to control the periodic execution of the adaptation mechanism. An LCM compares user-defined references with observed magnitudes' values to detect irregular status. It then generates a preliminary decision of application's algorithmic configuration and sends it to the GCM. The latter is responsible for global architectural implementation decisions. It decides the new system configuration according to user requirements, magnitudes' values, and the received LCM local decision. The decision strategies of both GCM and LCM are mode switch based. They use rules and conditions for mode transitions. Therefore, their behavior can be modeled using state machines. Six LCM rules are defined for the object tracking application. An example of rule mentioned in the paper is as follows.

[*If the noise level is larger than a given threshold (Th9), then activate the adaptive threshold task T9.*] As for the GCM, it decides a new global configuration j such that

$$\left[yj\left(t+1\right)=\max k\left(yk\left(t+1\right)\mid yk\left(t+1\right)\leq u\left(t\right)+x\left(t\right)\right)\right], \tag{1}$$

where $yk(t+1)$ is the controlled magnitude value for the configuration k. $u(t)$ is the magnitude reference and $x(t)$ is the output of the proportional regulator.

When the next configuration CID is determined, it undergoes a refinement step through the execution of a mask-based CID analysis by each task to get information about the new configuration to run. The *Adaptation Plan* refinement results in one of four predefined atomic *AdaptationActions*. These actions are assigned to *Effectors*, which are the HAL and the HW task legal representative (LR), to be applied on tasks.

The final functionality of case study's control loop is the performance evaluation of the self-adaptation behavior. Self-adaptivity must have a negligible overhead compared to the gain it brings. Performance *Metrics* used in the example are system stability and avoidance of reconfiguration. A PI regulator and a LMS observer are used to evaluate system stability. The reconfiguration acceptance is related to a minimum delay Tk required to accept the reconfiguration overhead compared to expected benefits. TK is calculated as follows:

$$\mathrm{Tk}=\max\left(\frac{\mathrm{TR}}{\mathrm{GT}};\frac{\mathrm{ER}}{\mathrm{GE}}\right), \tag{2}$$

where TR is the reconfiguration delay, GT is the performance gain between the new and the previous configuration, ER is the energy required for a reconfiguration, and GE is the energy gain.

An example of assessment decision is a restriction of configuration space. If Tk $\leq$ 0, then costly hardware reconfigurations (SW $\rightarrow$ HW and HW $\rightarrow$ HW) are not allowed.

We notice from the previous study that the adaptation scenario of the case study is very similar, in both structure and behavior, to our proposed patterns, used in conjunction with each other in one model. We present in the next section the pattern-based specification of the case study application to show how our proposed patterns are used and combined together.

5.1.3. Patterns Application to the Self-Adaptive Object Tracking System. The general structure of the closed-loop self-adaptive object tracking system, presented in Figure 13, is composed of 4 basic elements: the controlled system, a control function, an observer, and user references. The corresponding high-level pattern-based specification is presented in Figure 14.

The controlled system S is composed of two configurationmanagers (an LCM for the tracking application and a GCM for the whole system) and a set of tasks and sensors observing the controlled magnitude $y(t)$. The LCM encompasses the monitoring and analyzing activities. Therefore, it is modeled using the RTE *Monitor* and the RTE *Analyzer* patterns. The observed *ContextElements* represent the controlled magnitudes provided by sensors and tasks. These magnitudes, together with user references, which are represented by *Thresholds*, form the input of the LCM. The output of this latter is an *AdaptationRequest* sent to the RTE *DecisionMaker* pattern. Decision-making is insured mostly by the GCM and partly by the LCM. Parametric adaptation strategy, performed by the LCM, is applied to application tasks which represent *ChangeableElements* of the adaptation loop. The GCM is responsible for structural adaptation strategy, applied for the *AdaptableSystem*, the coordination between local and global configuration decisions, and the generation of final *Adaptation Plan* which is an architectural configuration CID.

The *Adaptation Plan* is the input of the RTE *Actor* pattern responsible for the realization of the next configuration CID. The refinement of the selected configuration, which is performed by each task using a mask-based CID analysis, is insured by an *Actor*. An *AdaptationAction* is generated for each task. The *Actor* assigns actions to *Effectors*, which are the HAL and the HW task LR. The latter apply actions on tasks, which are modeled as *ChangeableElements*.

The remaining blocks in the case study structure are the system observer O and the control function R. The observer calculates estimates of controlled magnitude for the next time slot in order to predict its evolution to anticipate the right reconfiguration decision. It is a model-based estimator that replaces new sensors' measures when they are delayed or even not available. It is therefore linked to *ContextElements* and *Threshold*. As for the controller, it permits handling of system stability and avoidance of reconfiguration based on a PI regulator and a LMS observer. It performs the adaptation loop assessment and is consequently modeled using the RTE *Assessor* pattern. The *Assessor* evaluates performance *Metrics* which are the system stability and the minimum delay for reconfiguration acceptance. *Metrics* are evaluated using different parameters such as controlled magnitudes and adaptation cost. Relations between *Metric* and the *ContextElements* are then added. The *Assessor* generates an assessment *Report* which is applied on *ControlParameter* such as the restriction of system configuration space. System configurations are designed using the *AdaptableSystem* class.

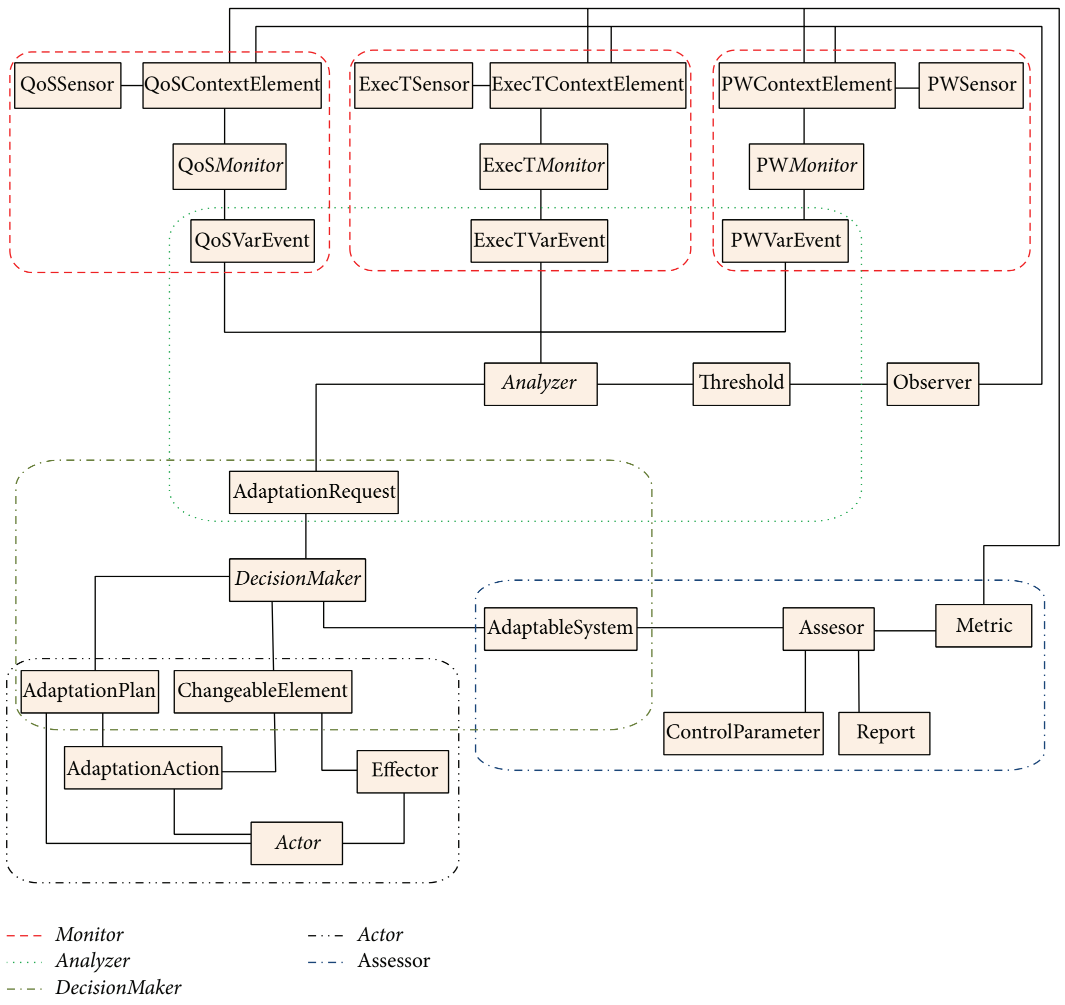

Figure 14: Patterns application to the closed-loop self-adaptive object tracking system.

5.1.4. Discussion. We can derive from the previous subsections that using a pattern-based specification for the development of a RTE system presents advantages upon the low-level development approach. Instantiating the patterns for the target system has simplified the system specification by hiding internal functional and technical details of system elements and lowering the system model size. That permits proving of the efficiency of the pattern-based solution regarding the provision of a generic and clear model since it is specified using standard modeling languages. Moreover, the use of the monitoring, analyzing, and deciding patterns permits, for instance, a better organization of both LCM and GCM activities which promotes the modularity and thus flexibility of the design. Additionally, the use of the acting pattern permits the separation between the adaptation functionalities (plan refinement and effectors assignment) and the adaptable elements (tasks), thus offering a design that is based on the external adaptation approach.

5.2. Case Study 2 (The Race Feedback Loop for Application QoS Control). This section deals with the specification of the control architecture of the resource allocation and control engine (RACE) [44]. RACE is an adaptive resource management framework for open distributed real-time embedded (DRE) systems, such as the NASA's magnetospheric multiscale mission system. The RACE's control engine employs a feedback loop permitting adapting of the system in response to variations in resource utilization and application QoS.

5.2.1. Patterns Application to the RACE Feedback Loop. In this section, we only describe the specification of the application QoS adaptation. Figure 15 illustrates the RACE's adaptation architecture. The feedback loop is composed of three principal components: *Monitor*, Controller, and Effector. The corresponding pattern-based specification is presented in Figure 16.

(i) *Monitors.* A hierarchical monitoring architecture is considered. *AppMonitors* measure end-to-end application delay using high resolution timers which periodically send the collected QoS delays to the node*monitor* of the same node. Each *nodeMonitor* tracks the QoS of all the applications running on its node and sends it, also periodically, to the *centralizedMonitor*. The latter tracks the overall system QoS

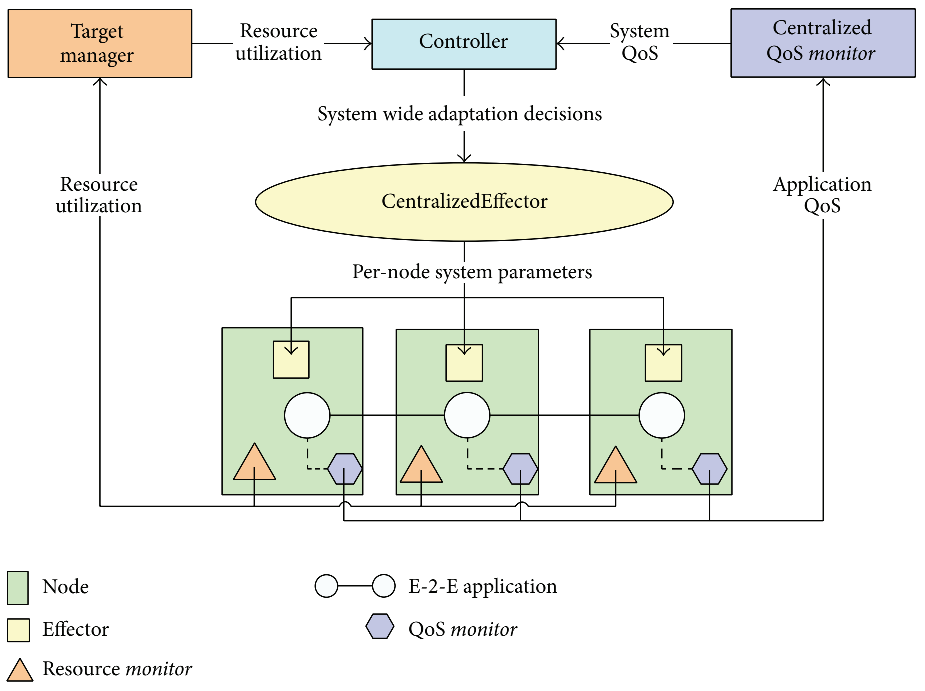

Figure 15: Structure of the RACE's feedback control loop [44].

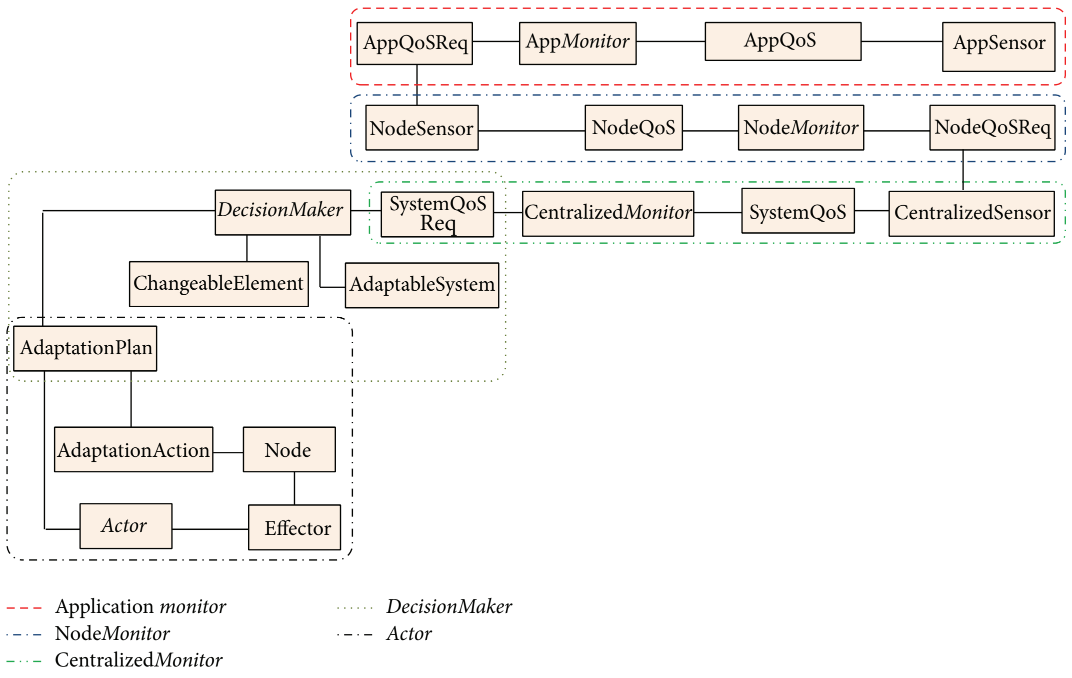

Figure 16: Patterns application to the RACE feedback loop for application QoS control.

and sends it to the controller. The RTE *Monitor* pattern can be applied hierarchically in the three levels (application, node, and system). A *HWSensor*, which represents the high resolution timer, is used to capture application QoS value. *AppMonitors* of the same node notify a *nodeSensor* by their application QoS. It is a *SWSensor*, such as a thread that is continuously waiting for events or messages from *AppMonitors* to get their data and transfer them to its *nodeMonitor*. The same entities are used for the *CentralizedMonitor* that is used to generate the system QoS.

(ii) *Controllers*. The controllers are responsible for adaptation decision-making. RACE disposes of several controllers which implement different decision algorithms that manage changes in the DRE system operational context. According to the control algorithm used, the controller modifies different configuration parameters of the system, such as applications execution rates and platform-specific QoS parameters like OS/middleware/network QoS settings for an application based on its QoS characteristics and requirements. It can also modify the allocation of resources to components. We notice from this description that both parametric and structural adaptation strategies are present in the studied adaptation engine. It is therefore convenient to use the RTE *DecisionMaker* pattern in the specification of the *Controller*. The *DecisionMaker* generates an adaptation plan that permits meeting of the system performance requirements sent by the *centralized-QoS-Monitor*. Based on the control algorithm it implements, the *DecisionMaker* selects the appropriate *ConfigurationManager* which permits modeling of the event/condition/action tuples as well as configurations details that are used to take reconfiguration decisions.

(iii) *Effectors*. The Effectors are responsible for modifying system parameters according to the adaptation decisions. Similarly to *Monitors*, *Effectors* are designed hierarchically. A *centralizedEffector* calculates the values of some system wide parameters. It also computes and transmits per-node parameters for each *nodeEffector* in the system. The RTE *Actor* pattern fits this task well. The Controller decisions represent the input for the *Actor*. The *Actor* represents the *centralizedEffector*. It examines adaptation decisions to refine parameters into a set of per-node parameters represented by *adaptationActions* and propagate them to *nodeEffector* represented by *Effectors*. The node is the *changeableElement* in this case.

5.2.2. Discussion. This case study permitted illustration of a different context (DRE systems) for the proposed patterns utilization. We equally presented a hierarchical structure of the patterns use, thus demonstrating a different possible combination of them and proving their independency.

6. Conclusion and Future Work

This paper deals with high abstraction level design of modular self-adaptive RTE systems using design patterns. The modular structure of the adaptive system is based on an external adaptation approach which separates the adaptable system from the adaptation logic in order to promote modularity and thus flexibility and maintainability of the model. The adaptation logic is modeled using an extended version of the MAPE loop. The extended loop is composed of the four common adaptation modules, the *Monitor*, *Analyzer*, *DecisionMaker*, and *Actor*, and an additional module, the Assessor, which permits the evaluation of the adaptive behavior.

We developed five patterns representing generic modeling of the adaptation loop modules. The patterns take into consideration the adaptation cost as well as concurrency and real-time features of adaptation operations which are key concerns in RTE systems design. Then, the proposed patterns were used in the development of a MAPE loop-based self-adaptive RTE system. We combined, in a first case study, the five proposed patterns and applied them to a relevant existing adaptive system, an object tracking application implemented on an FPGA-based smart camera. We also illustrated, through a second case study, the application of the proposed patterns to another context with a different combination in order to prove their reusability and independency. The case studies showed the effectiveness of using design patterns for high abstraction level modeling of self-adaptive RTE systems. Such approach permits easing and fastening of designers' job by enabling the reusability and flexibility of the design.

We plan in future work to integrate the proposed patterns in an MDE-based approach for the automatic generation of self-adaptive RTE systems. In particular, we aim at automatically detecting our implemented patterns through applying patterns recognition techniques.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] S. Hallsteinsen, E. Stav, and J. Floch, "Self-adaptation for everyday systems," in *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managing Systems (WOSS '04)*, pp. 69–74, ACM, New York, NY, USA, November 2004.

[2] A. Kofod-Petersen and M. Mikalsen, "Context: representation and reasoning representing and reasoning about context in a mobile environment," *Revue d'Intelligence Artificielle*, vol. 19, no. 3, pp. 479–498, 2005.

[3] M. Salehie and L. Tahvildari, "Self-adaptive software: landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, article 14, 2009.

[4] B. Schätz, A. Pretschner, F. Huber, and J. Philipps, "Model-based development of embedded systems," in *Advances in Object-Oriented Information Systems*, vol. 2426 of *Lecture Notes in Computer Science*, pp. 298–311, Springer, 2002.

[5] M. B. Said, Y. H. Kacem, N. B. Amor, and M. Abid, "High level design of adaptive real-time embedded systems: a survey," in *Proceeding of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD '13)*, pp. 341–350, Barcelona, Spain, February 2013.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, Mass, USA, 1995.

[7] A. J. Ramirez and B. H. C. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*, pp. 49–58, ACM, New York, NY, USA, May 2010.

[8] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[9] OMG Object Management Group, "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems," ptc/2011-06-02, Object Management Group, June 2011.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor et al., "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems and Their Applications*, vol. 14, no. 3, pp. 54–62, 1999.

[11] T. Frikha, N. Ben Amor, I. Benhlima, K. Loukil, M. Abid, and J. P. Diguet, "Self-adaptive on-chip system based on cross-layer

adaptation approach," *International Journal of Reconfigurable Computing*, vol. 2013, Article ID 141562, 17 pages, 2013.

[12] J. P. Diguet, Y. Eustache, and G. Gogniat, "Closed-loop-based self-adaptive hardware/software-embedded systems: design methodology and smart cam case study," *Transactions on Embedded Computing Systems*, vol. 10, no. 3, article 38, 2011.

[13] L. Ye, J. P. Diguet, and G. Gogniat, "Rapid application development on multi-processor reconfigurable systems," in *Proceeding of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 285–290, Milano, Italy, September 2010.

[14] V. Vardhan, W. Yuan, A. F. Harris III et al., "Grace-2: integrating fine-grained application adaptation with global adaptation for saving energy," *International Journal of Embedded Systems*, vol. 4, no. 2, pp. 152–169, 2009.

[15] W. Yuan and K. Nahrstedt, "Energy-efficient CPU scheduling for multimedia applications," *ACM Transactions on Computer Systems*, vol. 24, no. 3, pp. 292–331, 2006.

[16] N. P. Ngoc, W. van Raemdonck, G. Lafruit, G. Deconinck, and R. Lauwereins, "A qos framework for interactive 3d applications," in *Proceeding of the 10th International Conference on Computer Graphics and Visualization (WSCG '02)*, pp. 317–324, Plzen-Bory, Czech Republic, February 2002.

[17] I. Rafiq Quadri, H. Yu, A. Gamatié, E. Rutten, S. Meftali, and D. Jean-Luc, "Targeting reconfigurable FPGA based SoCs using the UML MARTE profile: from high abstraction levels to code generation," *International Journal of Embedded Systems*, vol. 4, no. 3-4, pp. 204–224, 2010.

[18] I. R. Quadri, S. Meftali, and D. Jean-Luc, "High level modeling of dynamic reconfigurable FPGAs," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 408605, 15 pages, 2009.

[19] J. Vidal, F. de Lamotte, G. Gogniat, J. Diguet, and P. Soulard, "UML design for dynamically reconfigurable multiprocessor embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, pp. 1195–1200, European Design and Automation Association, Leuven, Belgium, March 2010.

[20] F. Krichen, B. Hamid, B. Zalila, and M. Jmaiel, "Towards a model-based approach for reconfigurable dre systems," in *Software Architecture*, vol. 6903 of *Lecture Notes in Computer Science*, pp. 295–302, 2011.

[21] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Proceeding of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 371–380, ACM, Shanghai, China, May 2006.

[22] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[23] G. Gogniat, J. Vidal, L. Ye et al., "Self-reconfigurable embedded systems: from modeling to implementation," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '10)*, pp. 84–96, Las Vegas, Nev, USA, 2010.

[24] Famous Project, "Anr famous overview," http://www.lifl.fr/~meftali/famous/.

[25] N. Shankaran, J. S. Kinnebrew, X. D. Koutsoukas, C. Lu, D. C. Schmidt, and G. Biswas, "An integrated planning and adaptive resource management architecture for distributed real-time embedded systems," *IEEE Transactions on Computers*, vol. 58, no. 11, pp. 1485–1499, 2009.

[26] M. Mikalsen, N. Paspallis, J. Floch, E. Stav, G. A. Papadopoulos, and A. Chimaris, "Distributed context management in a mobility and adaptation enabling middleware (MADAM)," in *Proceeding of the ACM Symposium on Applied Computing (SAC '06)*, pp. 733–734, Dijon, France, April 2006.

[27] L. Capra, W. Emmerich, and C. Mascolo, "Carisma: context-aware reflective middleware system for mobile applications," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 929–945, 2003.

[28] S. M. Sadjadi and P. K. McKinley, "A survey of adaptive middleware," Tech. Rep. MSU-CSE-03- 35, Department of Computer Science, Michigan State University, East Lansing, Mich, USA, 2003.

[29] L. Tan, "Model-based self-adaptive embedded programs with temporal logic specifications," in *Proceeding of the 6th International Conference on Quality Software (QSIC '06)*, pp. 151–158, Beijing, China, October 2006.

[30] W. Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, and R. H. Kravets, "Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems," in *Multimedia Computing and Networking*, vol. 5019 of *Proceeding of SPIE*, pp. 1–13, Santa Clara, Calif, USA, January 2003.

[31] T. Vogel and H. Giese, "Model-driven engineering of adaptation engines for self-adaptive," Tech. Rep. 66, Hasso Plattner Institute for Software Systems Engineering, University of Potsdam, Potsdam, Germany, 2013.

[32] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair, "Genie: supporting the model driven development of reflective, component-based adaptive systems," in *Proceeding of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 811–814, ACM, New York, NY, USA, May 2008.

[33] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture*, vol. 2 of *Patterns For Concurrent and Networked Objects*, Wiley, 2000.

[34] D. Weyns, B. Schmerl, V. Grassi et al., "On patterns for decentralized control in selfadaptive systems," in *Software Engineering for Self-Adaptive Systems II*, R. Lemos, H. Giese, H. Müller, and M. Shaw, Eds., vol. 7475 of *Lecture Notes in Computer Science*, pp. 76–107, Springer, Berlin, Germany, 2013.

[35] H. Gomaa and K. Hashimoto, "Dynamic self-adaptation for distributed service-oriented transactions," in *Proceeding of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '12)*, pp. 11–20, Zurich, Switzerland, June 2012.

[36] M. Puviani, G. Cabri, and F. Zambonelli, "A taxonomy of architectural patterns for self-adaptive systems," in *Proceedings of the International C∗ Conference on Computer Science and Software Engineering (C3S2E '13)*, pp. 77–85, ACM, New York, NY, USA, July 2013.

[37] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. ORyan, "Virtual component—a design pattern for memory-constrained embedded applications," in *Proceedings of the 9th Conference on Pattern Language of Programs (PLoP '02)*, 2002.

[38] S. Dobson, S. Denazis, A. Fernández et al., "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223–259, 2006.

[39] B. H. Cheng, R. de Lemos, H. Giese et al., "Software engineering for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, vol. 5525 of *Lecture Notes in Computer Science*, pp. 1–26, Springer, Berlin, Germany, 2009.

[40] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[41] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, New York, NY, USA, 1996.

[42] J. Andersson, R. de Lemos, S. Malek, and D. Weyns, “Software engineering for self-adaptive systems,” in *Modeling Dimensions of Self-Adaptive Software Systems*, vol. 5525 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2009.

[43] M. B. Said, Y. H. Kacem, N. B. Amor, M. Kerboeuf, and M. Abid, “Fine-grain adaptation for real time embedded systems using uml/marte profile,” in *Proceeding of the Forum on Specification Design Languages (FDL '13)*, pp. 1–8, Paris, France, September 2013.

[44] N. Shankaran, D. C. Schmidt, X. D. Koutsoukos, Y. Chen, and C. Lu, “Design and performance evaluation of configurable component middleware for end-to-end adaptation of distributed real-time embedded systems,” in *Proceeding of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '07)*, pp. 291–298, Santorini Island, Greece, May 2007.